MŰEGYETEM 1782

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

# HW-SW Co-verification of Concurrent Programs

**Scientific Students' Association Report**

Author:

Levente Bajczi

Advisors:

András Vörös
Vince Molnár

2018

# Contents

# Kivonat

A folyamatosan fejlődő technika világában egyre több olyan szituáció fordul elő, ahol sok ember élete múlik számítógépek helyes működésén - legyen szó akár egy önvezető autóról, repülőről vagy atomreaktorok biztonsági rendszeréről. A legfontosabb elvárás az ilyen rendszerekkel szemben az, hogy lehetőleg elkerüljék a kritikus hibákat. Tekintve, hogy számos alkalommal történt már emberéleteket követelő katasztrófa rosszul működő programok vagy számítógépek miatt, elfogadhatjuk, hogy az ilyen rendszerek működésének alapos ellenőrzése különösen indokolt éles használat előtt - az ellenőrzés egy precíz módja az, hogy matematikailag bizonyítjuk, hogy a rendszer előre látható működése során nem történhet hiba.

Szoftverek ellenőrzésére számos sikeres eszköz létezik, amelyek a programokat formális modellként ábrázolva képesek bizonyos követelmények teljesülését vizsgálni. Azonban a teljes rendszer verifikációjával kevesen próbálkoztak, pedig hiába tökéletes a szoftver, ha egy hardveres hiba képes akár az egész rendszert térdre kényszeríteni. A hardver és szoftver együttes verifikációja még nehezebb több processzormagos környezetekre. Ennek a kutatásnak a fő célja a többmagos eszközök és a rajtuk futó konkurens programok együttes ellenőrzése. Egy olyan megközelítést fejlesztettem ki, ami nem csak a konkurens programok verifikációjával foglalkozik, de a többmagos rendszereken futó magasszintű nyelvek memóriamodelljének (MCM) megsértését is vizsgálja. A témám alapja egy korábbi kutatás, ahol a szerzők megmutatták, hogy sok, potenciálisan hibát okozó inkonzisztencia fordul elő a modern architektúrákon.

Ezen projekt keretein belül egy olyan megközelítést mutatok be, ami az ilyen biztonságkritikus rendszereket fejlesztő programozókat segíthet a formális verifikáció és MCM validáció eszközeinek segítségével, ezzel biztonságosabb, megbízhatóbb kódot eredményezve. Továbbá bemutatok egy prototípus implementációt ami ezen munkafolyamat több lépését automatizálja, ezzel a következő funckionalitást biztosítva:

- Egy {mikroarchitektúra, MCM, magasszintű nyelv} kombináció automatikus verifikációja a TriCheck segítségével, felderítve az MCM-et sértő szituációkat.
- C11 kód formális modellbe való átalakítása, ezzel lehetővé téve a formális verifikációs eszközök használatát.
- Automatikus lekérdezés generálás a fent említett verifikációs eszközök használatához.
- Visszajelzés a potenciálisan hibát okozó kódsorokról.
- Mitigáció automatikus ajánlása mutex zárak és feltételes szinkronizáció használatával.

# Abstract

In the ever-developing world of technology, more and more situations arise where the life of many people lay in the hands of computers - be it the processor of a self-driving car or an airplane, or the command center of a nuclear reactor. The most important expectation of critical software is that they should never fail in a way that could have been prevented. As there have already been many accounts of catastrophes that were caused by malfunctioning computers or programs, we need to verify these systems before deployment - which could mean, among others, to prove mathematically that no unintended outcome can ever occur in the foreseeable operational circumstances.

On the software side, there have been many successful attempts at creating a verification framework that takes a formal model and verifies whether it conforms to specified criteria. However, there is a lack of approaches targeting system level correctness - even if the software is perfect, a hardware bug can still occur that could render the whole system unsafe. The co-verification of HW-SW systems is even more challenging for multi-core systems. The main goal of this research is the co-verification of multi-core hardware and concurrent software running on top of them. I have developed an approach that not only takes multithreading capabilities into account but does so while checking for memory consistency problems. My research is based on a previous research where authors showed that there are many situations violating the Memory Consistency Model (MCM) of programming languages running on modern architectures.

In the scope of this project, I propose an approach that can help programmers developing software for such mission-critical systems leverage the tools of formal verification and MCM validation resulting in safer, more reliable code. Furthermore, I provide a proof-of-concept implementation of a tool that enables the automation of the introduced verification workflow, providing the following features:

- Automated MCM verification of a specified {Microarchitecture, MCM, High-level language} combination resulting in a list of possible MCM violations using TriCheck.
- Parsing and transformation of C11 code into a formal modelling language, enabling the use of traditional model verification frameworks.
- Automated query generation for use in above-mentioned model checking frameworks.
- Feedback of error-prone lines of code in the editor itself.
- Suggested corrections using conditionals and mutex locks for safer code.

# Chapter 1

# Introduction

Since the second half of the last century, there have been more and more occasions where the lives of many people depended on the correct operation of computers or programs running on them. This kind of dependence got built into our everyday life and now we cannot go anywhere without relying on the safe and correct operation of embedded devices. Everyone is used to the ”technical gremlins” of personal computers, for example the well-known Blue Screen of Death, but no-one would like to see that on the on-board computer in an Airbus A380-800, where 853 passengers could die if such an error happened.

In order to avoid such an accident, software engineers have two possibilities while developing mission-critical software: One is to test the system they created so rigorously that they can be sure they have not made a mistake in development, which is not only tiresome and time-consuming, but also unreliable as it introduces yet another human element. The other option is much more reliable, takes less time, but is also the more complex one: using mathematically justifiable rules one needs to prove that no unintended outcome can ever occur in the foreseeable operational circumstances. This, while being the superior option, is not always that easy to achieve as such a method is only applicable to specific types of problems. There is no general method for the formal verification of software that does not take any assumptions about the subject itself.

Even though there is no such thing as a generally applicable formal verification tool, the number of situations where we can apply them is growing day to day. In this paper I propose an approach that can broaden this variety by introducing the formal verification of multi-threaded applications by validating the Memory Consistency Models (MCM) of high-level languages running on modern architectures, then applying the found inconsistencies to actual source code to mitigate violations.

MCM violations are actually very common in modern microarchitectures - at least more common than it would be desirable. The infamous load-load hazard [2] in ARM processors is a good example of a bug that was present in an architecture so long that it actually got to the end user before a mitigation technique was released (in this case the compiler mappings were redefined in a stricter way).

The main idea behind being able to mitigate MCM violations as opposed to changing the architecture itself is that there are many situations where neither the hardware nor the underlying OS can be changed, but developers still need to create new programs for it - the simplest being an update service of an embedded IoT device that cannot get replaced every time a new hardware bug is found, but also cannot be left vulnerable. If we can solve the problem by adding a bit of an overhead, but without changing the core components of the system itself, then we potentially saved a fortune for our business while being able to concentrate on supporting older products as well as developing new ones. As an interesting comparison, developers of most modern kernels used such mitigation techniques when dealing with the Spectre/Meltdown vulnerabilities of microprocessors - taking a small performance hit [19] is worth it if we can avoid investing in new hardware for all our systems.

The report is structured as follows: Chapter 2 introduces the concepts later chapters build on, such as Memory Consistency Models. Chapter 3 provides a short overview on the proposed verification approach, which is then expanded in the next chapter, Chapter 4, going into detail about the theoretical part of the verification methodology. Chapter 6 explains the details of the implementation itself through the tool *MCMEC*, which is used to prove the applicability of the approach. This tool was used in a series of tests in Chapter 7 to provide benchmarking results showing that the approach is in fact applicable to actual problems. In Chapter 8 I introduce some approaches that are related to mine as I partly utilize them in my work, then finally in Chapter 9 I summarize the findings of this work.

# Chapter 2

# Background

In this chapter I introduce the core concepts of microarchitectures, multithreaded applications and formal model checking in order to establish the definitions that I will be referring to later on.

## 2.1 Formal Verification

**2.1.1. Definition (Formal verification).** In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics [7]. ▪

The definition of formal verification includes all the necessary information about the subject:

- It operates on an ”intended algorithm”, which needs to be represented by a *formal model* (see Section 2.1.1)
- It takes a property (or rather the *formal specification* of said property) and either proves or disproves it
- It uses the (formal) methods of mathematics, leaving no place for ambiguity on the correctness of the proof

The goal of such verification methods is to eliminate the uncertainty of the human element, meaning that it provides a way of checking if the developer of a system or algorithm made any mistakes by comparing the resulting model to the specification of the expected product.

In the context of this work this means that the verification tool takes the formal model generated from the source code of a program and the specification which states which operations cannot ever be executed at the same time (generated from the formal specification of the microarchitecture, ISA, MCM and compiler mappings) and checks if these properties are observable in the model. If any such property holds, one of the inputs must be refined to avoid such a situation.

### 2.1.1 Formal Models

**2.1.2. Definition (Formal model).** A formal model is a description method of abstract systems and algorithms that uses a well-defined, unambiguous and mathematically precise structure. ▪

In other words, everything can be called a formal model if the content thereof is specified in a way that can be interpreted in only one way.

These formal models are always an abstraction of the subject they are depicting, as we omit the irrelevant details while transforming them into such representations (see modelling in general). For example when transforming source code to a formal model one might lose the exact structuring of the source file or even operations that are unrelated to the goal of the verification process.

In this research two different formal models will be used to represent the C11 source we are aiming at verifying: UPPAAL Timed Automaton (XTA) [8] and Control Flow Automaton (CFA) [21]. The former is used to represent timed automata (meaning they have a clock variables that advance the execution), and therefore it is more suitable for more abstract systems, while the latter is better for generic program representations. Furthermore, XTAs provide a way of synchronization between its processes, therefore this functionality does not need to be implemented using some other way (such as message passing). As both have their own advantages and disadvantages it is up to the implementation to decide, for which I have made the decision to use XTA while looking for ways to expand the functionality to support CFAs as well. Both implementations come from the tool *theta* [24], which is being developed at the BME-MIT Fault Tolerant Systems Research Group (FTSRG).

What both formal model types have in common is that they operate on locations and transitions between them. Transitions can have *guard expressions* that regulate if a transition should be taken at all by evaluating them and only allowing a transition if the value is **true**. Furthermore, they can use variable arithmetic by assigning to and reading back from variables.

## 2.2 Computer Architectures

As the proposed model checking workflow spans down to the core of a system and therefore logically includes the computer architecture thereof, some definitions about this abstract concept must be set first. Other, somewhat different definitions might exist for them, hence the need for clarification.

**2.2.1. Definition (ISA).** The Instruction Set Architecture (ISA) is the formal model describing the inner operations of a given microprocessor, providing axioms and deduced rules for instructions and their behaviour, such as memory ordering [18, 26]. ▪

**2.2.2. Definition (Microarchitecture).** Microarchitecture, commonly abbreviated as *$\mu$Arch* or *uarch* is one physical realisation of a particular Instruction Set Architecture (ISA) in a microprocessor. Mathematically, the relation $a \multimap A$ can be introduced as a way to express *The microarchitectural implementation **a** is a correct implementation of the ISA **A*** [6]. ▪

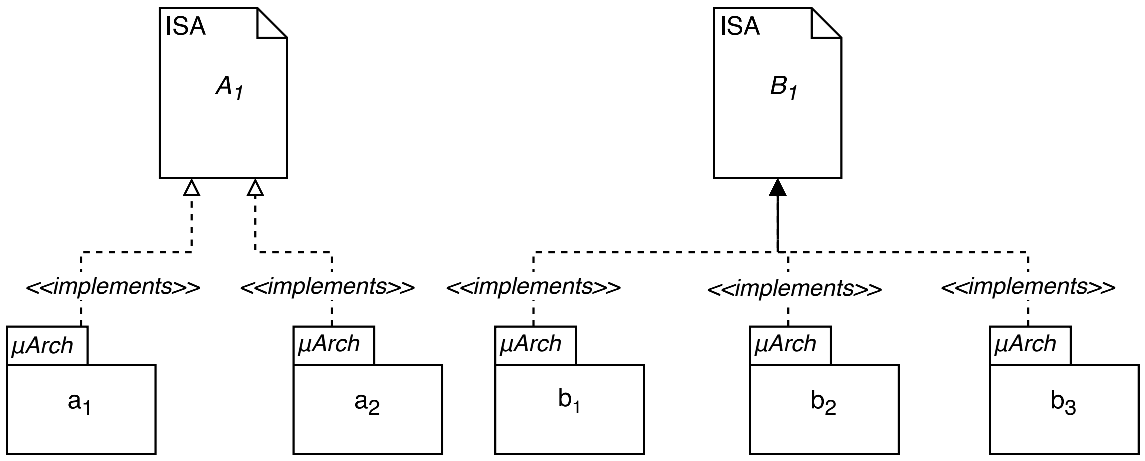


**Figure 2.1:** Two ISA definitions and uarch packages implementing them.

**2.2.3. Definition (Computer architecture).** One element of the Descartes-product of a given ISA $A$ and given microarchitectural realisations $\{a_i\}$. *Computer Architectures* $= \{a_i \times A \mid a_i \rightharpoondown\!\!\triangleright A\}$ [10]. ▪

In other words, what the ISA is for software, is the microarchitecture for hardware and these two concepts together are what we call a *computer architecture.*

As it can be seen in the three definitions above, the ISA and the microarchitecture itself are two separate and easily distinguishable description methods of a particular microprocessor. Even though the microarchitecture itself *might* differ from the ISA due to a mistake, this is negligible as in most cases the translation from ISA to $\mu Arch$ is done by automated generators (*compilers*, so to say), which (once formally verified, of course) do not output an erroneous representation - and even if this realisation is done manually, it is very easy to audit the result if it conforms to the ISA. Furthermore, as it can be seen in Figure 2.1, an ISA can have more than one correct microarchitectural implementation but generally not vice versa[1], and a single computer architecture $\Pi$ defined as the product of an ISA $A$ and a $\mu$Arch $a$ is only correct, when $a \rightharpoondown\!\!\triangleright A$. As programs interface with the processor via the ISA, it can generally be stated that any two *Computer Architectures* are binary compatible[2] when sharing the same ISA and therefore almost entirely transparent from the standpoint of software. An everyday example of this phenomenon is that even though big silicon companies (such as Intel or AMD) improve, modify and release new processors circa every quarter with a modified $\mu$Arch, there is no need for recompiling all used software after a CPU upgrade as they all share a common ISA [13].

Logically follows that if we want to verify a specific computer architecture for memory ordering violations, we can do so either by using the ISA directly, or indirectly by creating artificial microarchitectures that conform to that ISA. In this work, we will use the latter approach and create such implementations of the ISA in question that allows for the *loosest* memory constraints and therefore **if found correct, the ISA in question is also correct**.

[1]Of course if one constructs an abstraction $B$ of a specific ISA $A$, then all $a_i$ implementations of $A$ will also be correct implementations of $A$, but in the fewest of cases does this situation prevail.

[2]Once a program is compiled for a system, it can be run on any binary compatible different systems.

## 2.3 Multithreading on Multi-core Systems

There is no need for multiple threads for memory ordering violations, a bug in the pipelining system is enough, this situation has been verified countless of times (such as in [17]) and almost no further work is required until a brand new concept of pipelining processors are made up. Multithreaded applications, however, lack this saturation of verification tools and therefore much work can be done to further advance this field.

Most modern languages implement some kind of multithreading-capable standard library or class (In the C11 standard even C received official multithreading support [12]) without seriously constraining how many threads a given program can handle, with next to no respect for the computing cores that the system has control over. This is the reason why we need to differentiate between *real concurrency* and *apparent concurrency.*

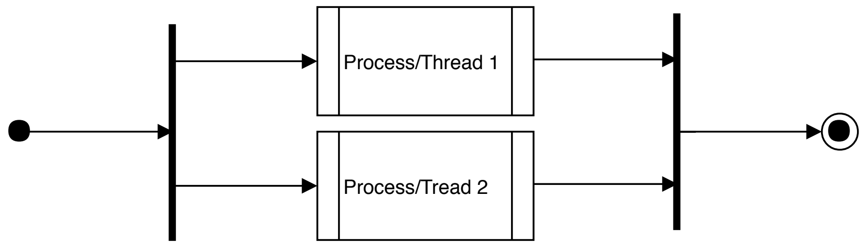


**Figure 2.2:** Representation of a multi-process/multi-thread program.

**2.3.1. Definition (Real concurrency).** A program starts on a single core as a single thread, then it might wake up other cores and specify the memory address where the program or part of a program resides that will be executed by the processing core (**fork**). Any thread can **join** another thread, which means the calling thread suspends execution until the other thread ends. Logically follows that the number of threads must not exceed the number of processing (logical) cores themselves. ▪

Different processing cores do not share registers, share some type of cache (not always), but each core has access to the same memory regions[3] (See Figure 2.3), therefore each architecture that allows multiple threads must have memory ordering rules in place to avoid memory collisions (See Definition 2.4.1).

**2.3.2. Definition (Apparent concurrency).** A program starts as a process on top of a lower-level software (generally referred to as an operating system) and can spawn other processes that behave in the same way. The maximal number of processes is only constrained by the operating system, and the number of physical cores does not influence program logic. ▪

Processes may or may not be running on different cores, this is up to the scheduler built into the operating system. If two processes share a processing core, then a technique called time-slice multithreading will be employed, which means the processor rapidly changes which process it is currently executing.

[3]In this work I do not take NUMA (Non-Uniform Memory Access) into account. It is not yet present in architectures where formal verification could occur as the obscurity of the standard does not allow for such an option.

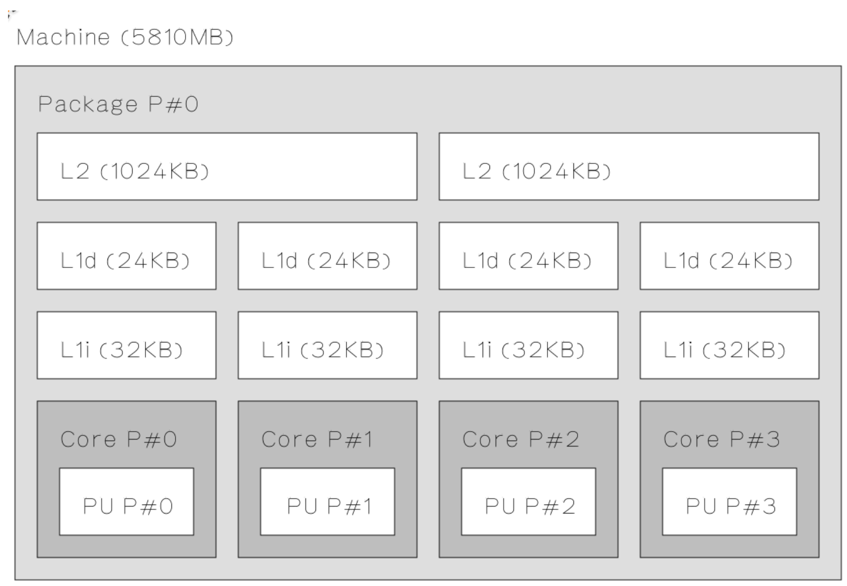


**Figure 2.3:** Output of the lstopo utility on a 4-core Intel Celeron N3450 processor with 6GB of system memory.

As a *data race* (See Definition 2.4.2) or *memory collision* (See Definition 2.4.1) can only occur when two threads access the same memory region at the exact same atomic time, a program sheerly employing apparent concurrency will never experience these among its processes[4]. However, if multiple cores are present, any two processes can collide and therefore the need for the formal verification of such situations arises. On multi-core systems these processes behave in a way that their **sequence of execution between a fork and join is undefined**. In the example Figure 2.2, *Thread 1* and *Thread 2* will both be executed, but we cannot say anything about the order they are being executed in.

### 2.3.1 Synchronization Possibilities

Even though different cores have access to different registers and top-level cache and they operate mostly independently, an upper level regulating entity might be required if their operation needs to be organized. The approach *most* (operating) system designers use is that rather than appointing this responsibility to a specific core, they all need to obey a ruleset which regulates when and how they should operate. The simplest example of such a rule is in the form a mutex variable, which provides two functions: **lock** and **unlock**

The rule here is that no two different threads can assume ownership of the mutex lock at any given time, which means successfully executing a lock operation, but not yet having executed an unlock operation.

The unlock part is mostly straightforward, it lets other threads lock the same variable. The lock part is somewhat more complicated, even so because it needs support from the underlying architecture.

The locking mechanism looks something like this:

[4]As in this case only one core is actually executing instructions and therefore all accesses are strictly ordered by design.

```
1 void mutex_lock(int* mutex)
2 {
3     while(*mutex);
4     *mutex = 1;
5 }
```

**Listing 2.1:** Locking the mutex variable pointed to by *mutex. A value of 0 means the mutex is free to lock.

This solution is lacking in two different aspects: on one hand it is using CPU-heavy active wait (in line 3) until the mutex becomes available and on the other hand it uses two distinct operations for the testing and setting of the mutex variable. When two different threads try to place a lock on the same variable, such a situation can happen very easily where both exit the while loop because the mutex variable got set in 0, and both do so at the exact same time. Then they can both assume that they are the sole owners of said lock, and keep executing tasks that should only be executed distinctly. This second problem can only be solved by a CPU instruction that tests and sets the value of a lock at the same time - such an instruction is available in most modern architectures, such as AMD64 (the BTS reg/mem{16,32,64}, {reg{16,32,64},imm8} instruction), or RISC-V (the amoswap.w.aq operation, see Listing 2.2 for the recommended mutex lock/unlock on RISC-V systems.). [11] [26]

```
li t0, 1                  # Initialize swap value.
again:
amoswap.w.aq t0, t0, (a0)   # Attempt to acquire lock by swapping t0 and (a0).
                          # If it is currently unlocked, a0 will point to a variable with value 0.
bnez t0, again            # Retry if held (t0 is still 1).
# Critical section
amoswap.w.rl x0, x0, (a0)   # Release lock by storing 0
```

**Listing 2.2:** Recommended code for mutual exclusion. a0 contains the address of the lock. [26], Page 44.

This, however, does not solve the other problem of ”actively idling”, meaning instructions are executed continuously until acquiring the lock. This is due to the level of abstraction the core is operating on, which is very low in the context of multi-threaded applications - we need to move a few levels up to be able to control inactive idling until said lock becomes available. This is solved at the level of the operating system with the help of system calls (the API to reach kernelspace services), such as the *futex* syscall in the Linux kernel[5]. If said mutex is available, the acquire operation will be executed in userspace, while trying to lock an already locked mutex results in the task being delegated to the kernel, which handles such cases safely and efficiently, but in a much costlier manner.

Another method of synchronization is called a *condition variable.* It provides three (or four, implementation dependent) functions:

- wait - waits until signalled, releases a mutex lock when called and re-acquires it when returning
- signal - gives off a signal to exactly 1 thread, if any is currently waiting
- broadcast - gives off a signal to all threads currently waiting
- timed_wait - waits until signalled or until it is past its expiration time, releases a mutex lock when called and re-acquires it when returning

[5]http://man7.org/linux/man-pages/man2/futex.2.html

These can be used to *wait for something and be notified when ready.* Listing 2.3 shows an example implementation of such a construct: The code will launch two threads, both only operate inside a mutex environment and one waits until a specified flag (*ready*) is 0, giving up the mutex environment in the cnd_wait line. The other sets the flag to 1 meaning an operation is ready, then signals the other thread which returns, exits the while loop and continues operation normally.

```
#include<threads.h>

mtx_t mutex;
cnd_t cond;

int ready = 0;

int thread_one(void *data){
    mtx_lock(&mutex);
    while(!ready)
        cnd_wait(&cond, &mutex);

    mtx_unlock(&mutex);
}
int thread_two(void *data){
    mtx_lock(&mutex);
    ready = 1;
    cnd_signal(&cond);
    mtx_unlock(&mutex);
}
int main(){
        thrd_t first, second;
        mtx_init(&mutex, mtx_plain);
        cnd_init(&cond);
        thrd_create(&first, thread_one, NULL);
        thrd_create(&second, thread_two, NULL);
        thrd_join(first, NULL);
        thrd_join(second, NULL);
}
```

**Listing 2.3:** Condition variables in C11

There are many other synchronization options on the level of the operating system - just a glance at the ./kernel/futex.c[6] reveals the countless opportunities that cater to everyone's needs. In this work, however, we only care about the two options described above - the **mutex lock** and **condition variable**. These are officially supported by the C11 standard [12] and therefore they should work the same way on any architecture or OS that employs this standard.

It is also advantageous to mention that all synchronization methods are subject to the same verification procedures as the programs using them, because otherwise issues in their implementation might be undiscovered during development. For the purpose of this work it is always to be assumed that all such tools work as described and were already successfully verified and proven to be correct.

### 2.3.2 Atomic Operations

Most problems about multi-threaded applications are caused by message passing between two or more threads. This is done by using a common memory region that one thread writes, and the other reads, which might be implemented the following way:

[6]https://github.com/torvalds/linux/blob/master/kernel/futex.c

```
void write(void* message, size_t length, void* envelope)
{
    memcpy(envelope, message, length);
}
void* read(size_t length, void* envelope)
{
    void* message = malloc(length);
    memcpy(result, envelope, length);
    return message;
}
```

**Listing 2.4:** Sending and receiving a message. *envelope* is the common memory region, *length* is the size of the message (known by both parties) and *message* is the message being passed (e.g. a struct).

As the operation *memcpy* cannot be implemented immediately for lengths bigger than the word size of the architecture (no such single register exists), therefore it is entirely possible that while one thread writes the data, another thread reads it back, but they collide and the reader gets the first half of the old value and the second half of the new value, which can trivially cause very serious problems.

In the previous section we have seen that two threads might synchronize their operations using a mutex lock. This is a very resource-expensive way of creating environments where threads can operate without worrying about reading memory values that were only written in part, and with the help of such locks the example above might be re-implemented the following way:

```
mtx_t mutex;
void write(void* message, size_t length, void* envelope)
{
    mtx_lock(&mutex);
    memcpy(envelope, message, length);
    mtx_unlock(&mutex);
}
void* read(size_t length, void* envelope)
{
    mtx_lock(&mutex);
    void* message = malloc(length);
    memcpy(result, envelope, length);
    mtx_unlock(&mutex);
    return message;
}
```

**Listing 2.5:** Sending and receiving a message with a memory region pointed to by *envelope* that can only be accessed from one thread at a time.

This issue of overwriting data while being read is not exclusively the trait of writing/reading arrays. For example, on the X86 architecture a 32-bit **MOV** operation is only written in one part if the address is naturally aligned[7], otherwise it is written in two parts, therefore allowing for (a very subtle) problem. The solution to this issue is using locks, which are, however, very expensive instruction-wise.

**Lock-free programming** The other, better solution for such a problem is using some kind of a data structure that can guarantee that within a very small given timeframe all data is either written to memory (*committed*), or the entire operation fails (*aborted*). This timeframe is called *atomic time*, and these data movement operations are called *atomic operations.*

[7] $32 \mid addr$

**2.3.3. Definition (Atomic operation).** A given operation is *atomic*, if its results turn visible in one instance rather than in separate, smaller "packages". *Atomic memory* operations are such memory operations that change/read data in/from the memory in exactly one block. ▪

It is easy to conclude that without such operations lock-free programming could not be possible.

**Memory ordering rules** When it comes to atomic memory operations (atomic operations in short), the order in which non-atomic memory operations happen is a very important factor as they might try and change the value, which will then be read incorrectly by an atomic memory read (or vice versa). To counter this possibility, we introduce a new type called *memory ordering type*.

**2.3.4. Definition (Memory ordering types).** The rules about when (after/before which operations) the atomic operation gets executed. ▪

Table 2.1 summarizes the atomic operations' ordering types.

| memory_order | load? | store? | Description |
|---|---|---|---|
| _relaxed | x | x | Loosest memory order, provides no guarantees for the ordering of other threads' memory accesses with respect to the atomic operation. |
| _consume | x | | The atomic operation will happen once all prior memory accesses of the *releasing* thread that have a dependency on the *releasing* operation have happened. |
| _acquire | x | | The atomic operation will happen once all prior memory accesses of the *releasing* thread have happened. |
| _release | | x | The atomic operation will happen before a *consume* or *acquire* load operation allowing for synchronization among the threads. |
| _acq_rel | x | x | Depending on the type of atomic operation this ordering will either mean *acquire load* or *release store.* |
| _seq_cst | x | x | Strictest memory order, only happens after all visible operations of other threads have already happened. |

**Table 2.1:** Memory ordering types [12]

The main reason why such ordering types need to actually be enforced is because even though multiple cores operate independently from each other (without synchronization constraints when employing methods of lock-free programming), they all share the system memory, more specifically the bus thereof. This bus is only a word-size number of wires spanning from the processor complex to the memory slots on the motherboard and therefore it can only carry a word-sized piece of data at a given time[8]. This means that even though processors operate in a *really concurrent* environment, their memory accesses by definition need to be only *apparently concurrent.* To achieve this, a very strict ordering ruleset needs to be in place in the architecture itself. See Figure 2.4 for an example.

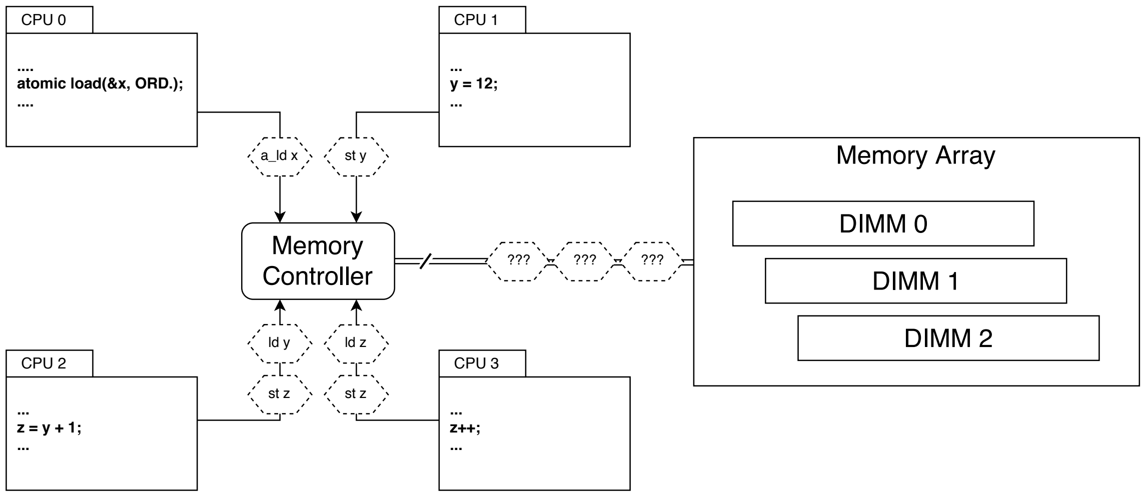


**Figure 2.4:** The memory controller puts the concurrently incoming requests into a linear order depending on the memory ordering rules.

Depending on the memory ordering **ORD.**, there is either no restriction and the memory accesses can go through in any sequence (**relaxed** ordering), or the *a_ld x* will be the last to execute (**seq_cst** ordering). **consume** and **acquire** do not have any influence on the situation as there are no previous memory accesses from CPU0 waiting for execution.

Back in Listing 2.2 we saw the recommended algorithm for acquiring and releasing mutex locks. Line 3 uses an *acquire* ordering, which means that it can only happen after all currently visible memory operations have already been dealt with, as acquiring a lock needs the pre-requirement of not being locked (not having 1 as the value of the variable). The same thoughts go into line 7, where we release our mutex lock and we do so by employing the *release* ordering type, which means that it happens as soon as it can, most likely before any other memory operation of other threads.

## 2.4 Memory Consistency Models

Memory consistency models (MCMs), which regulate memory operations in a shared memory system, are an absolute necessity of system design. They make the guarantees the developers can build on about the memory ordering ruleset (see previous section) and they provide a thorough specification on how shared memory systems should behave in specific situations. Much like mathematics, it builds on *axioms* and *derived rules*, which can easily be checked for inconsistencies, contradictions and incompleteness.

The main problem about shared memory systems is described above (See Section 2.3.2): multiple threads might try and operate on the same memory region at the exact same time. This situation is either called a *memory collision* or a *data race.*

[8]This time is the atomic time referred to above

**2.4.1. Definition (Memory Collision).** One or more threads try to read a memory region while it is being written by another thread. ▪

**2.4.2. Definition (Data Race).** Two or more distinct threads try to write memory at the exact same time. ▪

MCMs include specific rules for both (and more) situations, describing how it should be dealt with - but in most cases a sensible solution can only be found if all colliding threads are using atomic operations, because regular memory I/O **will** still cause the discrepancies discussed in previous sections. For atomic operations, however, the MCM provides the mathematical background for the ordering types in Table 2.1 - these are the guarantees that the MCM provides[9].

### 2.4.1 MCM Validation

MCMs are defined on the level of the programming language itself. The specification of said programming language must include the worded rules and guarantees of memory operations, which then must be implemented in the compilers used. This is first barrier the formal verification must overcome: Are the worded rules really implemented correctly in code? This is one aspect of the CompCert [5] project, which aims at the formal verification of compilers.

Down to the physical realization of the architecture every step along the way poses a new obstacle for the verification method to tackle. Most verification techniques care about a 2-element subset of these realizations and verify whether the memory orderings assumed in the source hold in the binary produced by the compiler, and whether the binary really executes correctly on the specific computer architecture.

In order to provide the full-stack verification of the MCM, a new methodology has been developed by researchers at Princeton University [25], which operates at the ”trisection of the MCM, the compiler and the hardware”, verifying the consistency among all three levels at the same time. It uses a description of situations called *litmus tests* that can be virtually executed on the goal target platform and the behaviour checked against memory ordering criteria. In its original meaning, litmus tests are used by chemists to determine a solution’s approximate pH value, but this concept later transferred to many other fields of science. In this work, when I refer to these tests, it will always mean the definition seen in Definition 2.4.3 and look like as seen in Listing 2.6

**2.4.3. Definition (Litmus test).** Formal description of simple multi-threaded programs that realize situations where memory ordering rules are in action. All threads are started at the same time and no synchronization can be observed among them. Given some variables’ initial state, the *forbidden* outcome(s) of the situation is described after the test as a mathematical formula, which when*true*, the MCM is breached. ▪

[9]It must also be mentioned that MCM not only provides these guarantees, but also rules along the pipeline for successive same-address memory operations.

```
C iriw_R_acquire_acquire_relaxed_relaxed_W_release_relaxed
{
[x] = 0;
[y] = 0;
}
P0 (atomic_int* x) {
  atomic_store_explicit(x, 1, memory_order_release);
}
P1 (atomic_int* y) {
  atomic_store_explicit(y, 1, memory_order_relaxed);
}
P2 (atomic_int* x, atomic_int* y) {
  int r1 = atomic_load_explicit(x, memory_order_acquire);
  int r2 = atomic_load_explicit(y, memory_order_acquire);
}
P3 (atomic_int* x, atomic_int* y) {
  int r3 = atomic_load_explicit(y, memory_order_relaxed);
  int r4 = atomic_load_explicit(x, memory_order_relaxed);
}
exists
(2:r1 = 1 /\ 2:r2 = 0 /\ 3:r3 = 1 /\ 3:r4 = 0)
```

**Listing 2.6: iriw_R_acquire_acquire_relaxed_relaxed_W_release_relaxed** litmus test

# Chapter 3

# Co-verification Workflow

The violation of memory ordering constraints might mean the difference between a functional piece of software that we can reasonably trust no to fail, and a dangerous, completely disfunctional product. The best solution is to avoid using flawed architectures, but that options is not likely to be available in all situations, such as an update service for an autonomous car's software, where the hardware should not be changed if any other solution is available. If we know what these flaws are, however, we can modify our current programs to mitigate these risks.

Consider the following program in Listing 3.1. It shows a program that will only return with 0 if the atomic operations function as expected, which means the releasing store will happen prior to the acquiring load. If, however, this is *not* the case, the program will return with 1 which is an *unintended behaviour*. However, if we know that the acquire and release orderings produce this unintended outcome in a given situation, we can forego this problem by replacing them with an other (maybe more costly), correct solution.

```c
#include <stdatomic.h>
#include <threads.h>

_Atomic int x;

int run(){
        return atomic_load_explicit(&x, memory_order_acquire);
            //the acquire load will happen after the releasing thread's store
}

int main(){
        thrd_t thread;
        x = 1;
        int y = -1;
        thrd_create(&thread, run, NULL);
        atomic_store_explicit(&x, 0, memory_order_release);
            //the release store will happen before an acquire load
        thrd_join(thread, &y);
        return y;
            //It will return with 0 if the store was before the load
}
```

**Listing 3.1:** Simple concurrent program written in C11 compliant C code

To summarize the main goal of this work, it is to help developers uncover all possible MCM (Memory Consistency Model) violations in a source file prior to deployment. Figure 3.1 includes all the main steps for that:

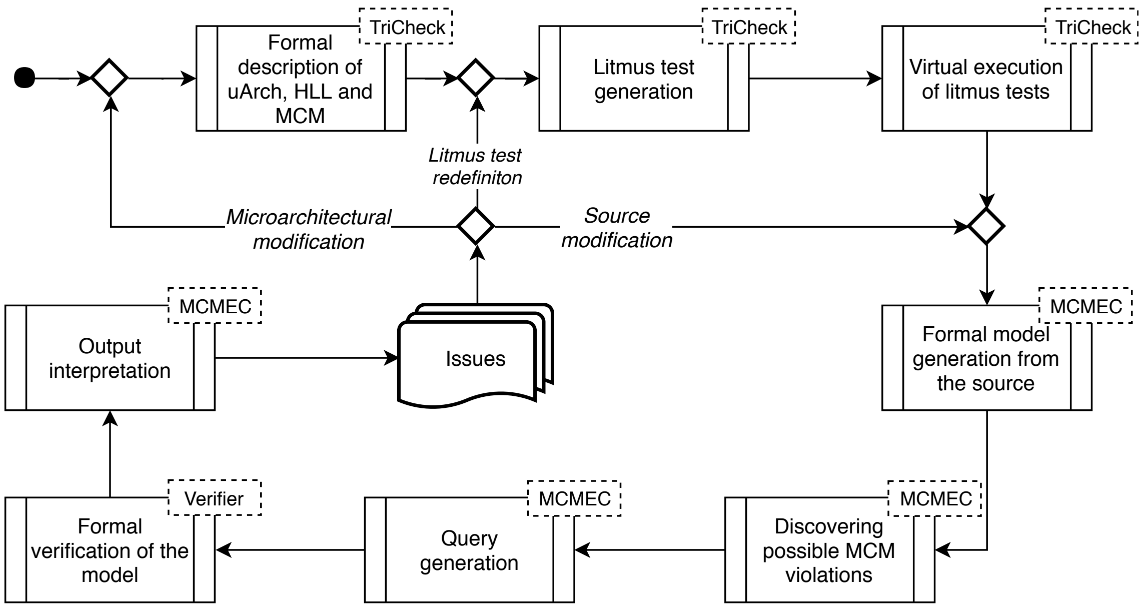

TriCheck
Formal description of uArch, HLL and MCM
TriCheck
Litmus test generation
TriCheck
Virtual execution of litmus tests
Litmus test redefiniton
Microarchitectural modification
Source modification
MCMEC
Output interpretation
Issues
MCMEC
Formal model generation from the source
Verifier
Formal verification of the model
MCMEC
Query generation
MCMEC
Discovering possible MCM violations

**Figure 3.1:** Overview of the approach

**Formal description of $\mu$Arch, HLL and MCM** As we are aiming at verifying software through the layers of abstractions down to the hardware itself, we need a way to describe each of these layers. As a bottom-up approach, first we need to describe the microarchitecture itself. Aiding this there is a domain-specific language called $\mu$Spec [14] that hardware designers can use to specify the constraints and axioms of the architecture they are working on.

The specification of the MCM is done with a tool called Herd [3]. Herd defines more abstract models that do not depend on the microarchitecture itself.

Finally, the compiler mappings specific to the high-level language are to be specified. These include (as opposed to a single MOV-like operation) prefixes and suffixes that can include fences or other safeguards. In many cases this is the part that is over- or underdefined, resulting in observable erroneous outcomes. An example for compiler mappings can be seen in Table 3.1.

| **C11 instruction** | **Power** | | |
|---|---|---|---|
| | **prefix** | **instruction** | **suffix** |
| load relaxed | - | ld | - |
| load acquire | - | ld | ctrlisync |
| load seq_cst | hwsync | ld | ctrlisync |
| store relaxed | - | st | - |
| store release | lwsync | st | - |
| store seq_cst | hwsync | st | - |

**Table 3.1:** Compiler mappings of C11 atomic operations on Power [15].

**Litmus test generation (by TriCheck [25])** After defining the rules in which we operate, litmus test templates are to be written next. These are blueprints of situations that we are interested in, leaving out places for e.g. memory orderings - this way it is easy to generate many different litmus tests with search-and-replace methods instead of having to write thousands of them by hand.

By specifying places in the templates where different strings can be inserted to create different possibilities, one can generate all the combinations of a specific type of litmus tests. These litmus tests include variable declarations, process definitions and specific criteria which signals if a litmus test passes or not - see Section 2.4.1.

**Virtual execution of litmus tests (by TriCheck [25])** After the generation of litmus tests the next logical step is to check which of them pass and which do not. First, they are translated into their assembly equivalents, then these modified litmus test are run on the Herd $\mu$Spec model to determine if an outcome is observable, after which it checks if it was permitted or forbidden. Forbidden, but observable outcomes are the ones we are looking for, therefore it is possible to create a visual representation of these litmus tests to see where the program might fail.

At this step, the generic part of the MCM validation is over. These steps were only to be run once (per architecture change), as the results can be saved and re-used in the following steps.

**Formal model generation from the source** In order to verify the program under development we need to parse the source code into a formal model. As this transformation is challenging to do in one step, it is broken up into two distinct phases: firstly an abstract syntax tree (AST) will be generated, as it can be done by reading the source file line-by-line, then this AST will be transformed into the type of formal model our verification tool accepts. For the sake of simplicity, it is to be assumed that this is a Control Flow Automaton (CFA), as it is very easy to represent entire programs in this format. Furthermore, there has been extensive research about the optimal transformation of programs into CFAs [21].

**Discovering possible MCM violations** In this step we remove all the elements from the model that are not interesting with respect to our goal. These include locations and transitions which have nothing to do with synchronization, multithreading or atomic operations, as these only litter our state space as they are irrelevant for our purposes. After that, locations with the above-mentioned types of instructions will be labeled with their type of operation and the number of the line of code that generated them. Then we find all locations alike to the generic litmus tests, which we found violating, resulting in sets of problematic lines present in the source file.

**Query generation** Using the sets from the previous step a formal criterion-set can easily be assembled that can serve as the second input of the formal verification of our choice. These are all labeled with the name of the corresponding litmus test as to provide a way for the developer to sanity-check if the results this tool provides are correct or not.

**Formal verification of the model** As long as a compatible model and query was generated, it does not matter which verification tool is being used - as a minimal requirement it needs to be able to handle multiple processes in the same system and provide feedback whether a situation can happen or not.

**Output interpretation** As output we are going to get the numbers of lines in the query which are observable in the model together at the same time. As these include the label of states affected, it is easy to trace back which lines are violating the MCM, and therefore they can be dealt with accordingly - either just by displaying those lines, or providing possible mitigation techniques, which (depending on the implementation) can even automate the correction of such programs.

**Mitigation possibilities** After interpreting the output, developers have three possibilities:

- Modify the microarchitecture when working on new, still-in-development microarchitectures
- Redefine litmus tests if suspecting missing problems
- Modify the source to mitigate the problems (with the use of mutex locks, conditional synchronization or a similar solution)

Out of these three the third option is generally the most applicable, as changing software is almost always the easiest to do (in the context of hardware vs. software), and therefore an automatically generated solution might be applicable.

# Chapter 4

# Formal Description of Hardware and Software

## 4.1 Source Code to Formal Model

Transforming a source file into a formal model is not as straightforward as it might sound. The state space can get crowded very easily therefore hindering the performance of verification tools and the correctness of the transforming tool itself cannot be easily verified. The former problem can be somewhat dealt with by using tools and techniques such as in [21], but as this approach requires specific lines of code to remain locations (atomic operations would otherwise be kept as labels on transitions, as they are memory operations, but we would like to observe these locations explicitly), most of the optimization work cannot be applied. The latter problem is even more troublesome, as failing to prove the correctness of the tool, manual examination needs to take place.

Furthermore, these problems are the easier to overcome in contrast with the strict conditions one needs to set if aiming at the generation of a formal model from a source file. The most important challenges are the following:

**Undefined number of threads** Consider the following functions (written in C11 compliant C code):

```
#include <stdatomic.h>
#include <threads.h>
#include <stdio.h>
#include <stdlib.h>

/*
Generates a random integer 'max' and starts that many threads, each performing a relaxed store to 'x
    '.
*/

int _thread()
{
    _Atomic int x;
    atomic_store_explicit(&x, 0, memory_order_relaxed);
    return 0;
}

int main()
{
    int max;
    srand(time(0));
    max = rand();
    thrd_t* threads = (thrd_t*)malloc(sizeof(thrd_t)*max);
    for(int i = 0; i<max; i++)
        thrd_create(&threads[i], _thread, NULL);
    for(int i = 0; i<max; i++)
        thrd_join(threads[i], NULL);
    return 0;
}
```

A number of threads will spawn (ranging from 0 to MAX_RAND), which means there is no way of knowing if there will be *any* stores to the address of $x$, and if yes, then *how many.* This is the disadvantage of many modelling languages (such as CFAs or XTAs), as they cannot handle a previously unknown number of threads/components.

**Dynamic memory allocation** Consider the following functions (written in C11 compliant C code):

```
void foo(unsigned int c)
{
    int* arr = (int*)malloc(sizeof(int)*c);
    for(int i = 0; i<=c;i++)
        arr[i] = i;                            //out of range index when i == c
}
```

We will try to store the value $c$ at the address $(arr + c)$ in memory, even though we only have access to the addresses $[arr,\ arr+c)$. At best, this will throw a runtime error (which, in C, means *segmentation fault*, then quitting) and at worst this will overwrite something critical that was stored next to it. Even though the former will outright end the execution of the program (which in itself is unacceptable for fault-tolerant systems), the latter is much more dangerous: assume there was a variable used in a control expression or an atomic variable that carried mission-critical importance. As per these concerns dynamic memory allocation and usage is not allowed in the subject of verification.

**Unknown depth of recursion** Consider the following functions (written in C11 compliant C code):

```
#include <stdatomic.h>

/*
When called with a non-null parameter this function will call itself that many times recursively,
     each time writing the current depth of recursion into a variable.
Atomics and void* parameters are only used because the next example will utilize a similar situation
     , and that will need these elements.
*/

_Atomic int x;

int foo(void * depth)
{
    atomic_store_explicit(&x, *(unsigned int*)depth, memory_order_relaxed);

    if(*(unsigned int*)depth == 0 ) return 0;

    unsigned int next_depth = *(unsigned int*)depth - 1;

    return foo((void*)&next_depth);
}
```

Here an arbitrary depth (provided in value *depth* as an unsigned int value) can be achieved and therefore it is very easy to accidentally overfill the stack and therefore end the execution. A similar, but even more dangerous approach is the following (C11 compliant C code):

```
#include <stdatomic.h>
#include <threads.h>
#include <stdio.h>
#include <stdlib.h>

/*
When called with a non-null parameter this function will start a new thread of itself that many
    times, each time writing the current depth of recursion into a variable.
*/

_Atomic int x;

int foo(void * depth)
{
    atomic_store_explicit(&x, *(unsigned int*)depth, memory_order_relaxed);

    if(*(unsigned int*)depth == 0 ) return 0;

    thrd_t thread_id;
    unsigned int i = *(unsigned int*)depth - 1;

    thrd_create(&thread_id, foo, (void*)&i);
    thrd_join(thread_id, NULL);

    return 0;
}
```

Even though technically this is **not** recursive, one can see why it shares some similarity with the former approach. Yet again, this is the problem of not knowing how many threads there are until the code is actually executed, and even then it might not be deterministic (consider the *rand*() function or a read from a floating GPIO).

**The boundedness of the chosen model type** Even without implementing any unsafe operation from the list of situations above there might be some boundaries that are set by the type of formal model the source needs to be transformed into. For example, in the case of the UPPAAL Timed Automata (XTA) [8], there is no way of spawning/detaching processes and therefore all threads need to be started at the beginning of the model (those that should not start immediately need to employ some kind of block, such as conditional synchronization), after which they can be unblocked by the otherwise spawning process. This, and similar workarounds lead to the non-invertibility of the transformation process, which means any modification must be done in the source code rather than on the model itself.

To summarize this section, there are many obstacles to overcome when dealing with source code to formal model transformers. Even though only a few were discussed here, there are many other corner cases that most generators do not handle. My implementation (see Chapter 6) for example does not allow threads to take up more than one function, as that would make transformation too difficult for a simple proof-of-concept implementation.

### 4.1.1 Optimization Possibilities

As most formal model types provide their modelling capabilities for general problems rather than highly specified ones, a number of optimization steps can take place by creating an abstraction of the original model in a way that no important information is lost. These are, among others, the following:

**Loop elimination** Consider the following code (written in C11 compliant C code):

```
void unoptimized() {
    int _control = 5;
    for(;_control!=0;_control--)
        atomic_store_explicit
            (&x, 0, memory_order_relaxed);
}
```

In this function the body of the for-loop will be repeated exactly 5 times. This code might produce the following XTA model (visualized in Figure 4.1):

```
int _control;

process unoptimized() {
clock x;
state
    _init,
    _merge,
    relaxed_store,
    _decision,
    _final;
init _init;
trans
    _init -> _merge { assign _control = 5;},
    _merge -> relaxed_store { },
    relaxed_store -> _decision { },
    _decision -> _final {guard _control == 0;},
    _decision -> _merge {guard _control != 0;
     assign _control = _control - 1;};
}
system unoptimized;
```

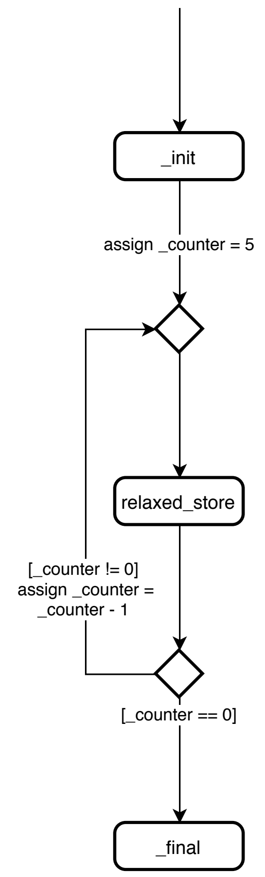


**Figure 4.1:** Unoptimized model.

And after optimization:

```
process optimized() {
clock x;
state
    _init,
    relaxed_store,
    _final;
init _init;
trans
    _init -> relaxed_store { },
    relaxed_store -> _final { },
}
system optimized;
```

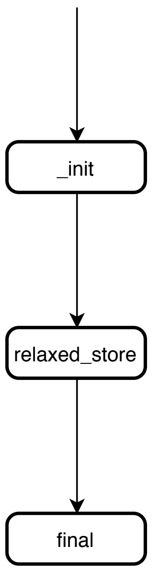


**Figure 4.2:** Optimized model.

As the only type of criteria we set for the code is in the form of an N-Tuple[1] that will be interpreted as the question *Can it ever occur that all of the members of the tuple happen at the exact same atomic time?*, it does not matter if a member happens once, twice or five times in a block, all that matters is if it happens at all. With this information we can construct a new model (again in the form of an XTA, but it is relevant to all similar modelling languages) and representation in Figure 4.2.

**Function call elimination** If a function contains no atomic, synchronization or control flow content it can be easily ignored. This dramatically reduces the state space as in a real-world program these types of function calls will be a negligible amount and therefore even huge, multi-thousand lines of code can be verified as if it only consisted of the few relevant lines.

[1]Such as *(relaxed load, release store)*. This means that if an atomic relaxed load on one thread happens while another atomically stores (with memory ordering type *release*) a value at the same address, the behaviour might be unexpected with respect to memory ordering rules. More information in Section 5.1.

# Chapter 5

# Verification and Mitigation

In this chapter I will go through the steps from the interpretation of litmus tests through the verification process to the mitigation generation. These sections will provide the answer to the theoretical *How does this approach actually work?* question, leaving out implementation-specific information (which are discussed in Chapter 6).

## 5.1 Interpreting Litmus Tests

Litmus tests are the key to the theoretical verification of a given microarchitecture. These are source-code-like descriptions of situations we want to observe. Of course manually writing each litmus test is not only time-consuming but also error-prone and therefore discouraged - hence the need for litmus test generation as seen in Figure 5.1.

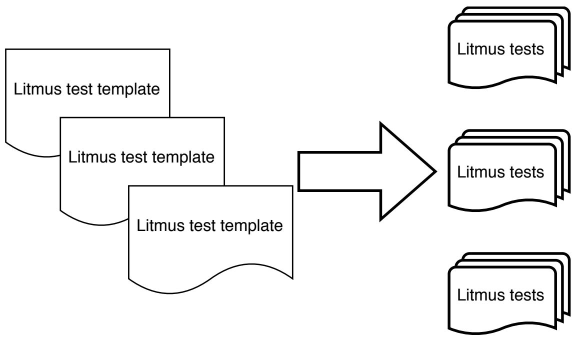


**Figure 5.1:** Litmus test generation mock-up.

The litmus test templates are similar to the litmus tests themselves but they include placeholders used to mark places where specific types (such as memory ordering) might be placed. An example of such a template can be seen in Listing 5.1.

```
C <TEST>
{
[x] = 0;
[y] = 0;
}
P0 (atomic_int* y, atomic_int* x) {
    atomic_store_explicit(x,1,memory_order_<ORDER_STORE>);       // ordering placeholder
    int r0 = atomic_load_explicit(y,memory_order_<ORDER_LOAD>);  // ordering placeholder
}
P1 (atomic_int* y, atomic_int* x) {
    atomic_store_explicit(y,1,memory_order_<ORDER_STORE>);       // ordering placeholder
    int r1 = atomic_load_explicit(x,memory_order_<ORDER_LOAD>);  // ordering placeholder
}

exists (0:r0=0 /\ 1:r1=0)                                     // forbidden outcome
```

**Listing 5.1:** The *sb* (store-buffering) litmus test template.

In the example above it can be seen that the placeholder strings *<ORDER_STORE>* and *<ORDER_LOAD>* signal the place where the enumeration constants of the type *memory_order* should be placed. The other noteworthy thing about the template is the last line, in which the *forbidden* outcome is described in a mathematical form. Figure 5.2 summarizes the template above.

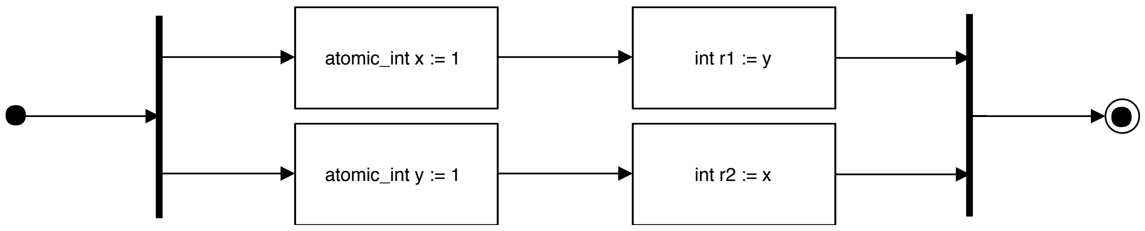


**Figure 5.2:** Visualization of the *sb* litmus test template. Forbidden outcomes are: r1 = 0 and r2 = 0 (at the same time), as each thread loads the variable after it has already stored it atomically, meaning at least one needs to be 1.

After writing the litmus test template a utility will take it as input and by replacing the placeholders create the actual litmus tests. The memory_order enum's constants are summarized in Table 5.1 and are a subset of Table 2.1 by leaving out *memory_order_acq_rel* as it is just a shorthand for *acquire if loading, release if storing.* In our use-case this provides no actual benefit and therefore it can be left out.

| **ORDER_LOAD** | | |
|---|---|---|
| _RELAXED | _ACQUIRE | _SEQ_CST |
| **ORDER_STORE** | | |
| _RELAXED | _RELEASE | _SEQ_CST |

**Table 5.1:** Memory ordering primitives used in place of placeholders.

It is easy to determine that from the *sb* litmus test template (which has two loads and two stores) a number of $3^4 = 81$ litmus tests will be generated, and in the situation of the litmus test template *iriw* (Independent Reads of Independent Writes ) (see Figure 5.3), this number is even higher at $3^6 = 729$.

*In this work I use 5 different litmus test templates.* These cover basic use-cases of atomic operations [22] and are therefore suitable for verification. In addition to the two above, the three litmus tests on Figures 5.4, 5.5 and 5.6 were employed.

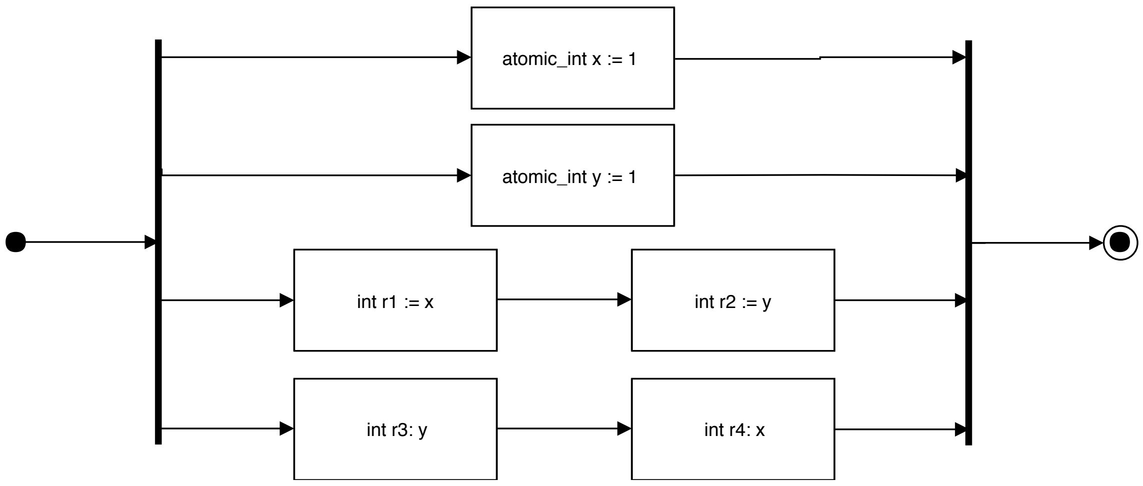


**Figure 5.3:** Visualization of the *iriw* litmus test template. Forbidden outcomes are: r1 = 1 and r2 = 0 and r3 = 1 and r4 = 0 (at the same time).

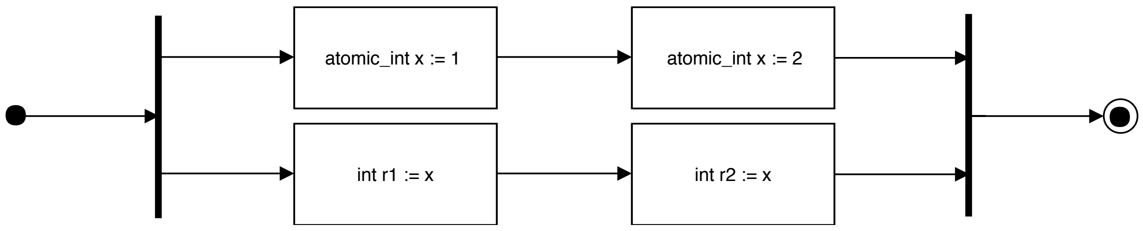


**Figure 5.4:** Visualization of the *corr* (coherent read-after-read) litmus test template. Forbidden outcomes are: r1 = 2 and r2 = 1 (at the same time).

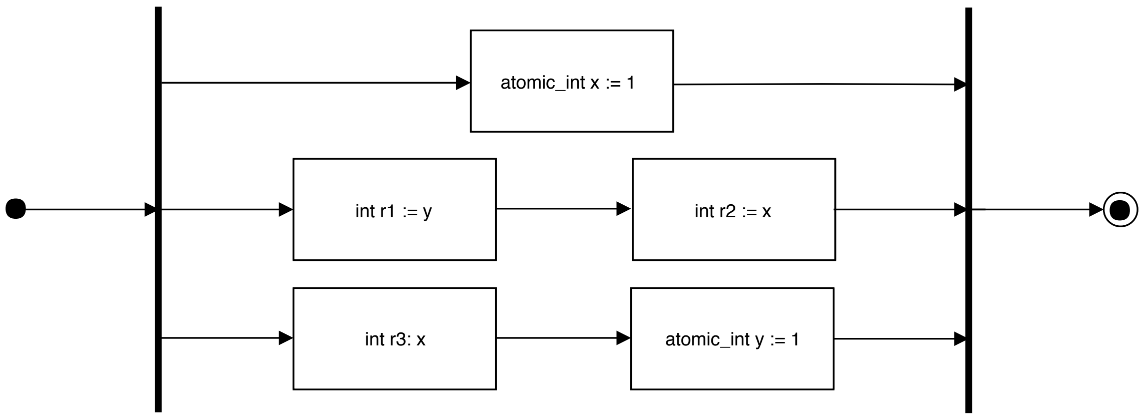


**Figure 5.5:** Visualization of the *wrc* (write-to-read casualty) litmus test template. Forbidden outcomes are: r1 = 1 and r2 = 0 and r3 = 1 (at the same time).

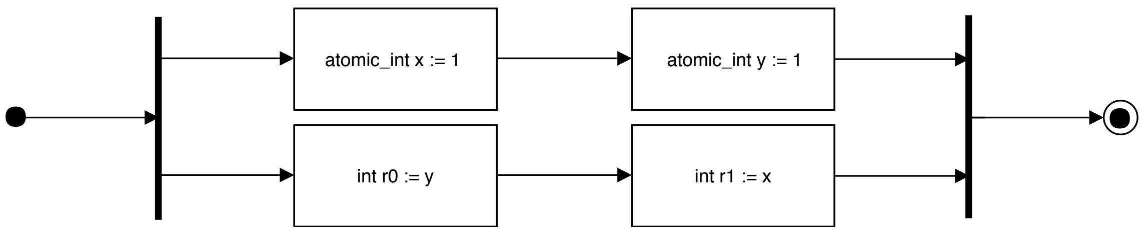


**Figure 5.6:** Visualization of the *mp* (message passing) litmus test template. Forbidden outcomes are: r0 = 1 and r1 = 0 (at the same time).

All together $3^4 + 3^6 + 3^4 + 3^5 + 3^4 = 1215$ litmus tests will be generated.

### 5.1.1 A More Convenient Representation of Litmus Tests

Even though litmus tests are in a perfect form for testing and running them (as they are written mostly in C), they are not as convenient when it comes to interpreting them.

After finding out which litmus tests are problematic by running them through TriCheck, they can be collected in a separate place and used for the verification of a source file itself. For this, we need to create a new representation of said litmus tests. At this step we do not care what their forbidden outcome is as whether or not they pass has already been decided, so we can leave that information out of the new representation.

As the only relevant part of the tests are the threads and the order of the operations, we only need to store these. A proposed structure can be seen in Listing 5.2, where each litmus test has its own line, in which there is the list of operations (semicolon separated) of each thread, which are in a colon-separated list. A big advantage is that they are in one file, which minimizes the number of I/O operations and therefore speeds up later steps.

```
issue tricheck_0 { seq_cst store : seq_cst load : seq_cst load }

issue tricheck_1 { acquire load : release store }

issue tricheck_2 { seq_cst store : relaxed load : relaxed load }

issue tricheck_3 { seq_cst store ; seq_cst store  : relaxed load ; relaxed load }
```

**Listing 5.2:** Proposed representation of problematic litmus tests in one file

This is the set of the previously mentioned N-Tuples that describe potentially dangerous situations.

## 5.2 Query Generation and Verification

| | |
|---|---|
| fork | Spawns a new thread |
| join | Synchronizes with the end of the other thread's life |
| cnd_wait | Synchronizes with another thread calling cnd_signal or cnd_broadcast and unlocks a mutex lock (locking it again after synchronization) |
| cnd_timedwait | Synchronizes with another thread calling cnd_signal or cnd_broadcast and unlocks a mutex lock (locking it again after synchronization). If no threads call cnd_wait before a given time, the calling thread tries to lock the mutex and then returns. |
| cnd_signal / cnd_broadcast | Synchronizes with one / all of the threads that called cnd_wait previously (if no-one called it so far, continues execution normally) |
| mtx_lock | Either continues execution normally (when first caller), or synchronizes with another thread's mtx_unlock |
| mtx_unlock | Synchronizes with one of the threads that are trying to mtx_lock (if no-one is waiting on it, continues execution normally) |

**Table 5.2:** The subset of C11 functions about threads and their synchronization capabilities.

In Section 5.1, the formal model representing the program under observation and the list of potential issues have already been constructed. The next logical step is to combine these two and find out which lines of code might be problematic together. For this, first we need to find out which lines of code can be executed at the same time at all. The elements influencing this are summarized in Table 5.2, and these are the previously mentioned synchronization instructions. An example can be seen in Figure 5.7.

```
_Atomic int x;
int firstThread(void *data){        //performs two stores to the variable x.
    atomic_store_explicit(&x, 0, memory_order_relaxed);
    atomic_store_explicit(&x, 1, memory_order_relaxed);
    return 0;
}
int secondThread(void *data){       //performs two loads from the variable x.
    int r1 = atomic_load_explicit(&x, memory_order_relaxed);
    int r2 = atomic_load_explicit(&x, memory_order_relaxed);
    return r1+r2;
}
```

**Listing 5.3:** Allowing the same-time execution of atomic operations.

```
_Atomic int x;
mtx_t mutex; //Needs to be initialized
int firstThread(void *data){    //performs two stores to the variable x.
    mtx_lock(&mutex);           //creates a thread-safe execution environment.
    atomic_store_explicit(&x, 0, memory_order_relaxed);
    atomic_store_explicit(&x, 1, memory_order_relaxed);
    mtx_unlock(&mutex);         //lets others lock the environment.
    return 0;
}
int secondThread(void *data){   //performs two loads from the variable x.
```

```
    mtx_lock(&mutex);            //creates a thread-safe execution environment.
    int r1 = atomic_load_explicit(&x, memory_order_relaxed);
    int r2 = atomic_load_explicit(&x, memory_order_relaxed);
    mtx_unlock(&mutex);          //lets others lock the environment.
    return r1+r2;
}
```

**Listing 5.4:** Avoiding same-time execution of atomic operations using a mutex lock.

```
_Atomic int x;
mtx_t mutex; //Needs to be initialized
cnd_t cond; //Needs to be initialized
int firstThread(void *data){  //performs two stores to the variable x.
    mtx_lock(&mutex);         //creates a thread-safe execution environment.
    atomic_store_explicit(&x, 0, memory_order_relaxed);
    cnd_signal(&cond);        //unblocks the other thread if it is blocked.
    cnd_wait(&cond, &mutex);  //suspends execution until signalled
    atomic_store_explicit(&x, 1, memory_order_relaxed);
    cnd_signal(&cond);        //unblocks the other thread if it is blocked.
    mtx_unlock(&mutex);       //lets others lock the environment.
    return 0;
}
int secondThread(void *data){ //performs two loads from the variable x.
    mtx_lock(&mutex);         //creates a thread-safe execution environment.
    int r1 = atomic_load_explicit(&x, memory_order_relaxed);
    cnd_signal(&cond);        //unblocks the other thread if it is blocked.
    cnd_wait(&cond, &mutex);  //suspends execution until signalled
    int r2 = atomic_load_explicit(&x, memory_order_relaxed);
    cnd_signal(&cond);        //unblocks the other thread if it is blocked.
    mtx_unlock(&mutex);       //lets others lock the environment.
    return r1+r2;
}
```

**Listing 5.5:** Avoiding same-time execution of atomic operations using a mutex lock and a condition variable. This solution provides fine-grade control of the execution order.

As it can be seen on the examples above, there are operations that *allow* same-time execution of following operations and there are operations that *forbid* such execution paths. The former group can be defined as $A = \{fork,\ cnd\_signal,\ cnd\_broadcast,\ mtx\_unlock\}$ from Table 5.2, while the latter can be $B = \{cnd\_wait,\ cnd\_timedwait,\ mtx\_lock,\ return\}$. Any $a \in A$ will influence another thread (denoted by *other_thread*).

With the definitions above, we can create an algorithm that will *always* tell us which lines of code are allowed to be executed at the same time. For this, we can create a Graph $\vec{G}(V, \vec{E})$ in the way seen in Listing 5.6 (Originally $n := null$, $|V| = 0$, $|\vec{E}| = 0$ and $op$ is the first operation of the main() function). If the algorithm ends before all operations have been processed, meaning all threads are in a state of synchronization, restart the algorithm with $op$ := *any remaining thread's next operation* and $n$ := *null*.

```
ADD(NODE n, OPERATION op):
    IF ($op \in A$): ADD(n, other_thread.next_operation).
    ELSE IF ($op \in B$): RETURN.
    ELSE: Create a new node n' connected to n with label 'thread_name'

    ADD(n', this_thread.next_operation)
```

**Listing 5.6:** Graph constructing algorithm in pseudo-code

After constructing the Graph $\vec{G}(V, \vec{E})$ with the algorithm above, all such node-pairs ($a$, $b$) need to be extracted for which *(1) no common ancestor node $n_i$ will be either $a$ or $b$* (so they are not immediate descendants of each other), and *(2) they have different labels*

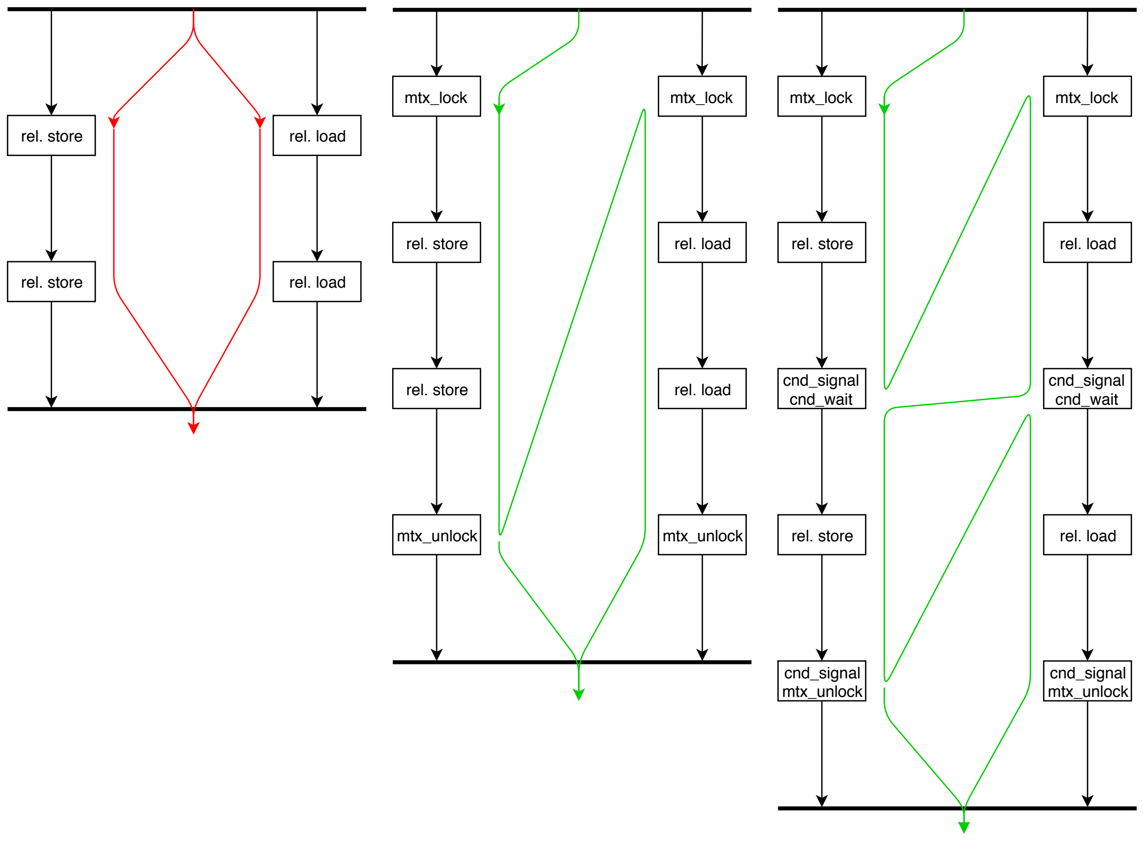


**Figure 5.7:** Visualization of Listings 5.3, 5.4 and 5.5. The lines show an example execution order: red, when two atomic operations can execute at the same time and green otherwise. The extra lines from mtx_unlock locations show that parallel execution is allowed again.

(because if they do, then they are in the same thread and therefore cannot be executed at the same time). These pairs will have the *opportunity* to be executed at the exact same time, while if two nodes are immediate descendants of each other then there is *no* chance they would ever be executed in the exact same moment. See Listing 5.7 and Figure 5.8 for an example on this algorithm.

```
#include <threads.h>
#include <stdatomic.h>

_Atomic int x;
mtx_t mutex;

int firstThread(void *data){
        mtx_lock(&mutex);
        atomic_store_explicit(&x, 0, memory_order_relaxed);
        mtx_unlock(&mutex);
        atomic_store_explicit(&x, 1, memory_order_relaxed);
        return 0;
}
int secondThread(void *data){
        mtx_lock(&mutex);
        int r1 = atomic_load_explicit(&x, memory_order_relaxed);
        mtx_unlock(&mutex);
        return r1;
}
int main(){
        thrd_t first_id, second_id;
        int val1, val2;
        mtx_init(&mutex, mtx_plain);
        thrd_create(&first_id, firstThread, NULL);
        thrd_create(&second_id, secondThread, NULL);
        thrd_join(first_id, &val1);
        thrd_join(second_id, &val2);
        return 0;
}
```

**Listing 5.7:** On thread stores data in variable x while the other reads from it.

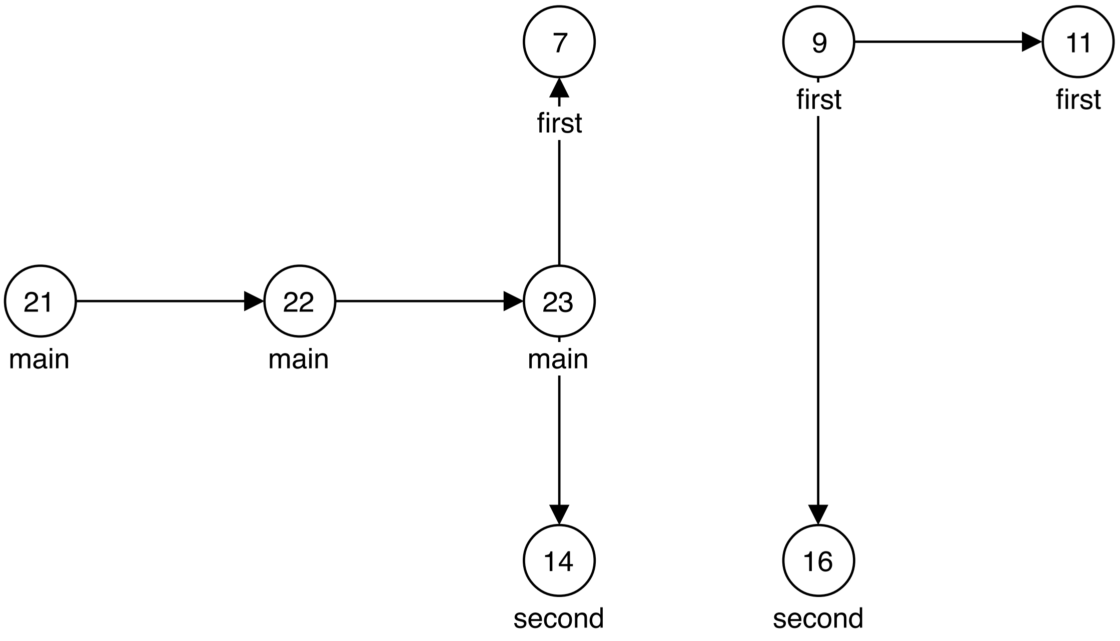


**Figure 5.8:** The graph after running the algorithm in Listing 5.6 on the C11-program found in Listing 5.7.

As it can be deduced from the graph in Figure 5.8, the lines (7, 14), (11,16), (14,9), (7,16), (14,11) *might* execute at the exact same time, but no other lines (that are in neither $A$ nor $B$).

Even though we *could* start by generating all pairs of nodes and then test if the two criteria hold, it is at least an $O(n^2)$ algorithm (n ≈ lines of code) as we need to test criteria on all 2-element subsets of $V$, which will include $\frac{|V|*(|V|-1)}{2}$ different sets, where $V = \{$all lines of code outside those in $A$ or $B\}$. Of course reducing the number of nodes help a lot (by substituting all such paths with edges that only include nodes with valency $\leq 2$ and do not include atomic operations, see program slicing [27]), but it is not the most optimal way.

A more elegant way is to take the list of issues from the previous section, and use that to collect all relevant lines of code. Let's define $I$ set as the set of issues and the relevant problematic tuples of line numbers in the following way: I={issues}, issue={tuples}, tuple={line numbers}. An example of such a set (each new level in the tree means a new level of sets inside the parent set):

```
I
|-- issue 1
|   |-- tuple 1
|   |   |-- line 9
|   |   `-- line 16
|   `-- tuple 2
|       |-- line 11
|       `-- line 16
|-- issue 2
|   `-- tuple 2
|       |-- line 9
|       `-- line 11
`-- issue 3
```

This means there are 3 issues (*(relaxed load : relaxed store)*, *(relaxed store : relaxed store)*, *(relaxed load : relaxed load)*), for which there are 3 possibly problematic lines: (9, 16) and (11, 16) for the first, while (9, 11) for the second. Such a small set can be easily checked against the criteria defined above, and we can see that the only actually problematic tuple is (11, 16), because 9 is immediate ancestor of 16, furthermore 9 and 11 are in the same thread.

Even though the proposed algorithm might differ from the implementation of well-known verification tools, the idea behind generating queries for them stays the same. They accept the formal model generated at the beginning of this chapter as input, and evaluate mathematical statements about given locations/states (here, lines of code) - the example above could be formulated for example using the UPPAAL Model Checker's query language the following way:

```
E<>(line_9  & line_16)
E<>(line_11 & line_16)
E<>(line_9  & line_11)
```

If any of the lines come back as "allowed", the issue persists and should be dealt with accordingly. Section 5.3 will discuss mitigation possibilities.

## 5.3 Mitigation Possibilities

After discovering the problematic lines the developer needs to come up with a solution for avoiding them. Such a solution might exist on three levels:

- Change the microarchitecture (least likely to be applicable to a mere software developer)
- Change compiler mappings to use with litmus tests (might be applicable if using an open source compiler such as a member of the GNU Compiler Collection)
- Change the source to employ some kind of synchronization among the violating threads (almost always applicable, but will introduce an overhead)

### 5.3.1 Changing the Microarchitecture

Even though this option is very seldom available to software developers, there might be some exceptions. An example can be seen in Listings 5.8 and  5.9, which (using $\mu$Spec as a description language) takes an existing microarchitectural specification called TSO and relaxes the LD->LD ordering, creating the new architecture TSO-RR. Note that this specific change in the microarchitecture did not solve all our problems, therefore further work is required.

```
...
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ EdgeExists ((i1, Fetch),  (i2,
     Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "
     darkgreen").
...
Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch),       (f, Execute),      "
     path");
          ((f, Execute),     (f, Writeback), "
     path")]
/\
(
  forall microops "w",
    (IsAnyWrite w /\ ProgramOrder w f) =>
      AddEdge ((w, (0, MemoryHierarchy)), (f,
     Execute), "fence", "orange")
).
```

**Listing 5.8:** TSO.uarch

```
...
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ ~(IsAnyRead i1 /\ IsAnyRead i2)
      /\
EdgeExists ((i1, Fetch),  (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "
     darkgreen").
...
Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f /\ AccessType MMFENCE f =>
AddEdges [((f, Fetch),       (f, Execute),      "
     path");
          ((f, Execute),     (f, Writeback), "
     path")]
/\
(
  forall microops "w",
    (IsAnyWrite w /\ ProgramOrder w f) =>
      AddEdge ((w, (0, MemoryHierarchy)), (f,
     Execute), "fence", "orange")
).
Axiom "Addr_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ IsAnyRead i1 /\ IsAnyRead i2 /\
      HasDependency addr i1 i2 =>
  AddEdge ((i1, Execute), (i2, Execute), "
     addr_rr_dependency").

Axiom "CtrlIsb_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ IsAnyRead i1 /\ IsAnyRead i2 /\
      HasDependency ctrlisb i1 i2 =>
AddEdge ((i1, Execute), (i2, Execute), "ctrlisb")
     .
```

**Listing 5.9:** TSO-RR.uarch

This solution, even though elegant and carries the least performance impact, will not be available for the use-case of this project, which is *using a given microarchitecture eliminate all problematic line combinations*, therefore it should not be taken into account. If someone develops programs for a newfound architecture that still has not left the FPGA testing environment, then contacting the developers about a needed change might not be out of question, but this is far from a general use-case.

### 5.3.2 Changing the Compiler Mappings

The compiler mappings are used to transform the litmus tests into machine-parsable code that can be interpreted by the tools in our workflow. With the help of these one can define prefixes and suffixes for the atomic operation, such as MMFENCE (memory->memory fence) that will cause the program to take a performance hit, but will always avoid a memory collision. For example, the change of the suffix for Read acquire in the following example will help avoid collisions in 101 different litmus tests when using the TSO-RR uarch and the 5 litmus test templates introduced above.

```
C11/C++11 op    | prefix;prefix | suffix;suffix
Read relaxed    | NA            | MMFENCE
Write relaxed   | NA            | NA
Read acquire    | NA            | NA
Write release   | NA            | NA
Read seq_cst    | NA            | MMFENCE
Write seq_cst   | NA            | MMFENCE
```

**Listing 5.10:** Original compiler mappings.

```
C11/C++11 op    | prefix;prefix | suffix;suffix
Read relaxed    | NA            | MMFENCE
Write relaxed   | NA            | NA
Read acquire    | NA            | MMFENCE
Write release   | NA            | NA
Read seq_cst    | NA            | MMFENCE
Write seq_cst   | NA            | MMFENCE
```

**Listing 5.11:** Modified compiler mappings.

This solution *might* be available, but should be discouraged in any situation where the used compiler should ever change (with a version update, for example) that would reset the modification. Such tools can even be developed that would insert the required fence operation in the binary itself, but tampering with a compiled program should not be considered safe or fault-tolerant to any degree.

### 5.3.3 Changing the Source Code

This is the most likely available option for most software developers. No need to change the microarchitecture or the compiler being used, the problematic lines can simply be placed inside safe execution environments using mutex locks or condition variables to synchronize the calling threads. These *will* introduce a serious overhead, because now instead of a single atomic memory operation a mutex needs to be locked and unlocked, but will **guarantee** that the violation will not occur in the program when deployed. For example, using the flowchart in Figure 5.9 the code from Listing 5.7 might be fixed in the following way (lines 10 and 11 have changed position):

```
1  #include <threads.h>
2  #include <stdatomic.h>
3
4  _Atomic int x;
5  mtx_t mutex;
6
7  int firstThread(void *data){
8          mtx_lock(&mutex);
9          atomic_store_explicit(&x, 0, memory_order_relaxed);
10         atomic_store_explicit(&x, 1, memory_order_relaxed);
11         mtx_unlock(&mutex);
12         return 0;
13 }
```

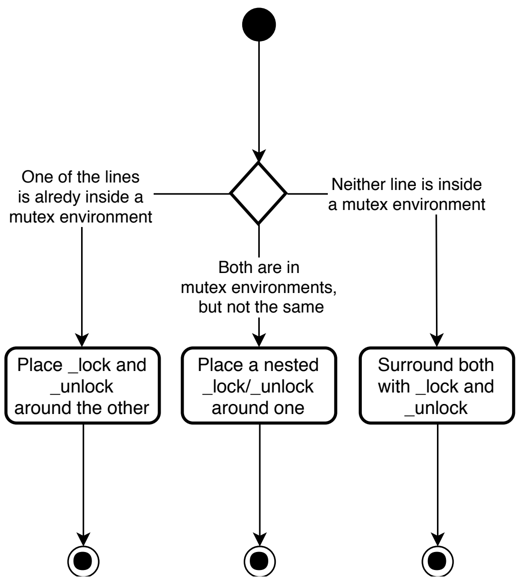

One of the lines is alredy inside a mutex environment
Neither line is inside a mutex environment
Both are in mutex environments, but not the same
Place _lock and _unlock around the other
Place a nested _lock/_unlock around one
Surround both with _lock and _unlock

**Figure 5.9:** Automatic correction generation using mutex environments.

```
14 int secondThread(void *data){
15         mtx_lock(&mutex);
16         int r1 = atomic_load_explicit(&x, memory_order_relaxed);
17         mtx_unlock(&mutex);
18         return r1;
19 }
20 int main(){
21         thrd_t first_id, second_id;
22         int val1, val2;
23         mtx_init(&mutex, mtx_plain);
24         thrd_create(&first_id, firstThread, NULL);
25         thrd_create(&second_id, secondThread, NULL);
26         thrd_join(first_id, &val1);
27         thrd_join(second_id, &val2);
28         return 0;
29 }
```

The solution for the (artificial) example was to place the 11th line inside the mutually exclusive execution environment that was right above it. Figure 5.9 shows an example of an algorithm that solves such problems - but the usage of such a tool should always be checked by a developer to see if the performance impact can be reduced. Of course, more sophisticated solutions might exist for this problem that take this into account and optimize the solution in such a way that the developer cannot come up with a better alternative, but this is outside the scope of this project.

# Chapter 6

# Implementation

In this chapter I introduce a proof-of-concept implementation of the software suite described in previous chapters called *MCMEC*, which is an abbreviation of *Memory Consistency Model Error Checker*.

## 6.1 Implementation Details

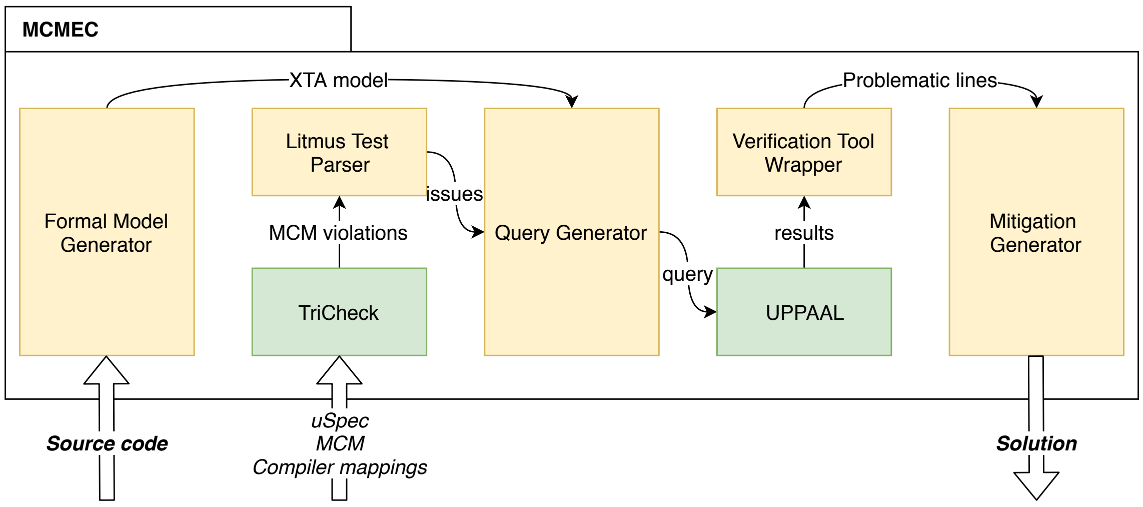


**Figure 6.1:** The architecture of the MCMEC tool with data flow among its components, which are yellow for the modules developed in this project and green for third-party software.

As it can be seen in Figure 6.1, the architecture of the MCMEC tool can be broken up into *modules*, which are either modules developed in this project (in yellow), or third-party software employed as modules. The figure describes the data flow among these components, from the inputs (which are the ***source code*** and the inputs of TriCheck {*$\mu$Spec*, *MCM* and *compiler mappings*}) to the output (which is a set of proposed ***solution***s to the discovered problems. Figure 3.1 still holds, and an assignment of processes to components can be seen in Figure 6.2.

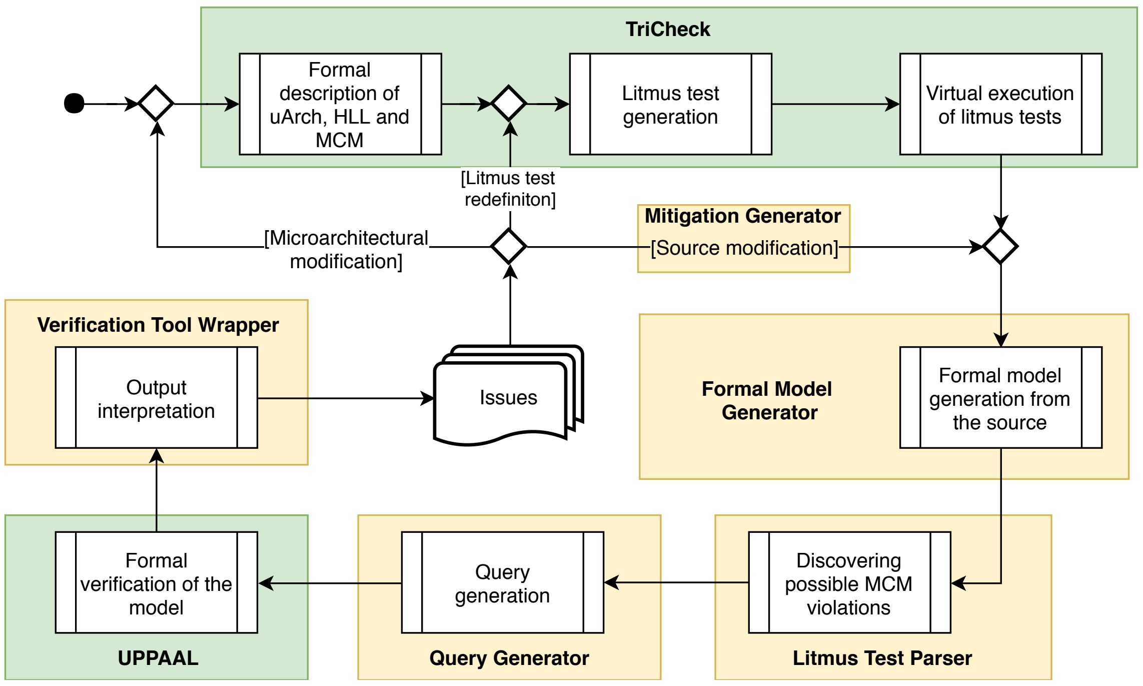

TriCheck
Formal description of uArch, HLL and MCM
Litmus test generation
Virtual execution of litmus tests
[Litmus test redefiniton]
[Microarchitectural modification]
Mitigation Generator
[Source modification]
Verification Tool Wrapper
Output interpretation
Issues
Formal Model Generator
Formal model generation from the source
Formal verification of the model
UPPAAL
Query generation
Query Generator
Discovering possible MCM violations
Litmus Test Parser

**Figure 6.2:** Figure 3.1 re-drawn with the corresponding modules for each step, showing the optimal work- and toolflow of MCMEC.

To aid the use of this tool, I have created an extension for the widely used Visual Studio Code text editor/IDE[1]. This creates shortcuts so that the tool can be used much easier and the output can be visualized in a more elegant fashion - See Figures 6.3 and 6.4.

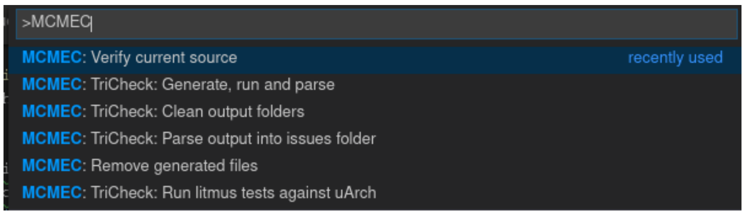

```
>MCMEC
MCMEC: Verify current source                                   recently used
MCMEC: TriCheck: Generate, run and parse
MCMEC: TriCheck: Clean output folders
MCMEC: TriCheck: Parse output into issues folder
MCMEC: Remove generated files
MCMEC: TriCheck: Run litmus tests against uArch
```

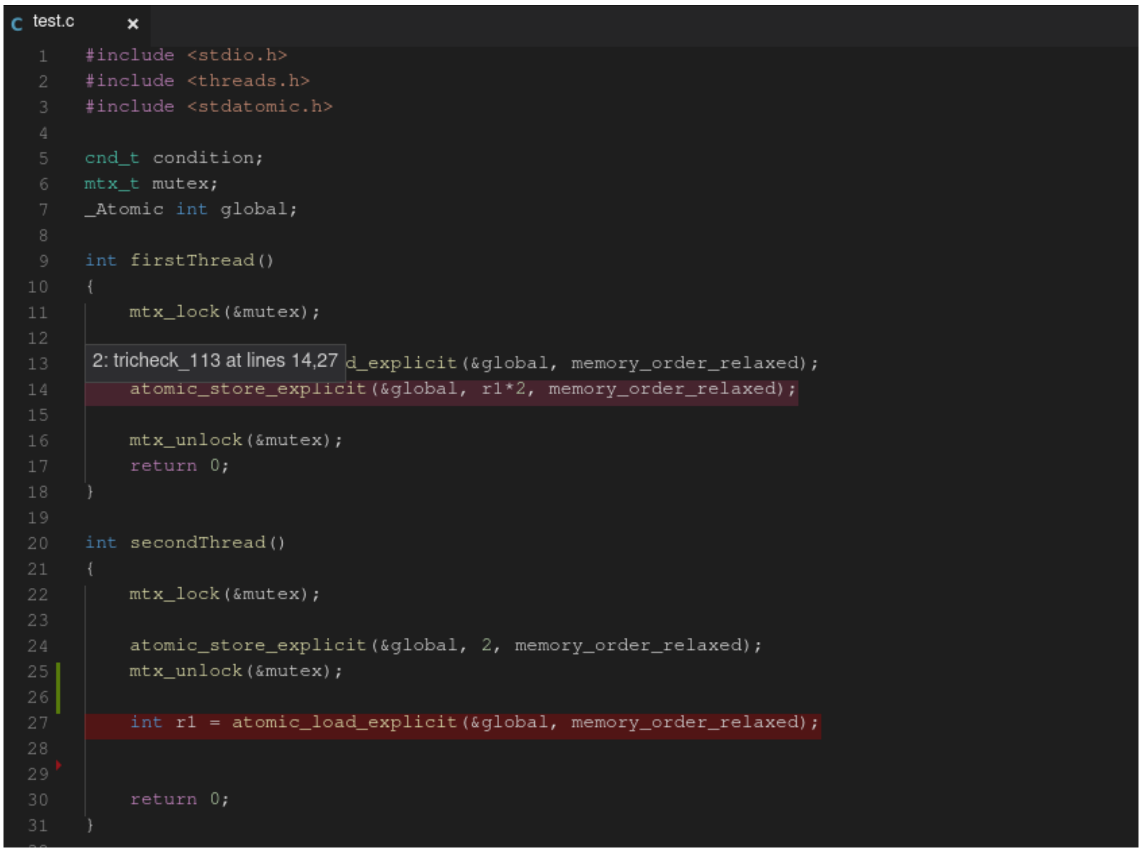
**Figure 6.3:** The possible commands to be run with the MCMEC VSCode extension.

## 6.2 Hardware-Level Memory Validation

This section is about the inner functionality of the tool **TriCheck** [25], which is an essential part of the approach I propose.

Consistency from the compiler downwards is something we mostly take for granted. However, as it has been shown, there are some possible violations even in modern architectures when it comes to corner cases - which is mostly anticipated as until recently there has been no general-purpose verification tool that could test if the memory model of high-level languages keeps intact down to the metal itself. This, however, changed with the release of TriCheck [25] and its dependencies PipeCheck [14] and Herd [3] – these tools attempt

[1]https://code.visualstudio.com/



```
test.c  ×
 1  #include <stdio.h>
 2  #include <threads.h>
 3  #include <stdatomic.h>
 4
 5  cnd_t condition;
 6  mtx_t mutex;
 7  _Atomic int global;
 8
 9  int firstThread()
10  {
11      mtx_lock(&mutex);
12
13  2: tricheck_113 at lines 14,27 d_explicit(&global, memory_order_relaxed);
14      atomic_store_explicit(&global, r1*2, memory_order_relaxed);
15
16      mtx_unlock(&mutex);
17      return 0;
18  }
19
20  int secondThread()
21  {
22      mtx_lock(&mutex);
23
24      atomic_store_explicit(&global, 2, memory_order_relaxed);
25      mtx_unlock(&mutex);
26
27      int r1 = atomic_load_explicit(&global, memory_order_relaxed);
28
29
30      return 0;
31  }
```

**Figure 6.4:** The feedback of erroneous lines with the MCMEC VS-Code extension.

to formally verify the memory models of multithreading-capable programming languages and the hardware they are running on.

As a top-down approach firstly the Memory Consistency Model (MCM) of the high-level language needs to be formally described. As it can be seen in Listing 6.1, this formal description builds on axioms and derived rules, allowing for a mathematically backed certainty of the tool. This model is the basis of all our tests, as it will describe if a given situation is allowed or forbidden. The model can be constructed by following the worded memory model from [12].

```
"C++11" withinit

show po
let sb = po
let mo = co

let cacq = acq | (sc & (R | F)) | acq_rel
           | (F & con)
let crel = rel | (sc & (W | F)) | acq_rel
let ccon = R & con
let fr = rf^-1 ; mo
let dd = (data | addr)+

let fsb = [F] ; sb
let sbf = sb ; [F]

let rs_prime = int(_ * _) | (_ * (R & W))
let rs = mo & rs_prime \ ((mo \ rs_prime) ; mo)
```

**Listing 6.1:** Specifying the MCM of C11 for Herd (excerpt)

As a next step we need to specify what tests to run against that ruleset. As we are trying to test memory consistency, these tests will include a specific number of threads and each of those threads are going to include some atomic operations (either loads or stores). This process is automated to a degree - given a *litmus test template*, a generator will produce all possible *litmus tests* by substituting placeholders with values from a list. In this particular case this is done by leaving out places for the exact memory ordering of each atomic operation. A list of memory ordering types can be found in Table 2.1.

Even though at this step we already have the entire set of litmus tests, they cannot be processed as they are not in a runnable form. So, in order to change that we need to compile them into their assembly variants using compiler mappings in the form of Listing 6.2. These can be interpreted by the tool we are using to run the tests on the microarchitecture of our choice - but it is important to state that this is not done in a virtual machine of any sort, but rather interpreted by a program which then either proves or disproves the expected outcome of the test.

```
C11/C++11 op    | prefix;prefix | suffix;suffix
Read relaxed    | NA            | MMFENCE
Write relaxed   | NA            | NA
Read acquire    | NA            | NA
Write release   | NA            | NA
Read seq_cst    | NA            | MMFENCE
Write seq_cst   | NA            | MMFENCE
```

**Listing 6.2:** Example of pre- and suffixes of compiler mappings.

Finally, the microarchitecture needs to be described. This is done in a language called $\mu$Spec as seen in Listing 6.3. This description includes the ordering axioms of the pipeline stages of the microarchitecture under observation. These rules will dictate the way memory values are passed between the different stages of the pipeline.

```
Axiom "Reads_Path":
forall microops "i",
IsAnyRead i =>
AddEdges [((i, Fetch),      (i, Execute),     "path");
          ((i, Execute),     (i, Writeback),    "path")].

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ EdgeExists ((i1, Fetch),  (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "darkgreen").
```

**Listing 6.3:** Specifying the $\mu$Arch (excerpt)

## 6.3 The Formal Model Generator

The first step on the route to the formal verification of a source file is choosing the type of formal model we want to transform our source into. Currently, the formal modelling language called XTA (Uppaal Timed Automaton) is employed as the formal verification of such models can easily be done through the software called Uppaal (See Section 6.6 for more information on this tool), but as it describes *timed* systems it does not suit our use-case fully. In the immediate future I plan on re-writing this part of the project to use a modelling language called CFA, which is much more suitable for this specific project.

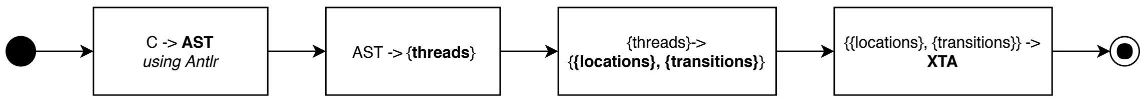


**Figure 6.5:** Flowchart of the C11 -> XTA transformation.

As this implementation is only to corroborate the theoretical work on a proof-of-concept basis, very heavy requirements are set against the type of source code we can transform into an XTA model. Among others, these are the most important ones[2]:

- There is a unique assignment of threads and functions, therefore every function describes exactly one thread.
- All synchronization related variables (such as mtx_t and cnd_t types) and atomic types should be defined globally.
- Every line has only one operation.
- No line can block indefinitely if it is not a synchronization operation.

The requirements set above are necessary as right now the transformation is only experimental and follows a very simple workflow, see Figure 6.5. First, the C11 code is parsed and an Abstract Syntax Tree (AST) is constructed from its contents. Then, this AST is transformed into a set of threads and their variables, which will be broken up into locations and transitions in the next step. The final step is to assemble all information (variables, locations, transitions) and create an XTA model. The only really interesting step is the third one: How is a set of threads (still in AST-like form) transformed into a set of transitions and locations?

The answer to this question is given with a set of rules:

- If an operation is atomic, create a location that includes the memory order and the line number of this operation, and also a transition from the last location to this one without any label.
- If an operation synchronizes with another thread, create a location that includes the type of synchronization and the line number, and also a transition with the correct synchronization option.
- If an operation changes the program flow, create helper locations and transitions to them to aid XTA generation.
- Create helper locations at the beginning and end of each thread to aid XTA generation (only one synchronization label can be placed on a transition, therefore otherwise a new thread (which starts in a blocked state) could not start with a synchronization operation such as mtx_lock).

As this is a proof-of-concept implementation, I did not include any serious optimization techniques other than excluding non-atomic and non-synchronization operations (which might influence the time of execution, but will never change whether two operations might be executed at the same time or not). This will create a fairly large state space which

[2]Here only the practical requirements are stated. However, one should not forget that theoretical requirements have also been set in Section 4.1.

could mean a worse verification time for big codebases heavy on atomic operations, but this issues is outside the scope of this prototype.

As an example, the code from Figure 6.4 was transformed to an XTA model using the MCMEC tool. This produced the following result:

```
int var_mutex;

chan mutex, condition, main_irstThread, main_econdThread, irstThread, econdThread, main_ain;


process firstThread() {
clock x;
state
    firstThread_in,
    firstThread_start,
    mtx_lock_0,
    relaxed_load_global_13,
    relaxed_store_global_14,
    mtx_unlock_1,
    final;
init firstThread_in;
trans
    firstThread_in  ->  firstThread_start{ sync irstThread?; },
    firstThread_start  ->  mtx_lock_0{ guard var_mutex == 0; assign var_mutex = var_mutex + 1; },
    mtx_lock_0  ->  relaxed_load_global_13{ },
    relaxed_load_global_13  ->  relaxed_store_global_14{ },
    relaxed_store_global_14  ->  mtx_unlock_1{ assign var_mutex = var_mutex - 1; },
    mtx_unlock_1  ->  final{ sync main_irstThread!; };
}

process main() {
clock x;
state
    main_in,
    relaxed_store_global_36,
    thrd_create_4,
    thrd_create_5,
    thrd_join_6,
    thrd_join_7,
    final;
init main_in;
trans
    main_in  ->  relaxed_store_global_36{ assign var_mutex = 0; },
    relaxed_store_global_36  ->  thrd_create_4{ sync irstThread!; },
    thrd_create_4  ->  thrd_create_5{ sync econdThread!; },
    thrd_create_5  ->  thrd_join_6{ sync main_econdThread?; },
    thrd_join_6  ->  thrd_join_7{ sync main_irstThread?; },
    thrd_join_7  ->  final{ sync main_ain!; };
}

process secondThread() {
clock x;
state
    secondThread_in,
    secondThread_start,
    mtx_lock_2,
    relaxed_store_global_24,
    relaxed_load_global_25,
    mtx_unlock_3,
    final;
init secondThread_in;
trans
    secondThread_in  ->  secondThread_start{ sync econdThread?; },
    secondThread_start  ->  mtx_lock_2{ guard var_mutex == 0; assign var_mutex = var_mutex + 1; },
    mtx_lock_2  ->  relaxed_store_global_24{ },
    relaxed_store_global_24  ->  relaxed_load_global_25{ },
    relaxed_load_global_25  ->  mtx_unlock_3{ assign var_mutex = var_mutex - 1; },
    mtx_unlock_3  ->  final{ sync main_econdThread!; };
}
system firstThread, main, secondThread;
```

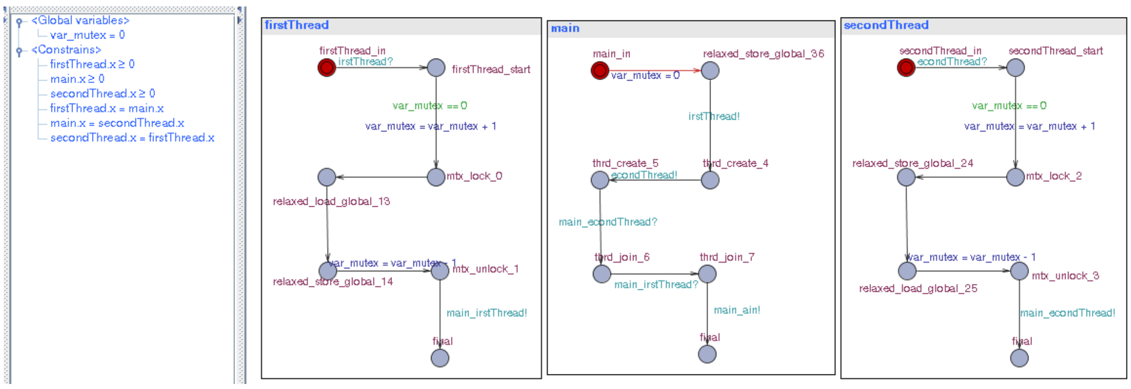


**Figure 6.6:** Visualization of the XTA system above with the tool UPPAAL.

## 6.4 The Litmus Test Parser

Not much can be said about litmus test parsing that was not already mentioned in Section 5.1, so only some Given a set of litmus tests that were found violating the MCM of the high-level language, we need to construct the following data structure:

```
issue <issue_name> {<atomic op. type of thread 1> ; <atomic op. type of thread 1> ; ... : <atomic op
    . type of thread 2> ; ... : ...}
```

An example output for the inputs uArch=TSO-RR.uarch, HLL = C11 and compiler mappings as in Listing 5.10 is the following:

```
issue tricheck_0 { seq_cst store : seq_cst load : seq_cst load }
issue tricheck_1 { acquire load : release store }
issue tricheck_2 { seq_cst store : relaxed load : relaxed load }
issue tricheck_3 { acquire load : seq_cst store }
issue tricheck_4 { relaxed store : seq_cst load : acquire load }
issue tricheck_5 { acquire load : seq_cst store }
issue tricheck_6 { release store : relaxed load : relaxed load }
[...]
issue tricheck_171 { seq_cst store : relaxed load : acquire load }
issue tricheck_172 { acquire load : release store }
issue tricheck_173 { release store : relaxed load }
issue tricheck_174 { seq_cst store : acquire load }
```

This structure is simply constructed by traversing the AST of litmus tests and writing found operations into the issues file as they came up, as the exact ordering of the threads does not play any role in the outcome of the tests.

## 6.5 The Query Generator

Even though the algorithm I proposed in Section 5.2 functions as expected for programs not utilizing any control flow modifiers (such as *if-else*, *loops*, etc), it can still serve as a filter for verification queries, limiting the combinations of probelmatic lines to check. For a general-purpose solver I am using the tool UPPAAL, which is a model checking tool designed to compute and analyze the state space of systems. As it operates on the specification language TCTL (Timed Computation Tree Logic), I need to generate the corresponding query for the formula $\forall\, p \in P : \sigma(p) \cap L_r^n = \emptyset$ (*where* $L_r^n$*: reachable state space,* $P$*: all sets of problematic lines and* $\sigma(x)$ *denotes all possible permutations of* $x$) with the following syntax:

```
E<>(p_11 & p_12 & ... & p_1n)
E<>(p_21 & p_22 & ... & p_2n)
...
E<>(p_m1 & p_m2 & ... & p_mn)
```

Here, locations can be identified with line numbers (which are $p_{ij}$, where $p_{ij}$ is the $j^{th}$ element of $p_i \in P$). In this query, *E<>* means we are proving if a state or set of states are reachable - which is exactly what we are trying to find out, as the original question can be formulated in the following way: *Are the problematic lines reachable all at the same time?*

When using the outputs of the previous two chapters as input, the following query file will be generated, which can be given to the formal verification tool UPPAAL that will determine which lines are reachable[3]:

```
//tricheck_113
E<>(firstThread.relaxed_load_global_13 & firstThread.relaxed_store_global_14)
E<>(firstThread.relaxed_load_global_13 & main.relaxed_store_global_36)
E<>(firstThread.relaxed_load_global_13 & secondThread.relaxed_store_global_24)
E<>(firstThread.relaxed_store_global_14 & secondThread.relaxed_load_global_25)
E<>(main.relaxed_store_global_36 & secondThread.relaxed_load_global_25)
E<>(secondThread.relaxed_load_global_25 & secondThread.relaxed_store_global_24)
//tricheck_149
E<>(firstThread.relaxed_load_global_13 & firstThread.relaxed_store_global_14 & secondThread.
    relaxed_load_global_27)
E<>(firstThread.relaxed_load_global_13 & main.relaxed_store_global_37 & secondThread.
    relaxed_load_global_27)
E<>(firstThread.relaxed_load_global_13 & secondThread.relaxed_load_global_27 & secondThread.
    relaxed_store_global_24)
```

**Listing 6.4:** The query file generated from the issue list in Section 6.4 and the source code in Figure 6.4.

## 6.6 The Verification Tool Output Parser

The generated query from the previous section, when fed into the UPPAAL Model Checker, will return with a result for each line that can be handled as if they were *answers* to the *questions* that each line represents. For the inputs {*XTA model*, *Query file*} from previous sections the following output is produced:

```
    Options for the verification:
    Generating shortest trace
    Search order is breadth first
    Using conservative space optimisation
    Seed is 1539680923
    State space representation uses minimal constraint systems
  ^[[2K
  Verifying formula 1 at line 30
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 2 at line 31
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 3 at line 32
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 4 at line 33
  ^[[2K -- Formula is satisfied.
  Showing example trace.
  ^[[2K
```

[3]Lines starting with // will be ignored by the parser and therefore they can include comments. These comments are the names of litmus tests which provide better feedback for the developer when displayed

```
  Verifying formula 5 at line 34
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 6 at line 35
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 7 at line 58
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 8 at line 59
  ^[[2K -- Formula is NOT satisfied.
  ^[[2K
  Verifying formula 9 at line 60
  ^[[2K -- Formula is NOT satisfied.
```

Even though this is hard to read (especially because of the escape sequences), it includes all the required information: If the string ”Formula is satisfied” is found in a line, then the line above it will have the number of the formula and which line it is in. These are the occurrences we are looking for, as these tell us that the specific set of states described in that line of the query file is *observable* at a given time, and therefore a solution must be found to mitigate this threat - otherwise there is the possibility that once the two atomic operations are going to get executed at the same time and then a *memory value will change in an undefined way*. Here, the only such statement is that in line 33 (line 5 in the previous section’s Listing), which has two locations: *firstThread.relaxed_store_global_14* and *secondThread.relaxed_load_global_25* which both include the line numbers from the original source file - therefore it is easy to formulate the set $I$ (given in JSON formatting):

```
{
  "issues": [
    {
      "name": "tricheck_113",
      "values": [
        14,
        27
      ]
    }
  ]
}
```

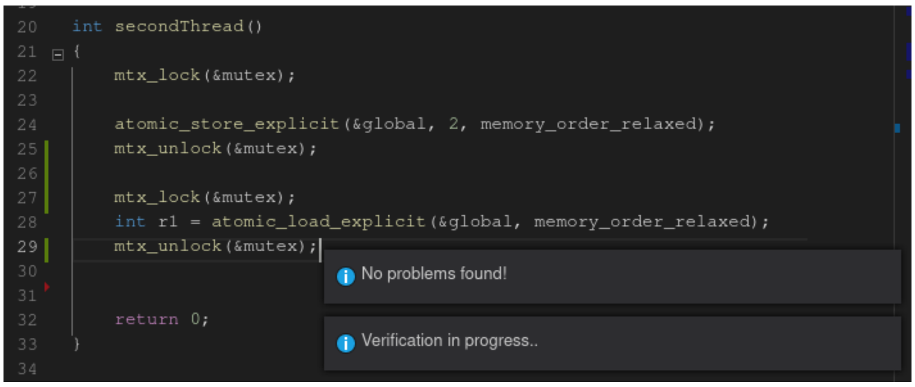
This means that the broken rule is called *tricheck_113* and the lines are {14, 27} in the original source file.

## 6.7 The Mitigation Generator

A mitigation for the discovered problem can be found by following the algorithm depicted by the flowchart in Figure 5.9. Here, one of the lines (Line 14) is already inside the mutex environment of *mtx_t mutex*, therefore the other line should be put into such an environment with the same *mtx_t mutex* variable. This can be done by substituting Line 27 with the following block:

```
    mtx_lock(&mutex);
    int r1 = atomic_load_explicit(&global, memory_order_relaxed);
    mtx_unlock(&mutex);
```

If the verification tool is run again on this refined input, the output will be empty, therefore no MCM violations will be observable after this program is deployed in the target environment.



```
int secondThread()
{
    mtx_lock(&mutex);

    atomic_store_explicit(&global, 2, memory_order_relaxed);
    mtx_unlock(&mutex);

    mtx_lock(&mutex);
    int r1 = atomic_load_explicit(&global, memory_order_relaxed);
    mtx_unlock(&mutex);


    return 0;
}
```

```
No problems found!
Verification in progress..
```

**Figure 6.7:** The output of the tool when run on this refined input.

# Chapter 7

# Evaluation

In this chapter I show metrics and benchmarks about the proof-of-concept implementation presented in Chapter 6. As this implementation did not aim to be the most optimized tool for this job, these numbers should only be taken as a proof of the applicability of the idea itself.

I will show the following metrics (in braces their abbreviation for later use):

- number of lines of code in the original source (LOC)
- number of locations in the generated XTA model (NOL)
- number of query lines (NOQ)
- number of issues found (NOI)
- overall time of execution (t)
- correctness of the mitigation (c)

### 7.0.1 General Tool Environment

In order to get reproducible results the precise circumstances must be set first. These are the following:

- High-level language: C11 with headers <threads.h> and <stdatomic.h>
- $\mu$Spec: TSO-RR[1]
- Compiler mappings: See Listing 6.2
- MCM specification: C11[2]
- UPPAAL version: UPPAAL 4.1.15
- TriCheck[3]

For the verification of source files the tool MCMEC will be used that I presented in detail in Chapter 6. For the above-defined environment 101 litmus tests (out of the theoretical 1215) were found violating the MCM.

[1]https://github.com/ctrippel/TriCheck/blob/master/uarches/TSO-RR-solution.uarch
[2]https://github.com/ctrippel/TriCheck/tree/master/util/herd
[3]https://github.com/ctrippel/TriCheck

### 7.0.2 The Benchmarks

The benchmarks I will be using come from the Mintomic test suite[4]. As these use the Mintomic atomic library (this was widely used before concurrency got introduced a standard feature in C11/C++11) and are written mostly in C++, they must be first re-written to be used by pure C11 compliant tools (such as MCMEC itself). The benchmarks I will be running are the following:

- add_triangle_64.c
- load_linkedlist_64.c
- generic_atomic_test.c (this one is *not* part of the Mintomic test suite but shows the use of the tool very well)
- generic_atomic_test_fail.c (this one is *not* part of the Mintomic test suite but shows the use of the tool very well)

The benchmarks' modified source code can be found in the Appendix. The results can be seen in Tables 7.1 and 7.2[5].

| | **LOC** | **NOL** | **NOQ** |
|---|---|---|---|
| add_triangle_64.c | 24 | 14 | 0 |
| load_linkedlist_64.c | 39 | 18 | 2 |
| generic_atomic_test.c | 49 | 23 | 9 |
| generic_atomic_test_fail.c | 47 | 21 | 9 |

**Table 7.1:** Metrics about the verification subjects.

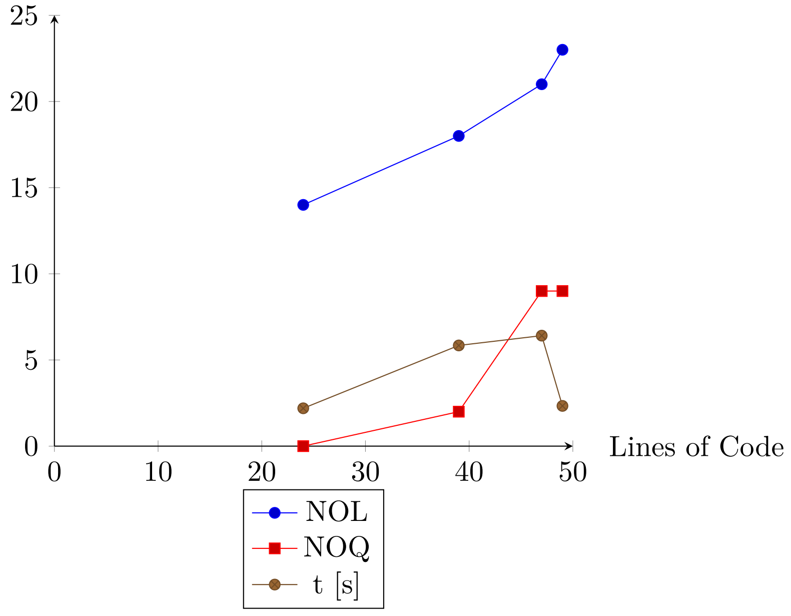
| | **NOI** | **t** | **c** |
|---|---|---|---|
| add_triangle_64.c | 0 | 2.196s | N/A |
| load_linkedlist_64.c | 1 | 5.843s | CORRECT |
| generic_atomic_test.c | 0 | 2.334s | N/A |
| generic_atomic_test_fail.c | 1 | 6.412s | CORRECT |

**Table 7.2:** Results of the verification and the mitigation generation.

In the tables above there are several noteworthy phenomena to be discussed. In order to analyze the he results, four metrics are summarized on the following plot:

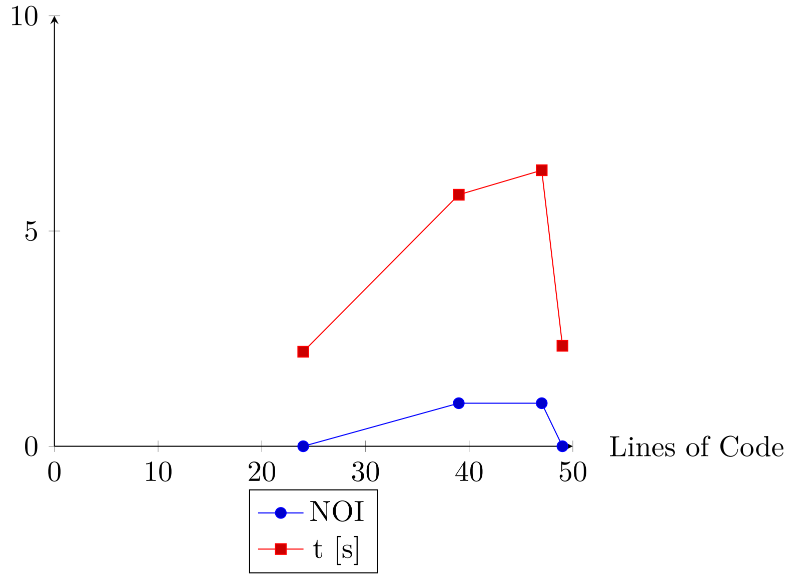
[4]http://mintomic.github.io/tests/

[5]Using Intel(R) Celeron(R) CPU N3450 @ 1.10GHz with 6GB of system memory and eMMC as its main storage device, running Linux kernel 4.18.12-arch1-1-ARCH.



Among these four metrics a line of correlation can be found:

- As the number of lines grows bigger, the number of locations (NOL) grows in an almost linear fashion (if the structure of the two programs resemble each other)
- As the number of locations grows, the number of query lines (NOQ) grows at almost the same pace (if the frequency of atomic operations does not differ in the two programs)
- As the number of query lines grows, the time of verification (t) does **not** grow at the same pace. The reason for this phenomenon can be discovered in the following plot by taking the number of issues (NOI) also into account:



As it can be seen, the verification time only stands out when an issue is found (even when the two programs only differ in two lines and therefore two states).

# Chapter 8

# Related Work

In this chapter I introduce some of the concepts and findings in other works that are related to this approach.

## 8.1 Software Verification

Software verification is an essential part of software engineering which aims at assuring if a software satisfies its specification fully [9, 7]. Two main approaches exist in this field, which are *static verification* and *dynamic verification*. The former provides a way to prove criteria without needing to execute the software just by looking at its source code (or rather a formal model thereof), while the latter executes the program and checks its output after a given set of inputs.

Dynamic tests are employed in almost all software development practices for example in the form of module tests. Static verification of programs is much more sophisticated as they operate on the theoretical level allowing for a deeper, but more circumstantial verification.

## 8.2 Full Stack Memory Consistency Model Verification

The full-stack verification of Memory Consistency Models (MCM for short) is a fairly new concept introduced in [25]. In short, it categorizes the ruleset of the MCM (defined by the language specification, e.g. [12]) in a specific situation into two groups: *allowed* and *forbidden*. The former group includes all outcomes that are *allowed* by the specific MCM, while the latter has all the *forbidden* ones. This is done on the abstraction level of the programming language itself, therefore simulating a program that would be in that specific situation. After that, it compiles these situations (called *litmus tests*, see Definition 2.4.3) into their assembly equivalents using the specified compiler mappings. This representation can be evaluated by the rules of the microarchitecture itself, yielding two groups for each litmus test: *observable* and *non-observable*. If the *observable* outcomes include any *forbidden* outcome then the MCM does **not** stay consistent, and if there are *non observable* but *allowed* outcomes then the MCM is too strictly enforced. In this work we only care about the first option as the second one does not cause any problems other than performance-related ones, which is not in the scope of this project.

The tool for this job (called **TriCheck**) was used to successfully unearth very serious memory ordering problems in the ISA of RISC-V, such as the lack of cumulative memory fences or absence of roach-motel movement for SC atomics.

In order to put the difference between their approach and mine into perspective, they supply the situations that theoretically violate the MCM, which I utilize to find actually offending situations in a given program, putting more emphasis on the practical side of the approach.

## 8.3 Program Source to Formal Model Transformation

The first barrier to overcome when verifying actual programs is the transformation of the source code into a model which can be given to verification tools. However, during this transformation many decisions have to be made - for example, if the target model is state-based, choosing what states to create will influence many aspects of the verification process, such as the time of verification. Often, this aspect is the most limiting factor when verifying complex programs.

The reduction in statespace can be the key to allow the verification of previously unverifiable software. As discussed in [21], there are many options in the hands of the developers creating source code to formal model generators, such as dead branch elimination, constant folding, or program slicing (see [21] for details).

In this work, I use a limited version of program slicing which allows for a great reduction in the state space of programs that include atomic or synchronization operations only in a small part (which is true for most real-life programs).

## 8.4 HW-SW Co-Simulation

As presented in [23], there have been approaches to HW-SW co-verification through simulation. Even though this is an inferior approach in contrast with the methods of formal verification with respect to completeness, but a violating situation *might* be found this way as well. It is much easier to simulate the behaviour of HW-SW systems (as it is enough to run a program on the target platform) and this method can be used to discover trivial flaws in chip/software design, but if we do not find any erroneous situations we cannot state with certainty that the components are *correct*, as opposed to formal verification through mathematical means. This latter is much harder to execute, but it gives a certainty to its outcome as we are using mathematically derived rules.

## 8.5 System on a Chip Verification

Many approaches have been published that verify the microcode running on SoCs of embedded systems, such as [1] or [16]. These are not general verification tools or approaches, but rather the description of the workflow one needs to employ in order to release a formally verified microcode for that SoC. These mitigate the risk that comes with hidden CPU cores (as the package does not allow for external debuggers to be attached due to its SoC-ness).

## 8.6 Multi-Core System Verification

Multi-core system verification is gaining more and more standing ground in the field of formal verification. Discovering dynamic deadlocks[1] is already possible as discussed in [4], and even the JML (Java Modelling Language[2]) was extended to include specification capabilities of multi-threaded programs [20].

[1]The system is in a state from which it cannot proceed anywhere with its currently programmed behaviour.

[2]https://www.openjml.org/

# Chapter 9

# Conclusion

Even though multi-core system verification is still in its early days, I believe it will eventually take over the scene as safety-critical systems get more and more complex - as multi-core processors took over the consumer market due to their outstanding performance compared to their single core counterparts, the same phenomenon will likely transfer itself to the embedded systems controlling automated cars, airplanes or nuclear reactors. The main reason this has not fully happened yet is the incompleteness of verifiability of such systems, as concurrency carries many potentially unforeseen properties with itself, such as the violation of the Memory Consistency Models of common languages.

The work presented in this paper tries to narrow the number of unverifiable aspects of multi-core systems. By using the formal description of the microarchitecture, the memory model of the programming language and its compiler mappings to generate a set of possible scenarios which violate the MCM ruleset, the source code of a program can be inspected if any of these scenarios might happen during its operation therefore automating the verification of software based on hardware and platform-specific factors. With this information a possible mitigation technique can either be generated automatically, or the developer of the program can modify it until none of the violating situations is not observable any more. This approach complements the usual way of verifying software, where the semantics of source code is assumed to be given and correct, by ruling out faults related to previously unconsidered issues.

As I demonstrated through a proof-of-concept implementation, the above described approach is in fact solvable. Even though the implementation has some restrictions on the type of software it can verify, the main idea is proven to be applicable to this situation as simple C11 programs could successfully be verified.

The limitations of this implementation (and even the approach itself) can be narrowed down greatly if the components (See Figure 6.1) themselves improve. For example, the formal model generator that takes the source as an input poses as the biggest limiting factor in the toolchain, and therefore a much greater set of programs might be verified if a more apt generator would take its place. Furthermore, the mitigation generation step might be improved in the future to fully automate this process of verification *and* correct the program as well. This could save many hours of precious work that would otherwise need to be done by an experienced developer.

# List of Figures

# Bibliography

[1] *Hardware/Software Co-verification*, pages 235–315. Springer US, Boston, MA, 2002. ISBN 978-0-306-46995-4. DOI: `10.1007/0-306-46995-2_6`.

[2] *Cortex-A9 MPCore, Programmer Advice Notice.* 2011.

[3] 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`.

[4] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, *Hardware and Software, Verification and Testing*, pages 208–223, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32605-2.

[5] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A concrete memory model for compcert. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 67–83, 2015. DOI: `10.1007/978-3-319-22102-1\_5`.

[6] Abdallah Cheikh, Gianmarco Cerutti, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. The microarchitecture of a multi-threaded RISC-V compliant processing core family for iot end-nodes. In *Applications in Electronics Pervading Industry, Environment and Society - APPLEPIES 2017, Rome, Italy, 21-22 September 2017*, pages 89–97, 2017. DOI: `10.1007/978-3-319-93082-4\_12`.

[7] Edmund M. Clarke. *Model checking.* The MIT Press, 2018.

[8] Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors. *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-22261-8. DOI: `10.1007/b98282`.

[9] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering.* Prentice Hall, 1991. ISBN 978-0-13-818204-5.

[10] John L. Hennessy, David A. Patterson, and Krste Asanovic. *Computer architecture: a quantitative approach.* Morgan Kaufmann, 2019.

[11] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, volume 2.

[12] ISO 9899:201x. Information technology — programming languages — C. Standard, International Organization for Standardization, April 2011.

[13] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for intel and AMD microarchitectures. *CoRR*, abs/1809.00912, 2018.

[14] 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`.

[15] Paul E. McKenney and Raul Silvera. Example power implementation for c/c memory model, Mar 2011.

[16] Ashok B. Mehta. *Hardware/Software Co-verification*, pages 243–253. Springer International Publishing, Cham, 2018. ISBN 978-3-319-59418-7. DOI: `10.1007/978-3-319-59418-7_12`.

[17] Hiroshi Nakamura, Takanori Arai, and Masahiro Fujita. Formal verification of a pipelined processor with new memory. In *9th Pacific Rim International Symposium on Dependable Computing (PRDC 2002), 16-18 December 2002, Tsukuba-City, Ibarski, Japan*, pages 321–324, 2002. DOI: `10.1109/PRDC.2002.1185653`.

[18] David A. Patterson. 50 years of computer architecture: From the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set. In *2018 IEEE International Solid-State Circuits Conference, ISSCC 2018, San Francisco, CA, USA, February 11-15, 2018*, pages 27–31, 2018. DOI: `10.1109/ISSCC.2018.8310168`.

[19] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther, and Jeremy Kepner. Measuring the impact of spectre and meltdown. *CoRR*, abs/1807.08703, 2018.

[20] Edwin Rodríguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending jml for modular specification and verification of multi-threaded programs. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 551–576, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31725-8.

[21] Gyula Sallai. Compiler optimizations for software verification, 2016.

[22] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 311–322, 2012. DOI: `10.1145/2254064.2254102`.

[23] Luc Séméria and Abhijit Ghosh. Methodology for hardware/software co-verification in C/C++ (short paper). In *Proceedings of ASP-DAC 2000, Asia and South Pacific Design Automation Conference 2000, Yokohama, Japan*, pages 405–408, 2000. DOI: `10.1145/368434.368712`.

[24] 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*, page 176–179, Vienna, Austria, 2017. FMCAD Inc., FMCAD Inc. ISBN 978-0-9835678-7-5.

[25] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Tricheck: Memory model verification at the trisection of software, hardware, and ISA. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 119–133, 2017. DOI: `10.1145/3037697.3037719`.

[26] Andrew Waterman and Krste Asanovic. The RISC-V instruction set manual. volume 1: User-level isa, version 2.2. *https://www.riscv.org*, May 2017.

[27] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984. DOI: `10.1109/TSE.1984.5010248`.

# Appendix

## A.1 Benchmarks

[1]

```c
#include<stdatomic.h>
#include<threads.h>

_Atomic int g_sharedInt;

int threadFunc()
{
    for (int i = 0; i < 10000000; i++)
    {
        atomic_store_explicit(&g_sharedInt, i, memory_order_relaxed);
    }
    return 0;
}

int main()
{
    g_sharedInt = 0;
    thrd_t threadFunc1Id, threadFunc2Id;
    thrd_create(&threadFunc1Id, threadFunc, NULL);
    thrd_create(&threadFunc2Id, threadFunc, NULL);
    thrd_join(threadFunc1Id, NULL);
    thrd_join(threadFunc2Id, NULL);
    return 0;
}
```

**Listing A.1.1:** test_add_triangle_64.c

```c
#include<stdatomic.h>
#include<threads.h>


struct Node
{
    Node* next;
};

_Atomic int g_head;

int threadFunc2(void*data)
{
    for (; iterator < 1999;)
    {
        atomic_store_explicit(&g_head, pointers[iterator++], memory_order_relaxed);
        struct Node* curr = (struct Node*) atomic_load_explicit(&g_head, memory_order_relaxed);
        curr->next = (struct Node*)pointers[iterator++];
    }
        return 0;
}
```

[1]The original benchmarks can be found in the repository https://github.com/mintomic/mintomic

```c

int main()
{
    g_head = 0;
    for(int i = 0; i<2000;i++){
        struct Node node;
        pointers[i]=(long)&node;
    }
    thrd_t threadFunc1Id, threadFunc2Id;
    thrd_create(&threadFunc1Id, threadFunc1, NULL);
    thrd_create(&threadFunc2Id, threadFunc2, NULL;
    printf("done");
    thrd_join(threadFunc1Id, NULL);
    thrd_join(threadFunc2Id, NULL);
    return 0;
}
```

**Listing A.1.2:** test_linkedlist_64.c

```c
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

cnd_t condition;
mtx_t mutex;
_Atomic int global;

int firstThread()
{
    mtx_lock(&mutex);

    int r1 = atomic_load_explicit(&global, memory_order_relaxed);
    atomic_store_explicit(&global, r1*2, memory_order_relaxed);

    mtx_unlock(&mutex);
    return 0;
}

int secondThread()
{
    mtx_lock(&mutex);

    atomic_store_explicit(&global, 2, memory_order_relaxed);
    mtx_unlock(&mutex);

    mtx_lock(&mutex);
    int r1 = atomic_load_explicit(&global, memory_order_relaxed);
    mtx_unlock(&mutex);


    return 0;
}

int main()
{
    thrd_t firstThreadId, secondThreadId;

    atomic_store_explicit(&global, 2, memory_order_relaxed);

    thrd_create(&firstThreadId, firstThread, NULL);
    thrd_create(&secondThreadId, secondThread, NULL);

    thrd_join(secondThreadId, NULL);
    thrd_join(firstThreadId, NULL);
    printf("%d\n", global);

    return 0;
}
```

**Listing A.1.3:** generic_atomic_test.c

```c
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

cnd_t condition;
mtx_t mutex;
_Atomic int global;

int firstThread()
{
    mtx_lock(&mutex);

    int r1 = atomic_load_explicit(&global, memory_order_relaxed);
    atomic_store_explicit(&global, r1*2, memory_order_relaxed);

    mtx_unlock(&mutex);
    return 0;
}

int secondThread()
{
    mtx_lock(&mutex);

    atomic_store_explicit(&global, 2, memory_order_relaxed);
    mtx_unlock(&mutex);


    int r1 = atomic_load_explicit(&global, memory_order_relaxed);



    return 0;
}

int main()
{
    thrd_t firstThreadId, secondThreadId;

    atomic_store_explicit(&global, 2, memory_order_relaxed);

    thrd_create(&firstThreadId, firstThread, NULL);
    thrd_create(&secondThreadId, secondThread, NULL);

    thrd_join(secondThreadId, NULL);
    thrd_join(firstThreadId, NULL);
    printf("%d\n", global);

    return 0;
}
```

**Listing A.1.4:** generic_atomic_test_fail.c