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 + 15
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:**	vii + 15
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

problems of concurrency

necessity of Weak Memory Models

- hardware wmms
- wmms of programming languages
- specific wmms like for kernel

wmm as a formal way to define guaranties that a hw, programming language, execution environment provide for programmers.

considering wmm as a set of allowed behaviours, the latter wmms are the supersets

wmm allows and disallows optimisations: partial sync of memory buffers, out-of-order execution (reordering), <more> => more behaviours that are unallowed in SC.

question possible to answer with wmm: which behaviours (in addition to SC) are allowed? which new states are allowed? Consequently, correctness, absence of data races, deadlocks or portability issues, etc.

existing tools: herd, diy7 – exhaustive search approach for exploring the state space.

another approach: using smt solver, e.g. for answering questions like

base paper: aims to investigate portability of small programs written in a C-like pseudocode and provides the proof-of-concept tool PORTHOS [link]. As input, it takes a program and two memory models in CAT language. Then it encodes programs and memory models into an smt formula and tries to solve it via z3. Current thesis aims to extend this tool functionality to process the real C code, therefore it proposes different modula program architecture and multiple optimisations.

1.1 Thesis structure

...

Chapter 2

Weak Memory Models

briefly what it is, as in Intro

Some examples of what wmms allow to do

Lamport's sequentially consistent memory model – global order "a global order, even if they are actually executed in different threads running in parallel in different processors. In this model, each read from some memory location is guaranteed to see a value written by the last write to this location." ([c11 by natasha]). Produces same result as sequential program.

2.1 The CAT language

the CAT language [ref to Jade's paper] (hard part: to decipher the Alglave's paper).

the event representation

2.2 Examples of WMM

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

Relationship between different models `http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2013/10c_ks`

Chapter 3

Implementation

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

3.1 Program Requirements

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

3.2 Program Components

Big view

3.3 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.

3.4 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);

3.4.1 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.

Although there are multiple techniques for converting cyclic directed graphs into DAGs (Directed Acyclic Graphs), most of them are either not

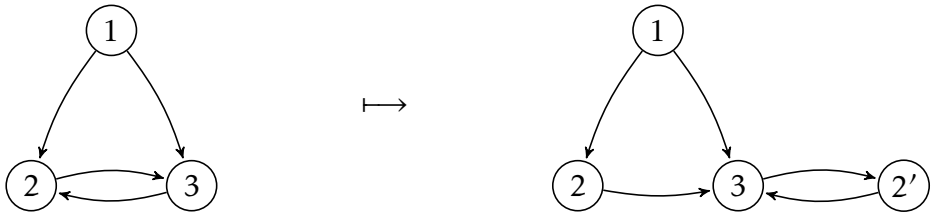

Figure 3.1: *Transforming irreducible flow graph into reducible graph [3]*

applicable to the problem of encoding graph into SMT formula, or work only for special cases (e.g., only for single-nested loops). // TODO: examples

In the work [1], the authors propose using the *loop unrolling* technique to solve this problem, however the loops in flow graph are restricted syntactically by using only `while` statement without loop-breaking statements. The general case (that uses jumps to arbitrary instructions, which may be produced by `C` instruction `goto`) requires more thorough control-flow analysis.

3.4.1.1 Identifying the loops

In order define the unrolling algorithm, we need to introduce some graph-theoretical terminology used in compiler theory. All edges of a flow graph may be divided into three categories: *advancing edges* that go from a node to its proper descendant in the tree constructed by the depth-first traverse starting at the source node; *retreating edges* that go from a node to its ancestor in the tree, and *cross edges* (edges that lead from a node to another node that is neither its descendant nor ancestor) [2]. In a control flow graph, branching statements (`if-then-else`, `switch`) introduce cross edges in the graph, while other control flow statements (`while`, `do-while`, `for`, `continue`, `break`, `goto`, etc.) introduce retreating edges.

If the head of retreating edge dominates its tail (i.e., *every* path from head to tail goes through head), such edge is called *back edge*. If all retreating edges of a flow graph are back edges, such graph is called *reducible* (otherwise *irreducible*). As it was noted in [2], the flow graphs that does not use jumps to arbitrary instructions are reducible. However, it is not hard to transform the irreducible graph to the reducible equivalent by cloning the node incident to two back edges to two nodes as it is shown in Figure 3.1. Therefore, the loop unrolling algorithm can assume that the flow graph is reducible.

The *loop* (or *natural loop*) in a program is the set of nodes that form a path in a flow graph with a single single-entry node (*loop head*) which

dominates all other loop nodes, and single back edge that goes to the head. The more precise definition of loop allows merging some cycles, caused by the set of back edges $E_i = \{(v_i, w_i) \mid (v_i, w_i) \in G.E \wedge w_i = h\}$ that point to the same head and having non mutually-nested set of nodes, into single loop:

Definition 3.4.1. The *loop* in a flow graph $G = (V, E)$ caused by some set of back edges $E_i \subset G.E$ is the tuple $l_i = (h, N, T, R)$ where

- the loop *head* $h \in G.N$,
- the set of *loop nodes* $N = \{u \mid u$ is reachable in reversed graph G^{-1} from origins of back edges $\{w \mid (w, h) \in E_i\}$ such that the reachability search does not go through the loop head $h\}$,
- the set of *tails* of the loop $T \subseteq G.N$ such that $T = \{w \mid (w, h) \in G.E\}$
- the set of *reference nodes* $R \subseteq G.N$ such that $R = \{(u, v) \mid u \in l_i.N \wedge v \notin l_i.N\}$

This constructive definition of the loop directly leads to the loop searching algorithm [2]:

Examples of possible mutual arrangements of two nested loops in a flow graph are shown in Figure 3.2 (the back edges are pictured with dotted lines). In the first example 3.2a, the loop $l_{2\to3}$ caused by back edge $2 \to 3$ has the set of nodes $\{2, 3\}$, while the loop $l_{4\to1}$ has the set of nodes $\{1, 2, 3, 4\}$, which is a superset of the set of nodes of $l_{2\to3}$, therefore the latter loop is the outer loop of the former one, and it should be unrolled firstly. The same reasoning applies to all other examples 3.2b – 3.2d.

Figure 3.3 illustrates two cycles $1 \to 2 \to 3$ and $1 \to 2 \to 4$ with the same head 1 that can be safely merged into one loop $\{1, 2, 3, 4\}$ with head 1 (lines 10 and 11 of Algorithm 1) and interpreted as a single loop with branching expression at the head node.

3.4.1.2 Binding unrolled loops

Once the set of loops of the flow graph is defined, they need to be unrolled and bounded into single graph. Unrolling is being performed up to the user-defined bound k, this means every loop is being unwound to the acyclic subgraph of size at most k. This bound is defined separately for each collected loop, so that the l-nested loop has at most k^l nodes. Note that the loop unrolling does not require any data-flow information: the reachability

Algorithm 1 Algorithm for finding loops in the flow graph

Input: The flow graph G with the source node v_s
Output: Set *Loops* of discovered loops $loop = (h, T, R, N)$ where h is the head of loop, T is the set of tails of the loop, N is the set of all nodes of the loop, $R = \{(u, v) | u \in N \wedge v \notin N\}$ is the set of edges (references) outside the loop

```
1: function FindLoops(G)
2:     E_b ← all back edges of the graph G
3:     for each back edge (v_i, h_i) ∈ E_b do
4:         mark the head h_i as visited    ▷ do not go outside the loop head
5:         for each edge (u, w^-1) ∈ G^-1.E explored while traversing the
   reversed graph G^-1 in depth-first order starting from node v_i do
6:             remember visited node u in the set N_i
7:             merged ← false
8:             for each already collected loop l ∈ Loops do
9:                 if (l.h = h_i) ∧ (l.N \ N_i ≠ ∅ ∧ N_i \ l.N ≠ ∅) then
   ▷ merge these two loops into one:
10:                    l.N ← l.N ∪ N_i
11:                    l.T ← l.T ∪ T_i
12:                    merged ← true
13:                    break
14:            if not merged then
15:                add l_i = (h_i, T_i, R_i, N_i) to Loops
   ▷ fill in the set l_i.R of references outside the loop:
16:        for each loop node v ∈ l_i.N do
17:            for each edge (v, w) ∈ G.E do
18:                if w ∉ l_i.N then
19:                    add v to the set l_i.R of references outside the loop
```

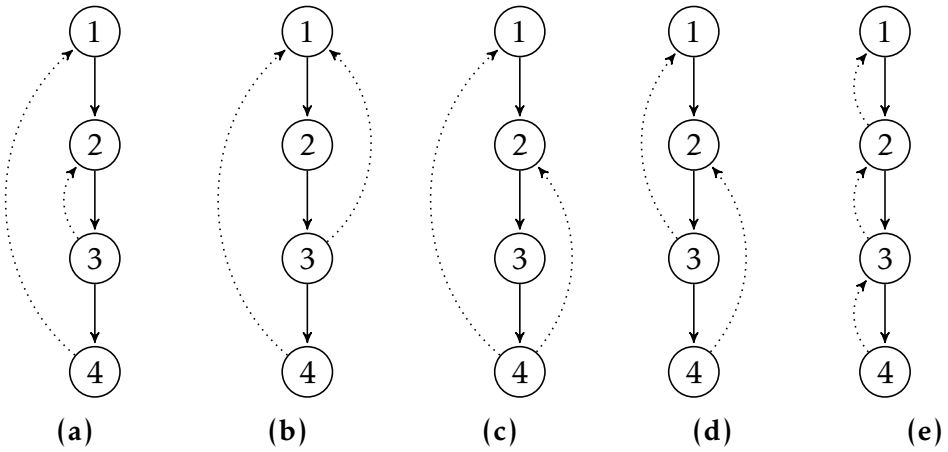

Figure 3.2: *Examples of possible mutual arrangements of two nested loops in a flow graph*

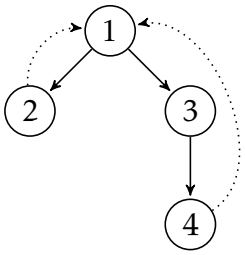

Figure 3.3: *Example of flow graph with multiple cycles merged into a single loop*

analysis of a particular node in the unrolled graph will be performed later by the SMT solver. This is rather rough underapproximation, which is to be handled by user by specifying the unrolling bound k.

The algorithm for unrolling and binding nested loops is as following:

Algorithm 2 Algorithm for unrolling and binding nested loops

Input: The ordered list *Loops* of nested loops l_i such that $l_1.N \subset l_2.N \subset ...l_t.N$

Output: Set *Loops* of discovered loops $loop = (h, N)$ where d is the head of loop, L is the set of nodes of the loop

1: **function** UnrollAndBindLoops(*Loops*)
2: //to be done

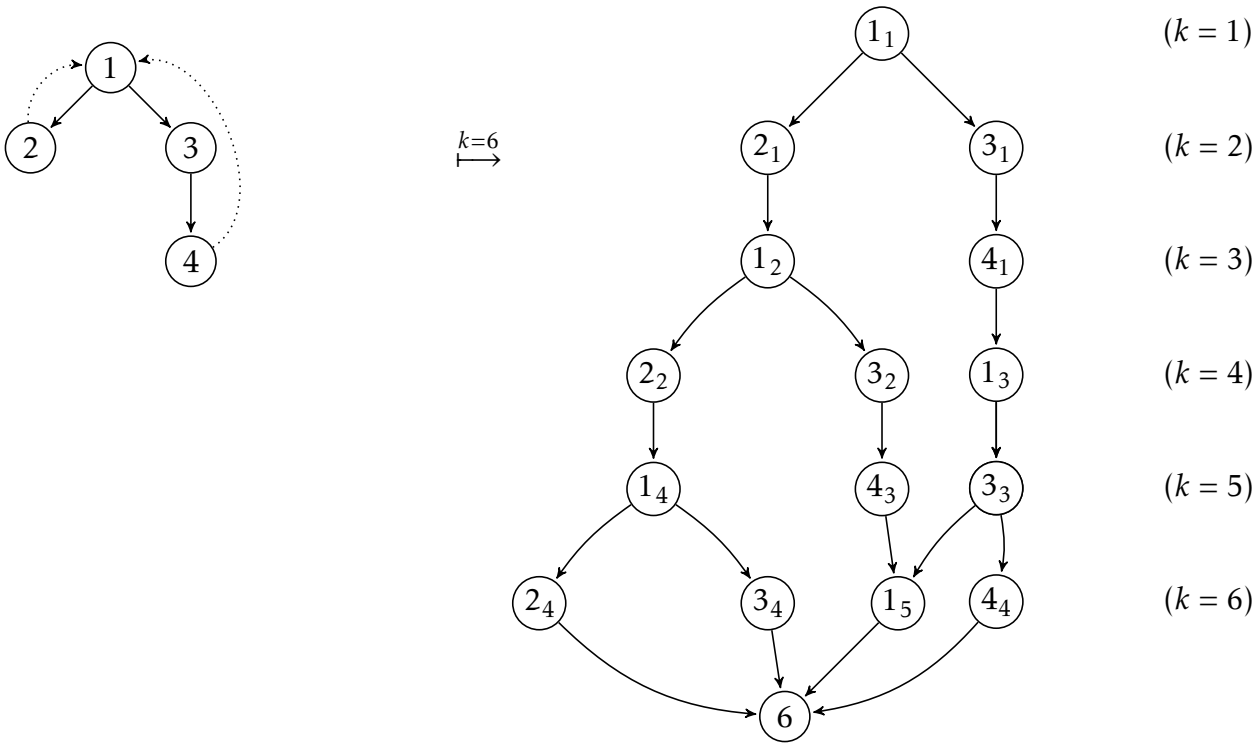

Figure 3.4: *Example of the flow graph from the Figure 3.3, unwinded up to the bound* $k = 6$

3.5 XGraph to ZFormula (SMT) encoder

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

3.6 Optimisations

... performed on each stage

Chapter 4

Evaluation

4.1 Comparison with PORTHOS

4.1.1 Unique Features

4.1.2 Performance

4.2 Comparison with HERD

4.2.1 Unique Features

4.2.2 Performance

Chapter 5

Summary

Bibliography

[1] Hernán Ponce de León et al. "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.

[2] A.V. Aho. In: *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007. Chap. 9.6 Loops in Flow Graphs. ISBN: 9780321486813.

[3] A.V. Aho. In: *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007. Chap. 9.7.6 Handling Nonreducible Flow Graphs. ISBN: 9780321486813.

[4] A.V. Aho. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007. ISBN: 9780321486813.