Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Artem YUSHKOVSKIY

Automated Analysis of Weak Memory Models

Master's Thesis
Espoo, ?.?.2018

Supervisor: Assoc. Prof. Keijo Heljanko
Instructor:

Aalto University
School of Electrical
Engineering

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF
MASTER'S THESIS

Author:	Artem YUSHKOVSKIY		
Title:	Automated Analysis of Weak Memory Models		
Date:	?.?.2018	**Pages:**	vi + 27
Professorship:		**Code:**	AS-116
Supervisor:	Assoc. Prof. Keijo Heljanko		
Instructor:			

In id fringilla velit. Maecenas sed ante sit amet nisi iaculis bibendum sed vel elit. Quisque eleifend lacus nec ipsum lobortis ornare. Nam lectus diam, facilisis eget porttitor ac, fringilla quis massa. Phasellus ac dolor sem, eget varius lacus. Sed sit amet ipsum eget arcu tristique aliquam. Integer aliquam velit sit amet odio tempus commodo. Quisque commodo lacus in leo sagittis vel dignissim quam vestibulum. Cras fringilla velit et diam dictum faucibus. Pellentesque at eros non mauris auctor euismod. Nullam convallis arcu vel lectus sollicitudin rutrum. Praesent consequat, nisl at pretium posuere, neque arcu dapibus lacus, ut sollicitudin elit velit ultricies libero. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

Nulla semper hendrerit molestie. Pellentesque blandit velit sit amet est vestibulum faucibus. Nullam massa turpis, venenatis non mollis fringilla, mattis et diam. Fusce molestie convallis elementum. Morbi nec lacus dapibus arcu mollis gravida. Aliquam erat volutpat. Nam vitae magna nunc. Nunc ut ipsum at massa porttitor vestibulum. Praesent diam lorem, ultrices nec vestibulum id, volutpat nec lacus.

Keywords:	Thesis template, master's thesis
Language:	English

Aalto-yliopisto
Sähkötekniikan korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
???

???

Tekijä:	Artem YUSHKOVSKIY		
Työn nimi: ?			
Päiväys:	?.?.2018	**Sivumäärä:**	vi + 27
Professuuri:	?	**Koodi:**	AS-116
Valvoja:	Assoc. Prof. Keijo Heljanko		
Ohjaaja:			

Cras tincidunt bibendum erat, vel tincidunt diam porttitor aliquam. Donec sit amet urna non felis placerat pharetra. Aenean ultrices facilisis nulla vitae semper. Nullam non libero quis dui fermentum aliquam id vel eros. Praesent elementum tortor quis sem congue iaculis sit amet eget nisl. Quisque erat tortor, condimentum eu volutpat et, blandit et augue. Phasellus erat turpis, pretium non feugiat id, posuere id velit. Vestibulum ut sapien felis, quis convallis dui.

In elementum est eu nulla hendrerit feugiat. In sodales diam vel lacus cursus tincidunt. Morbi nibh dui, imperdiet non vestibulum non, dignissim id risus. Sed sollicitudin neque lectus, porttitor sollicitudin elit. Nulla facilisi. Nullam in ante eu mi suscipit sollicitudin. Sed est velit, gravida facilisis varius eget, tempus sed urna. Aliquam erat volutpat. Nam semper condimentum nisi. Nullam scelerisque, metus nec sodales vulputate, purus augue venenatis urna, sit amet mattis turpis nisl ac metus. Mauris nec odio ut neque condimentum vulputate vel in turpis. Nulla facilisi. Nulla id tellus sapien, vitae blandit lorem.

Asiasanat:	Diplomityöpohja
Kieli:	Englanti

Acknowledgements

In id fringilla velit. Maecenas sed ante sit amet nisi iaculis bibendum sed vel elit. Quisque eleifend lacus nec ipsum lobortis ornare. Nam lectus diam, facilisis eget porttitor ac, fringilla quis massa. Phasellus ac dolor sem, eget varius lacus. Sed sit amet ipsum eget arcu tristique aliquam. Integer aliquam velit sit amet odio tempus commodo. Quisque commodo lacus in leo sagittis vel dignissim quam vestibulum. Cras fringilla velit et diam dictum faucibus. Pellentesque at eros non mauris auctor euismod. Nullam convallis arcu vel lectus sollicitudin rutrum. Praesent consequat, nisl at pretium posuere, neque arcu dapibus lacus, ut sollicitudin elit velit ultricies libero. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

Espoo, ?.?.2018

Artem YUSHKOVSKIY

Abbreviations

LI	Lorem Ipsum
ABC	Quisque et mi lacus, nec porta ante.
DEF	Proin pellentesque accumsan laoreet

Contents

Chapter 1

Introduction

Most modern computer systems contain large parts that operate concurrently. Though parallelisation of the system can improve its performance drastically, it opens numerous of problems connected to correctness, robustness and reliability, which makes the concurrent program design one of the most difficult problems of programming [McK17].

Traditionally, studies related to concurrent programming concern on more fundamental theoretical questions of designing race-free and lock-free parallel algorithms, asynchronous data structures and synchronisation primitives of a programming language. Unfortunately, when it comes to the real-world concurrent programs, the algorithmic level of abstraction is not enough for guaranteeing their properties of correctness and reliability. The reasons of this fact lie in the code optimisations that both compiler and hardware perform in order to increase performance as much as possible. For instance, Figure 1.1 provides simple example of reachability of the state '`(0:EAX=0 /\ 1:EAX=0)`' on x86 machines (such little examples that illustrate specific behaviour of a WMM are called *litmus tests*). This state is allowed because in x86 architecture each processor may cache the write to shared memory variable into its local write buffer, so that they do not become visible by other processes immediately. In the example, the write '`MOV [x],1`' performed by process `P0` stores value `1` to the shared variable `[x]` into the write buffer of process `P0`. Meanwhile, the write cache of the process `P1` may not have updated version of the variable `[x]`, neither may have the main memory, so that the read '`MOV EBX,[x]`' performed in the process `P1` may read the initial value `0` even if this variable has been already updated in another thread. These problems have lead to the need for formalisation of

{ x=0; y=0; }	
P0	P1
MOV [x],1 MOV EAX,[y]	MOV [y],1 MOV EAX,[x]
exists (0:EAX=0 /\ 1:EAX=0)	
x86-TSO: allow	

Figure 1.1: *Store buffering (SB): a litmus test on write-read reordering allowed under the x86-TSO and forbidden under the SC memory model*

semantics of memory operations within different concurrent architectures defined by *weak memory models (WMM)*.

Research of weak memory models firstly aims to *formalise* develop the formal approach of understanding programs with respect to weak memory models which is systematic, sound and complete. The first (and so far the only) such a framework was presented in 2010 [Alg10]. In addition to developing rather theoretical basis, researchers work on extracting the WMMs for hardware architectures from existing implementations of from their specifications, which are written in natural language and thus suffer from ambiguities and incompleteness. Over last decade the memory models have been defined for most mainstream multiprocessor architectures, such as x86-TSO and Sparc-TSO (for *Total Store Order*) model for x86 and Sparc architecture formalised in 2009 [OSS09], much more relaxed memory model for Power and ARM architectures [AMT14] [SSA+11] [AFI+09], and others. There are projects for validating hardware architectures wrt. a memory model, e.g. [LSM+16] [LPM14].

Most modern high-level programming languages rely on relaxed memory model as well. Thus, the memory model for Java is based on the *happens-before* principle [Lam78], it was introduced in J2SE 5.0 in 2004 [MPA05]; the C++11 standard [ISO12] has introduced the set of hardware-independent synchronisation fences and atomic operations, whenever the C++17 memory model [BOS+11] is based on the relation *strongly happens-before*. Weak memory are being formalised for even more abstract software environments, the notable project in this area is the project on formalising the Linux kernel memory model, which is being actively developing these days [MAM+17]. Furthermore, there is a wide range of tools that perform program verification wrt memory models (see [AKN+13], [LFH+17]).

The first memory model for concurrent systems was formulated by Leslie Lamport back in 1979 [Lam79]. This memory model, called the *sequential consistency (SC)*, allows only those executions (interleavings) that produce the same result as if the operations had been executed by single process. This means that the order of operations executed by a process is strictly defined by the program it executes. The SC model does requires the write to a shared variable performed in one process to become visible by all other processes not instantly, but simultaneously. This means each process communicates to the shared memory directly, without local buffering. Another important requirement of SC memory model is that it forbids memory operations reordering within single process (the order is strictly defined by the program).

The SC model is considered to be the strong memory model in the sense that it provides strong guarantees regarding the ordering and caused effect of memory operations. Different relaxations of this model lead to the class of *weak memory models (WMM)*. They specify how threads interact through shared memory, when a write becomes visible to other threads and what value a read can return. Therefore, WMMs serve as set of guarantees made by designers of execution environment (hardware, programming language, compiler, database, operation system, etc.) to programmers on which behaviours of their concurrent code they may expect.

Although weak memory studies is rather young research area, there exist frameworks and tools for exploring WMMs and examining simple programs with respect to the them. The state-of-the-art tool is `diy` (for *do it yourself*), developed by the researchers from INRIA institute, France and University of Cambridge, UK. The `diy` [1] is a software suite for designing and testing weak memory models. It is firstly released back in 2010, and since that time it remained to be the only tool for testing weak memory models. The `diy` consists of several modules: the litmus tests generators `diy`, `diycross` and `diyone`, the litmus tests concrete executor `litmus` that runs tests on a physical machine while collecting its behaviours, and the weak memory models simulator `herd` that implements reachability analysis for exploring states reachable under specified WMM.

All the `diy` tools work only with single memory model, however, in real life we face serious engineering problems involving necessity to model more than one execution environment. One of these problems is the *portability* of the program from one hardware architecture to another. A program written

[1] Project web site: `http://diy.inria.fr/`

in a high-level language is then compiled for different hardware. Even if all the compiler optimisations were disabled (which is rare case nowadays), the behaviour of two compiled versions of the same program may differ due to differences between hardware memory models. As the result, a program compiled under the platforms T can reach states that are unreachable on the platform S, which is a *portability bug* from the source platform S to the target platform T [LFH+17].

The first tool that performs the WMM-aware portability analysis is `PORTHOS` introduced in April 2017 [LFH+17]. This tool reduces described problem to a bounded reachability problem, which can be solved with help of an SMT-solver. This approach allows to capture symbolically the semantics of analysing program and both weak memory models into single SMT-formula, augmented by the reachability assertion. As most modern SMT-solvers are efficient enough to be able to operate the state space of size millions of variables bounded by millions of constraints ([MZ09]), the used method can be applicable in solving the real-world problems.

Current work aims to rework the proof-of-concept tool `PORTHOS` by extending the input language, which currently represents the minimum subset of C, and revising the general architecture of the tool in order to enhance performance, reliability and maintainability. We called the new tool the `mousquetaires` framework as now it not only performs the portability analysis, but can serve as the basis for SMT-solver driven weak memory model analysis.

1.1 Thesis structure

The thesis is organised as following. Chapter 2 gives a general view on the weak memory model-aware analysis. Chapter ...

Chapter 2

Memory model-aware analysis

In general, analysis of concurrent programs with respect to axiomatic memory models is performed in several stages. Firstly, the control-flow and data-flow of a program is encoded as the set of possible *candidate executions*. Obtained model of the program is called an anarchic semantics, which is a truly parallel semantics with no global time that describes all possible computations with all possible communications [ACM16]. Thereafter, the anarchic semantics is constrained by the *weak memory model* specification which is a set of axiomatic constraints for filtering out executions inconsistent in particular architecture.

2.1 The event-based program representation

The classical approach for modeling concurrent programs is to use the *global time*, a single order of interleavings among all events happened in different threads. Although these models are easy to understand, it may be impossible to process *all* possible states, number of which is exponentially large. However, there exist equivalence classes such that the result of execution different interleavings from single equivalence class is the same (for instance, computations performed by a processor locally do not affect the global state). One such model is the *event-based* representation of a program, which models a program as a directed graph of events (the *event-flow graph*). The vertices of such a graph represent *events* (independent low-level instructions; see Section 2.1.1), and edges represent *relations* over the events (see Section 2.1.2).

2.1.1 Events

A *memory event* $e_m \in \mathbb{E}$ represents the fact of access to the memory. Since memory is the crucial low-level resource shared by multiple processes, most relations are defined over memory events. The processes can access a shared memory location (denoted by l_i, for *location*), or a local one (denoted by r_i, for *register*). A memory event can access at most one shared memory location, high-level instructions that address more than one shared variable must be transformed into a sequence of events. A memory event is specified by its direction with respect to the shared variable, its location $\mathtt{loc}(e_m)$, its processor label $\mathtt{proc}(e_m)$, and a unique event label $\mathtt{id}(e_m)$ [Alg10].

The set of memory events $\mathbb{M}$ is divided into write events $\mathbb{W}$ (that write values to shared-memory locations) and read events $\mathbb{R}$ (that read values stored in shared-memory locations). We add a restriction that each memory event uses at most one shared location, so that the write instruction $i = write(l_1, l_2)$, that encodes the write from the shared location l_2 to the shared location l_1, is represented as two consequent events $e_1 = \mathtt{load}(r_1 \leftarrow l_2)$; $e_2 = \mathtt{store}(l_1 \leftarrow r_1)$. Also, it is important to separate the set of initial write events $\mathbb{IW} \subset \mathbb{W}$ that perform initialisation of program variables.

A *computation event* $e_c \in \mathbb{C} \subseteq \mathbb{E}$, represents a low-level assembly computation operation performed solely on local-memory arguments. An example of computation event may be the event $e_c = r_1 \leftarrow add(r_2, 1)$ that writes the sum of values stored in register r_2 and constant 1 (which is modelled as a register as well) to the register r_1. For modelling branching statements, we distinguish the set $\mathbb{C}_! \subseteq \mathbb{C}$ of *predicative* computation events (also called as *branching events*), that are evaluated as a boolean value.

The synchronisation instructions (fences) cause the *barrier events*, that do not perform any computation or memory value transfer, instead, they add new relations to the program model that restrict the set of allowed behaviours. Functionally, a fence may be a synchronisation barrier or a instruction of flushing the local memory caches, etc.

2.1.2 Relations

The relation $\xrightarrow{r}: \mathbb{E} \times \mathbb{E}$ is a binary function over events (set of pairs of events). There are two kinds of relations between events: *basic relations* that capture semantics of the program, and *derived relations* that are defined from the basic relations and events in the weak memory model specification.

Constraints over relations that are specified by weak memory models are defined as requirements of acyclicity, irreflexivity or emptiness of specific relations [ACM16].

The basic relations are the following [Alg10]:

- The *control-flow* of a program is defined by the *program-order* relation $\mathtt{po} \subset \mathbb{E} \times \mathbb{E}$, which represents the total order of events of same process. For instance, if the instruction i_1 generates the event e_1 and the instruction i_2 follows i_1 and generates the event e_2, then $e_1 \overset{\mathtt{po}}{\rightarrow} e_2$.
- The *data-flow* of a program is defined by *communication relations*:
  - the *read-from* relation $\mathtt{rf} \subset \mathbb{W} \times \mathbb{R}$ that maps each write event to the read event that reads its value;
  - the *coherence order* relation $\mathtt{co} \subset \mathbb{W} \times \mathbb{W}$ that defines the total order on writes to the same location across all processes (also called the *write serialisation* ws-relation);
- Events from the same process are related by the *scope relation* $\mathtt{sr} \subset \mathbb{E} \times \mathbb{E}$. In contrast to the `herd` tool, the `mousquetaires` does not use hierarchy of scopes (depicted as the scope tree); instead, it uses simple lables that indicate which process has produced certain event.

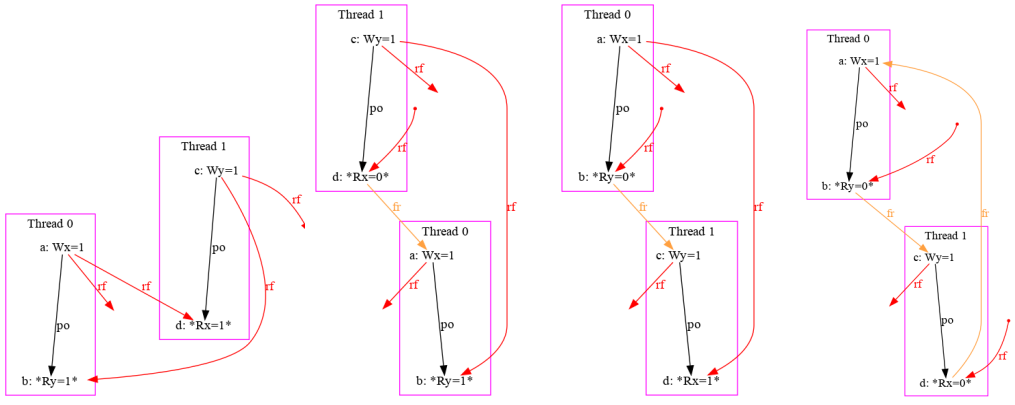
Below we enumerate some derived relations [Alg10]:

- the *from-read* relation $\mathtt{fr} \subset \mathbb{R} \times \mathbb{W}$ that maps a read to writes preceding the write event from which the read reads the value: $r \overset{\mathtt{fr}}{\rightarrow} w = (\exists w'.w' \overset{\mathtt{rf}}{\rightarrow} r \wedge w' \overset{\mathtt{co}}{\rightarrow} w)$.
- the *data dependency* relation dp, which is a subset of po-relation that always has a read at its source (it connects the read to the write which it depends on).
- the *external* (and *internal*) *read-from* relations that restrict the rf-relation to the different (respectively, same) processes.
- the po-loc relation that is the po-relation over events that access to the same shared variable: $m_1 \overset{\mathtt{po-loc}}{\rightarrow} m_2 = (m_1 \overset{\mathtt{po}}{\rightarrow} m_2 \wedge \mathtt{loc}(m_1) = \mathtt{loc}(m_2))$.
- the semantics of fences and barriers specific for different architectures may be defined as derived relations.

The work [Alg10, Chapter 2] provides definition of basic properties of relations, such as *reflexivity* and *irreflexivity*, *transitivity* and *transitive closure*, *acyclicity*. Thereafter, weak memory models make asserts over these properties, thus restricting the set of allowed behaviours of the system.

2.1.3 Executions

The *candidate execution* is a path in the event-flow graph defined by `po`- and `rf`-relations and set of final writes to a given memory location that is valid under certain memory model [AMT14]. Figure 2.1 illustrates four possible candidate executions for the litmus test Example 1.1 (the pictures are generated by the `herd7` tool, version 7.47). Since there are no conditional jumps, the `po`-relation is defined and we do not need to guess it. Since each thread performs single write followed by a single read, the `co`-relation is also defined (it relates the initial write event with the write event to the same location). Thus, there are only four possible executions defined by the choice of `rf`-relation. The candidate executions pictured in Figures 2.1a–2.1c are consistent both under strong memory model SC and under relaxed memory models x86-TSO, Power, ARM, and some others. However, the execution shown in Figure 2.1c is still consistent under relaxed-memory architectures, but it becomes inconsistent under SC architecture as it forbids cycles over $\texttt{fr} \cup \texttt{po}$.



(a) *Final state:* `(0:EAX=1 /\ 1:EAX=1)` **(b)** *Final state:* `(0:EAX=1 /\ 1:EAX=0)` **(c)** *Final state:* `(0:EAX=1 /\ 1:EAX=1)` **(d)** *Final state:* `(0:EAX=0 /\ 1:EAX=0)`

Figure 2.1: *Possible candidate executions for the litmus test Example 1.1*

2.2 The cat language

Weak memory models are defined via the cat language [ACM16]. This is a domain specific language for describing consistency properties of concurrent programs. The cat language combines expressive power of a functional language (it is inspired by OCaml and adopts its types, first-class functions, pattern matching and other features) with types, expressions and assertions that are specific for operating with relations and executions.

The derived relations can be defined via the keyword `let` and the following operations over relations [ACM16]:

- *the union* of two relations `r1` and `r2` is `r1 | r2`
- *the intersection* of two relations `r1` and `r2` is `r1 & r2`
- *the difference* of two relations `r1` and `r2` is $\texttt{r1} \setminus \texttt{r2}$
- *the sequence*[1] of two relations `r1` and `r2` is `r1;r2`

For instance, the `fr`-relation is defined as follows: $\texttt{fr} = (\texttt{rf}^{-1}; \texttt{co})$.

Figure 2.2 contains part of x86-TSO model [OSS09] that asserts acyclicity of communication relation and `po-loc` relation:

```
...
let com = rf | fr | co
let po-loc = po & loc
acyclic po-loc | com
```

Figure 2.2: *Excerpt from the x86-TSO memory model in* `cat` *language*

[1] The sequence of two relations `r1` and `r2` is defined as the set of pairs `(x,y)` such that there exists an intervening `z`, such that $\texttt{(x,z)} \in \texttt{r1}$ and $\texttt{(z,y)} \in \texttt{r2}$

Chapter 3

Portability analysis as a SAT problem

As it has been discussed in Chapter 1, the program may behave differently when compiled for different parallel hardware architectures. This may cause the portability bugs, the behaviour that is allowed under one architecture and forbidden under another. In this Chapter, we describe the general task of analysing the concurrent software portability as a *bounded reachability* problem, which in turn can be reduced to a SAT problem [LFH+17] (more precisely, to an SMT problem).

3.1 Model checking and reachability analysis

The model checking is the problem of verifying the system (the model) against the set of constraints (the specification). As the state machine model is the most widespread mathematical model of computation, most classical model checking algorithms explore the state space of a system in order to find states that violate the specification. The general schema of model checking is the following: firstly, the analysing system is being represented as a transition system, a finite directed graph with labeled nodes representing states of the system such that each state corresponds to the unique subset of atomic propositions, that characterise the behavioral properties of each state. Then, the system constraints are being defined in terms of a modal temporal logic with respect to the atomic propositions. Commonly, the Linear Temporal Logic (LTL) or Computational Tree Logic (CTL), along with their extensions, are used as a specification language due to the expressiveness and verifiability of their statements. In the described schema,

the model checking problem is reducible to the reachability analysis, an iterative process of a systematic exhaustive search in the state space. This approach is called *unbounded model checking (UMC)*.

However, all model checking techniques are exposed to the *state explosion problem* as the size of the state space grows exponentially with respect to the number of state variables used by the system (its size). In case of modeling concurrent systems, this problem becomes much more considerable due to exponential number of possible interleavings of states. Therefore, the research in model checking over past 40 years was aimed at tackling the state explosion problem, mostly by optimising search space, search strategy or basic data structures of existing algorithms.

One of the first technique that optimises the search space considerably major was the symbolic model checking with binary decision diagrams (BDDs). Instead of by processing each state individually, in this approach the set of states is represented by the BDD, efficient data structure for performing operations on large boolean formulas [CKN+12]. The BDD representation can be linear of size of variables it encodes if the ordering of variables is optimal, otherwise the size of BDD is exponential. The problem of finding such an optimal ordering is known as NP-complete problem, which makes this approach inapplicable in some cases.

The other idea is to use satisfiability solvers for symbolic exploration of state space [CBR+01]. In this approach, the state space exploration consists of sequence of queries to the SAT-solver, represented as boolean formulas that encode the constraints of the model and the finite path to a state in the corresponding transition system. Due to the SAT-solver. This technique is called *bounded model checking (BMC)*, because the search process is being repeated up to user-defined bound k, which may result to incomplete analysis in general case. However, there exist numerous techniques for making BMC complete for finite-state systems (e.g., [Sht00]).

3.2 Portability analysis as a bounded reachability problem

In general, a BMC problem aims to examine the reachability of the "undesirable" states of a finite-state system. Let $\vec{x} = (x_1, x_2, ..., x_n)$ be a vector of n variables that uniquely distinguishes states of the system; let $Init(\vec{x})$ be an *initial-state predicate* that defines the set of initial states of the system; let $Trans(\vec{x}, \vec{x}')$ be a *transition predicate* that signifies whether there the

transition from state $\vec{x}$ to state $\vec{x}'$ is valid; let $Bad(\vec{x})$ be a *bad-state predicate* that defines the set of undesirable states. Then, the BMC problem, stated as the reachability of the undesirable state withing k steps is formulated as following: $\mathsf{SAT}(Init(\vec{x_0}) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Trans(\vec{x}_{k-1}, \vec{x}_k) \wedge Bad(\vec{x}_k))$.

Portability analysis problem may also be stated as a reachability problem, where the undesirable state is the state reachable under the target $\mathcal{M}_\mathcal{T}$ memory model and unreachable under the source memory model $\mathcal{M}_\mathcal{S}$. However, unlikely the BMC problem, the portability analysis does not require to call the SMT-solver repeatedly, since (imperative) programs may be converted as acyclic state graph (by reducing the loops, see Section 5.4.1) and the $Trans$ predicate may be stated only for the final state of a program.

Consider the function $cons_\mathcal{M}(P)$ calculates the set of executions of program P consistent under the memory model $\mathcal{M}$. Then, the program P is called portable from the source architecture (memory model) $\mathcal{M}_\mathcal{S}$ to the target architecture $\mathcal{M}_\mathcal{T}$ if all executions consistent under $\mathcal{M}_\mathcal{T}$ are consistent under $\mathcal{M}_\mathcal{S}$ [LFH+17]:

Definition 3.2.1 (Portability)**.** Let $\mathcal{M}_\mathcal{S}$, $\mathcal{M}_\mathcal{T}$ be two weak memory models. A program P is portable from $\mathcal{M}_\mathcal{S}$ to $\mathcal{M}_\mathcal{T}$ if $cons_{\mathcal{M}_\mathcal{T}}(P) \subseteq cons_{\mathcal{M}_\mathcal{S}}(P)$

Note that the definition of portability requirements against *executions* is strong enough, as it implies the portability against *states* (the *state-portability*) [LFH+17]. The result SMT formula ϕ of the portability problem should contain both encodings of control-flow ϕ_{CF} and data-flow ϕ_{DF} of the program, and assertions of both memory models: $\phi = \phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}_\mathcal{T}} \wedge \phi_{\neg\mathcal{M}_\mathcal{S}}$. If the formula is satisfiable, there exist a portability bug.

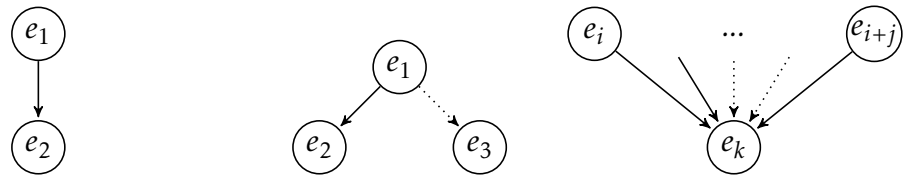
3.2.1 Encoding for the control-flow

The control-flow of a program is represented in the *control-flow graph*, a directed acyclic connected graph with single source and multiple sink nodes, obtained by the *loop unrolling* (see Section 5.4.1).In control-flow graph, there are two types of transitions (edges): *primary transitions* that denote unconditional jumps or if-true-transitions (pictured with solid lines), and *alternative transitions* that denote if-false-transitions (pictured with dotted lines). Each node on graph can have either one successor (primary) or two successors (both primary and alternative); only computation events can serve as a branching point). However, each merge node can have any positive number of predecessors, where each edge may be either primary or alternative.

While working on the `mousquetaires`, we applied some modifications of the encoding scheme for the control-flow. The changes are conditioned by the need to be able to process an arbitrary control-flow produced by conditional and unconditional jumps of C language. For that, we compile the recursive abstract syntax tree (AST) of the parsed C-code to the plain (non-recursive) event-flow graph. We show that the new encoding is smaller than the old one used in `PORTHOS` since it does not produces new variables for each high-level statement of the input language. For instance, `PORTHOS` uses the encoding scheme where the control-flow of the sequential instruction $i_1 = i_2; i_3$ was encoded as $\phi_{CF}(i_2; i_3) = (cf_{i_1} \Leftrightarrow (cf_{i_2} \wedge cf_{i_3})) \wedge \phi_{CF}(i_2) \wedge \phi_{CF}(i_3)$, and control-flow of the branching instruction $i_1 = (c\,?\,i_2 : i_3)$ was encoded as $\phi_{CF}(c\,?\,i_2 : i_3) = (cf_{i_1} \Leftrightarrow (cf_{i_2} \vee cf_{i_3})) \wedge \phi_{CF}(i_2) \wedge \phi_{CF}(i_3)$ (here we used the notation of C-like ternary operator `x?y:z` for defining the conditional expression `if`x`then`y`else`z). In contrast, the new scheme implemented in `mousquetaires` firstly compiles the recursive high-level code into the linear low-level event-based representation, that is then encoded into an SMT-formula. The encoding of branching nodes depends on the *guards*, the value of conditional variable on the branching state, which in turn is encoded as data-flow constraint (see Section 3.2.2).

Let $\mathbf{x} : \mathbb{E} \to \{0, 1\}$ be the predicate that signifies the fact that the event has been e**x**ecuted (and, consequently, has changed the state of the system). Let $\mathbf{v} : \mathbb{C} \to \mathbb{R}$ be the function that returns the value of the computation event (evaluates it) that will be computed once the event is executed (strictly speaking, it retuns the *set* of values determined by the $\xrightarrow{\text{rf}}$-relation; see Chapter **?TODO?** for the relations encoding). We distinguish the function $\mathbf{v}_p : \mathbb{C}_{\text{I}} \to \{0, 1\}$ that evaluates the predicative computation event. In the result formula, all symbols $\mathbf{x}(e_i)$ and $\mathbf{v}(e_i)$ are encoded as boolean variables.

Consider the following possible mutual arrangement of nodes in a control-flow graph:



(a) *Sequential events (*`seq`*)* **(b)** *Conditional branching (*`br`*)* **(c)** *Branch merging (*`mer`*)*

Figure 3.1: *Linear and non-linear cases of control-flow graph*

For listed cases, below we propose the encoding scheme that uniquely encodes each node of graph and allows to encode partially executed program. Equation 3.1 encodes the sequential control-flow represented in Figure 3.1a and reflects the fact that the event e_2 can be executed iff the event e_1 has been executed. Equation 3.2 encodes the branching control-flow depicted in Figure 3.1b by allowing only following executions: $\{\emptyset, (e_1), (e_1 \rightarrow e_2), (e_1 \rightarrow e_3)\}$. In encoding 3.3 of the merge-point represented in Figure 3.1c, the event e_k is executed if either of its predecessors was executed, regardless of type of the transition.

$$\phi_{CF_{seq}} = \mathbf{x}(e_2) \rightarrow \mathbf{x}(e_1) \tag{3.1}$$

$$\begin{aligned}\phi_{CF_{br}} = {} & [\mathbf{x}(e_2) \rightarrow \mathbf{x}(e_1)] \wedge [\mathbf{x}(e_3) \rightarrow \mathbf{x}(e_1)] \wedge \\ & [\mathbf{x}(e_2) \rightarrow \mathbf{v}(e_1)] \wedge [\mathbf{x}(e_3) \rightarrow \neg\mathbf{v}(e_1)] \wedge \\ & \neg[\mathbf{x}(e_2) \wedge \mathbf{x}(e_3)]\end{aligned} \tag{3.2}$$

$$\phi_{CF_{mer}} = \mathbf{x}(e_k) \rightarrow \Big(\bigvee_{e_p \in \mathtt{pred}(e_k)} \mathbf{x}(e_p) \Big) \tag{3.3}$$

For sake of encoding correctness, we require all branches to have at least one event. Thus, for branching statements that do not have any events in one of the branches (such a branch represents a conditional jump forward), we add the synthetic nop-event as it is shown in Figure 3.2:

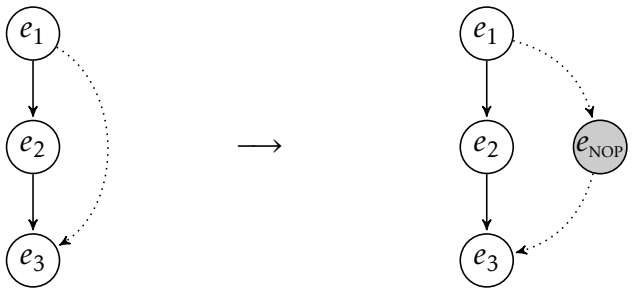


Figure 3.2: *Transformation of the empty-branch nonlinear control-flow*

3.2.2 Encoding for the data-flow

To encode the data-flow constraints, we use the *static single-assignment (SSA) form* in order to be able to capture an arbitrary data-flow into a single SMT-formula. The SSA form requires each variable to be assigned only once within entire program. In contrast, PORTHOS used the dynamic

single-assignment (DSA) form, that requires indices to be unique within a branch. Although the number of variable references (each of which is encoded as unique SMT-variable) on average is logarithmically less in case of the DSA form than the SSA form, the result SMT-formula still needs to be complemented by same number of equality assertions when encoding the data-flow in merge points [LFH+17].

Following [LFH+17], the indexed references of variables are computed in accordance with the following rules: (1) any access to a shared variable (both read and write) increments its SSA-index; (2) only writes to a local variable increment its SSA-index (reads preserve indices); (3) no access to a constant variable or computed (evaluated) expression changes their SSA-index. These rules determine the following encoding of load, store and computation events within single thread:

$$\phi_{DF_{e=\mathrm{load}(r\leftarrow l)}} = \mathbf{x}(e) \rightarrow (r_{i+1} = l_{i+1}) \tag{3.4}$$

$$\phi_{DF_{e=\mathrm{store}(l\leftarrow r)}} = \mathbf{x}(e) \rightarrow (l_{i+1} = r_i) \tag{3.5}$$

$$\phi_{DF_{e=\mathrm{eval}(\ldots)}} = \mathbf{x}(e) \rightarrow \mathbf{v}(e) \tag{3.6}$$

To convert the program into SSA form, for each event each variable that is declared so far (either local or shared) is mapped to its indexed reference; this information is stored in the SSA-map "event *to* variable *to* SSA-index". The SSA-map is computed iteratively while traversing the event-flow graph in topological order as it is described in Algorithm 1.

Algorithm 1 Algorithm for computing the SSA-indices

Input: The event-flow graph $G = \langle N, E \rangle$ where V is the set of nodes (events), E is the set of control-flow transitions, e_0 is the entry node
Output: The SSA-map of the form "{ event : { variable : index }}"

```
1: function COMPUTE-SSA-MAP(G)
2:     S ← empty map; S[e_0] ← empty map
3:     for each event e_i ∈ G.N in topological order do
4:         for each predecessor e_j ∈ pred(e_i) do
5:             S[e_i] ← copy(S[e_j])
6:             for each variable v_k ∈ set of variables accessed by e_i do
7:                 S[e_i][v_k] ← max(S[e_i][v_k], S[e_j][v_k])
8:                 if need to update the index of v_k then        ▷ cases (1)-(2)
9:                     S[e_i][v_k] ← S[e_i][v_k] + 1
```

The time of described algorithm is linear of the size of event-flow graph since it performs only single traverse of the graph.

As it has been described before, the `rf`-relation links data-flow between events of data-flow stored in equivalence assertions over the SSA-variables. The encoding of this linkage left untouched as it is implemented in PORTHOS: for each pair of events e_1 and e_2 linked by the `rf`-relation, we add the following constraint:

$$\phi_{DF_{mem}}(e_1, e_2) = \mathtt{rf}(e_1, e_2) \rightarrow (l_i = l_j) \tag{3.7}$$

where the variable of location l is mapped to the SSA-variable l_i for event e_1, and to the SSA-variable l_j for event e_2; and the predicate $\mathtt{rf}(e_1, e_2)$ is encoded as a boolean variable, which itself equals $true$ if e_2 reads the shared variable that was written in e_1.

3.2.3 Encoding for the memory model

todo

Chapter 4

The input language

The PORTHOS tool used a small subset of C as an input language as described in Figure 4.1 [LFH+17]. It supports recursive definitions of control-flow instructions, that were directly encoded into SMT-formula. In mousquetaires, the set of supported instructions is extended in order to support expressions valid in C. Also, for the purpose of simplicity, all kinds of fence instructions and different types of variable assignments (shared variable assignment ':=' or local variable assignment '←') were defined directly in the grammar for PORTHOS, whereas the semantics of function invocations need to be determined dynamically during the interpretation (see Chapter **??**).

Both PORTHOS and mousquetaires use the parser generator ANTLR [Par13], the powerfull language processing tool. The (hand-written) ANTLR grammar used in PORTHOS is available at Appendix A, whereas the mousquetaires uses the grammar of C language proposed in the C11 standard [ISO11] (the ANTLR grammar can be found in the official ANTLR repository on GitHub [1]).

[1] https://github.com/antlr/grammars-v4

```
<prog> : program <thrd>*
       ;
<thrd> : thread <tid> <inst>
       ;
<inst> : <atom>
       | <inst> ; <inst>
       | while <pred> <inst>
       | if <pred > then <inst> else <inst>
       ;
<atom> : <reg> ← <exp>
       | <reg> ← <loc>
       | <loc> := <reg>
       | mfence
       | sync
       | lwsync
       | isync
       ;
```

Figure 4.1: *Syntax of an input language of* PORTHOS

Chapter 5

The mousquetaires : implementation

This Chapter describes the architecture of the mousquetaires framework.

The programming language choice for mousquetaires was made in favour of java in order to reuse some parts of its predecessor PORTHOS that was written in java. However this language does not show best results in performance benchmarks (comparing to, for example, C++) **TODO**, the performance cornerstone of mousquetaires (as well as any other SMT-based code analyser) is the phase of solving the SMT-formula, which is left to the third-party SMT-solver called from mousquetaires via java API. However, considering the perspective of using mousquetaires as a static analyser for real-world programs, we also must to take into account memory problems at both encoding and solving stages.

5.1 Requirements

The main reason for starting the work on re-implementing the PORTHOS tool was the need to extend input language to be able to process real C programs in perspective. The existing PORTHOS architecture mixes high-level recursive *instructions* (statements of C) with low-level non-recursive *events*, to which the high-level instructions are compiled (see classes of package 'dartagnan.program' in the PORTHOS project [1]). Both instructions and events are then encoded into the SMT-formula. However, this approach is poorly extensible as to adding support for a high-level control-flow instruction (say, the do-while loop in C) requires making changes in many places of code from parser to encoder. Moreover, we encounter serious problems

[1] Project web site: http://github.com/hernanponcedeleon/PORTHOS

when we need to add support for control-flow jumps (as `continue`, `break`, `goto` in C).

Therefore, while planning the architecture of `mousquetaires`, we decided to clearly separate high-level recursive code representation as Abstract Syntax Tree (AST) and low-level compiled event-based representation as a flow-graph. Such a modular architecture allows to use multiple input-language parsers and convert parsed syntax trees to our AST, thus having support for multiple languages (for instance, the original input language of `PORTHOS` tool, two syntaxes of litmus tests used by `herd`, and even assembly language for any supported architecture).

The target **requirements** for the new tool was set up as following (in descending order of priority):

1. stability and transparency (KISS principle)
    - choice in favour of simplicity and readability at every step;
    - using software design patterns if necessary;
    - using immutable data structures for intermediate representations;
    - high code coverage by unit and functional tests;
2. efficiency
    - keeping trade-off between execution time and memory consumption;
3. extensibility
    - clear modular architecture

5.2 Program Components

Generally, `mousquetaires` uses following ...

5.3 Parsing the input language as YTree

below: mostly mock text.

- The language-dependent syntax tree: - for now it's the C subset language which I called 'Cmin'; as a base, I used the C11 grammar from ANTLR github repository, then I simplified it a lot, cutting off many unnecessary C

syntax features and making it more convenient for parsing. When developing the Cmin language, I kept in mind C elements that are necessary for processing the linux kernel code, though for now not the whole grammar element described in file 'Cmin.g4' are being implemented; - later I am going to add the litmus grammar as well; - in future, it will be not a problem to add any new C-like language;

- The language-independent abstract syntax tree (aliased 'Ytree', where 'Y' resembles branching of the tree): - all tree nodes in my code are prefixed with 'Y', see tentative (yet almost complete) class hierarchy in picture 'YEntity.png'; - this AST contains very basic language elements according to the C execution model (statements and expressions); - converting the language-dependent syntax tree to the language-independent syntax tree is performed by Visitor pattern (e.g., for Cmin->Ytree conversion is made by 'CminToYtreeConverterVisitor') - minor changes are performed by converting to ytree representation: desugaring the target code, etc.

5.4 YTree to XGraph event converter

- Then, the AST is being interpreted and converted to event-based representation (aliased 'Xrepr' for eXecution representation): - more low-level code representation (or high-level assembly); - I try to keep this representation close to the one you described in your papers: basic load & store events, branching events, fence events; - this representation is being implementing these days, I've just started doing it (see current class hierarchy in the picture 'XEntity.png');

- After we acquired the event-based representation, we can perform some modifications/simplifications/optimisations on it (separately, allowing user to manage them): - converting to SSA form as one of necessary steps before encoding; - (more? – I'm not thinking about it yet);

5.4.1 Loop unrolling

The original program encoded into the `XGraph` represents a *flow graph*, a connected cyclic directed graph with single source node (`ENTRY`) (usually for convenience all leaves are connected to the sink node (`EXIT`)). The cycles are caused by low-level jump instructions, obtained from non-linear high-level control-flow statements (such as `while`, `do-while`, `for`, etc.). How-

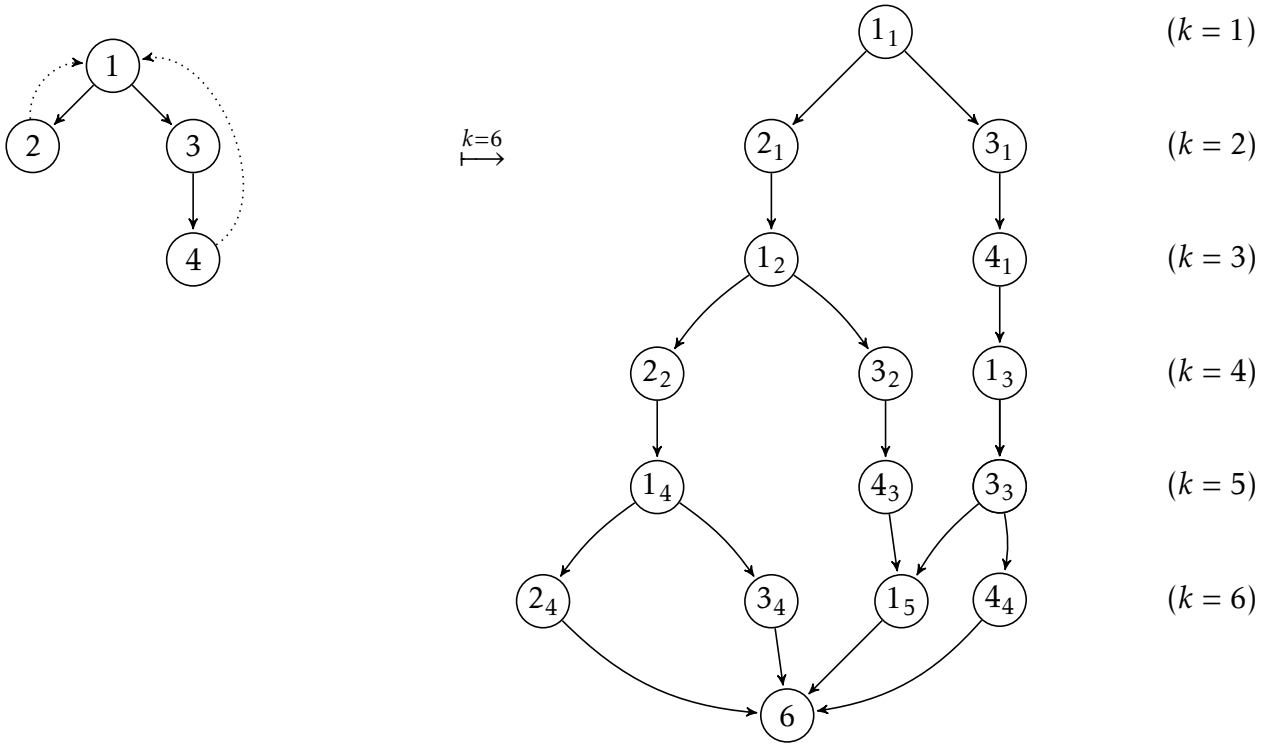


Figure 5.1: *Example of the flow graph from Figure* **??**, *unwinded up to the bound* $k = 6$

ever, the cyclic flow graph cannot be encoded into SMT formula since ... //TODO:REFERENCE.

5.5 XGraph to ZFormula (SMT) encoder

- Then, this modified event-representation is being encoded to SMT formula and sent to the solver.

5.6 Optimisations

... performed on each stage

Chapter 6

Evaluation

6.1 Comparison with PORTHOS

6.1.1 Unique Features

6.1.2 Performance

6.2 Comparison with HERD

6.2.1 Unique Features

6.2.2 Performance

Chapter 7

Summary

Bibliography

[LFH+17] H Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. "Portability Analysis for Axiomatic Memory Models. PORTHOS: One Tool for all Models". In: *CoRR* abs/1702.06704 (2017). arXiv: 1702.06704. URL: http://arxiv.org/abs/1702.06704.

[McK17] Paul E McKenney. *Is parallel programming hard, and, if so, what can you do about it?(v2017. 01.02 a)*. 2017.

[MAM+17] Paul E. McKenney, Jade Alglave, Luc Maranget, Andrea Parri, and Alan Stern. *A formal kernel memory-ordering model (part 1)*. 2017. URL: https://lwn.net/Articles/718628/.

[ACM16] Jade Alglave, Patrick Cousot, and Luc Maranget. "Syntax and semantics of the weak consistency model specification language cat". In: *arXiv preprint arXiv:1608.07531* (2016).

[LSM+16] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. "Coatcheck: Verifying memory ordering at the hardware-OS interface". In: *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 233–247.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding cats: Modelling, simulation, testing, and data mining for weak memory". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.2 (2014), p. 7.

[LPM14] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2014, pp. 635–646.

[AKN+13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. "Software verification for weak memory via program transformation". In: *European Symposium on Programming*. Springer. 2013, pp. 512–532.

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[CKN+12] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. "Model checking and the state explosion problem". In: *Tools for Practical Software Verification*. Springer, 2012, pp. 1–30.

[ISO12] ISO ISO. "IEC 14882: 2011 Information technology—Programming languages—C++". In: *International Organization for Standardization, Geneva, Switzerland* 27 (2012), p. 59.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ concurrency". In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 55–66.

[ISO11] ISO/IEC. "SC22/WG14. ISO/IEC 9899: 2011". In: *Information technology – Programming languages –C. http://www.iso.org/iso/iso_-catalogue/catalogue_ tc/catalogue_detail. htm* (2011).

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER multiprocessors". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 175–186.

[Alg10] Jade Alglave. "A shared memory poetics". In: *La Thèse de doctorat, L'université Paris Denis Diderot* (2010).

[AFI+09] Jade Alglave et al. "The semantics of Power and ARM multiprocessor machine code". In: *Proceedings of the 4th workshop on Declarative aspects of multicore programming*. ACM. 2009, pp. 13–24.

[MZ09] Sharad Malik and Lintao Zhang. "Boolean satisfiability from theoretical hardness to practical success". In: *Communications of the ACM* 52.8 (2009), pp. 76–82.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A better x86 memory model: x86-TSO". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 391–407.

[MPA05] Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*. Vol. 40. 1. ACM, 2005.

[CBR+01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded model checking using satisfiability solving". In: *Formal methods in system design* 19.1 (2001), pp. 7–34.

[Sht00] Ofer Shtrichman. "Tuning SAT checkers for bounded model checking". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 480–494.

[Lam79] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess program". In: *IEEE transactions on computers* 9 (1979), pp. 690–691.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

Appendices

Appendix A
The ANTLR grammar for the PORTHOS input language

```
grammar Porthos;

main
  :   program
  ;

bool_expression
  :   bool_atom
  |   bool_atom BOOL_OP bool_atom
  ;

bool_atom
  :   'true'
  |   'false'
  |   '(' arith_expr COMP_OP arith_expr ')'
  |   '(' bool_expression ')'
  ;

arith_expr
  :   arith_atom ARITH_OP arith_atom
  |   arith_atom
  ;

arith_atom
  :   DIGIT
  |   register
  |   '(' arith_expr ')'
  ;

register
  :   WORD
  ;

location
  :   WORD
  ;

local
  :   register '<-' arith_expr
  ;

load
  :   register '<:-' location
  ;

store
  :   location ':=' register
  ;

read
  :   register '=' location '.' 'load' '('
     ATOMIC ')'
  ;

write
  :   location '.' 'store' '(' ATOMIC ','
     register ')'
  ;

statement
  :   expression
  |   sequential_stmt
  |   while_stmt
  |   if_stmt
  ;

expression
  :   local
  |   load
  |   store
  |   FENCE
  |   read
  |   write
  ;

sequential_stmt
  :   expression  ';' statement
  |   while_stmt  ';' statement
  |   if_stmt     ';' statement
  ;

if_stmt
  :   'if' bool_expression 'then' '{' statement
      '}' ('else' '{' statement '}')?
  ;

while_stmt
  :   'while' bool_expression '{' statement '}'
  ;

program
  :   '{' location (',' location)* '}'
       ('thread t' DIGIT '{' statement '}'
         ('exists' (location '=' DIGIT ','
         | DIGIT ':' register '=' DIGIT ',')
         )*
       )*
  ;

ATOMIC
  :   '_na' | '_sc' | '_rx' | '_acq' | '_rel' |
      '_con'
  ;

FENCE
  :   'mfence' | 'sync' | 'lwsync' | 'isync'
  ;

COMP_OP : '==' | '!=' | '<=' | '<' | '>=' |
      '>';
ARITH_OP : '+' | '-' | '*' | '/' | '%';
BOOL_OP : 'and' | 'or';

DIGIT : [0-9];
WORD : (LETTER | DIGIT)+;
LETTER : 'a'..'z' | 'A'..'Z';
```