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:** | vii + 24 |
| **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
Perustieteiden korkeakoulu
???

???

| | | | |
|---|---|---|---|
| **Tekijä:** | Artem YUSHKOVSKIY | | |
| **Työn nimi:** ? | | | |
| **Päiväys:** | ?.?.2018 | **Sivumäärä:** | vii + 24 |
| **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 unexpected state '`(0:EAX=0 /\ 1:EAX=0)`' reachable 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.

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

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

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 sence 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 sofware suite for designing and testing weak memory models. It is firstly released back in 2010, and

[1] `diy.inria.fr`

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 `diy7`, `diycross7` and `diyone7`, the litmus tests concrete executor `litmus7` that runs tests on a physical machine while collecting its behaviours, and the weak memory models simulator `herd7` 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 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, the program compiled for the platform $X$, can reach states that are unreachable on the platform $Y$, or some states that are reachable on $Y$ can become unreachable on $X$. We declare these cases as *portability bugs*.

The first tool that performs the WMM-awared 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 mantainability. 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 Chapter 2 gives more detailed description of the weak memory model-awared analysis and provides description of memory models for some common architectures (x86, ARM and POWER, Sparc, ???). Chapter ...

# Chapter 2

# Weak Memory Models

A *weak memory model* is a set of predicative constraints over so-called *derived relations*, which defines the set of possible executions of a concurrent program. Usually, these constraints include requirements of the reflexivity or acyclicity of derived relations. The study of formalisation weak memory models for different architectures has been rapidly developed over last decade.

Firstly, the research of WMM aimes to formalise the weak memory models and provide systematic, sound and complete formal approach of defining WMMs in order to be able to verify systems with respect to them. A generic framework of program analysis with respect to weak memory models was presented in 2010 [Alg10]. Similar approach is used PORTHOS as it is described further in current Chapter.

Secondly, researchers work on extracting the WMMs of hardware architectures from existing implementations of from their specifications, that are written in natural language and thus suffer from ambiguities and incompletenesses. Over last decade the memory models have been extracted for most mainstream multiprocessor architectures, such as x86-TSO (*Total Store Order*) model for x86 architecture formalised in 2009 [OSS09], much more relaxed memory model for Power and ARM architectures [SSA+11] [AFI+09], for Alpha **?** and Sparc **?** memory models were defined in the specification. Moreover, most modern high-level programming languages rely on relaxed memory model. Thus, the memory model for Java that is based on the *happens-before* principle [Lam78] was introduced in J2SE 5.0 in 2004 [MPA05]; the C++11 standard **?** has introduced the set of hardware-independent synchronisation fences and atomic operations, whenever the C++17 memory model is based on the relation *strongly happens-before*.

Thirdly, important research direction targets the problem of verifying (or at least finding bugs in) existing software systems with respect to weak memory models. Perhaps, the most notable work in this field is the project for defining the Linux kernel memory model, which is being actively developing these days [MAM+17].

## 2.1 The event-based program representation

The classical approach to model the concurrent programs is to use the *global time*, a single order of interleavings of all actions happened in different threads. However these models are easy to understand, it may be hard to consider *all* possible states, number of which is exponentially large. Another way to do this is to to use non-deterministic computation-centric models defined in [Fri97], one of which represents the program as the graph of *memory events*. The idea in this class of models is based on the fact that the behaviour of a concurrent system is defined only by the interleavings of shared-memory operations, while being independant from the order of local computation events. These models may be further restricted by constraints of a weak memory model, adding *relations* to the memory events.

The event-based program model represents the directed graph (the *event-flow graph*), where vertices represent *events*, and edges represent *relations* over the events. An event is something, that, after being executed, changes the state of an abstract machine executing the concurrent program. An *execution* (trace, run) of a given program is an ordered set of events. The order of events in particular execution is denoted as '$\rightarrow$', an empty execution is denoted as $\emptyset$. An execution is considered to be *valid* if the memory events follow a single global timeline, textit.e., can be embedded in a single partial order allowed by the memory model restrictions [Alg10]. An execution to be checked on validity is called the *candidate execution*.

Below we describe some basic types of events and relations.

### 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 devided 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 event $e_m = write(l_1, l_2)$ that writes value from shared location $l_2$ to the shared location $l_1$ is represented as two consequent events $e'_m = \mathtt{load}(r_1 \leftarrow l_2)$; $e''_m = \mathtt{store}(l_1 \leftarrow r_1)$.

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 (emulated 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, that are evaluated as a boolean value.

The third class of events is *barrier events*, events caused by the synchronisation instructions (called *fences*). Barrier events 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. Technically, a fence may be represented either as a synchronisation barrier, or a flush local memory caches, etc.

### 2.1.2 Relations

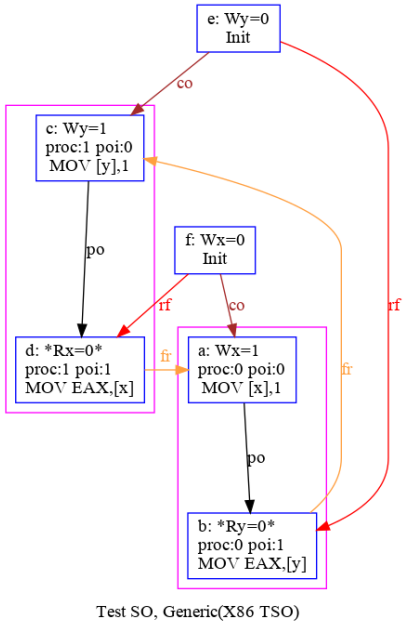
In this section, we describe basic derived relations used in memory model-awared program analysis.

The *control-flow* of a program is defined by po-relation $\subset \mathbb{E} \times \mathbb{E}$ (*program-order*), which represents the total order of events *from same process*, which never relates events from different processes. Thus, if a program specifies the memory instruction $i_2$ to follow immideately the memory instruction $i_2$, then there exist an edge $e_1 \xrightarrow{\text{po}} e_2$ in the event-flow graph where event $e_1$ is caused by the instruction $i_1$ and $e_2$ is caused by the instruction $i_2$. This relation encodes the control-flow of the program into the event-flow graph.

The *data-flow* of a program is defined by *communication relations*: the rf-relation $\subset \mathbb{W} \times \mathbb{R}$ (*read-from* relation) that maps a write to a read reading its value, the co-relation $\subset \mathbb{W} \times \mathbb{W}$ (*coherence order*, sometimes called ws-relation for *write serialisation*) defines the total order on writes

to the same location across all processes, and the fr-relation $\subset \mathbb{R} \times \mathbb{W}$ (*from-read order*) that maps a read to possible writes preceding the current write event (this relation is the inversion of the rf-relation: $\mathtt{rf} = \mathtt{fr}^{-1}$).

Figure 2.1.2 illustrates the candidate execution for the Example 1.1, that reaches the state `(0:EAX=0 /\ 1:EAX=0)` within x86-TSO memory model as it forbids cycles over po∪rf(union) relation (the picture is generated by the herd7 tool, version 7.47).




Test SO, Generic(X86 TSO)

### 2.1.3 Executions

The semantics of a concurrent program is represented by the set of allowed executions. An execution is uniquely defined by the set $\mathbb{X}$ of events have been executed in each thread (the *control-flow* of a program), and the relations rf and co (*data-flow* of a program) [Alg10]. As it was shown in [WBS+17], it is enough for memory models to constrain the executions independently instead of constraining the program as a whole.

## 2.2 The CAT language

One common way to define a weak memory model in a systematic way in to formalise it in *the CAT language* proposed in [ACM16]. The CAT language

allows to axiomatically define new relations, new fences and restrictions over relations.

briefly some hw memory models: X86-TSO, Alpha, POWER, ;
language memory models: Java, C++;
library-level kernel memory model, ref to github with tests

## 2.3 Some examples of WMM

//axioms of TSO wmm

// example of sets: rf, co, ... for the code snipped used before

# Chapter 3

# The Portability of Concurrent Software

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 the satisfiability modulo theories (SMT) problem [LFH+17].

## 3.1 The model checking problem

The classical model checking algorithms explore the state space of an abstract automata or a transition 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 of the system. 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 The portability as a bounded reachability problem

In general, the 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: $\mathsf{SAT}(Init(\vec{x_0}) \wedge Trans(\vec{x_0}, \vec{x_1}) \wedge \cdots \wedge Trans(\vec{x}_{k-1}, \vec{x_k}) \wedge Bad(\vec{x_k}))$.

The portability probram may also be formulated 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.

As we have said before, a program $P$ is called portable from the source architecture (defined as a 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 formulation of portability requirements against *executions* is strong enough, as it implies the portability against *states* (the *state-portability*) [LFH+17].

The result SMT formula $\phi$ of the portability problem should contain encodings of control-flow ($\phi_{CF}$) and data-flow ($\phi_{DF}$) 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 The control-flow encoding

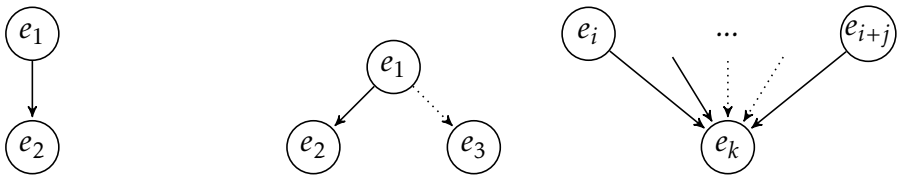
The control-flow of a program is represented in the *control-flow graph*, a directed acyclic connected graph with single source and multiple sink nodes, obtained by the *loop unrolling* (see Section 5.4.1).In control-flow graph, there are two types of edges: *primary edges* that denote unconditional jumps or if-true-transitions (pictured with solid lines), and *alternative edges* that denote if-false-transitions (pictured with dotted lines). Each node on graph can have either one successor (primary) or two successors (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 showthat 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, in old scheme 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 further in current Chapter).

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 $\mathbf{v} : \mathbb{C} \to \mathbb{N}$ be the function that returns the value of the computation event that will be computed once the event is executed (strictly speaking, it retuns the *set* of values determined by the `rf`-relation; see Chapter ? for the relations encoding). We distinguish the function $\mathbf{v}_p : \mathbb{C}_! \to \{0, 1\}$ 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 grpah:




**(a)** *Sequential events* **(b)** *Conditional branching* **(c)** *Branch merging*

**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. The Equation 3.1 for encoding the sequential control-flow represented on Figure 3.1a reflects the fact that the event $e_2$ can be executed iff

the event $e_1$ was executed. The Equation 3.2 for encoding the branching control-flow depicted on Figure 3.1b allows only following executions: $\{\emptyset, (e_1), (e_1 \rightarrow e_2), (e_1 \rightarrow e3)\}$. In encoding 3.3 of the merge-point represented on Figure 3.1c, the event $e_k$ is executed if either of its predecessors was executed, regardless of type of the transition.

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

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

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

### 3.2.2 The data-flow encoding

While encoding 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. That is, on each state of the program a variable (either local or shared) is mapped to its indexed reference, which represents a new boolean variable in the formula. On merge-points, the $\phi$-function is emulated as the disjunction of values of the variable that is computed as the *reaching definition*. For that, we carry the map that for each event stores mapping from all declared variables to their SSA-indices. All indices are globally unique within single program analysis.

As it is implemented in [LFH+17], the SSA-indices 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 expression changes their SSA-index. These rules determine the following encoding of load, store and computation events within single thread:

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

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

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

Unlikely the PORTHOS tool, the mousquetairesuses the algorithm of converting the program into the SSA form was also adapted for processing the plain flow-graph data structure. That is, for each node we keep track the set of its predecessors, and then, while traversing the graph in topological order, we propagate the SSA-indices for each variable defined so far, and update these indices according the rules listed above.

The time of described algorithm is linear of the size of event-flow graph since it performs only single traverse of the graph. Notwithstanding the overhead of storing (or, equivalently but undesirably, computing withlinear time) the predecessors of each node in order to be able to convert the program into an SSA form, the time of such a tranformation reducedcomparing to the algorithm implemented in PORTHOS , which recomputed SSA-indices recursively for each instruction.

Moreover, the data-flow encoding scheme used in PORTHOS faced the problem when the branching statements had different number of assignments of certain variable in their branches. //TODO: why? why don't we have this problem now?

As it has been stated before, the rf-relation links the partsof data-flow stored in equivalence assertions over the SSA-variables. The encoding of this linkage left untouch 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 The memory model encoding

The encoding of a weak memory model itself does not depend on the analysing program. The problem here...

recursive definitions and Kleene iteration

As it is described in [LFH+17], the is based ...

# Chapter 4

# The input language

the old input language syntax
    new one iwth changes highlighted

# Chapter 5

# The `mousquetaires`: implementation

This Chapter describes the architecture of the tool `mousquitaires` ...
language: java

## 5.1 Program Requirements

- stability (tests)
- scalability (new features of language, new models, new tasks for a program )
- transparency
- efficiency

## 5.2 Program Components

Big view

## 5.3 C11 to YTree parser

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

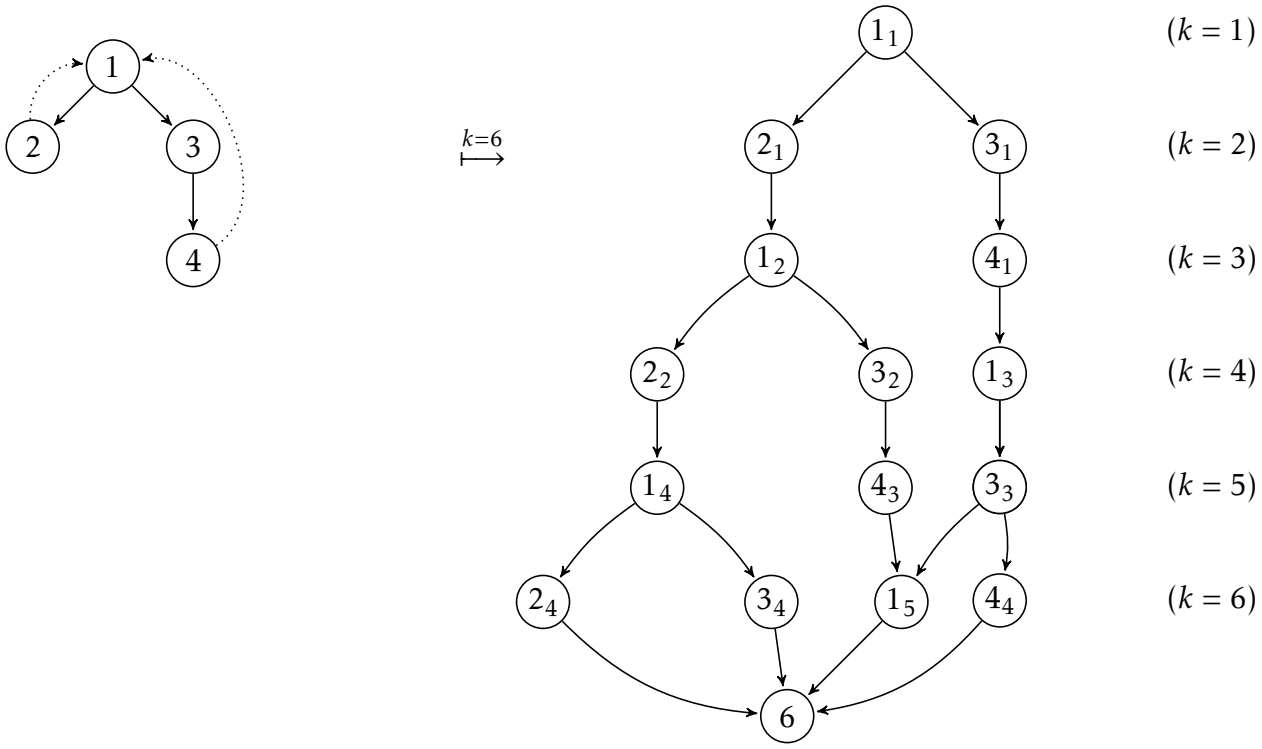



**Figure 5.1:** *Example of the flow graph from the Figure* **??**, *unwinded up to the bound* $k = 6$

## 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/.

[WBS+17] John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. "Automatically comparing memory consistency models". In: *ACM SIGPLAN Notices*. Vol. 52. 1. ACM. 2017, pp. 190–204.

[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).

[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.

[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: *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.

[Fri97] M. Frigo. "The weakest reasonable memory model". MA thesis. MIT, Oct. 1997.

[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.