Technische Universität Braunschweig

Institute of Theoretical Computer Science

Master's Thesis

Memory-Model-aware Static Analysis of Concurrent Programs

René Maseli

June 18, 2021

Institute of Theoretical Computer Science
Prof. Dr. rer. Nat. Roland Meyer

Supervisor:
Thomas Haas, M. Sc.

Statement of Originality

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, June 18, 2021

_____

Contents

Introduction

Static analysis is the examination of properties that all executions of a given program share. Oftentimes this entails a description of a feature and a collection of algorithms to detect it. More so than its efficiency, the most important property of a program is its safety, the absence of reachable error states. The attached problem bears the name *Verification* and was proven to be undecidable, so there is no algorithm that, given an arbitrary program and a well-defined set of error states, eventually either finds an unintended behavior, or proofs the absence of such.

The instructions of a program include jumps, which redirect the process not only forwards but also backwards, enabling executions of infinite length. Binding to a maximal length leads to the decidable problem *Bounded Model Checking* asking whether a violation occurs in the first number of instructions of a program. In other words, only programs are accepted which never jump back and so never repeat an instruction.

This thesis discusses static analyses specialised for bounded programs, that distribute onto multiple threads assignable to different processing units of a computer and executable in parallel. All participating units share a common memory space and may use it to communicate to each other: If addresses coincide a thread can read a value that has been previously written by another thread.

A wide variety of processor architectures exists and vendors seek clever methods to accelerate computers. Programmers and compilers aim to make use of any mean that the hardware provides to accelerate the program. Communication with the shared storage is slow in comparison to regular instructions. Processing units possess their own local storage, the *registers*, usually with lower capacity and lifetime and invisible to other units. Efficient programs will prioritise the register space over the shared memory.

The shared memory is also accelerated. Processing units tend to use the time while waiting for such slow operations to finish. Most architectures cache operations in yet another intermediate storage, the *cache*, serving as a direct surrogate to the shared memory. On one hand, this entails that write operations of a unit may be delayed and do not immediately become visible to other units and that read operations may return a stale value and on the other hand, that they arrive at other threads in a different order. The capacity of the cache depends on the actual hardware and is unknown to the program at compile time.

Barrier instructions counter this behavior, restricting the freedom of the processor on certain program states. For example does an *acquire* barrier issue a refresh of the cache before the next read operation, preventing skipping over the latest updates from other threads. And a *release* barrier delays subsequent write operations until all recent writes have been committed, thus becoming visible to the others. More types exist and the set

of barrier types is yet unexplored. A programmer will retain the placement of barrier instructions to where it is necessary for consistent inter-thread communication.

The subject of this thesis is the class of concurrent programs in machine code, that only sparsely impose restrictions, whether or in which order inter-thread communication happens. The bounded model checker *Dartagnan* searches for semantic anomalies like reachable error states (assertions placed by the programmer), violations to liveness (eventual progress of all threads) or data race freedom (all communication is synchronisfed).

Research in this field describes the guarantees of an architecture with an *axiomatic memory model*. Such a model is evaluated on a program trace, for each thread a sequence of *events* that were caused by it, and a mapping from read events to the respective source event of the read value. The memory model defines binary relations over the events and constraints over those relations. For instance, a model may impose that no thread shall read from its own write events which were not yet dispatched (Local Consistency). If some constraint is violated, the candidate trace is impossible in the associated architecture. Axiomatic memory models ignore technical limitations, like the previously-mentioned cache capacity, and thus over-approximate the set of possible traces.

The candidate trace can be constructed non-deterministically. Dartagnan encodes a program, its safety specification, and a memory model into a comprehensive proposition. Any model of the formula exposes a trace of the program that may be possible in the generalised model and witnesses some violation of the specification. If the proposition turns out unsatisfiable, the decision procedure yields a formal proof that no witness exists inside the specified bounds. This thesis will dive into the structure of the encoding.

The task of solving a formula is done by an *SMT prover* like Z3 [10] and is $\mathcal{NP}$-complete for the usual value domains. The static analyses in this thesis aim to reduce the complexity of the produced formula to ease the decision process and shorten the combined runtime. They limit on structural properties of the program and memory model so the proposition ideally keeps the kernel problem: to exclude all candidates that are no witnesses.

This thesis tightly orients on the Dat3M project [9, 26, 25, 27], which is currently maintained by Hernán Ponce-de-Léon on Github. It consists of Java applications preferably accepting programs in the boogie programming language, as produced from the SMACK project [29] given an *LLVM intermediate representation* as input. The Java application recompiles and unrolls its input into an acyclic program, then embeds it into an axiomatic model given in a reduced dialect of the modelling language cat [2]. This results in a quantifier-free boolean formula modulo strict integer order (see section 3.1), and modulo theories that support value operations used in the program (currently bit-vectors and integers). If, for example, the application targets state-reachability, the produced formula should be satisfiable if and only if some bounded execution reaches an error state. Dat3M supports reachability of failed assertions, reachability of race

conditions (pairs of accesses to the same location with unspecified memory order), and memory model portability [26].

Chapter 1 introduces the kind of concurrent program that will serve as the target for analysis, as well as required memory-model-unaware static analyses. In chapter 2 the language of memory models is amended, connecting to the already defined notion of traces, and completing the ingredients of the verification problem. We continue to flesh out the problem in chapter 3, together with a formal description of Dartagnan's approach, the SMT encoding. The core of this thesis is discussed in chapter 4: the analysis of *static* relationships between events of a program under a model. We summarise contributions of other authors that inspired our approach in chapter 5 and conclude this thesis with an evaluation through the analysis on real programs and models in chapter 6.

The evaluation shows a reduction in resource consumption in nearly all test instances, appearing to scale well with the size of the program and the complexity of the memory model. We confirm that this particular analysis enhances the capabilities of the verifier by relying on the well-structuredness of axiomatic memory models.

1. Program

In this chapter we define the class of programs we will examine in this thesis. We will use a generalised description of a program that makes use of a shared memory storage. It is heavily inspired by RISC and does not exactly reflect the current internal representation of the Dartagnan project. Furthermore, sections 1.3 and 1.4 describe memory-model-unaware analyses that we will have to perform in order to encode the dataflow later in section 3.2 and to communicate certain thread-local circumstances influencing a concurrent execution.

Concurrent programs distribute their tasks onto different threads, which may or may not execute in parallel. Usually, execution starts with one main thread, sometimes along with a garbage collector thread (see section 6.2), and provides the ability to create new threads by passing a program location to its constructor. Since we will quickly drop unbounded programs and switch to a bounded setting (section 1.5), where even the thread count will be fixed, and since we want to avoid dynamic jumps, where data may freely determine the successive control state, we define a purely static program where an already determined count of threads is already assigned to their disjoint instruction lists.

Definition 1 (Program)**.** *Let D be a set of values with a computable subset $\emptyset \subset \text{true} \subset D$ and $\mathcal{T}$ be a finite set of tags. A $D\mathcal{T}$-program, or just 'program', is a list of $D\mathcal{T}$-threads. A $D\mathcal{T}$-thread, or just 'thread', $\langle R, I\rangle$ consists of a set of registers R and a list of instructions $I \in \mathcal{I}(D,R,\mathcal{T})^*$ with $\mathcal{I}(D,R,\mathcal{T}) := (\{\top\} + R) \times S(D,R) \times T$.*

An instruction $\langle \varphi, s, t\rangle \in \mathcal{I}(D,R,\mathcal{T})$ consists of an optional condition $\varphi \in \{\top\} + R$, a statement $s \in S(D,R)$ and a subset of tags $t \subseteq T$. Let $S(R,D)$ denote the set of events over values D and registers R:

$$
\begin{aligned}
S(R,D) &:= && S_{\texttt{load}}(R,D) \cup S_{\texttt{store}}(R,D) \cup S_{\texttt{eval}}(R,D) \cup S_{\texttt{goto}} \cup \{\texttt{fail}\} \\
S_{\texttt{load}}(R,D) &:= && \{\texttt{ld}\ v \leftarrow k \mid k, v \in R\} \\
S_{\texttt{store}}(R,D) &:= && \{\texttt{st}\ v \rightarrow k \mid k, v \in R\} \\
S_{\texttt{eval}}(R,D) &:= && \{z \leftarrow f(x,y) \mid f \in D^{D\times D}, x, y, z \in R\} \\
S_{\texttt{goto}} &:= && \{\texttt{goto}\ n \mid n \in \mathbb{N}\}
\end{aligned}
$$

A statement is either

- *a load* $\texttt{ld}\ v \leftarrow k$ *on the address k depositing the read value in v*
- *a store* $\texttt{st}\ v \rightarrow k$ *on the address k writing the value v*
- *an evaluation $z \leftarrow f(x,y)$ of a computable operator f consigning its result in z*

- *a jump* `goto` n *to the instruction with index* n
- *an error state* `fail` *to avoid for safety*

Let X be some set and $x \in X^*$ be some list. $|x| \in \mathbb{N}$ denotes the number of elements in x. For $i \in \mathbb{N}$ let x_i denote the i-th element of x, which is defined if and only if $i < |x|$.

This purely syntactical construction misses semantics that we will detail later in section 1.5. A *computation* will carry thread-local register states through a path of each thread.

In the context of verification, the data domain may be infinite like $\mathbb{N}$ or even $\mathbb{C}$, although most practical instances will use bit vectors $\{0,1\}^n$. The label will be used only by the condition φ, where the statement is executed only if $\varphi \in$ true. Usually, all values will be convertible to true ($\top$), with the exception of one 'zero' value 0, convertible to false ($\bot$): true $:= D \setminus \{0\}$ As expected, instructions without a condition (with $\varphi = \top$) shall be executed unconditionally if reached by the control flow (see 1.3).

Note that in contrast to an assembly language, we do not allow dynamic jumps to any address. We assume that the program is present in its entirety. Concepts like *virtual functions* could be implemented by collecting all overriding implementations (thus all allowed destinations for the jump) and replace the dynamic jump with a list of conditionals which is targeted at runtime.

Example 1. *Figure 1 describes a two-threaded litmus test, a program with 'predefined' read-from relationships. Note how a special register* `null` *can avoid unintended dependencies (see section 1.4).*

The meaning of locations $\{x, y, z\}$ will be detailed in definition 1.5. From this point onwards, we will express programs in a more readable fashion, by inlining once-used evaluations and short jumps. Note how definition 1 stays valid, even if we state the equivalence of the program with the form in figure 1.

1.1. Program Order

Each thread consumes the instructions of its program in order of appearance. This total order in which a thread in a trace *issues* the events is called the *program order* of the thread. We extend this term to the program order of the entire trace as the union of program orders of the participating threads.

x po y implies that the events $x, y \in \mathcal{E}$ originate from the same thread. In fact, the axiomatic memory model may derive the internal equivalence relation int of events from the same thread: int $:=$ po $\cup$ po^{-1} $\cup$ id.

The C and C++ standards [28] define a relation called *sequenced-before*, which is related to the program order. It is a partial order over expressions in a program's source code that compilers have to reflect in the order of instructions produced for the expressions, especially if visible side effects are at play.

Figure 1.1.: source code of a litmus test based on Power assembly

```
PPC DETOUR0666
"LwSyncdWW Rfe DpDatadW Rfi DpCtrlIsyncdR Fre"
Cycle=Rfi DpCtrlIsyncdR Fre LwSyncdWW Rfe DpDatadW
Prefetch=0:x=F,0:y=W,1:y=F,1:x=T
Com=Rf Fr
Orig=LwSyncdWW Rfe DpDatadW Rfi DpCtrlIsyncdR Fre
{
0:r2=x; 0:r4=y;
1:r2=y; 1:r4=z; 1:r7=x;
}
 P0           | P1           ;
 li r1,1      | lwz r1,0(r2) ;
 stw r1,0(r2) | xor r3,r1,r1 ;
 lwsync       | addi r3,r3,1 ;
 li r3,1      | stw r3,0(r4) ;
 stw r3,0(r4) | lwz r5,0(r4) ;
              | cmpw r5,r5   ;
              | beq  LC00    ;
              | LC00:        ;
              | isync        ;
              | lwz r6,0(r7) ;
exists
(1:r1=1 /\ 1:r5=1 /\ 1:r6=0)
```

1.1.1. Barriers

Barriers are synchronisation primitives that do not manipulate the register state but enforce the processing unit to perform some form of synchronisation with a shared resource before continuing with the program [22, 4, 31]. All barriers we encountered share being executed by a thread and affect all events independently of their address, but dependently of their relative position in the program. The tool `herd` treats them as distinct events at first, but immediately encapsulate them into relations:

$$\texttt{fence}_F := \text{po} \,;\, (F\_ \cap \text{po})$$

Heavyweight SYNC and DMB possess similar semantics [31]: All preceeding writes become visible to all other threads and all preceeding reads have to be satisfied. This behavior is imposed by MFENCE from TSO, as well, and marks what we assume to be the strongest type of barrier. Apart from this, Power's 'Lightweight SYNC' allows preceeding writes to be reordered with succeeding reads and EIEIO only enforces order on

Figure 1.2.: program of figure 1 in terms of definition 1

$\{\texttt{null}, r_1, r_2, x, y, z\}$

$\langle \top, r_1 \leftarrow (1)(\texttt{null}, \texttt{null}), \emptyset \rangle$
$\langle \top, \texttt{st}\ r_1 \rightarrow x, \emptyset \rangle$

$\langle \top, r_2 \leftarrow (1)(\texttt{null}, \texttt{null}), \{lwsync\} \rangle$
$\langle \top, \texttt{st}\ r_2 \rightarrow y, \emptyset \rangle$

$\{\texttt{null}, \texttt{cmp}, r_3, r_4, r_5, r_6, x, y, z\}$

$\langle \top, \texttt{ld}\ r_3 \leftarrow y, \emptyset \rangle$
$\langle \top, r_4 \leftarrow (\neq)(r_3, r_3), \emptyset \rangle$
$\langle \top, r_4 \leftarrow \text{inc}(r_4, \texttt{null}), \emptyset \rangle$
$\langle \top, \texttt{st}\ r_4 \rightarrow z, \emptyset \rangle$
$\langle \top, \texttt{ld}\ r_5 \leftarrow z, \emptyset \rangle$
$\langle \top, \texttt{cmp} \leftarrow (=)(r_5, r_5), \emptyset \rangle$
$\langle \texttt{cmp}, \texttt{goto}\ 6, \emptyset \rangle$
$\langle \top, \texttt{ld}\ r_6 \leftarrow x, \{isync\} \rangle$

The operators encountered in the program are defined as below.

$$(1) := (x, y) \mapsto 1 \qquad \text{inc} := (x, y) \mapsto x + 1$$

$$(\neq) := (x, y) \mapsto \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases} \qquad (=) := (x, y) \mapsto \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

writes. The function of ISYNC as well as ISB seem to narrow on restricting speculation (see section 1.4.1).

This thesis'definition of program and trace do not include visible barrier instructions or events. Instead, memory events have to be tagged appropriately. In C for example, stores tagged as (rel) are usually compiled into regular stores prepended by a releasing barrier and loads with (acq) would be regular loads immediately followed by an acquiring barrier. Whether the associated barrier `F` appears program-ordered before or after the tagged instruction is defined by the underlying memory model in assignments respectively:

$$\texttt{f}_{\text{before}} := (\text{po} \cap \_F)\ ;\ (\text{id} \cup \text{po})$$
$$\texttt{f}_{\text{after}} := (\text{id} \cup \text{po})\ ;\ (F\_ \cap \text{po})$$

A multi-processor is able to execute an arbitrary number of programs at the same time. Including GPUs, it distributes them onto a fixed number of processing units. In situations where the former number exceeds the latter, a *scheduling* process has to dynamically reassign units to threads, ensuring liveness for every single thread. Whenever such a reassignment happens, the entire context of a thread will usually be synchronised with the shared memory space of its program. In a sequentially-consistent memory model, a

Figure 1.3.: program of figure 1 in an abbreviated form

Wx: `st` $1 \rightarrow x$	Ry: `ld r1` $\leftarrow y$
Wy: `st` $1 \rightarrow y$ ($lwsync$)	Wz: `st` $1 + 0 * $ `r1` $\rightarrow z$
	Rz: `ld r5` $\leftarrow z$
	Rx: `if r5` $=$ `r5 ld r6` $\leftarrow x$ ($isync$)

context switch like this happens every time the executing thread in the sequence of the trace changes. As a consequence, synchronisation is not restricted to those primitives described in this section, but should be allowed to trigger spontaneously. Vafeiadis et al. [36] demonstrate how verifiers struggle if this rule happens to be violated.

1.2. Atomicity

The common programmer would assume each instruction to be atomic by nature. This is certainly not the case if an instruction does either access an array of (even consecutive) addresses, or performs a *read-modify-write* operation. However, a single load or store on one memory address is considered atomic in the sense that there is no further separation of events that describe the operation. One would consider separating the start of a write event and all of its ends at each other thread symbolising its visibility. With the kind of axiomatic memory model described in chapter 2, this is unnecessary. Instead we define atomicity as a property of subsets of events of a common thread. Taking the notion of time out of the equation, we want no other thread to perceive one element of an atomic set of events, while missing out some other.

Architectures come with a minimal size of memory space that can be modified at once, as well as a maximal size. Usually the minimal size will still be one byte, leading to inconsistencies when trying to concurrently modify elements in a packed bitset. Language constructs such as the function `atomic_store` of the standard C library `stdatomic.h` [28, section 7.17] may guarantee atomicity for arbitrary object sizes, requiring the compiler to insert synchronisation primitives for accesses to larger structures. In case of verification, their safety guarantee would appear questionable itself. We count three basic approaches to this subproblem: First would be to remain skeptical and treat the primitives as potential safety risks. Second introduces an equivalence relation to the memory model between events of an atomic group. Dartagnan follows a third approach and tries to merge those cells into one address, while reasoning over values of arbitrary bit count.

The memory model should restrict communication while an atomic operation is happening: No consistent execution shall imply that some event of an atomic operation

became visible strictly before another event of that operation. It is required to be supported by the instruction set associated with the memory model (see example 4). In addition, the information which operations exhibit atomicity must therefore be communicated, serving the model some means to specify the restrictions.

```
void VERIFIER_begin_atomic()
void VERIFIER_end_atomic()
```

SVCOMP defines two primitives for arbitrary atomic operations [8]. The program must declare all events sequenced-between calls to that functions as pairwise-atomic, such that none of them is perceived individually. Of course, no architecture will support all combinations in its instruction set. Dartagnan deals with this by using a simple mutual exclusion algorithm on a static lock.

1.2.1. Load-Reserved / Store-Conditional

Based on the *load-reserved/store-conditional* paradigm for instruction sets [22], we decorate load and store instructions with a special pair of tags. A store that is tagged (cnd) may only be executed, if and only if there is a load tagged $(rsrv)$ that is program-ordered before it, without any other such load or store in-between, and the memory location accessed by the load was not manipulated since it was read. If the store also accesses the same location as the load, this results in a read-modify-write pair (see example 4). For instance, AARCH64 defines a fallback scenario in which the store fails and the thread receives a respective status value in its register state, allowing retrys until the operation succeeds.

This paradigm is usually abstracted by higher-level programming languages. Since the programs in chapter 6 were written in C, we will use a popular primitive for *lock-free* algorithms, the atomic CAS procedure. It accepts an address $k \in D$, an expectation $x \in D$ and an output $y \in D$. y is stored into memory at k only if it overwrites x.

Figure 1.4.: program fragment performing a single compare-and-swap

$$\texttt{ld}\ r \leftarrow k\ (rsrv)$$
$$\texttt{if}\ r = x\ \texttt{st}\ y \rightarrow k\ (cnd)$$

Processor architectures just have to implement this type of *sequential atomicity* to allow *spinlocks*, *lock-free* linked lists and more to be safe under *sequential consistency* (see example 7). Meaning, as long as any instruction including CAS is atomic, and any store immediately becomes a visible side effect for any other thread, programs like the one below will never fail their assertion.

Figure 1.5.: spinlock mutex accessed by two threads

0: `ld` $r \leftarrow m$ $(rsrv)$	0: `ld` $r \leftarrow m$ $(rsrv)$
`if` $r = 0$ `st` $1 \rightarrow m$ (cnd)	`if` $r = 0$ `st` $1 \rightarrow m$ (cnd)
`if` $r \neq 0$ `goto` 0	`if` $r \neq 0$ `goto` 0
`st` $1 \rightarrow y$	`st` $2 \rightarrow y$
`ld` $s \leftarrow y$	`ld` $s \leftarrow y$
`if` $s \neq 1$ `fail`	`if` $s \neq 2$ `fail`
`st` $0 \rightarrow m$	`st` $0 \rightarrow m$

Programming Languages - C [28] additionally describes a weak kind of CAS that is allowed to fail spontaneously. We assume the weak operation was introduced with Load-reserved/Store-conditional in mind, where a conditional store also fails if intervened by another thread storing the expected value at the reserved location.

1.3. Control Flow Analysis

Given a plethora of possible executions of a program one has to mind that instructions can be executed an arbitrary number of times, including zero times. If the program is acyclic, each instruction is optional if the structure allows to skip it when the runtime meets conditions contained by the program. Usually this is expressed by `if` control statements or *conditional jump* instructions.

Definition 2. *The control graph of a DT-thread* $\langle R, I\rangle$ *is a directed graph* $\langle Q, \rightarrow\rangle$*, consisting of control states* $Q := \{0 \dots |I|\}$ *and edges* $\rightarrow :=\rightarrow_f \cup \rightarrow_t$ *consisting of*

$$\rightarrow_f := \{\langle q, q+1\rangle \mid q < |I|, \neg\exists n, t.\, I(q) = \langle \top, \texttt{goto}\ n, t\rangle\} \tag{1.1}$$

$$\rightarrow_t := \{\langle q, \mathrm{dest}(q)\rangle \mid q < |I|\} \tag{1.2}$$

Hereby, dest *denotes the destination of the instruction if executed:*

$$\mathrm{dest}(q) := \begin{cases} n & \text{if } I(q) = \langle \varphi, \texttt{goto}\ n, t\rangle \text{ and } n < |I| \\ |I| & \text{if } I(q) = \langle \varphi, \texttt{goto}\ n, t\rangle \text{ and } |I| \leq n \\ |I| & \text{if } I(q) = \langle \varphi, \texttt{fail}, t\rangle \\ q+1 & \text{otherwise} \end{cases}$$

All non-halting control states $q \in Q \setminus \{|I|\}$ are associated with an instruction $I(q)$. Note that each control state yields at least one and at most two outgoing edges in the control graph. Only those states associated with conditional jump instructions may yield two edges. Only the halting state $|I|$ has no outgoing edge.

Definition 3. *Let $\langle Q, \rightarrow \rangle$ be a control graph. A finite sequence $p \in Q^*$ of control states is a (finite)* control path *if and only if it satisfies all following conditions:*

$$\begin{aligned} q_0 &= 0 \\ \forall i \in \{0 \dots |p| - 2\}.\, q_i &\rightarrow q_{i+1} \\ q_{|q|-1} &= |I| \end{aligned}$$

Reasoning over a collection of executions will require some intermediate results. Let us define an order between events of a program on the basis of control flow. It relates all those events whose control flows are implied. This notion will be useful later in section 4.3.

Definition 4. *Let $x, y \in Q$ be two instructions of the same thread. x* includes *y, written as $x \gg y$, if and only if all control paths passing through x also pass through y.*

Exclusion is an irreflexive symmetric relation between control states, proposing that no control path contains both. This relation is strongly constrained by cycles of the control graph and therefore benefits later when assuming a bounded setting beginning from section 1.5.

Definition 5. *Let $x, y \in Q$ be two instructions of the same thread. x* excludes *y, written as $x \otimes y$, if and only if there is no control path passing through both x and y.*

Lemma 1. *Given control states $x, y, z \in Q$, if $x \gg y$ and $y \otimes z$, then $x \otimes z$.*

Proof. Follows directly from definitions 4 and 5. □

1.4. Dependency

Computer architectures are limited in their capacity for reordering by a dependency order. The axiomatic model will define three distinct types of dependencies between operations: control, data and address dependency, which will be discussed in detail in this section.

These relations originate alone from structural features of the program and are not to be eliminated even if the result of the dependent operation is invariant to the required value, as some artificial dependencies are intentionally placed [1]. However some optimisation techniques may result in removal of said artificial dependencies, unintentionally adding more possibilities for unsynchronised inter-thread communication [27].

As a preparation, We use the simplicity of our definition of a program (definition 1) to build the transitive dependency chain. This relation is comparable to the internal dependency relation `idd` of Dartagnan. We will relate instructions to the involved registers with $\delta_{\text{provide}}, \delta_{\text{require}}, \delta_{\text{guard}}, \delta_{\text{address}}, \delta_{\text{value}} \subseteq \mathcal{I}(D, R, \mathcal{T}) \times R$. Example 13 will demonstrate how to compute an over-approximation of the set of direct dependencies.

Definition 6. *Let $i = \langle \varphi, s, t \rangle \in \mathcal{I}(D, R, \mathcal{T})$ be an instruction, $r, x, y \in R$ registers and $f \in D^{D \times D}$ a function.*

$$\begin{aligned}\delta_{provide} :=& \{\langle\langle \varphi, v \leftarrow f(x, y), t\rangle, v\rangle \mid \varphi, v, x, y \in R \wedge t \subseteq \mathcal{T}\} \\ &\cup \{\langle\langle \varphi, \texttt{ld}\ v \leftarrow k, t\rangle, v\rangle \mid \varphi, v, k \in R \wedge t \subseteq \mathcal{T}\}\end{aligned}$$

Let $p, q \in Q \setminus \{|I|\}$ be control states in a control graph $\langle Q, \rightarrow \rangle$, and $\delta \subseteq \mathcal{I}(D, R, \mathcal{T}) \times R$ a relation between instructions and registers.

*p is a δ-*dependency candidate *of q, if and only if q is reachable from p, $I(p)\ \delta_{provide}\ r$ and $I(q)\ \delta\ r$.*

p is a direct δ-dependency *of q if and only if p is a δ-dependency candidate of q and there is no other δ-dependency candidate o of q, that is reachable from and implied by p.*

$$\begin{aligned}\delta_{require} :=& \{\langle\langle \varphi, y \leftarrow f(r, x), t\rangle, r\rangle \mid x, y \in R, f \in D^{D \times D}\} \\ &\cup \{\langle\langle \varphi, y \leftarrow f(x, r), t\rangle, r\rangle \mid x, y \in R, f \in D^{D \times D}\}\end{aligned}$$

*p is a δ-*dependency *of q if and only if p is a direct δ-dependency of q or there is some control state $o \in Q$ where p is a $\delta_{require}$-dependency of o and o is a δ-dependency of q.*

1.4.1. Control Dependency

Since control graphs have just recently been introduced (definition 2), We start with events on control path fragments guarded by a conditional jump instruction depending on data read from memory. Branching on a thread may impose stronger constraints on the preservation of the program order, as demonstrated by architectures like ARM and Power [22].

Definition 7 (Control dependency)**.** *Let $g \in Q \setminus \{|I|\}$ be a control state associated with the instruction $\langle \varphi, s, t \rangle$. g* guards *another state $q \in Q$, if and only if $\varphi \in R$, $s = \texttt{goto}\ n$ and q is reachable from g.*

$$\delta_{guard} := \{\langle\langle \varphi, s, t\rangle, \varphi\rangle \mid \varphi \in R \wedge s \in S(D, R) \wedge t \subseteq T\}$$

A control state $p \in Q$ is a control dependency of q if and only if p is associated with a load statement and there is g guarding q where p is a δ_{guard}-dependency of g.

Example 2. *The following thread exhibits two control dependencies, $\langle 0 \rangle 2$ and $\langle 0 \rangle 3$ because 0 is a δ_{guard}-dependency of 1 and 1 guards 2 and 3.*

$$\begin{aligned}&0\colon \texttt{ld}\ a \leftarrow 0 \\ &1\colon \texttt{if}\ a = 0\ \texttt{goto}\ 3 \\ &2\colon \texttt{st}\ 1 \rightarrow 1 \\ &3\colon \texttt{st}\ 2 \rightarrow 1\end{aligned}$$

Strict models will require all dependency loads to be satisfied before the branch is available for processing. Weaker models like ARM and Power use pipelining to anticipate instructions whose execution is yet unclear in the context before a branching event. Those instructions then might already influence the cache and be saturated with values from the cache, before the processing unit determines which path to follow. Of course this will require the processor to roll back any side effect it might have done in case that the other branch is chosen, which will limit the possibilities for side effect of this feature. Most important limit would be that side effects of provisional operations should not be visible to other processing units.

Note that this notion of 'guarding' does not always apply. The Linux kernel [4] follows a more restrictive definition, additionally requiring that the dependent must not be included by one of both successor instructions of the guard (see definition 4). This means that the dependency spans until both branches rejoin. Fortunately, since the kernel imposes a weaker model than your usual underlying architecture, this should never lead to new inconsistencies after compilation. But architecture-modular verifiers should keep in mind that this other major definition exists.

1.4.2. Data Dependency

Dataflow permits combining values together that were fetched from shared memory, resulting in new values that can be sent back. This usually entails that all read accesses, that contribute to the value about to be written, have to be finished beforehand. In order for a memory model to recognise this property, its dependencies have to be provided by the program.

Definition 8 (Data dependency)**.**

$$\delta_{data} := \{\langle\langle \varphi, s, t\rangle, r\rangle \in \mathcal{J}(D, R, \mathcal{T}) \times R \mid \exists x.\, s = \texttt{st}\ r \to x\}$$

A control state $p \in Q$ is a data-dependency *of another control state $q \in Q$ if and only if p is a δ_{data}-dependency of q.*

1.4.3. Address Dependency

Even the destination of a memory event does not have to be static and can be computed using communicated data.

Definition 9 (Address dependency)**.**

$$\delta_{address} := \{\langle\langle \varphi, s, t\rangle, r\rangle \in \mathcal{J}(D, R, \mathcal{T}) \times R \mid \exists x.\, s \in \{\texttt{ld}\ x \leftarrow r, \texttt{st}\ x \to r\}\}$$

A control state $p \in Q$ is an address-dependency *of another control state $q \in Q$ if and only if p is a $\delta_{address}$-dependency of q.*

1.5. Boundedness

This section deals with the restriction of programs with bounded-length executions. Together with the set of relevant traces dramatically shrinking, verifiers gain several opportunities to abstract some components of the program. We will see that because of the bound, a symbolic representation of the set of executions becomes computable.

Definition 10 (Boundedness)**.** *A program $\mathcal{P}$ is called* bounded*, if and only if all control graphs of threads of $\mathcal{P}$ are acyclic.*

The first advantage concerns the set of valid addresses. In definition 1 there was no concept of a heap or a stack; the entire memory space was assumed to be already initialised and accessible. This is inherently problematic for verifiers, not able to handle such a sizable state space. As instructions are limited to one address, the set of accessible memory addresses in a bounded program becomes finite, as well, rendering a form of pointer analysis feasible (see section 4.1).

Definition 11 (Location)**.** *Bounded programs are associated with a finite $\mathbb{N}$-labeled set L of locations, such that all threads may potentially access them: $L \subseteq R$ for all register sets R of threads of $\mathcal{P}$.*

The label of L associates the location $l \in L$ with its array size $|l|$, thus defining an arbitrary distribution of blocks of accessible addresses. Therefore, given some placement $p\colon L \to D$ appended to a computation, the set of *valid* addresses shall be $\{p(l)+i \mid l \in L \wedge i \in \{0 \dots |l|-1\}\} \subseteq D$. Section 4.1 will describe how this notion can be used to abstract the otherwise unbounded address space into a closed and manageable domain.

Definition 12 (Program events)**.** *A bounded program $\mathcal{P}$ yields a finite set $\mathcal{E}$ of* events*.*

$$
\begin{aligned}
\mathcal{E} &:= \mathcal{E}_I + \mathcal{E}_M \\
\mathcal{E}_I &:= \{\langle l, i\rangle \mid l \in L \wedge i \in \{0 \dots |l|-1\}\} \\
\mathcal{E}_M &:= \mathcal{E}_R + \mathcal{E}_W \\
\mathcal{E}_R &:= \{\langle T, i\rangle \mid \mathcal{P}(T) = \langle R, I\rangle \wedge I(i) = \langle \varphi, \texttt{ld}\ k \leftarrow v, t\rangle\} \\
\mathcal{E}_W &:= \{\langle T, i\rangle \mid \mathcal{P}(T) = \langle R, I\rangle \wedge I(i) = \langle \varphi, \texttt{st}\ k \rightarrow v, t\rangle\} \\
\mathcal{E}' &:= \mathcal{E} + \mathcal{E}_L + \mathcal{E}_B + \mathcal{E}_A \\
\mathcal{E}_L &:= \{\langle T\rangle i \mid \mathcal{P}(T) = \langle R, I\rangle \wedge I(i) = \langle \varphi, r \leftarrow f(x,y), t\rangle\} \\
\mathcal{E}_B &:= \{\langle T\rangle i \mid \mathcal{P}(T) = \langle R, I\rangle \wedge I(i) = \langle \varphi, \texttt{goto}\ n, t\rangle\} \\
\mathcal{E}_A &:= \{\langle T\rangle i \mid \mathcal{P}(T) = \langle R, I\rangle \wedge I(i) = \langle \varphi, \texttt{fail}, t\rangle\}
\end{aligned}
$$

For each tag $t \in T$, $\mathcal{E}_t \subseteq \mathcal{E}_R + \mathcal{E}_W$ shall denote the set of t-tagged events.

An event $x \in \mathcal{E}'$ is unconditional*, if and only if $x \in \mathcal{E}_I$ is an initialisation or the associated instruction $\langle \top, s, t\rangle$ has no condition.*

A visible event is either an initialisation, a read event or a write event. Other events include local evaluations, branchings and assertions. Note that the events in $\mathcal{E} \setminus \mathcal{E}'$ will not be visible to the model in chapter 2, but have to be accounted in the encoding section 3.2. Furthermore, all but the halting control states are covered by events (see definition 2), lifting the definitions of inclusion, exclusion, as well as dependency to the domain of events.

Instructions can communicate intra-thread messages through registers. Different to memory locations, registers do not share owners, and the donators of a value read from a register can always be determined by backtracking the sequence of executed events in reverse program order. Overwriting an old value of a register does not allow any later event to read the old value from it. In a non-branching program, the donors of each register access are determined by the program. Only if two branches join does the determinism of the donor require more insight on the execution.

In a bounded program, each register state has a finite set of predecessors. This means that the program does not need to overwrite register names with new values, and could directly refer to the previous operations that provide the respective value. In section 1.4, this idea was formalised into a memory-unaware static analysis that will build the basis of the pointer analysis in section 4.1.

Definition 13 (Trace)**.** *Let $\mathcal{T}$ be a finite set of event tags. A $\mathcal{T}$-trace τ consists of*

- *a finite set* $\mathbb{E}_I$ *of* initialisations
- *a distinct finite set* $\mathbb{E}_R$ *of executed* read events
- *another distinct finite set* $\mathbb{E}_W$ *of executed* write events
- *a location map* $\mathrm{location} \colon \mathbb{E}_W \to \mathbb{E}_I$
- *a relation* $\mathrm{tag} \subseteq \mathbb{E}_M \times \mathcal{T}$
- *program order relations po, addr, ctrl, data* $\subseteq \mathbb{E}_M \times \mathbb{E}_M$
- *a 'read-from' map* $\mathrm{rf} \colon \mathbb{E}_R \to \mathbb{E}_W + \mathbb{E}_I$

The set $\mathbb{E}_M := \mathbb{E}_R + \mathbb{E}_W$ *of* memory events *contains all events issued by a thread. The set* $\mathbb{E} := \mathbb{E}_I + \mathbb{E}_M$ *contains all events of the trace. The set* $\mathbb{E}_t := \{x \mid x \,\mathrm{tag}\, t\}$ *contains all t-tagged memory events. The location map is extended to all events with following definitions:*

$$\begin{aligned} \mathrm{location}(i) &:= i && \text{for } i \in \mathbb{E}_I \\ \mathrm{location}(r) &:= \mathrm{location}(\mathrm{rf}(r)) && \text{for } r \in \mathbb{E}_R \end{aligned}$$

The program order is a union of total orders. The remaining relations are sparse subsets of po with read events on the domain side and data is also restricted to write

events on its range side.

$$\begin{aligned} po \cap id &= \emptyset \\ po \,;\, po &\subseteq po \\ (po \,;\, po^{-1}) \cup (po^{-1} \,;\, po) &\subseteq po \cup po^{-1} \\ addr \cup ctrl \cup data &\subseteq po \cap R\_ \\ data &\subseteq \_W \end{aligned}$$

Due to memory models relating pairs of events of a trace, literature prefers to illustrate traces as labeled graphs with events as nodes, presumably to highlight cycles or the absence of such. For instance, the trace targeted by the program in figure 1 would look like below.

$$\mathrm{Wx} \xrightarrow{\text{lwsync}} \mathrm{Wy} \xrightarrow{\text{rf}} \mathrm{Ry} \xrightarrow{\text{data}} \mathrm{Wz} \xrightarrow{\text{rfi}} \mathrm{Rz} \xrightarrow{\text{ctrlisync}} \mathrm{Rx}$$

Definition 14 (Computation)**.** *Let $\mathcal{P}$ be a bounded $D\mathcal{T}$-program and τ a trace. A family of control paths $p_t \colon \{0 \dots |I|\} \to (\{\bot\} + D^R)$ for $t \in \{0 \dots |\mathcal{P}| - 1\}$ is called a* computation *of $\mathcal{P}$ with trace τ, if and only if it satisfies all of the following conditions:*

- *Tags and types coincide $\mathbb{E}_I \subseteq \mathcal{E}_I$, $\mathbb{E}_R \subseteq \mathcal{E}_R$, $\mathbb{E}_W \subseteq \mathcal{E}_W$ and $\mathbb{E}_t \subseteq \mathcal{E}_t$ for $t \in \mathcal{T}$.*
- *Events in τ imply control flow through their associated instruction. $p_t(i)$ denotes the register state immediately before the execution of $\langle t, i \rangle$ and maps exactly all unreached events to $\bot$. Manipulation of register states shall follow the expected rules.*
- *The equivalence relation of accessing the same address is accurately supported.*
- *Each read event r updates its associated register's value in p_t with the same value that* $\mathrm{rf}(r)$ *has taken from its register state.*
- *Program order is consistent with the instruction sequence and each dependency relationship is supported by a chain of direct dependencies of the dependency analysis (see section 1.3).*

The computation is further a witness *for $\mathcal{P}$, if and only if some* `fail` *statement is executed.*

A $\mathcal{T}$-trace τ originates from a bounded $D\mathcal{T}$-program $\mathcal{P}$ if and only if there is some computation of $\mathcal{P}$ with τ.

2. Memory

Now that we have worked out the kind of programs that we will examine, along with what properties are queried, it is time to define the kind of model we are testing against. Although there are languages for operational models out there, as briefly discussed in section 2.2, this thesis concentrates on the axiomatic approach.

We will encounter a few examples of in-use memory models, and some extremely weak models with a bare minimum of guarantees that turn out to be implied by the real-world examples, hinting at the existence of a partial order over models as studied by Alglave [1] and Ponce de León et al. [26].

A *litmus test* is a minimal program equipped with an embedded specification that aims to exhibit one particular trace that both originates from the program and witnesses the specification. When examined under a subject model or architecture, a verifier will decide either that trace to be consistent with the model or that no consistent witness exists. This type of program is used extensively by architecture documentations to showcase behavior unexpected to programmers, and which program patterns act as expected.

2.1. Axiomatic Semantics of the Model

The research in this field needs a modelling language that is general enough to dive deep into weakness properties already discovered by the industry and properties probably exploitable in the future, while at the same time keeping its computability. Alglave, Cousot, and Maranget [2] claim to already have found such a language, which they implemented into the `herd` verification tool.

The `cat` language uses features of the functional programming paradigm, providing construction of more complex objects by using a type system with function types, set types and more. For example, the widely used *memory order* co (see example 6) is defined in a standard library header `cos.cat` using the primitives `linearisations` and `partition`.

Memory models are given as a set of constraints for binary relationships over events the threads of a computation commit. Candidate traces of the program can then be evaluated with respect to the model, formulating the problem of *memory model checking* (definition 17): Given a memory model $\mathcal{M}$ and a trace τ, is τ consistent under $\mathcal{M}$? The constraint types of `cat` include *emptiness*, *irreflexivity* and *acyclicity*. A verifier for robustness (see definition 18) might also be interested in their respective negative forms in order to characterize traces that are consistent under one model, but inconsistent under another [26].

For simplicity of the definition below, we reduce those three types to one type. Note

Figure 2.1.: herd's definition of memory order

```
let fold f =
  let rec fold_rec (es,y) = match es with
  || {} -> y
  || e ++ es -> fold_rec (es,f (e,y))
  end in fold_rec
let map f = fun S -> fold (fun (e,y) -> f e ++ y) (S,{})
let rec cross S = match S with
  || {} -> { 0 } (* 0 is the empty relation *)
  || S ++ R -> let s = cross R in fold (fun (e,r) -> map (fun t -> e | t) s | r) (S,{})
  end
let co0 = loc & (IW*(W\IW))
let co from cross (let f w = linearisations(w,co0) in map f (partition(W)))
```

that especially for acyclicity, such a reduction is not always preferable, as there might exist better approaches when the goal is to express the constraint in SMT, as discussed by Gebser, Janhunen, and Rintanen [14].

Lemma 2. *A binary relation R is irreflexive if and only if $R \cap id$ is empty.*

Lemma 3. *A binary relation R is acyclic if and only if R^+ is irreflexive, where R^+ denotes the smallest relation that satisfies $R^+ \supseteq R \cup (R^+ ; R^+)$.*

Dartagnan implements a strongly reduced cat-like dialect, that omits the functional fragment and maintains the declarative fragment [24]. co is built into the verifier, as it specialises for cache-consistent models, including but not limited to Sequential Consistency and Total Store Order, ARM and Power [22, 3], and the Linux kernel [4]. This thesis uses this dialect as a prototype for memory models, well-aware of its reduced expressiveness in relation to cat.

The modelling language still bases on a particular trace to be verified as consistent with the model. Its expressions are symbolic binary relations over the finite set of executed memory events. The actual location memory events access is not important to the model, as we assume that all addresses are equitable in visibility, accessibility and value capacity. However, we have to keep the information, whether two different events access a common address, and whether two events were issued by the same thread.

An axiomatic memory consistency model imposes a stratified fixed point problem over terms of binary relations. 'Stratified' means that although non-monotonic semantics of expressions are syntactically allowed using a set difference operator, there must always be a linear order in which variables may be refined in an iterative algorithm, such that monotonicity is preserved. In other words, a stratified fixed point problem is a finite sequence of monotonic fixed point problems, where each element is bounded by the solution of its predecessors. As Kleene's recursion theorem states that monotonic fixed point problems have a computable solution, this effectively also applies to such an entire sequence.

Definition 15 (Memory model)**.** *Let $\mathcal{T}$ be a finite set of tags. An axiomatic memory model $\mathcal{M}$ consists of*

- *a finite set $\mathcal{R}$ of variables for binary relations.*
- *a term mapping* $\mathrm{def}\colon \mathcal{R} \to \{\bot\} + C + \mathrm{terms}(\mathcal{R})$
- *a set $\mathcal{C} \subseteq \mathcal{R}$ of constrained relations.*

A variable $R \in \mathcal{R}$ is free*, if and only if* $\mathrm{def}(R) = \bot$.
The set of constants C is defined as

$$C := \{\_\_, id, po, addr, ctrl, data, rf, loc\} + \bigcup\{\{t\_, \_t\} \mid t \in \mathcal{T} \cup \{I, R, W\}\}$$

The set of terms $\mathrm{terms}(\mathcal{R})$ *includes exactly*

- *$S \cup T$ as the union of $S \in \mathcal{R}$ and $T \in \mathcal{R}$.*
- *$S \cap T$ as the intersection of $S \in \mathcal{R}$ and $T \in \mathcal{R}$.*
- *$S \setminus T$ as the difference of $T \in \mathcal{R}$ from $S \in \mathcal{R}$.*
- *$S\,;T$ as the post-composition of $S \in \mathcal{R}$ with $T \in \mathcal{R}$.*
- *S^{-1} as the inverse of $S \in \mathcal{R}$.*

Definition 16. *The fixed point problem imposed by the memory model $\mathcal{M}$, the trace τ and binding $\beta\colon \mathcal{R} \to 2^{\mathbb{E}\times\mathbb{E}}$ consists of a family of variables $X := (X_R)_{R\in\mathcal{R}}$ and a rule $X_R = [\![\mathrm{def}(R)]\!]_X^{\beta(R)}$ for each $R \in \mathcal{R}$.*
Let $t \in \mathcal{T} + \{I, R, W\}$, $P \in \{po, addr, ctrl, data\}$ and $R, S, T \in \mathcal{R}$.

$$\begin{aligned}
[\![\bot]\!]_X^R &:= R \\
[\![\emptyset]\!]_X^R &:= \emptyset \\
[\![__]\!]_X^R &:= \mathbb{E} \times \mathbb{E} \\
[\![id]\!]_X^R &:= \{\langle x, x\rangle \mid x \in \mathbb{E}\} \\
[\![P]\!]_X^R &:= P \\
[\![rf]\!]_X^R &:= \{\langle \mathrm{rf}(r), r\rangle \mid r \in \mathbb{E}_R\} \\
[\![loc]\!]_X^R &:= \{\langle x, y\rangle \mid \mathrm{location}(x) = \mathrm{location}(y)\} \\
[\![t_]\!]_X^R &:= \mathbb{E}_t \times \mathbb{E} \\
[\![_t]\!]_X^R &:= \mathbb{E} \times \mathbb{E}_t \\
[\![S \cup T]\!]_X^R &:= X_S \cup X_T \\
[\![S \cap T]\!]_X^R &:= X_S \cap X_T \\
[\![S \setminus T]\!]_X^R &:= X_S \setminus X_T \\
[\![S\,;T]\!]_X^R &:= \{\langle x, z\rangle \mid \exists y.\, \langle x, y\rangle \in X_S \wedge \langle y, z\rangle \in X_T\} \\
[\![S^{-1}]\!]_X^R &:= \{\langle x, y\rangle \mid \langle y, x\rangle \in X_S\}
\end{aligned}$$

Definition 17 (Consistency)**.** *A trace τ is* consistent *under an axiomatic memory model $\mathcal{M}$, written as $\tau \vDash \mathcal{M}$, if and only if there is some binding $\beta \in \left(2^{\mathbb{E}\times\mathbb{E}}\right)^{\mathcal{R}}$ for free variables of $\mathcal{M}$ that evaluates all constrained relations to the empty set. Hereby let $X \colon \mathcal{R} \to 2^{\mathbb{E}\times\mathbb{E}}$ denote the associated least fixed point.*

$$\exists \beta.\, \forall c \in \mathcal{C}.\, [\![\tau]\!]^{\beta}_{\mathcal{M}}(c) = \emptyset$$

As for definition 1, the models and model fragments in this thesis are abbreviated and inlined: We introduce a new relation $R \in \mathcal{R}$ associated with a term by $R := T$ and specify a constrained relation $S \in \mathcal{C}$ by $\emptyset = S$.

Example 3. *All models must satisfy those conditions: Only write events can send values to read events (1), and only if both address the same location (2). Each read event can communicate with at most one (3) and at least one write event (4). The internal relation is an equivalence relation, thus reflexive (5), transitive (6) and symmetrical (7). The external relation contains all other relationships (8) and (9).*

$$\begin{aligned}
int :=& id \cup po \cup po^{-1} \\
ext :=& ___ \setminus int \\
\emptyset =& (po \setminus id) \cup (po \,;\, po \setminus po) \cup \left(po \setminus po^{-1}\right) && \text{po is a union of total orders} \\
\emptyset =& rf \setminus (W_ \cap _R \cap loc) && \text{rf is typed accordingly} \\
\emptyset =& \left(rf^{-1} \,;\, rf\right) \setminus id && \text{reads read from at least one write} \\
\emptyset =& _R \setminus (___ \,;\, rf) && \text{reads read from at most one write} \\
\emptyset =& id \setminus loc \cup loc \,;\, loc \setminus loc \cup int \setminus int^{-1} && \text{loc is an equivalence relation}
\end{aligned}$$

Example 4. *Atomic* read-modify-write *operations are declared in the program. Of course their existence in a program requires that no execution shall feature an interference to their atomicity. Without a notion of time, the axiom could be stated as follows: If in a trace, two operations atomically read and write to the same location, they cannot read from the same event, since one of them will have happened beforehand and overwritten that value.*

$$\begin{aligned}
rmw_R :=& (rsrv)_ \\
rmw_W :=& _\ (cnd) \\
rmw :=& (rmw_R \cap po \cap rmw_W) \setminus (po \,;\, (rmw_R \cap po) \cup (po \cap rmw_W) \,;\, po) \\
\emptyset =& \left(rf^{-1} \,;\, rf\right) \cap ((rmw \cap loc) \,;\, ___) \setminus id
\end{aligned}$$

Example 5. Local Consistency *is the weakest constraint there is, requiring that each thread perceives its own side effects in the order they were issued.*

$$\begin{aligned}
\emptyset &= rf \cap po^{-1} \\
\emptyset &= rf \cap ((po \cap loc \cap W_ \cap _W) \,;\, po) \\
\emptyset &= rf \cap ((loc \cap I_ \cap _W) \,;\, po)
\end{aligned}$$

Note that the rules explicitly disallow internal communication with other writes than the most recent.

Definition 18. *Let $\mathcal{M}$ be a memory model. A memory model $\mathcal{N}$ is* robust *against $\mathcal{M}$, written as $\mathcal{N} \vDash \mathcal{M}$ if and only if $\forall \tau. (\tau \vDash \mathcal{N}) \Rightarrow (\tau \vDash \mathcal{M})$*

Definition 19 (Local consistency)**.** *A memory model $\mathcal{M}$ is* locally-consistent*, if and only if $\mathcal{M} \vDash$ Local Consistency*

Example 6. Cache Consistency *is one of the weakest practical models, requiring some total orderings of all accesses with common addresses. This is implied by most industrial processor architectures, including x86, ARM and Power.*

$$\emptyset = po^{-1} \cap (rf \cup co \cup (co\,;\,rf) \cup fr \cup (fr\,;\,rf))$$

To refer to the total ordering of writes in a trace, we define the *memory order* or coherence relation co, with expected properties described below. Cache consistent models make use not only of co, but also of the *from-read* relation fr, effectively lifting the trichotomy of the total order into the realm of read events: Each read-write pair accessing the same location, is either in rf, (co ; rf), or fr^{-1}.

$\text{fr} := \text{rf}^{-1}\,;\,\text{co}$	
$\emptyset = \text{co} \cap \text{id}$	irreflexivity
$\emptyset = (\text{co}\,;\,\text{co}) \setminus \text{co}$	transitivity
$\emptyset = \text{co} \cap \text{co}^{-1}$	asymmetry
$\emptyset = \text{co} \setminus (\text{loc} \cap W\_ \cap \_W)$	relating writes on common addresses
$\emptyset = (\text{loc} \cap W\_ \cap \_W) \setminus (\text{id} \cup \text{co} \cup \text{co}^{-1})$	totality
$\emptyset = \text{co} \cap \_I$	initialisations are minimal

Definition 20 (Cache consistency)**.** *A memory model $\mathcal{M}$ is* cache-consistent*, if and only if $\mathcal{M} \vDash$ CacheConsistency*

Example 7 (SC)**.** Sequential consistency *is a model where all threads share a common view on the memory: all stores are immediately visible to all other threads and there is a centralised memory device whose state is visible to all threads.*

The popular definition from Lamport [20]: [...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

We express that by requiring the graph made of edges in po, co, rf and the fr to be acyclic.

$$hb := (hb\,;\,hb) \cup po \cup co \cup rf \cup fr$$
$$\emptyset = hb \cap id$$

Example 8 (TSO)**.** Total Store Ordering *is implemented by modern x86 processors, allowing each thread to keep an arbitrary amount of recent store operations in a queue before updating the shared memory. The order of store operations per thread stays and each write becomes visible for all other threads at the same time. However, each write becomes immediately visible to the issuing thread, thus ensuring sequential consistency with single-thread applications. Additionally, there is the* mfence *barrier, flushing all writes into the shared memory in the order they were issued. Its tag F is attached to the first memory event* after *the barrier, so that such a write event would possess* release *semantics. In fact, the model guarantees sequential consistency over all writes that are separated to both next and previous writes by a barrier.*

$$
\begin{aligned}
rfe &:= rf \cap ext \\
com &:= rfe \cup co \cup fr \\
ppo &:= (po \cap R\_) \cup (po \cap W\_ \cap \_W) \\
fence &:= (po \cap W\_ \cap \_\,(rsrv)) \cup (po \cap\ (cnd)\_ \cap R\_) \cup ((po \cap \_F)\ ;\ (po \cup id)) \\
hb &:= (hb\ ;\ hb) \cup com \cup ppo \cup fence \\
\emptyset &= id \cap hb
\end{aligned}
$$

Example 9 (AARCH64)**.** *Besides being cache-consistent and supporting atomicity, the AARCH64 memory model relies on three classes of memory barriers: The* Data Memory Barrier *and the* Data Synchronization Barrier*, although the latter being slightly weaker thread-locally, exhibit the same behavior in regards to the memory model, prohibiting reordering. The* Instruction Synchronization Barrier *flushes the instruction pipeline, effectively limiting the speculation described in section 1.4.1.*

$$
\begin{aligned}
rfi &:= rf \cap int \\
coi &:= co \cap int \\
obs &:= ext \cap (rf \cup co \cup fr) \\
dob_{addr} &:= addr \cup (addr\ ;\ po \cap \_W) \cup (addr\ ;\ rfi) \\
dob_{data} &:= data \cup (data\ ;\ rfi) \cup (data\ ;\ coi) \\
dob_{ctrl} &:= (ctrl \cap \_W) \cup (ctrl\ ;\ coi) \\
dob_{isb} &:= (ctrl \cup (addr\ ;\ po))\ ;\ (\ (isb)\_ \cap \_R \cap (id \cup po)) \\
aob &:= rmw \cup (rfi \cap\ (cnd)\_ \cap \_\,(acq)) \\
bob &:= (po \cap\ (acq)\_) \cup ((po \cap \_\,(rel))\ ;\ (id \cup coi)) \cup (po \cap\ (rel)\_ \cap \_\,(acq)) \\
bob_{dmb} &:= (po \cap \_\,(dmb))\ ;\ (id \cup po) \\
hb &:= (hb\ ;\ hb) \cup dob_{addr} \cup dob_{data} \cup dob_{ctrl} \cup dob_{isb} \cup aob \cup bob \cup bob_{dmb} \\
\emptyset &= id \cap hb
\end{aligned}
$$

Alglave, Maranget, and Tautschnig [3] provided a different model for the former series of ARM architectures, along with what we now use as the model of IBM Power. The actual `cat` files are appended to this thesis and can alternatively be fetched from the Dartagnan Project [9].

2.2. Operational Semantics of the Model

The issue with the axiomatic model is that it is not optimised for trace search but for model checking. One can use the program for enumerating traces, then filter those that are consistent with the model like Alglave, Maranget, and Tautschnig [3] do or like described in section 3.6. An operational model describes its subject as a transition system, consisting of a class of configurations together with a transition relation over them. Consistent executions correspond to the paths through the system.

Alglave, Maranget, and Tautschnig [3] defines an *intermediate machine* and compare it to a previously proposed *PLDI machine* from Sarkar et al. [32]. The intermediate machine is based on an axiomatic memory model with memory order, preserved program order, propagation order and happens-before relation and differentiates when a store is committed (becomes perceivable by some other thread) and when it reaches 'coherence point' (all threads agree on its position in the memory order), as well as when a read is satisfied (determined from which write it reads) and when it is committed (the thread inserts the value into its register state). Its configurations consist of a map from a finite set of memory events to their location in D, a memory order over a subset of writes and a map from the subset of satisfied reads to their write. Transitions either commit a new write, insert a committed write into the memory order, satisfy a new read with some committed write or commit a satisfied read.

Combining such a transition system with that of a program appears to be manageable. Although satisfying its requirements based on an arbitrary axiomatic model is a non-trivial task and remains yet to be automatised, it may provide useful heuristics for the case that the input model has a fitting form.

2.3. Reordering

Alglave, Maranget, and Tautschnig [3] use the term *reordering*. It basically refers to the program order which is served as a linearisation of instructions given by the program, as explained in section 1.1. Instructions are fed to a processing unit in program order, but their events do not have to follow it when committed or becoming visible to other threads. The axiomatic model allows expressing weaker forms for those situations. Kusano and Wang [18] distinguish four general cases:

In a *read-read-reordering* some thread issues two loads, but the latter returns earlier and was able to read from a store that was not yet visible at the time the first read accessed the memory.

read-write-reordering describes a situation where a thread issues a load and a store in that order, but the store operation becomes a visible side effect (to at least some other thread) before the load operation is resolved. This enables other threads to read from the store operation and afterwards store some value which is read from the earlier load operation.

When a *write-read-reordering* happens, some thread issues a write w and then a read r, but r returns before w becomes visible to other threads. This is common for TSO.

Finally, *write-write-reordering* concerns pairs of write operations of the same thread that are committed to the shared memory in the opposite order.

Reorderings have no direct correspondence with the axiomatic model, but some models are designed to describe a *preserved program order* [3], explicitly enumerating the pairs of program-ordered events that can never be reordered, may it be grounded on types or because of a barrier in-between.

Local consistency (definition 19) can be alternatively described as the requirement that reorderings of events on the same address do not happen.

2.4. Out of Thin Air

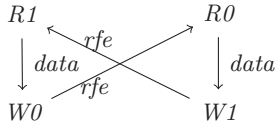
The term 'Out of Thin Air' refers to a special scenario where some thread reads a value from a write, whose value depends on the read. Since it is practically impossible to know a value before it is computed, all models should prohibit it. However, the axiomatic model described in this chapter does allow this until explicitly constrained.

It is a common understanding that committing a memory event is only possible as soon as its address value is completely computed. The same usually holds for the values being written, as architectures hardly support reserving cells for future values. In an architecture like this, a processing unit would promise to other threads to have written a certain value, allowing them to continue as if that value has indeed been written, while itself waiting for some of them to provide the resources to compute it.

Example 10.

$$R0\colon \texttt{ld}\ r \leftarrow x \qquad\qquad R1\colon \texttt{ld}\ r \leftarrow y$$
$$W0\colon \texttt{st}\ r \rightarrow y \qquad\qquad W1\colon \texttt{st}\ r \rightarrow x$$

This simple program consists of two threads, sharing memory at two addresses x and y. Each store is data-dependent on the previous load. Without explicitly forbidding behavior like this, the following execution is consistent under Local Consistency.



Forbidding out-of-thin-air values takes the form of an acyclicity constraint that includes at least rf, addr and data.

$$\begin{aligned} \text{hb} &:= (\text{hb} \,;\, \text{hb}) \cup \text{rf} \cup \text{addr} \cup \text{data} \\ \emptyset &= \text{hb} \cap \text{id} \end{aligned}$$

Considering control dependency: Events are usually not committed until the processing unit has secured a path that dictates their eventual issue. This means that all read events that contribute to the condition should have *happened before* any event in the branch becomes a visible side effect. When Dartagnan processes computations of a program under a memory model, the presence of any event in the trace (its satisfied execution variable), including its participation in relations of the memory model, implies its inclusion in a consistent control path (its satisfied control flow variable), and therefore mutually excludes any event in alternative paths.

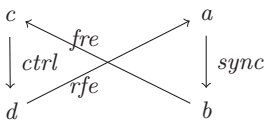
The ARM and Power [1] architectures impose a kind of weakness in their memory model as they implement an anticipating approach (see section 1.4.1). Instructions in a branch may be issued before their path is secured. Store instructions with evaluated address and value may alter the cache, as long as no synchronisation is performed before the path is decided. This allows rolling back, should the path finally be discarded. Loads can also communicate with the cache. Their result can be stored in an unused register and finalised as soon as their path is secured. However, as the branch condition awaits synchronisation with the shared memory, it may turn out more up-to-date than its dependents. In summary, write events must still await their control dependencies in a global *happens-before* relation, but read events may happen before their control dependencies.

Example 11 (MP+sync+ctrl)**.**

a: `st` $1 \to x$ $\qquad$ c: `ld` $r \leftarrow y$

b: `st` $1 \to y$ $(sync)$ $\qquad$ d: `if` $r = 1$ `ld` $s \leftarrow x$

In this program with a situation described above, the following trace has been observed on a Power processor [32].



3. Verification

This chapter refers to the bounded verifier Dartagnan [25, 27]. It aims to decide *bounded reachability* in-so-far that in its default behavior it tries to find executions of the input program that somehow reach a call passing 0 to `assert(int)`, like when an assertion placed by the programmer fails.

The tool specialises in concurrent programs and it is aware of memory models in-so-far as it accepts an axiomatic model in the `cat` file format [2]. It participates in the SVCOMP competition for software verification [8] as a verifier.

`Dat3M` was designed to analyse compiled machine code, preferably grounded on a platform-independent instruction set, but accepts it in a disassembled format, the verifier-oriented *Boogie* programming language [21]. This means that the input may have already undergone various forms of optimisation.

The program is unrolled a specified number of times so that it no longer exhibits unbounded executions. Each backward jump results in a copy of the program fragment beginning at the destination and ending at the jump itself, which is then being unrolled with a decreased bound. Therefore, the bound effectively describes a maximal recursion depth, or a maximal total number of duplicate instructions in an execution. This process happens explicitly, resulting in a data structure that is linear in the size of the program, but exponential in the size of the bound appended to the input.

As expected from the premise of boundedness, dynamic linking is not supported, as is an arbitrarily-sized initial state for the input program. Therefore, the to-be-analysed program code has to be completely present.

3.1. Satisfiablility

Instances of the problem tend to be expressible in a way that is intuitive for most people. SAT instances are simple as they consist only of variables and junctors. Using the conjunctive normal form, an SAT instance can be implemented by a finite set of clauses, each clause being a finite set of literals, and each literal being either a variable or its negation.

There are two major goals: Searching some model for the instance, thus proving that the instance is satisfiable, or otherwise returning a formal proof that there is no instance, meaning a finite sequence of lemmas where each lemma directly derives from the instance and previous lemmas. The Davis-Putnam-Logemann-Loveland algorithm (DPLL) is a sound and complete backtracking-based algorithm originally tailored for instances in conjunctive normal form, It is the most popular algorithm used today due to two properties: it supports the construction of both goals and it is extensible for

SMT.

DPLL accepts quantifier-free formulas in conjunctive normal form, and builds on partial maps from variables to a truth value. In each iteration of the *splitting* rule, the algorithm guesses a value for some variable. When an unsatisfied clause raises a *conflict*, the most recently guessed variable is *backtracked* and reassigned. To avoid this costly rule, clauses with only one literal trigger the *unit propagation* rule, specifying a value for the associated variable. Variables that appear only in positive or only in negative literals are *pure literals* and do not have to be guessed, as well.

Satisfiability Modulo Theories (SMT), according to Barrett and Tinelli [6], means the satisfiability problem extended to formulas of at least first-order logic with fixed interpretation of predicates. The term hints that theories can be loaded into a general-purpose solver like a module. De Moura and Bjørner [10] present `Z3` as such an implementation of a theory-modular DPLL, treating predicates as variables. Theory modules accept a conjunction of literals and try to detect theory-specific inconsistencies, raising a conflict for the current iteration if that is the case. The `SMT-Lib` project collects numerous defined theories [33].

Example 12. Integer difference logic*, as suggested by Gebser, Janhunen, and Rintanen [14], has order* 1*, no functions, but additional binary predicates* $k \leq x - y$ *for* $k \in \mathbb{Z}$*. Specialisations IDL(K) exist that allow just* $k \in K \subseteq \mathbb{Z}$*, including negation. For instance, IDL(*$\{1\}$*) adds just the predicate* $x < y$*.*

3.2. Encoding

In this section, we present a naïve encoding of a bounded program $\mathcal{P}$ with event set $\mathcal{E}$ under a memory model $\mathcal{M}$ with relations $\mathcal{R}$, term map def and constraint set $\mathcal{C} \subseteq \mathcal{R}$.

Each model of the formula shall correspond to a trace τ, together with a computation testifying that τ witnesses $\mathcal{P}$ and a binding testifying that τ is consistent with $\mathcal{M}$.

The encoding will make use of the *extended* event set $\mathcal{E}'$, as well as an extended relation set:

$$\mathcal{R}' := \mathcal{R} + \left\{ dep_{\text{require}}, dep_{\text{guard}}, dep_{\text{address}}, dep_{\text{value}} \right\}$$

The additional relation symbols are associated with δ_{require}, δ_{guard}, δ_{address} and δ_{value} from section 1.4. It should be sufficient to mention that the three dependency relations are factorised into dep^{+}_{require} ; dep_{x}. Note that besides this *extended* set of relations $\mathcal{R}'$, there is also the extended set of events $\mathcal{E}'$.

It uses boolean variables $x\ R\ y$ for all visible events $x, y \in \mathcal{E}$ and $R \in \mathcal{R}$, symbolising the binding β from definition 17. The additional relations $R \in R' \setminus R$ also yield boolean variables $x\ R\ y$ for all events $x, y \in \mathcal{E}'$. The transitivity of the dependency chain is expressed with variables $x\ dep^{+}_{\text{require}}\ y$ for $x, y \in \mathcal{E}'$.

$$\bigwedge_{x,z\in\mathcal{E}'} x\ dep^{+}_{\text{require}}\ z \Leftrightarrow x\ dep_{\text{require}}\ z \vee \bigvee_{y\in\mathcal{E}'} \left(x\ dep^{+}_{\text{require}}\ y \wedge y\ dep^{+}_{\text{require}}\ z\right) \tag{3.1}$$

$$\bigwedge_{\substack{\text{def}(R)=\text{addr}\\ x,z\in\mathcal{E}}} x\ R\ z \Leftrightarrow x\ dep_{\text{address}}\ z \vee \bigvee_{y\in\mathcal{E}'} \left(x\ dep^{+}_{\text{require}}\ y \wedge y\ dep_{\text{address}}\ z\right) \tag{3.2}$$

$$\bigwedge_{\substack{\text{def}(R)=\text{ctrl}\\ x,z\in\mathcal{E}'}} x\ R\ z \Leftrightarrow x\ dep_{\text{guard}}\ z \vee \bigvee_{y\in\mathcal{E}'} \left(x\ dep^{+}_{\text{require}}\ y \wedge y\ dep_{\text{guard}}\ z\right) \tag{3.3}$$

$$\bigwedge_{\substack{\text{def}(R)=\text{data}\\ x,z\in\mathcal{E}'}} x\ R\ z \Leftrightarrow x\ dep_{\text{value}}\ z \vee \bigvee_{y\in\mathcal{E}'} \left(x\ dep^{+}_{\text{require}}\ y \wedge y\ dep_{\text{value}}\ z\right) \tag{3.4}$$

3.3. Data Flow

Starting with the encoding of the memory, since $L \subseteq R$ for all thread's register sets, let us assume that pairs of threads only share registers in L, while other registers are well-distinguishable, and create λ_r for all registers to represent their initial value. Let $|l|$ denote the size of the location $l \in L$.

$$\bigwedge_{\substack{a,b\in L\\ a\neq b}} \lambda_a + |a| \leq \lambda_b \vee \lambda_b + |b| \leq \lambda_a \tag{3.5}$$

Being sensitive to the control flow (section 1.3) requires that events issued by a thread are optional. For each event $x \in \mathcal{E}'$, the variable χ_x shall symbolise that the modelled trace contains $x \in \mathbb{E}$. To simplify the rules of execution, boolean variables of the form γ_x shall denote that control flow passes through the instruction of x.

$$\bigwedge_{x\in\mathcal{E}'} \gamma_x \Leftrightarrow \begin{cases} \top & \text{if } x \in \mathcal{E}_I \\ \bigvee_{y\to_t x} \chi_y \vee \bigvee_{y\to_f x} (\gamma_y \wedge \neg\chi_y) & \text{otherwise} \end{cases} \tag{3.6}$$

The program order relation is predetermined by the control graph. As mentioned in section 1.5, we lift the defined relations over instructions of a common thread, especially the control graph and the dependency relations, into the domain of events. For example, an event $x \in \mathcal{E}'$ *reaches* another event $y \in \mathcal{E}'$, if and only if $x = \langle t, i\rangle$ and $y = \langle t, j\rangle$ for a common thread t and j is reachable from i in the associated control graph.

$$\bigwedge_{\text{def}(R)=\text{po}} x\ R\ y \Leftrightarrow \begin{cases} \chi_x \wedge \chi_y & \text{if } x \text{ reaches } y \\ \bot & \text{otherwise} \end{cases} \tag{3.7}$$

Concerning the data flow, we utilise the lifted dependency analysis of section 1.4. Register states are expressed by integer variables r_x for an event x and a register r of

the associated thread t and shall be bound to $p_t(x)(r)$ for the modelled computation p. The result value of a provider $x \in \mathcal{E}_L + \mathcal{E}_R$ is symbolised by an integer variable υ_x.

$$\bigwedge_{x \in \mathcal{E}_L \text{ evaluates } f(y,z)} \upsilon_x = f(y_x, z_x) \tag{3.8}$$

$$\bigwedge_{\substack{x \text{ provides } r \\ z \delta_d r}} x \; dep_d \; z \Rightarrow \upsilon_x = r_z \tag{3.9}$$

Conforming to definition 14, the most recent executed provider of a required register determines the respective state. If there is no such, the initial value is fetched instead.

$$\bigwedge_{x,z \in \mathcal{E}'} x \; dep_d \; z \Leftrightarrow \begin{cases} \chi_x \wedge \chi_y \wedge \neg \bigvee_{\substack{x \text{ reaches } y \\ y \text{ provides } r}} \chi_y & \text{if } x \text{ provides } r \wedge z \; \delta_d \; r \\ \bot & \text{otherwise} \end{cases} \tag{3.10}$$

$$\bigwedge_{z \delta_d r} r_z = \lambda_r \vee \bigvee_{y \text{ provides } r} \chi_y \tag{3.11}$$

We already mentioned that there is a difference between an instruction $x \in \mathcal{E}'$ passed through by the control path (γ_x) and that instruction being executed (χ_x), if it is equipped with a condition. Now that we have defined the register state, we can complete the specification for event execution.

$$\bigwedge_{x \in \mathcal{E}'} \chi_x \Leftrightarrow \gamma_x \wedge \begin{cases} \varphi_x \in \text{true} & \text{if } x \; \delta_{\text{guard}} \; \varphi \\ \top & \text{otherwise} \end{cases} \tag{3.12}$$

3.4. Communication

Let us add another set of variables $x \text{ rf } y$ for $x, y \in \mathcal{E}$, describing the read-from mapping of the modelled trace.

$$\bigwedge_{\substack{\text{def}(R)=\text{rf} \\ x,y \in E}} x \; R \; y \Leftrightarrow x \text{ rf } y \tag{3.13}$$

We demand totality and determinism of the modelled read-from map.

$$\bigwedge_{y \in \mathcal{E}_R} \chi_y \Rightarrow \bigvee_{x \in \mathcal{E}_W} x \text{ rf } y \tag{3.14}$$

$$\bigwedge_{x,y,z \in \mathcal{E}} \neg (y \text{ rf } x \wedge z \text{ rf } x) \tag{3.15}$$

We alias the address expression α_x of each memory event $x \in \mathcal{E}$ to either $\alpha_x := \lambda_l + i$, if $x = \langle l, i\rangle \in \mathcal{E}_I$ is an initialisation, or $\alpha_x := r_x$, if $x = \langle t, i\rangle$ is a load or store and $i\ \delta_{\text{address}}\ r$.

$$\bigwedge_{\substack{\text{def}(R)=\text{loc}\\ x,y\in\mathcal{E}}} x\ R\ y \Leftrightarrow \begin{cases} \chi_x \wedge \chi_y \wedge \alpha_x = \alpha_y & \text{if } x \in \mathcal{E} \wedge y \in \mathcal{E} \\ \bot & \text{otherwise} \end{cases} \tag{3.16}$$

$$\bigwedge_{x,y\in\mathcal{E}} x\ \text{rf}\ y \Rightarrow \begin{cases} \chi_x \wedge \chi_y \wedge \alpha_x = \alpha_y \wedge r_x = \upsilon_y & \text{if } x\ \delta_{\text{value}}\ r \wedge y \in \mathcal{E}_R \\ \bot & \text{otherwise} \end{cases} \tag{3.17}$$

Up to this point, we have effectively transferred definitions 13 and 14 into a proposition. Each trace τ originating from $\mathcal{P}$ corresponds to a model of the formula and vice-versa. To express our intention to test reachability of some error state, we simply propose that the models shall correspond with witnesses:

$$\bigvee_{x\in\mathcal{E}_F} \chi_x \tag{3.18}$$

3.5. Consistency

Compound relations follow simple rules that can easily be encoded.

$$\bigwedge_{\substack{\text{def}(R)=D\\ x,y\in\mathcal{E}}} x\ R\ y \Leftrightarrow [\![D]\!](x, y) \tag{3.19}$$

$$\begin{aligned}
[\![__]\!](x, y) &= \chi_x \wedge \chi_y \\
[\![\text{id}]\!](x, y) &= \begin{cases} \chi_x & \text{if } x = y \\ \bot & \text{otherwise} \end{cases} \\
[\![t_]\!](x, y) &= \begin{cases} \chi_x \wedge \chi_y & \text{if } x \in \mathcal{E}_t \\ \bot & \text{otherwise} \end{cases} \\
[\![_t]\!](x, y) &= \begin{cases} \chi_x \wedge \chi_y & \text{if } y \in \mathcal{E}_t \\ \bot & \text{otherwise} \end{cases} \\
[\![S^{-1}]\!](x, y) &= y\ S\ x \\
[\![S \cup T]\!](x, y) &= x\ S\ y \vee x\ T\ y \\
[\![S \cap T]\!](x, y) &= x\ S\ y \wedge x\ T\ y \\
[\![S \setminus T]\!](x, y) &= x\ S\ y \wedge \neg\,(x\ T\ y) \\
[\![S\ ;\ T]\!](x, z) &= \bigvee_{y\in\mathcal{E}} (x\ S\ y \wedge y\ T\ z)
\end{aligned}$$

Finally, the constrained relations are assumed to be empty:

$$\bigwedge_{\substack{R \in \mathcal{C} \\ x,y \in \mathcal{E}}} \neg (x \; R \; y) \tag{3.20}$$

3.6. Refinement

Beside a full reduction to SMT, where a verification instance is transformed to one formula, the verifier may divide its task into several components. For instance, one may extract the memory-aware part from section 3.5, thus keeping the dataflow-oriented formula $\varphi_{\mathcal{P}}$ from sections 3.3 and 3.4. The models of the remaining proposition will be executions of $\mathcal{P}$ that witness $\mathcal{S}$, but are not bound to be consistent to $\mathcal{M}$.

Once the SMT-solver finds such an execution, the memory model has to be consulted. This is an instance of *memory model checking* as described in definition 17 and exactly the kind of task that Alglave, Cousot, and Maranget [2] described in their `cat` specification.

Figure 3.1.: refinement algorithm

Let $R = \emptyset$
loop
    Guess a trace τ of $\varphi_{\mathcal{P}} \wedge \bigwedge R$.
    if unsatisfiable **then**
        $\mathcal{P}$ is $\mathcal{S}$-safe under $\mathcal{M}$. **return**
    else
        Model check τ under $\mathcal{M}$.
        if consistent **then**
            τ witnesses that $\mathcal{P}$ is not $\mathcal{S}$-safe under $\mathcal{M}$. **return**
        else
            Let T be an unsatisfiable core of τ.
            $R \leftarrow R \cup (\neg T)$
        end if
    end if
end loop

4. Analysis

With a working basis for an encoding of the verification problem into SMT in mind, this chapter identifies some useful properties of a concurrent program that may aid the reasoning process. As we have done in sections 1.3 and 1.4, one may attempt to *approximate* the desired property based on a more feasible decision procedure. Note how our shallow knowledge on the control flow lets us over-approximate the set of possible paths, as conditions may be non-trivially unsatisfiable or unfalsifiable. As a consequence, some dependencies may unnoticably exclude related dependents or be always overwritten, in retrospective resulting in encoded relationships that do not contribute to the solving process.

In the context of bounded verification, it seems natural to over-approximate the set of possible and relevant relationships that may appear in a fixed point of the memory model checking problem, as so many variables of the encoding presented in section 3.2 can be immediately falsified. Gavrilenko et al. [13] describes how the fixed point problem can be lifted from traces to programs, effectively yielding a family of 'maximal' relations, consisting of possible relationships. More precision can be achieved by filtering the relevant relationships, ignoring those that cannot contribute to a violation of any constraint.

On the other hand, we expect some collection of trivially-satisfied relationships, allowing us to further truncating the set of variables in the encoding. Since we have to treat events as optional, a more generalised definition of *static relationships* will be necessary. This will be discussed in section 4.3.

However, memory instructions may compute their target addresses at runtime, and sometimes with more complex expressions. In order to confine the set of variable loc pairs in more complex programs, and ideally populate its static set, some preemptive analysis is necessary.

The *pointer analysis* requires knowledge about the control flow (chapter 1.3) and especially the dependency relationships (section 1.4).

Introduced by Kam and Ullman [16] as an approach for data flow analysis, monotone frameworks describe properties of a program with fixed point solutions. The control flow graph $\langle Q, \to \rangle$ of the to-be-analysed thread is considered, where it is possible to invert the direction in order to analyse backwards, if required. Let $\langle I, \sqsubseteq \rangle$ be a complete lattice. Each block or instruction $q \in Q$ is assigned a homomorphism $h_q \colon I \to I$ describing the propagation of information. A family of variables $(X_q)_{q \in Q}$ has then to be assigned values of I such that the following equation holds:

$$X_q = \bigsqcup_{p \to q} h_p(X_p)$$

If all transfer functions h_q are monotone, a least fixed point exists. Its computability requires that all transfer functions are computable and that the encountered monotone chains in the lattice must stabilize.

Definition 21. *A dataflow system $\langle Q, \to, I, \sqsubseteq, i, f\rangle$ consists of*

- *A control graph $\langle Q, \to\rangle$*
- *A complete lattice $\langle I, \sqsubseteq\rangle$ where all monotone chains $(d_i)_{i\in\mathbb{N}}$ with $\forall i \in \mathbb{N}.\, d_i \sqsubseteq d_{i+1}$ satisfy $\exists n \in \mathbb{N}. \forall i \in \mathbb{N}.\, d_n = d_{n+i}$*
- *Transfer functions $h_q \colon I \to I$ for each block $q \in Q$*

The system induces a fixed point problem over $X \in I^Q$:

$$X = q \mapsto \bigsqcup_{p\to q} h_p(X(p))$$

Example 13. *With control graphs of chapter 1 in mind, consider the complete lattice $\langle R \times Q, \subseteq\rangle$ where R is the set of all registers of the thread. Transfer functions h_q are specified below and the union of all joining states is considered.*

$$\begin{aligned}
&h(q) = X \mapsto X \cup \{\langle r, q\rangle\} && \text{if } q\ \delta_{provide}\ r \wedge q\ \delta_{guard}\ \varphi\\
&h(q) = X \mapsto (X \setminus (\{r\} \times Q)) \cup \{\langle r, q\rangle\} && \text{otherwise, if } q\ \delta_{provide}\ r\\
&h(q) = \mathrm{id} && \text{otherwise}
\end{aligned}$$

The least fixed point maps each register for each instruction to a set of providers for that register. It corresponds to direct δ-dependencies in definition 6.

For acyclic control graphs, the least fixed point would be easy to determine, as no recursion takes place. As soon as we take the shared memory into account, together with multiple threads, recursion reappears. We have to define behavior of load and store instructions, which communicate outside the control flow. Note that although architectures may reorder thread-local events at runtime (see section 2.3), we assume that this has no influence on the active register state.

We get a multi-threaded framework by adding a special block M for the memory, add $x \to M$ for all stores x and $M \to x$ for all loads x, effectively connecting the formerly-isolated thread graphs. In the worst case we cannot know anything about the address being accessed, and over-approximate the analysed behavior. f_M should project memory-specific information when propagated from stores to loads (or vice versa during backwards analyses). The fixed-point variable X_M shall serve as a pool for inter-thread communicated information.

Example 14. *Consider the lattice $\langle (R + D) \times D, \subseteq\rangle$ with register set R and transition functions as follows:*

$X \mapsto X \setminus (\{v\} \times D) \cup \{\langle v, f(a,b)\rangle \mid \langle x,a\rangle, \langle y,b\rangle \in X\}$	*for evaluations* $v \leftarrow f(x,y)$
$X \mapsto X \setminus (\{v\} \times D) \cup \{\langle v, x\rangle \mid \langle a,x\rangle, \langle k,a\rangle \in X\}$	*for loads* `ld` $v \leftarrow k$
$X \mapsto X \cup \{\langle r,a\rangle \mid \langle v,r\rangle, \langle k,a\rangle \in X\}$	*for stores* `st` $v \rightarrow k$
$X \mapsto X \cap (D \times D)$	*for block* M
id	*for all others*

The analysis over-approximates the dataflow independently of the memory model. The memory block collects an address-value relation, while loads fetch the relevant portion of it and stores insert new relationships into it. Also, the scopes of register states are taken into account. The least fixed point, if computable, relates each memory location and each register in the local register state to values it could take in some computation. However, for instance with $D = \mathbb{N}$*, the lattice does not satisfy the chaining condition.*

4.1. Alias Analysis

Pointer analyses relate memory events of the program to the addresses they may access. They are useful to find accesses to uninitialised memory like null pointer dereferences, and dangling pointers.

A relevant subclass of pointer analyses is the class of alias analyses, relating memory events which can refer to the same memory location. Results are useful not only to other analyses like *Available Expressions*, which is primarily used for optimisation, but also for the memory-model-aware verification. Additionally, it turns out to be implementable in a memory-model-aware manner itself (see section 4.1.4).

$$\texttt{st}\ a \rightarrow b$$
$$\texttt{ld}\ a \leftarrow c$$

With an alias analysis determining $b = c$ for the above program fragment, an optimizer will be able to eliminate both events (see section 4.2), i.e. in C, as long as none of them is explicitly tagged as sensible to side effects. In the applications relevant to this thesis, all memory event are implicitly marked as sensitive and require to be proven insensitive.

Dartagnan makes use of an abstraction layer of the program, the dynamic allocation with the heap storage. The analysis may take into account that (valid) addresses do not have to be statically predefined by the program, like by elements placed into a program's data segment, but may also be allocated by a single thread using `malloc` or similar factories and may itself be communicated to other threads via the shared memory. As long as addresses are used in a well-defined manner, no thread will be able to access a location that some other thread has allocated dynamically without having it been communicated directly or indirectly by that thread.

Sound and complete alias analysis turns out to be undecidable, as shown by Reps [30]. Therefore, one aims for performant approximations. The most common approach would be the pointer analysis by Andersen [5], a fixed point problem over the *expression may point to address* relation.

4.1.1. Context-Insensitivity

Pointer analysis quickly reaches its capacity for reasoning when faced with procedures of a larger program. Since any call to a procedure may introduce new variables to the analysis, the associated fixed-point problem would have to be adapted to a infinite domain or would no longer be monotonic, such that the problem's decidability becomes questionable. However, we focus on bounded programs in an assembly-like language, whose call structure is eliminated via inlining, if there was any to begin with.

The original program may have exhibited a call stack, usually implemented by a growing array in memory and dedicated *stack pointer* registers for management. This structure will have been decompiled for the verification task [29]. Note that since the program is bounded, it will never be able to make use of a growing stack structure. This means that the feature of context-sensitivity is not applicable in the context of this thesis.

4.1.2. Flow-Insensitivity

Dependency analysis was able to determine the thread-local communication via of registers (section 1.4). But it is the shared memory that introduces the lion's share of data flow. Threads will exchange addresses of dynamically allocated arrays in order to increase the bandwidth in which messages can be passed. The pointer analysis has to take this into account.

When considering weak memory models, possibilities of perceiving out-of-thin-air values and inter-branch communication become unwanted side effects that have to be tolerated in favor of performance. The result is an over-approximation that is insensitive to memory states.

Example 15.

$$l_0 \texttt{:st}\ x \to y$$
$$l_1 \texttt{:ld}\ r \leftarrow y$$
$$l_2 \texttt{:st}\ y \to y$$

Although any such execution would not be locally-consistent*, the flow-insensitive analysis deduces that r may receive the address y, by reading from l_2.*

Example 16.

$$l_0 : \texttt{if } \varphi \texttt{ goto } l_3$$
$$l_1 : \texttt{st } y \to x$$
$$l_2 : \texttt{goto } l_4$$
$$l_3 : \texttt{ld } r \leftarrow x$$

This simple program lets a guard condition φ decide either to write to a location x or to read from it. Control flow analysis has determined that executions through l_1 and executions through l_3 are mutually exclusive. But since l_1 is no dead code, its effects on the memory are considered, relating location x to the value y. Now in the next iteration, we would conclude that l_3 can read y.

$$l_0 : \texttt{if } \varphi \texttt{ goto } l_3$$
$$l_1 : s \leftarrow y$$
$$l_2 : \texttt{goto } l_4$$
$$l_3 : r \leftarrow s$$

Note that control flow analysis has managed to fix equivalent behavior when accessing registers. In this variant, the dependency list of l_3 does not include l_1, hence from this program fragment alone, there is no reason for r to be able to contain y.

4.1.3. Field-Sensitivity

Given an arithmetics on D with at least $(+) \in D^{D \times D}$, the analysis should take into account that memory is allocated in blocks of consecutive addresses, sometimes by arrays, primarily by data structures. Oftentimes the addresses of memory events take the form $r + k$ for a variable $r \in R \cup D$ and a constant offset $k \in \mathbb{Z}$. It will be desirable for the alias analysis to take into account that adjacent fields or members of a structure will receive different values and communicate different addresses.

As already stated in definition 11, bounded programs possess a finite set of valid addresses. Some of them will originate from static storage, and directly accessed by multiple threads from the beginning. Others were compiled from dynamic allocations performed by a thread and are to be revealed to others via inter-thread communication.

Example 17.

$$l_0 : \texttt{st } 1 \to x \qquad\qquad l_3 : \texttt{ld } r_0 \leftarrow x$$
$$l_1 : x_1 \leftarrow x + 1 \qquad\qquad l_4 : x_1 \leftarrow x + 1$$
$$l_2 : \texttt{st } 2 \to x_1 \qquad\qquad l_5 : \texttt{ld } r_1 \leftarrow x_1$$

Consider this message-passing program under the premise that x points to some static array of size at least 2 and is known to both threads. This means that the address constants $x+0$ and $x+1$ are valid addresses. Alias analysis determines already during its preprocessing that l_0 must access the same location as l_3, since both always access the location of x and only that. Both evaluations now conform to the offset pattern and each yields the address constant $x+1$. The may-result sets of both operations are equal.

During the first iteration, l_2 and l_5 both receive the computed address of the respective may-result set. At this point we have derived that those two may *access the same location. Since the first iteration already reaches a fixed point, we not only conclude that l_2 and l_5 must access the same location, but also that $\langle l_0, l_2\rangle$, $\langle l_0, l_4\rangle$, $\langle l_2, l_3\rangle$ and $\langle l_3, l_5\rangle$ must not. On one hand we have increased the static set of loc, on the other hand its maximal set was reduced. In this particular example, both sets even turn out identical, leaving no space for non-determinism in this relation.*

Example 18.

l_0 :st $x \to x$		
l_1 :ld $r \leftarrow x$	l_4 :ld $r \leftarrow y$	l_7 :ld $r \leftarrow z$
l_2 :$r \leftarrow r+1$	l_5 :$r \leftarrow r+1$	l_8 :$r \leftarrow r+1$
l_3 :st $r \to y$	l_6 :st $r \to z$	l_9 :st $r \to x$

The above program lets the flow-insensitive analysis deduce l_1 to read $x+3k$ from x. In order to keep computability along with field-sensitivity, the address space has to be bounded again.

Introducing address validity: With each location carrying a size property, valid addresses can be decomposed into a base address and an offset. An address $l+i$ associated with the location l and with offset $i \in D$ added to some integral constant $k \in D$ yields another address, that is also associated with l and holds offset $i+k$. Beyond addition, the operations under which address validity is closed reaches its limits at more complex operations like multiplication and bitwise manipulation. We abstract all invalid addresses into a single value ω, leading to a still-finite domain of abstract addresses.

Definition 22 (Valid address)**.** *Let $\mathcal{P}$ be a bounded $D\mathcal{T}$-program with finite set of locations L and sizes $|l| \in D$ for $l \in L$. The set of valid address constants is defined as follows.*

$$A := \{l+i \mid l \in L \wedge i \in D \wedge 0 \leq i < |l|\}$$

Let ω be some distinguishable object that is no address. $A_\omega = A + \{\omega\}$ denotes the abstracted address domain.

All operators $f \in D^{D\times D}$ are mapped to $f_\omega \in {A_\omega}^{A_\omega \times A_\omega}$, such that arithmetics based on

addition is defined as a family of endomorphisms $k+$ for $k \in D$:

$$k+ \colon A + \{\omega\} \to A + \{\omega\}$$
$$k + (\omega) = \omega$$
$$k + (l + i) = \begin{cases} l + (i + k) & \text{if } 0 \leq i + k < |l| \\ \omega & \text{otherwise} \end{cases}$$

Problems may arise for more complex address transformations. Take for example hash tables that use addresses as the identity of stored objects. In this case, usually a location is combined with an unpredictable offset. If the verifier is not able to deduce that the result is an appropriately-bounded offset, it has to fall back to ω.

Dartagnan would benefit from avoiding this fallback scenario, that some memory event's address expression was too complicated to statically compute. Once this happens anywhere in the program, the common behavior of all events that may access a given address has to be generalised.

Overall Algorithm We utilise the results of the dependency analysis from chapter 1.3, previously performed on $\mathcal{P}$.

The initial register state of location $l \in L$ can initially be l. Any other register state initially cannot be any address. Each memory event initially cannot access any address. Each memory address initially cannot contain any address.

For each write event `st` $v \to k$, if k can be some address a, the location at a can contain any address that v can be.

For each read event `ld` $v \leftarrow k$, if k can be some address a, v can be any address that the location at a can contain.

For each evaluation $r \leftarrow x + y$, if x can be some address a, and y is either a constant n or a register that can be some constant n, then r can be $a + n$. Otherwise, if the evaluation is too complicated, r can be any address.

Definition 23 (Pointer analysis)**.** *The pointer analysis of a bounded program $\mathcal{P}$ is a fixed point problem with variables for 2^{A_ω}. Each memory event $e \in \mathcal{E}_M$ has an address variable A_e. Each write event $e \in \mathcal{E}_W$ has a value variable W_e. Each register provider $e \in \mathcal{E}_R + \mathcal{E}_L$ has a result variable R_e.*

To treat initial values as providers, abstract locations $l \in L + \{\omega\}$ provide a constant $R_l := \{l\}$. Let δ^e denote all direct δ-dependencies of $e \in \mathcal{E}$ and in case that the initial value of the associated register r is readable by e, let δ^e additionally contain the initial address, which is either $r \in L$, or otherwise ω if $r \notin L$.

$$A_e = \bigcup \{R_d \mid d \in \delta^e_{address}\}$$
$$W_e = \bigcup \{R_d \mid d \in \delta^e_{value}\}$$
$$R_e = \bigcup \{W_w \mid w \in \mathcal{E}_W \wedge ((\omega \in A_w \cup A_e) \vee (A_w \cap A_e \neq \emptyset))\} \quad \text{if } e \in \mathcal{E}_R$$
$$R_e = \bigcup \{f_\omega[R_a, R_b] \mid a, b \in \delta^e_{require} \wedge a\ \delta_{provide}\ x \wedge b\ \delta_{provide}\ y\} \quad \text{if } e \text{ evaluates } f(x, y)$$

The least fixed point relates events to abstract addresses they *may* interact with. With this information, if two memory events end up with distinct address sets, their equivalence in loc is impossible for all traces.

4.1.4. Model Checking

Let us test consequences of a relationship w rf r for some $w \in \mathcal{E}_W$ and $r \in \mathcal{E}_R$: First direct consequence is that both events are executed. That means that all events x with $w \gg x$ or $r \gg x$ must be executed, as well. Also, all excluded events y with $x \otimes y$ must not be executed.

On the model side, rf usually contributes to compound relations, enabling more relationships between w and r, preferably up to some axiom. In case that one relationship alone violates an axiom of the model, this would usually be accomplished by an internal read opposing the program order. This observation leads us to the conjecture that, if the memory model is locally-consistent, the pointer analysis described above may ignore thread-local communication opposing the program order.

This reasoning could be extended to arbitrary sets of rf relationships, resulting in a series of partial traces of the program being model-checked, in order to gain precision for the running pointer analysis. However, we will require the model to have supportive properties. If not verified, the analysis would ignore behavior that would turn out to be consistent with the model, given some other communication that was not considered, rendering the process unsound.

4.2. Optimisation

Blocks that are unreachable from the initial block in a control graph (see chapter 1.3) will not be included in any execution and therefore will never exhibit side effects neither on the register state of the associated thread nor on the shared memory, and can therefore be removed from the instance without influencing the semantics of the program [7].

This well-known class of anomalies, structural features of a program, is called *unused code* or *dead code* and introduces one of the simplest forms of code elimination, or *slicing*. Slicing belongs to the family of refactoring techniques, aiming to transform programs such that effects to their semantics are manageable. This particular form aims to shrink the program's complexity.

Another class of anomalies that can be detected with control flow analysis are *unused variables*. Thread-local evaluations that are never queried do not effectively influence the register state of the thread. These anomalies would propagate to later analyses and the resulting SAT instance without contributing to the decision procedures. On the contrary, concerning alias analysis (4.1) for example, an unused variable would result in a sink node of the variable graph, its information never able to enrich that of memory operations.

Definition 24. *An event x of a program $\mathcal{P}$ is* obsolete *under a model $\mathcal{M}$ if and only if the program $\mathcal{P} - x$ that results by 'removing' x exhibits the same behavior under $\mathcal{M}$:*

$$\forall \tau.\, (\tau \vDash \mathcal{M} \wedge \tau \text{ is a witness of } \mathcal{P}) \Leftrightarrow (\tau - x \vDash \mathcal{M} \wedge \tau - x \text{ is a witness of } \mathcal{P} - x)$$

Slicing an event from $\mathcal{P}$ may be implemented by replacing it with $r \leftarrow r$ for some $r \in R$, with the intention that the indexes of the thread's instruction list do not change. Removing it from τ is a no-op if $x \notin \mathbb{E}$, but non-deterministic if $x \in \mathbb{E}_W$ is a write that is read by at least one read.

For instance, unconditional jumps whose destination is the next instruction are obsolete, since they support no control dependency.

Lemma 4. *Thread-local evaluations whose result is unused are obsolete under any model.*

Proof. Let $x \in \mathcal{E}_L$ be such an event in a program $\mathcal{P}$ providing the register $r \in R$ and $\tau \vDash \mathcal{M}$ a trace. x is invisible to the memory model by definition 13. Therefore, the sets of visible events $\mathcal{E}$ of $\mathcal{P}$ and $\mathcal{P} - x$ are identical. If τ originates from $\mathcal{P}$, with computation p_t of the containing thread, then p_t modified on all control states beginning with x and ending before the next providers of r or with the halting state testifies that τ originates from $\mathcal{P} - x$. Vice versa can be shown analogously. □

The next step would be to extend this notion to memory events. However, the definition 15 of axiomatic models allows patterns like 'All loads prevent reordering': $\text{po} ; \text{po} \cap R\_ \subseteq \text{hb}$. Consider this rule added to TSO, the resulting model does not allow arbitrary loads to be removed even if they do not contribute to the dataflow, as that could introduce new consistent behavior.

Kokologiannakis, Raad, and Vafeiadis [17] encountered a similar issue when they formulated a proof of correctness of the `GenMC` project. Their approach included assuming a series of properties on the input model. Although it would have been favorable to provide a decision procedure for the property on arbitrary models, over the course of this thesis we did not manage to develop one.

Definition 25 (Unawareness)**.** *A memory model $\mathcal{M}$ is* unaware *if and only if for each T-trace $\tau \vDash \mathcal{M}$ with event set $\mathbb{E}$, tag relation $tag \subseteq \mathbb{E} \times T$ and read-from map $\text{rf} \colon \mathbb{E} \to \{\bot\} + \mathbb{E}$, and for each untagged and unread event $x \in \mathbb{E}_R + \mathbb{E}_W$ with $\neg\exists t \in T.\, x \ tag\ t$ and $\neg\exists r \in \mathbb{E}.\, \text{rf}(r) = x$, the trace $\tau - x$ with x removed is also consistent $\tau - x \vDash \mathcal{M}$.*

We expect Sequential Consistency, Total Store Ordering, ARM and Power to be unaware, since their preserved program order is defined by barriers or other tags. We further suspect the C model described by Vafeiadis et al. [36] to be unaware, if non-atomic operations are tagged 'non-atomic' and untagged events mean relaxed-atomic (see 5.6).

Theorem 1. *If $\mathcal{M}$ is unaware, untagged loads that result in an unused variable are obsolete under $\mathcal{M}$.*

Proof. Let $x\mathcal{E}_R$ be such a read in the program $\mathcal{P}$ and $\tau \vDash \mathcal{M}$ be a trace. Since $\mathcal{M}$ is unaware and x is untagged, $\tau - x \vDash \mathcal{M}$. Since x has no effect on the dataflow, $\tau - x \sim \tau$. If τ originates from $\mathcal{P}$, we construct a computation for $\tau - x$ on $\mathcal{P} - x$ in a similar manner as in the proof of lemma 4. Starting from $\tau - x$ originating from $\mathcal{P} - x$ works analogously. □

Theorem 2. *If $\mathcal{M}$ is unaware, untagged stores that can never be read are obsolete under $\mathcal{M}$.*

Proof. Let $x\mathcal{E}_W$ be such a write event in the program $\mathcal{P}$ and $\tau \vDash \mathcal{M}$ be a trace. By premise, x is not read by any event of τ, therefore $\tau - x$ is well-defined. Again we deduce $\tau - x \vDash \mathcal{M}$ as before. This time, showing witness-equivalence does not require modification, as stores do not manipulate the register state, and we conclude, $\tau - x$ originates from $\mathcal{P} - x$. □

Not all addresses have to be shared between threads. If some location could be proven only accessible by one thread, it could be replaced with a new register in following procedures, including verification. Any means of *release-acquire*-like synchronisation on this location could also be omitted since it would require some other thread to access it. Note that the required premise entails that the pointer analysis was *perfect*, such that no memory event was related to ω. Yet we were not able to implement a pointer analysis that returned perfect results on any benchmark and therefore have not evaluated the impact of the following optimisation on the verification procedure.

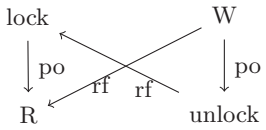
Theorem 3. *If the memory model is locally-consistent and unaware, an address that is accessible by only one thread and only by untagged events, is 'obsolete': It can be replaced by a new register.*

Proof. Let $a = \langle l, i \rangle \in L \times D$ be such an address. First transform the owning thread by adding the new register r. Prepend $r \leftarrow i$ to the thread's instruction list. Each store $\texttt{st}\ v \to k$ that can access a is appended $\texttt{if}\ k = l + i\ r \leftarrow v$. Each load $\texttt{ld}\ v \leftarrow k$ that can access a is prepended $\texttt{if}\ k = l + i\ v \leftarrow r$. If the event must access a, the condition may be $\top$, but this has no effect on this proof. This is a refactorisation: the evaluations added to the stores are obsolete by lemma 4, and removing them reveals the remainders to be obsolete, as well.

Now swap each load with its prepended evaluation, so that it may override the load. Local consistency ensures that this is a refactorisation, as the communicated value always coincides with the value in r.

Finally, for each involved memory event $x \in \mathcal{E}_M$ in reversed instruction order, if x must access a, it is obsolete analogously to theorem 1 or 2, respectively. Otherwise, adding the condition $k \neq l + i$ to x is a refactorisation. We then can remove a from the set of reachable addresses of the modified x and continue with the previous memory event. When the first event was processed, a has become unaccessible. □

The insights gained by pointer analysis extend from what we have discussed. We might observe that programs use multiple addresses for compound assignments and ensure atomicity with locks. In those fragments, clustering multiple communications might reduce the expected problem size significantly: in terms of chapter 3, this would mean directly relating multiple relationships of rf. For instance, consider a pair of critical sections on a buffer, one updating all elements, the other copying all elements. When a thread locks with the intention to copy, and synchronises with a previous unlock of an update, the sources of the read events will be determined.



This process would have to answer two essential questions: Which valid addresses are *protected* by some kind of synchronisation mechanism? And what kind of synchronisation safety does the mechanism guarantee? Question is which accesses actually just concern the data-flow and which are important to the order of events.

4.3. Relation Analysis

This chapter focuses on the task of shrinking the encoding of the memory model under the context of a bounded program. The reduction to a formula presented in 3 introduced many variables that are directly or indirectly equalised to a false statement. This starts with the atomic relations, as demonstrated by *read-from*, where only write events may appear as domain-side event of this relation, and only read events can ever read from them and fill the range-side. Impossible relationships propagate upwards through compound relations, thus defining an over-approximation of possible pairs.

Additionally, not all possible relationships have to be relevant for a constraint. In fact, irreflexivity just focuses on relationships events have with themselves, while the subset of heterogenous pairs may be ignored. As a special case of irreflexivity, acyclicity is unaffected by possible pairs that will never contribute to any cycle. Iteratively marking possible relationships as relevant yields an even more precise over-approximation of possible-and-relevant pairs.

Those steps of reducing the encoding with an over-approximation of possible and relevant event pairs were already discussed by Gavrilenko et al. [13]. Basically, Relationships that are irrelevant to verification, or impossible due to some static precondition being violated, can be encoded with $\bot$ and ignored by the solver.

We can examine the behavior of a bounded program under an axiomatic memory consistency model by filling out the relations defined in the model. Axiomatic memory consistency models lay constraints on relations between events of a program. Taking

branching and conditional instructions into account, events issued by threads of the program become optional.

With events $x, y \in \mathcal{E}'$ and relation $R \in \mathcal{R}'$ of some program and model, we denote the proposition that a trace τ executes x by $\tau \vDash x$ and the proposition that τ relates x to y in R by $\tau \vDash x\ R\ y$. Note that by definition 17, $\tau \vDash x\ R\ y$ always implies $\tau \vDash x$ and $\tau \vDash y$.

The performance of Dartagnan highly depends on the complexity of the produced formula. As shown in chapter 6, the running time of its static analysis is towered by the solving time by several magnitudes. This is due to the fact, that the solver's problem still is $\mathcal{NP}$-complete, at best.

The relationships between events of a program are expressed by boolean variables that are decorated with preconditions. For instance, the preconditions of loc relationships include the equality of the participating event's address values, as does rf together with the communicated value. However, as the membership of an event in the trace can be optional, all relationships require at least that both events are executed.

Would that not be the case, we would experience some relationships to always hold unconditionally. This section targets such relationships, as this property exhibits interesting propagation properties similar to the maximal tuple set.

Definition 26 (Static relationship)**.** *Let $\mathcal{P}$ be a bounded program with extended event set $\mathcal{E}'$, and $\mathcal{M}$ be memory consistency model with extended relation set $\mathcal{R}'$.*

The triple $\langle R, x, y\rangle \in \mathcal{R} \times \mathcal{E} \times \mathcal{E}$ is called static*, if and only if it denotes a relationship that always holds as long as both events are executed:*

$$\forall \tau.\, \tau \vDash \mathcal{P} \land \tau \vDash x \land \tau \vDash y \Rightarrow \tau \vDash x\ R\ y$$

The static relation mapping *maps relations of the model to a subrelation.*

$$\text{static}_{\mathcal{M},\mathcal{P}}(R) := \{\langle x, y\rangle \mid \langle R, x, y\rangle \text{ is static}\}$$

$R \in \mathcal{R}'$ is called static *if and only if $R = \text{static}_{\mathcal{M},\mathcal{P}}(R)$*

Example 19. *Some relations defined in the memory consistency modelling language are static in the sense that all possible relationship variables are bound to that minimal premise. We already know po completely beforehand, given the unrolled program in its entirety. We already know the types and tags of events, therefore domain and range constructions yield static relations.*

Lemma 5. *Let $R \in \mathcal{R}'$ and $t \in T \cup \{I, R, W\}$ be a tag. If $\text{def}(R) \in \{po, \_\_\_, id, t\_, \_t\}$, then R is static.*

Proof. Follows directly from definitions 14 and 17. □

Lemma 6. *Let $x, y \in \mathcal{E}'$ be events and $R, S, T \in \mathcal{R}'$ be relations.*

- *If $\text{def}(R) = S^{-1}$, $\langle R, x, y\rangle$ is static if and only if $\langle S, y, x\rangle$ is static.*

- *If* $\mathrm{def}(R) = S \cup T$, $\langle R, x, y\rangle$ *is static if and only if at least one of* $\langle S, x, y\rangle$ *and* $\langle T, x, y\rangle$ *is static.*
- *If* $\mathrm{def}(R) = S \cap T$, $\langle R, x, y\rangle$ *is static if and only if both* $\langle S, x, y\rangle$ *and* $\langle T, x, y\rangle$ *are static.*
- *If* $\mathrm{def}(R) = S \setminus T$, $\langle R, x, y\rangle$ *is static if and only if* $\langle S, x, y\rangle$ *is static and* $\langle S, x, y\rangle$ *is not even maximal.*

With the information gained from control flow analysis (section 1.3), staticality can propagate onto compositions.

Theorem 4. *Let* $x, y, z \in \mathcal{E}'$ *be events and* $R, S, T \in \mathcal{R}'$ *be relations with* $\mathrm{def}(R) = S\,;T$. *If* $\langle S, x, y\rangle$ *and* $\langle T, y, z\rangle$ *are static,* y *is unconditional and at least* $x \gg y$ *or* $z \gg y$, *then* $\langle R, x, z\rangle$ *is static.*

Proof. Let $\tau \vDash x$ and $\tau \vDash z$ and $x \gg y$ (or $z \gg y$). Since y is unconditional, we infer $\tau \vDash y$ (analogously). $\tau \vDash x\ S\ y$ and $\tau \vDash y\ T\ z$ follow from the presumed staticality. We conclude $\tau \vDash x\ R\ z$. □

This is indeed helpful for barrier relations, as oftentimes barriers are placed into the same control path as immediately preceding or succeeding memory events. Architectures whose instruction set does not support memory-ordered loads and stores like `atomic_load(A)` [28] will instead prepend or append a fence to regular memory instructions. Nevertheless, barrier relations require the execution of a third event and will not always be static.

The ARM and Power models define their *preserved program order* recursively [22]. The preserved program order is a term widely used in literature and denotes a subset of the program order that has not been reordered in the trace. Static tuples propagate nicely through recursion and may eliminate vast portions of the encoding.

Considering dependencies, there is also a useful static characterisation, based on the fact that the latest executed register provider will always determine the current value. Note that for this purpose, definition 6 was lifted to the domain of events.

Theorem 5. *Let* $x, y \in \mathcal{E}$ *be events and* $R \in \mathcal{R}' \setminus \mathcal{R}$ *a direct dependency relation associated with* $\delta \subseteq \mathcal{I}(D, R, T) \times R$. *If* x *is a direct* δ*-dependency of* y *and for all other direct* δ*-dependencies* $z \in \mathcal{E}$, z *is not reachable from* x, *then* $\langle R, x, y\rangle$ *is static.*

Proof. Let $\tau \vDash x$, $\tau \vDash y$ and $p_t \colon \{0 \dots |I|\} \to (\{\bot\} + D^R)$ a computation from definition 14. By premise, there is no provider between x and y for the register r in question. Therefore, the most recent manipulation of $p_t(y)(r)$ must have been that of x. We conclude $\tau \vDash x\ R\ y$ by the definition of direct dependency. □

Now we know that for each dependent event and required register, there will always be at least one static relationship for the direct dependency relation. In combination with theorem 4, this can propagate up to the general dependency relations.

Example 20.

$$0\colon \texttt{ld}\ r \leftarrow a$$
$$1\colon \texttt{if}\ 0 \leq r\ r \leftarrow -r$$
$$2\colon \texttt{st}\ r \rightarrow a$$

This program fragment updates a stored value by its absolute value. By the theorems above, the set of static $dep_{require}$-relationships include at least $\langle 0, 1\rangle$, and $\langle 1, 2\rangle$. But $\langle data, 0, 2\rangle$ is also static, taking all possible control paths into account. We have not yet implemented an analysis covering such a scenario.

At last, the memory variable loc exhibits static relationships, that we discussed in section 4.1. Since they tend to be highly connected to the dataflow of the program, there is little to expect from rf that is not already covered in section 4.2.

Lemma 7. *Let $x, y \in E$ be events, $l \in L$ a location and $i \in \{0 \dots |L| - 1\}$ an offset. If alias analysis determines $A_x = A_y = \{l + i\}$, then $\langle loc, x, y\rangle$ is static.*

Proof. Let $x = \langle t_x, i_x\rangle \in \mathcal{E}_M$, $y = \langle t_y, i_y\rangle \in \mathcal{E}_M$, $i_x\ \delta_{\text{address}}\ k_x$, and $i_y\ \delta_{\text{address}}\ k_y$. Since the algorithm over-approximates the dataflow, we infer $p_{t_x}(i_x)(k_x) = l + i = p_{t_y}(i_y)(k_y)$ for all computations (p_t) relating τ to the program, thus location(x) = location(y) and finally $\tau \vDash x$ loc y.

Considering initialisation, from $z \in \mathcal{E}_I$ follows $z = \langle l, i\rangle$, thus $\tau \vDash x$ loc z and $\tau \vDash z$ loc y, as well. □

Lemma 8. *Let $t \in \{0 \dots |\mathcal{P}| - 1\}$ be a thread with instruction list I, $Q := \{0 \dots |I| - 1\}$ and $d := \{\langle p, q\rangle \in Q \times Q \mid p$ is an address-dependency of $q\}$. If $d(x) = d(y) \neq \emptyset$ for some $x, y \in Q$ and $\langle t, x\rangle, \langle t, y\rangle \in \mathcal{E}_M$, then $\langle loc, \langle t, x\rangle, \langle t, y\rangle\rangle$ is static.*

Proof. Note that both events use the same register r with $x\ \delta_{\text{address}}\ r$ and $y\ \delta_{\text{address}}\ r$. Again consider a trace originating from the program and some testifying computation p. Since the subset of executed events represented by $d(x)$ stays equal, the respective latest provider is the same for both events and we conclude $p_t(x)(r) = p_t(y)(r)$. □

Computing the exact static tuple sets for a program under some model would require exhausting analysis. In the context of verification, we can settle for an *under-approximation*, in order to effectively shrink the encoding without changing its semantics: For all triples that were identified as static, the subformulas carrying their original definitions can be replaced by

$$\bigwedge_{\langle R,x,y\rangle \text{ is static}} x\ R\ y \Leftrightarrow \chi_x \wedge \chi_y \tag{4.1}$$

The expected impact on the solving process for the truncated proposition entails a reduced set of variables, as static 'inner' relationships may become obsolete after this analysis, and a *shift* of unit propagation invocations into earlier iterations of the solver.

5. Related Work

This chapter discusses recent work done in the field that we consulted during this thesis. We provide summaries for several publications.

5.1. Thread-Modular Static Analysis for Relaxed Memory Models

Kusano and Wang [18] check reachability in their own decision procedure. They define mandatory relationships like 'must happen before' and 'must not read from' in order to reason about consistency with the model. Speaking of which, models are described in a more restricted language where RR- RW- WR- WW-reordering can either be permitted or prohibited. SC, TSO, PSO, RMO of the SPARC-hierarchy are expressible, but AARCH64 and Power are not.

Must Not Read From uses non-static premises. When transferred to a SAT instance, results will look like $\neg S_1 \models \vee\neg(L_1 \mathrm{rf} S_0) \vee \neg(L_2 \mathrm{rf} S_0)$ and will be added to the instance. When the effort is done to compute a static part of an acyclicity constraint, the results should be honored when encoding the rules.

When collecting the must-part of a relation, we effectively decide a part of the SAT instance that is linearly decidable (in DPLL solvable using unit propagation). While this part does not have to be encoded anymore, the time advantage is not impressive.

Moreso the restrictions gained from the static analysis possible with it. There is a fraction of the program order that must not be reordered by any memory model.

5.2. SAT modulo Graphs: Acyclicity

Gebser, Janhunen, and Rintanen [14] define the problem ACYC-SAT, where instances consist of a main proposition and a graph with edges annotated with propositions. A model of the main proposition models the instance, if and only if the subgraph of enabled edges is acyclic. They discuss four different approaches to reduce ACYC-SAT to SAT, and briefly argue the contribution of the approaches to the solving process via unit propagation. The authors propose and prove that ACYC-SAT is linear-time interreducable to IDL(1)-SAT, a first-order logic with integer terms and one predicate of the form $x < y$, which additionally is not allowed to appear in negated context. They claim to have inspected the source code of Z3 unsuccessfully to confirm specialised behavior if faced with acyclicity instances.

5.3. A Shared Memory Poetics

Alglave [1] bases her understandings on the SPARC architecture and applies it to the Power ISA specification, for which she aimed to formulate a memory consistency model. The thesis is accompanied by the `diy` testing tool for a model of the Power architecture, which contributes to the early stages of the memory-model-aware `herd` tool (section 5.5), and adds more groundwork for the class of axiomatic memory models that at that time had yet to be further formalised (chapter 2). She also discusses the DEC Alpha architecture, exposing its own weak memory model, which she shows to be incomparable to Relaxed Memory Order.

The *uniproc* axiom $\emptyset = \mathrm{id} \cap ((\mathrm{po} \cap \mathrm{loc}) \cup \mathrm{com})^+$ frequently provided by the cat files of Dartagnan seems to have gotten its name from this contribution.

5.4. Understanding POWER Multiprocessors

Sarkar et al. [32] discussed the processor series of IBM Power. They document experiments performed on such a device and compared it to the expected behavior of the specification. They formulated a generalised operational model in which instructions become *in-flight* before being committed. Instructions in this state may even become visible past a control dependency (see chapter 1.3). They compare Power with ARM, which implements similar methods of speculation and similar families of barriers.

5.5. Herding cats: Modelling, Simulating, Testing and Data-mining for Weak Memory

Alglave, Maranget, and Tautschnig [3] present the tool `herd`, which enumerates consistent runs for a given program and axiomatic memory model, as well as the tool `mole`, which extracts multi-thread-sensitive fragments from larger code bases written in C. Motivated by the machine defined in [32], they define an operational memory model on the basis of an axiomatic one, prove its equivalence to its base, and compare it to the machine. In addition to this, they evaluated the tools on Debian Linux version 7.1 of which pattern search was performed with `mole`, resulting in a variety of smaller test cases that were then verified with `herd`.

5.6. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it

Vafeiadis et al. [36] show that the memory model defined in the C and C++ standard from 2011 has flaws that result in some common optimisations *and* compilations unsound and provide means to fix it. To us, it came unexpected, that the standard model seems to not take spontaneous synchronisations into account when dealing with non-atomic memory accesses. Such a read must be satisfied by the latest write that happened before it, where inter-thread happens-before just requires explicit synchronisation (i.e. using release-acquire read-from relationships). Non-atomic accesses seem to follow a stronger memory order than relaxed atomic accesses. Standard-conform architectures are thereby required to cache arbitrarily.

$\texttt{ld}\ r \leftarrow y\ (relaxed)$	$\texttt{ld}\ r \leftarrow z\ (relaxed)$
$\texttt{if}\ r\ \texttt{st}\ 1 \rightarrow z\ (relaxed)$	$\texttt{if}\ r\ \texttt{st}\ 1 \rightarrow y\ (relaxed)$

The first program (called 'CYC') demonstrates that the model allows speculation with relaxed accesses: There is a consistent execution where both threads read from each other.

$\texttt{st}\ 1 \rightarrow x\ (non-atomic)$	$\texttt{ld}\ r \leftarrow x\ (non-atomic)$

The second program is not data-race-free, therefore its behavior is undefined and contains an execution in which the load reads from the store.

	$\texttt{ld}\ r \leftarrow y\ (relaxed)$	$\texttt{ld}\ r \leftarrow z\ (relaxed)$
$\texttt{st}\ x \rightarrow 1\ (non-atomic)$	$\texttt{ld}\ r \leftarrow x\ (non-atomic)$	
	$\texttt{st}\ 1 \rightarrow z\ (relaxed)$	$\texttt{st}\ 1 \rightarrow y\ (relaxed)$

The third program (called 'SEQ') would include a data race like the former if the load is executed. However, this would require non-atomic communication through x, which is only allowed with synchronisation that the program does not provide. Therefore, there shall be no execution with the load and the program is data-race-free.

6. Evaluation

This chapter contains some example programs that were verified using Dartagnan. This includes mechanisms for mutual exclusion (section 6.1) as well as popular implementations of lockfree datastructures (section 6.2).

Section 6.3 compares the usage of static tuples (section 4.3) to the original encoding with just the over-approximation using

6.1. Locking

The most popular method of inter-thread communication. When shared data structures that are distributed into several distant locations have to be manipulated in an atomic manner, or when the manipulations performed on them extend the limits of regular atomic operations, programmers want to make sure that no other thread will be able to intervene the process once it has started and until it has ended. The program fragment that describes the operation is called a *critical section*. Its start and end may each consist of several control states, and have to be properly occupied by subprocedures to enforce temporary mutual exclusion. Those subprocedures are called `lock` and `unlock` respectively.

For instance, the contribution of Lamport [19] would consist of two rounds m and $m+1$ and a flag $m+2+t$ for each thread t. This implementation requires at least cache consistency for its liveness property, as otherwise a scenario could arise that two candidate threads disagree on the current value of the second round.

Szymański [34] proposed an implementation for mutual exclusion that consists of two waiting rooms for locking candidates, each implemented by spin-locks. During locking, a thread communicates its progress through a system of five states to its concurrents. Initially threads reside in state 0, as long as they do not intend to lock. State 1 marks the first waiting room, 3 the second, 2 in-between the rooms and 4 for the critical section.

Taking C memory order [28] into account, `lock` must develop acquire semantics, as `unlock` must have release semantics, in order to let the modifications in a critical section appear atomic from the viewpoint of the successive threads.

The locking tests consist of five worker threads, all performing the same instruction list, beginning with acquiring ownership on the lock m, performing a simple concurrent but non-atomic operation on the protected location x, and releasing m again. This particular number of threads was chosen specifically to stay solvable in a reasonable amount of time for our setup while also being able to exploit any vulnerability of the implementation, although for the selection described below, three threads might have sufficed.

Figure 6.1.: thread of the TTAS program

$$
\begin{aligned}
\texttt{lock:}\ & \texttt{ld}\ r \leftarrow m\ (x) \\
& \texttt{if}\ r\ \texttt{goto lock} \\
& \texttt{ld}\ r \leftarrow m\ (rsrv)\ (acq) \\
& \texttt{st}\ \top \rightarrow m\ (cnd) \\
& \texttt{if}\ r\ \texttt{goto lock} \\
\texttt{critical:}\ & \texttt{st elem} \rightarrow x \\
& \texttt{ld}\ r \leftarrow x \\
& \texttt{if}\ r \neq \texttt{elem fail} \\
\texttt{unlock:}\ & \texttt{st}\ \bot \rightarrow m\ (rel)
\end{aligned}
$$

6.1.1. Spinlock

The most simplistic lock consists of a single boolean value which controls access to its critical section. It starts initially unset, as a set value would signal to other candidates that some thread currently owns the lock. TTAS means *test and test-and-set* and differs to the description above in a preliminary test loop before the slightly more expensive read-modify-write operation that implements the locking.

Each thread enters a critical section by locking m and exits it by unlocking it. The test uses one data variable x to verify mutual exclusion under the assumption that the memory model is at least locally-consistent (definition 19). Only if another thread is disturbing the computation, shall the locking thread be able to read from a different store than its own.

6.1.2. Fast userspace mutual exclusion

Franke, Watson, and Kirkwood [12] proposed an alternative mutual exclusion implementation that avoids the kernel mode of a Linux operating system. The lock has three states, stored at m: unlocked, locked, and locked-with-waiting-threads. It is equipped with a counter at $m + 1$, signaling to all waiting threads that the current owner has unlocked. This ensures that no thread polls for the duration of more than one critical section.

6.2. Lockfree Datastructures

For inter-thread communication that follows a simpler structure, like that of a homogenous collection of values, numerous contributions were added to the repertoire of a

Figure 6.2.: thread of the 'futex' test

$$
\begin{array}{rl}
\texttt{lock}: & \texttt{ld}\ r \leftarrow m\ (rsrv)\ (acq) \\
& \texttt{if}\ r = 0\ \texttt{st}\ 1 \rightarrow m\ (cnd)\ (x) \\
& \texttt{if}\ r = 0\ \texttt{goto critical} \\
l_0: & \texttt{ld}\ r \leftarrow m\ (rsrv)\ (x) \\
& \texttt{if}\ r = 1\ \texttt{st}\ 2 \rightarrow m\ (cnd) \\
& \texttt{ld}\ s \leftarrow m + 1\ (acq) \\
& \texttt{ld}\ r \leftarrow m\ (acq) \\
& \texttt{if}\ r \neq 2\ \texttt{goto critical} \\
l_1: & \texttt{ld}\ r \leftarrow m + 1\ (acq) \\
& \texttt{if}\ r = s\ \texttt{goto}\ l_1 \\
& \texttt{ld}\ r \leftarrow m\ (rsrv)\ (acq) \\
& \texttt{if}\ r \neq 0\ \texttt{goto}\ l_0 \\
& \texttt{st}\ 2 \rightarrow m\ (cnd)\ (x) \\
\texttt{critical}: & \texttt{st elem} \rightarrow x \\
& \texttt{ld}\ r \leftarrow x \\
& \texttt{if}\ r \neq \texttt{elem fail} \\
\texttt{unlock}: & \texttt{ld}\ r \leftarrow m\ (rsrv)\ (x) \\
& \texttt{if}\ r = 1\ \texttt{st}\ 0 \rightarrow m\ (cnd)\ (rel) \\
& \texttt{if}\ r = 1\ \texttt{goto}\ l_2 \\
& \texttt{st}\ 0 \rightarrow m\ (rel) \\
& \texttt{ld}\ r \leftarrow m + 1\ (rsrv)\ (x) \\
& \texttt{st}\ r + 1 \rightarrow m + 1\ (cnd)\ (rel) \\
l_2: &
\end{array}
$$

programmer. In this section, We discuss some popular approaches for shared LIFO and FIFO storage.

The tests follow a simple structure; each participating thread will add a value before attempting to remove some value. Each item of the data structure is dynamically allocated into the locations pointed to by `elem` of each thread and composed of a `next`-pointer at `elem` and a value at `elem` + 1.

The removed item is kept allocated by the thread, as our focus does not lie on storage reclamation mechanisms. The problem arises when a concurrent program insists on manual deallocation instead of automatic *garbage collection*, which would entail another thread performing a dynamic storage reachability analysis, as the worker threads perform their tasks. The garbage collector would synchronise regularly with the other threads and deallocate any memory that has become unreachable by any worker, thus avoiding dangling pointers while also enabling reclamation.

A single-threaded program should have no refutation to free its resources before falling out of scope. In a concurrent situation, however, some other thread might itself be in the process of popping, having loaded a now stale version of the location at `TOP`, and the address of its next item, its own progress currently at around `loop`: (give or take reordered instructions). Since no communication between the two threads has taken place yet, the thread that is about to deallocate the item would impose *undefined behavior*, so in the jargon of the C specification.

The magnitude in which the other thread might be delayed is unbound. It might even occur that other threads have enough 'time' to reallocate the memory of the item, assign a new value and next element to it, possibly push it into another stack or even give that address completely different semantics, without the paused thread noticing.

In a bounded setting, reclamation is unnecessary, and Dartagnan does not yet implement any verification on this matter. One should note, however, that by enforcing disjoint arrays to be returned by the function family of `malloc`, as the tool currently does, the possible behavior of the original program is under-approximated.

6.2.1. Treiber's stack

The stack by Treiber [35] consists of a singly linked list and one sole location `TOP` of size 1 per container that may store the address to the top item. Once successfully inserted into the stack, and until successfully popped from it, a list item is not manipulated. In addition, barriers are placed appropriately in order to ensure that no element gets lost during stack manipulation. The data structure heavily relies on the atomic compare-and-swap operation.

The `push` procedure accepts some address `TOP` $\in R$ to a stack and some value $v \in R$ to insert at the top of it. For simplicity, all involved arguments are already assumed to be present in the register state. Note that all lockfree examples discussed in this thesis contain at least one cycle in the control graph in order to allow polling.

The `pop` procedure accepts some address `TOP` $\in R$ to a stack and returns some value `result` $\in R$. If the stack was empty, the result register is not touched. Note again that for simplicity, safe memory reclamation has been abstracted out of the program.

Hendler, Shavit, and Yerushalmi [15] extend this implementation by an *elimination scheme*, invoked on an unsuccessful manipulation attempt. Threads are instructed to deposit an individual identifier at the lock using a CAS operation, effectively giving one of them a small head start for reattempting the stack manipulation, while the others are being delayed by another loop.

6.2.2. Michael and Scott's Queue

As typical for a queue, the header consists of two item references `HEAD` and `TAIL` for head and tail of the sequence, respectively, where the tail item's `next` pointer will be updated to a new item when `enqueue`d into the structure, and the head item indirectly accesses the whole sequence.

The queue starts with a dummy element, to which both `HEAD` and `TAIL` point and whose `next` pointer is `NULL`. This means that there will always be some item in the queue whose value is to be ignored and `dequeue` will have to check with `TAIL` before returning the item it found at `HEAD`.

As Treiber's stack in section 6.2.1, the algorithm requires the `cas` operation in order to verify that the view the thread has on the structure is current.

Suggesting an unnecessary failing condition in the `dequeue` procedure, Doherty et al. [11] proposed a slight derivation by preponing the final CAS on `HEAD`. The idea is that any thread, that had successfully incremented the head, would only be able to perceive `TAIL` pointing to the removed item, if another thread had inserted the next element and yet missed to synchronise. In this situation, the original would reattempt, while the new implementation just copies the missed update to tidy the queue. The `enqueue` procedure was left unchanged.

6.3. Results

The following tables show the verifier in two configurations on a different input, each input consisting of a program and an unroll bound ‘k’. Both configurations differ only in their usage of static tuples, first omitting their computation, and second performing it. Each table groups the results for one of the tested memory models SC, TSO, AARCH64 and Power.

‘k’ contributes to the size of the bounded program being verified, resulting in higher analysis time ‘pre’, measuring the entire transformation in seconds on our machine from when the tool finished parsing to when the proposition has been completed. The results of the relation analysis are summarised into the total number of maximal relationships between events, the total number of static relationships, if applicable, and the total

Figure 6.3.: lockfree stack, left T [35], right HSY [15]

```
push: st TID → elem + 1
  l0: ld t ← TOP (acq)
      st t → elem
      ld s ← TOP (rsrv) (acq)
      if s = t goto l2
      goto l0
  l2: st n → TOP (cnd) (rel)
 pop: ld t ← TOP (acq)
      if t = 0 result ← 0
      if t = 0 goto check
      ld n ← t
      ld s ← TOP (rsrv) (acq)
      if s = t goto l4
      goto pop
  l4: st n → TOP (cnd) (rel)
      ld result ← t + 1
check: if result = 0 fail
```

```
push: st TID → elem + 1
  l0: ld t ← TOP (acq)
      st t → elem
      ld s ← TOP (rsrv) (acq)
      if s = t goto l2
  l1: ld w ← WAIT (acq)
      ld s ← WAIT (rsrv) (acq)
      if s ≠ w goto l1
      st elem → WAIT (cnd) (rel)
      goto l0
  l2: st n → TOP (cnd) (rel)
 pop: ld t ← TOP (acq)
      if t = 0 result ← 0
      if t = 0 goto check
      ld n ← t
      ld s ← TOP (rsrv) (acq)
      if s = t goto l4
  l3: ld w ← WAIT (acq)
      ld s ← WAIT (rsrv) (acq)
      if s ≠ w goto l3
      st elem → WAIT (cnd) (rel)
      goto pop
  l4: st n → TOP (cnd) (rel)
      ld result ← t + 1
check: if result = 0 fail
```

Figure 6.4.: lockfree queues, left MS [23], right DGLM [11]

```
enqueue: st 1 → elem
     l0: ld t ← TAIL (acq)
         ld n ← t (acq)
         ld r ← TAIL (acq)
         if n = r goto l0
         if n = 0 goto l1
         ld r ← TAIL (rsrv) (acq)
         if r = t st n → TAIL (cnd) (rel)
         goto l0
     l1: ld r ← n (rsrv) (acq)
         if r ≠ n goto l0
         st elem → n (cnd) (rel)
         ld r ← TAIL (rsrv) (acq)
         if r = t st elem → TAIL (cnd) (rel)
dequeue: ld h ← HEAD (acq)
         ld t ← TAIL (acq)
         ld n ← h (acq)
         ld r ← HEAD (acq)
         if r ≠ h goto dequeue
         if n = 0 result ← 0
         if n = 0 goto check
         if h ≠ t goto l2
         ld r ← TAIL (rsrv) (acq)
         if r = t st n → TAIL (cnd) (rel)
         goto dequeue
     l2: ld r ← HEAD (rsrv) (acq)
         if r ≠ h goto dequeue
         st n → HEAD (cnd) (rel)
         ld result ← n + 1
  check: if result = 0 fail
```

```
enqueue: st 2 → elem
     l0: ld t ← TAIL (acq)
         ld n ← t (acq)
         ld r ← TAIL (acq)
         if n = r goto l0
         if n = 0 goto l1
         ld r ← TAIL (rsrv) (acq)
         if r = t st n → TAIL (cnd) (rel)
         goto l0
     l1: ld r ← n (rsrv) (acq)
         if r ≠ n goto l0
         st elem → n (cnd) (rel)
         ld r ← TAIL (rsrv) (acq)
         if r = t st elem → TAIL (cnd) (rel)
dequeue: ld h ← HEAD (acq)
         ld n ← h (acq)
         ld r ← HEAD (acq)
         if r ≠ h goto dequeue
         if n = 0 result ← 0
         if n = 0 goto check
         ld r ← HEAD (rsrv) (acq)
         if r ≠ h goto dequeue
         ld t ← TAIL (acq)
         st n → HEAD (cnd) (rel)
         if h ≠ t goto l2
         ld r ← TAIL (rsrv) (acq)
         if r = t st n → TAIL (cnd) (rel)
     l2: ld result ← n + 1
  check: if result = 0 fail
```

number of encoded relationships, contributing to the number of variables contained in the encoding.

Dartagnan usually uses two SMT instances for verification: One for reachability of an error state in the bounded context, and one for reachability of a higher bound. In the tables below, the second instance is ignored, as it roughly has a similar complexity. Instead, the column 'smt' contains the solving time of `Z3` for the first proposition in seconds and we added some insights from the solving process itself: 'vars' counts the number of boolean variables, including predicates. 'prop' stands for the `propagations` counter that was incremented each time a variable was eliminated using the *unit propagation* rule from DPLL. 'dec' does the same for the *splitting* rule, and was taken from the counter `decisions`. 'conf' for `conflicts` counts the number of times the backtracking had to be invoked when the solver reached a conflicting variable assignment.

Our machine runs Ubuntu 20.04.1 on an Intel Core i5-3450 with 7.7 GB RAM. Some of the tests not only exceeded our time threshold but also used up enough memory to block the timeout of `Z3`, resulting in the test being continued beyond 600 seconds and occasionally in the process being killed by the operation system, once it requested more than 150 percent of physical memory. The latter cells of those tests were left blank.

As expected, comparing the results grouped by memory model shows us that the order SC, TSO, AARCH64 and Power correlates with the order of complexity. Having been formulated in terms of Alglave, Maranget, and Tautschnig [3], together with recursive in-flight/commit relations, a preserved program order and a happens-before relation, the Power model turned out to exhibit the least beneficial structure. The 'pre'-analysis timer, while also measuring the fixed point problem for static relationships, which are omitted on each first entry, also highly depends on the number of encoded tuples. The additional relation analysis results in a smaller encoding that is faster to build. In contrast to the majority of tests, only four configurations exhibited additional costs, attesting that our approach does not generally benefit all aspects of the verifier.

Note how MS (bound 2) under AARCH64 and 'futex' (bound 3) under Power got solvable in our testing setup before timed out, where about 30 percent of the encoded relationships were replaced. We attribute this to the shifting of propagating clauses into earlier iterations of DPLL mentioned in section 4.3. It seems that our setup exhibits a rough limit at about 100K encoded tuples for feasibly solving the encoding.

While the usage of static relationships noticeably reduces the number of encoded ones by an average of around 30 percent as well as the number of variables in the encoding, this does not always propagate to the reasoning. The column 'conf' features occasional increases in conflicts during the solving process, for instance TTAS under SC and MS under AARCH64 and Power. We suspect that since the encoding replaced many interconnected but simple rules, the decision making of Z3 is confused by the increased number of execution variables, biased towards satisfying static relationships when other variables would faster lead to a proof.

Figure 6.5.: Tests under Sequential Consistency

	k	pre	max	static	encoded	smt	vars	prop	dec	conf
futex	1	1.09	17540		14178	0.51	23K	407	0	1
		1.01	17510	4658	13107	0.54	21K	407	0	1
	2	2.67	33500		27954	1.7	50K	55	0	1
		2.43	33458	9443	25149	1.9	45K	55	0	1
	3	5.26	55973		47442	4.1	96K	55	0	1
		4.95	55919	16661	42021	3.4	85K	55	0	1
ttas	1	0.29	5267		3896	0.14	8655	100K	283	50
		0.28	5261	1559	3554	0.14	7778	92K	75	49
	2	0.55	10208		7856	0.6	19K	644K	697	175
		0.51	10196	2816	7106	0.6	17K	776K	1173	228
	3	1.07	17687		13919	1.9	36K	2.3M	3549	333
		0.94	17669	4928	12467	1.9	31K	2.3M	4622	390
t	1	0.31	4878		3403	0.17	7959	238K	3469	172
		0.25	4870	2004	2905	0.22	7102	210K	4006	174
	2	0.53	10734		8313	1.2	20K	1.5M	13K	617
		0.49	10718	4100	6993	1.1	17K	1.3M	20K	629
	3	1.08	18880		15413	3.5	39K	3.8M	28K	992
		1.07	18856	6984	12855	3.3	34K	3.4M	24K	1079
ms	1	0.64	12986		9439	0.91	22K	1.3M	11K	424
		0.61	12966	5126	8083	0.89	20K	1.0M	18K	411
	2	4.63	53448		42233	117	103K	121M	193K	14K
		3.83	53404	22856	34205	109	88K	88M	179K	14K
	3	31.1	177844		144669					
		33.2	177768	85618	111873					
hsy	1	0.45	8394		5926	0.28	13K	225K	6900	149
		0.38	8378	3572	5084	0.27	12K	269K	5447	154
	2	1.80	28658		22824	3.5	52K	2.9M	29K	540
		1.68	28618	11590	19044	2.6	46K	2.0M	41K	490
	3	16.3	107256		91762	96	246K	77M	368K	3678
		14.9	107168	42862	74768	80	208K	68M	385K	3785
dglm	1	0.73	14839		10048	2.06	26K	2.3M	35K	752
		0.71	14819	5981	8640	1.76	23K	1.7M	31K	629
	2	3.85	51685		39166	54.7	103K	55M	633K	7473
		3.61	51641	20853	32398	53.7	89K	45M	345K	7037
	3	28.0	146843		117480					
		23.5	146767	63055	94100					

Figure 6.6.: Tests under Total Store Ordering

	k	pre	max	static	encoded	smt	vars	prop	dec	conf
futex	1	1.05	31336		16281	0.60	25K	407	0	1
		1.09	31336	12589	14823	0.53	23K	407	0	1
	2	2.60	60886		33258	1.7	56K	55	0	1
		2.32	60886	24775	29217	2.3	52K	55	0	1
	3	6.06	103573		57570	7.3	108K	55	0	1
		5.53	103573	43384	49416	4.1	100K	55	0	1
ttas	1	0.32	9900		4456	0.16	9218	127K	198	57
		0.31	9900	4468	3926	0.14	8604	119K	201	59
	2	0.63	18669		9178	0.51	20K	834K	935	183
		0.59	18669	7678	8063	0.56	19K	811K	713	185
	3	1.07	32373		16540	2.1	39K	2.6M	2932	369
		1.03	32373	13015	14375	1.7	37K	2.2M	4844	346
t	1	0.28	9598		4002	0.27	8422	297K	5467	177
		0.26	9598	5492	3176	0.22	7288	201K	4542	158
	2	0.62	20688		10122	0.95	22K	1.3M	23K	514
		0.53	20688	11124	7856	0.97	19K	1.2M	17K	599
	3	1.23	36140		19142	3.2	43K	4.2M	38K	873
		1.06	36140	18910	14650	3.2	36K	3.6M	35K	984
ms	1	0.74	25033		11241	0.84	24K	1.4M	6144	422
		0.63	25033	13577	9007	0.77	21K	903K	15K	359
	2	4.50	107389		55035	119	118K	138M	124K	15K
		4.19	107389	60163	39485	80	98K	83M	168K	11K
	3	47.1	378797		202421					
		39.0	378797	227345	132109					
hsy	1	0.42	16398		6813	0.30	14K	350K	5336	182
		0.40	16398	9624	5408	0.24	12K	210K	7669	132
	2	1.97	54890		27447	4.0	56K	3.6M	33K	681
		1.72	54890	30676	20736	2.7	47K	3.0M	56K	579
	3	15.3	205160		114475	86.0	267K	79M	435K	3548
		15.2	205160	112920	82806	65.2	221K	63M	392K	3486
dglm	1	0.768	28920		11964	1.88	27K	2.5M	66K	735
		0.678	28920	15980	9658	1.84	24K	2.5M	57K	841
	2	4.62	101712		49878	51.4	115K	58M	355K	7273
		4.10	101712	54392	37724	46.3	100K	50M	517K	6781
	3	29.0	299112		157692					
		27.0	299112	163686	112952					

Figure 6.7.: Tests under AARCH64

	k	pre	max	static	encoded	smt	vars	prop	dec	conf
futex	1	2.47	50187		21259	2.5	66K	473	0	1
		2.26	50187	22948	18066	2.1	57K	492	0	1
	2	5.80	104367		43843	14	217K	166	0	1
		5.61	104367	51250	35508	20	178K	295	0	1
	3	13.5	183336		76177	128	545K	211	0	1
		12.5	183336	93952	60105	28	439K	475	0	1
ttas	1	0.68	15196		6341	0.55	27K	292K	245	36
		0.64	15196	7335	4948	0.49	22K	276K	104	40
	2	1.21	28975		12482	2.2	69K	3.0M	736	175
		1.17	28975	13221	9931	2.4	59K	2.6M	860	190
	3	2.24	50536		21761	7.6	152K	12M	2572	413
		2.19	50536	22869	17515	6.6	132K	11M	1762	396
t	1	0.68	16190		6033	0.85	27K	1.2M	5068	198
		0.49	16190	9523	4013	0.62	20K	839K	4408	222
	2	1.46	35414		14919	7.2	88K	7.2M	16K	584
		1.40	35414	20027	10023	3.8	66K	5.6M	15K	582
	3	3.25	62490		27931	20	207K	30M	32K	1081
		2.90	62490	34861	18783	15	150K	19M	26K	1053
ms	1	1.77	42193		16431	5.8	96K	7.5M	6440	397
		1.67	42193	24254	11407	5.0	70K	5.2M	11K	492
	2	14.3	198151		80995					
		11.5	198151	116158	51113	403	602K	576M	120K	9508
	3	113	727341		298045					
		127	727341	443000	173185					
hsy	1	1.06	26879		9935	1.7	50K	1.6M	6646	193
		1.01	26879	15650	6942	1.4	38K	1.2M	4866	170
	2	5.49	96283		40013	41	341K	36M	14K	752
		5.03	96283	55376	27102	33	249K	23M	41K	715
	3	52.8	379197		167467					
		49.8	379197	220972	109458					
dglm	1	1.99	45770		17194	9.47	105K	13M	46K	868
		1.68	45770	25563	12114	7.13	79K	8.8M	43K	739
	2	11.9	176556		71876	335	744K	483M	194K	7250
		11.1	176556	97667	48100	211	538K	279M	271K	6297
	3	74.2	544572		228774					
		77.0	544572	308605	145588					

Figure 6.8.: Tests under Power

	k	pre	max	static	encoded	smt	vars	prop	dec	conf
futex	1	4.97	79064		40988	23	110K	995	0	1
		4.60	79064	31391	32527	12	93K	648	0	1
	2	15.9	177422		96626	113	388K	8065	0	1
		13.8	177422	77297	71482	92	306K	664	0	1
	3	42.1	328862		183035					
		37.1	328862	150338	132424	218	767K	1524	0	1
ttas	1	1.30	21759		11985	2.2	77K	1.2M	886	55
		1.11	21759	8445	9393	1.7	65K	1.0M	174	52
	2	2.99	43971		25941	8.0	222K	8.5M	928	201
		2.85	43971	16107	19986	10.4	186K	8.3M	1415	218
	3	7.13	81582		50301	53	558K	41M	7524	392
		6.27	81582	29589	37998	28	467K	33M	3767	380
t	1	1.50	29353		14632	3.9	86K	3.7M	3712	240
		1.29	29353	13631	10092	2.6	64K	2.6M	7513	195
	2	4.31	70543		38526	33	330K	45M	17K	728
		3.72	70543	34483	25160	25	232K	29K	26K	704
	3	10.1	130215		73924	132	826K	119M	19K	1209
		7.97	130215	66133	47128	73	564K	79M	39K	1054
ms	1	4.55	84463		41774	26	318K	19M	13K	402
		3.93	84463	41587	27444	24	232K	21M	13K	484
	2	62.5	475911		254282					
		46.2	475911	265867	148120					
	3	>24	1937897		1066304					
		>25	1937897	1167569	584668					
hsy	1	2.37	52378		25176	8.1	159K	7.0M	6910	227
		2.20	52378	24258	17526	7.1	122K	5.0M	11K	193
	2	18.5	207384		111898	248	1.3M	164M	27K	839
		14.3	207384	104038	72976	194	944K	99M	30K	657
	3	190	861552		493738					
		158	861552	464822	305690					
dglm	1	5.17	90974		43333	42.9	361K	42M	37K	756
		3.91	90974	42574	28869	25.9	264K	32M	39K	758
	2	50.2	402004		214445					
		38.5	402004	203360	132819					
	3	3111	1342436		749427					
			1342436	729328	444575					

Conclusion

The kernel of memory-model-aware verification consists of two decisions: Which assertion to violate and for each involved load, which store to read from, in order to enable the violation. Specialised algorithms exist for SC, TSO, as well as ARM and Power, and are based on an operational model. In a generalised approach, the axiomatic model restricts combinations of inter-thread communication in a well-structured manner, that is non-the-less difficult to automatically reverse-engineer.

Dartagnan heavily relies on boundedness of the program, presenting a quantifier-free boolean formula modulo at least some theory supporting arithmetics on some address space, like integers or bit vectors. Boundedness establishes a kind of relation analysis, a sound over-approximation of the set of possible and relevant relationships between events of the program.

Motivated by the usage of must-components in other verifiers, we have introduced another relation analysis to Dartagnan, under-approximating the set of static relationships which only require both participating events to be executed, effectively reducing the set of to-be encoded relationships and shifting their representing variables to be assigned a truth value into an earlier iteration of a DPLL-based solver. This corresponds to prepending a part of the solving process and supplying lemmas for the remainder based entirely on structural properties of the input.

We have modified the already present control-flow-, dependency and pointer analyses in order to present their results for successive tasks and to access those relevant structural properties. The under-approximation performed by our tool has been proven correct in this thesis.

Evaluation on some configurations show that the modified formulas can sometimes confuse the solving process, potentially resulting in even greater resource consumption. We will further examine this factor in the context of DPLL to understand its character and enable the development of a more performant verifier.

To uncover more on this field, we mentioned in section 4.1.4 that pointer analysis may be suggestible to memory-based information, enabling a higher precision for this approximative approach. In section 4.2, we studied the possibility of verification-exclusive slicing, once the pointer analysis reaches a satisfying level of precision. Those methods promise scalability to the verifier for larger programs and will be subject of our future reseach.

Bibliography

[1] Jade Alglave. "A Shared Memory Poetics". PhD thesis. Université Paris 7, 2010.

[2] Jade Alglave, Patrick Cousot, and Luc Maranget. "Syntax and semantics of the weak consistency model specification language cat". In: (Aug. 2016).

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory". In: *CoRR* abs/1308.6810 (2013). eprint: 1308.6810. URL: http://arxiv.org/abs/1308.6810.

[4] Jade Alglave et al. "Frightening Small Children and Disconcerting Grown-Ups: Concurrency in the Linux Kernel". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 405–418. ISBN: 9781450349116. DOI: 10.1145/3173162.3177156.

[5] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, May 1994.

[6] C. Barrett and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. 2018.

[7] Bernhard Beckert et al. "Using Relational Verification for Program Slicing". In: *Software Engineering and Formal Methods*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Springer International Publishing, 2019, pp. 353–372. ISBN: 978-3-030-30446-1.

[8] Dirk Beyer. *Software Verification: 10th Comparative Evaluation (SV-COMP 2021)*. 2021. URL: https://sv-comp.sosy-lab.org/2021.

[9] *Dat3M: Memory Model Aware Verification*. URL: github.com/hernanponcedeleon/Dat3M.

[10] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[11] Simon Doherty et al. "Formal Verification of a Practical Lock-Free Queue Algorithm". In: vol. 3235. Sept. 2004, pp. 97–114. ISBN: 978-3-540-23252-0. DOI: 10.1007/978-3-540-30232-2_7.

[12] H. Franke, T. J. Watson, and M. Kirkwood. "Fuss , Futexes and Furwocks : Fast Userlevel Locking in Linux Hubertus Franke IBM". In: 2005.

[13] N. Gavrilenko et al. “BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings”. In: *Dillig I., Tasiran S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science, vol 11561. Springer, Cham.* (2019). DOI: 10.1007/978-3-030-25540-4_19.

[14] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. “SAT modulo Graphs: Acyclicity”. In: Aug. 2014. ISBN: 978-3-319-11557-3. DOI: 10.1007/978-3-319-11558-0_10.

[15] Danny Hendler, Nir Shavit, and Lena Yerushalmi. “A scalable lock-free stack algorithm”. In: *SPAA 2004: Proceedings of the 16th annual ACM symposium on parallelism in algorithms and architectures*. ACM, 2004.

[16] John B. Kam and Jeffrey D. Ullman. “Monotone data flow analysis frameworks”. In: *Acta Informatica, Issue 3, Volume 7, Page 305-317* (1977). ISSN: 1432-0525. DOI: 10.1007/BF00290339.

[17] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. “Model Checking for Weakly Consistent Libraries”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 96–110. ISBN: 9781450367127. DOI: 10.1145/3314221.3314609.

[18] Markus Kusano and Chao Wang. *Thread-Modular Static Analysis for Relaxed Memory Models*. 2017. eprint: 1709.10077.

[19] Leslie Lamport. “A Fast Mutual Exclusion Algorithm”. In: *ACM Transactions on Computer Systems 5, 1 (February 1987), 1-11. Also appeared as SRC Research Report 7.* (Nov. 1985), pp. 1–11. URL: https://www.microsoft.com/en-us/research/publication/fast-mutual-exclusion-algorithm/.

[20] Leslie Lamport. “How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor”. In: *IEEE Trans. Comput.* 46.7 (July 1997), pp. 779–782. ISSN: 0018-9340. DOI: 10.1109/12.599898.

[21] K. Leino and M. Rustan. “This is Boogie 2”. June 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[22] Luc Maranget, Susmit Sarkar, and Peter Sewell. *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*. Aug. 2012. URL: http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

[23] Michael and Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. In: *PODC* (1996).

[24] Hernán Ponce de León et al. “BMC with Memory Models as Modules”. In: *Formal Methods in Computer Aided Design (FMCAD)* (2018), pp. 1–9.

[25] Hernán Ponce de León et al. “Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution)”. In: Feb. 2020.

[26] Hernán Ponce de León et al. “Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models”. In: (Aug. 2017).

[27] Hernán Ponce-de-León, Thomas Haas, and Roland Meyer. “Dartagnan: Leveraging Compiler Optimizations and the Price of Precision (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II* 12652 (Feb. 2021), pp. 428–432. URL: https://europepmc.org/articles/PMC7984541.

[28] *Programming Languages - C*. 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[29] Zvonimir Rakamaric and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7.

[30] Thomas Reps. “Undecidability of Context-Sensitive Data-Dependence Analysis”. In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 162–186. ISSN: 0164-0925. DOI: 10.1145/345099.345137.

[31] Susmit Sarkar et al. “Synchronising C/C++ and POWER”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: Association for Computing Machinery, 2012, pp. 311–322. ISBN: 9781450312059. DOI: 10.1145/2254064.2254102.

[32] Susmit Sarkar et al. “Understanding POWER Multiprocessors”. In: *PLDI* (2011).

[33] *SMT-LIB The Satisfiability Modulo Theories Library: Logics*. URL: smtlib.cs.uiowa.edu/logics.shtml.

[34] Bolesław K. Szymański. “A simple solution to Lamport’s concurrent programming problem with linear wait”. In: *Proceedings of the 2nd international conference on Supercomputing - ICS ’88*. ACM, June 1988, pp. 621–626. DOI: 10.1145/55364.55425.

[35] R. K. Treiber. *Systems programming: Coping with parallelism.* Tech. rep. RJ 5118. IBM Almaden Research Center, 1986.

[36] Viktor Vafeiadis et al. “Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do about It”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. Mumbai, India: Association for Computing Machinery, 2015, pp. 209–220. ISBN: 9781450333009. DOI: 10.1145/2676726.2676995.

A. Storage Device

The storage device contains a git repository `Dat3M`, containing a copy of our fork of the project [9]. The branches `final-no` and `final-yes` define the configurations that were evaluated and compared in chapter 6, one without and one with analysis of static relationships, respectively.

Besides the programmatic appendix, the directory `evaluation` contains copies of all measurements taken during the course of this thesis. The subdirectory `2021-06-15` especially contains the logs that the presented tables base on. The program and model definition files can also be found there.

In directory `ref` there is a collection of scientific articles and related work that were consulted for this thesis.

The git repository `thesis` maintains the source code of this document.