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 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 + 19
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 + 19
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 [1].

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 in x86 machines (such little examples that illustrate specific behaviours of a WMM are called *litmus tests*). This behaviour is caused by write buffers used by all processors in x86 architecture. These buffers cache writes to shared variables, so that the writes to shared memory does not become visible by other processes immediately. In the example, the write '`MOV [x],1`' performed in the process `P0` stores value `1` into the shared variable `[x]` in 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. More presice description of x86-TSO memory model is given in Chapter 2.3.1.

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

Figure 1.1: *Litmus test on memory operations reordering allowed by the x86-TSO weak memory model*

The first memory model for concurrent systems was formulated by Leslie Lamport back in 1979 [2]. 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)*. 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 and firstly released back in 2010. It is the sofware suite for designing and testing weak memory models, it consists of 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 aforesaid 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 [3]. This tool reduces described problem to the bounded model checking (BMC) problem, which can be solved via SMT-solver. This approach allows to capture symbolically the semantics of analysing program and both weak memory models into single SMT-formula. As most modern SMT-solvers are efficient enough to be able to operate the state space of size .... (**TODO**), the used method can be applicable in solving the real-world problems.

Current work aims to improve the proof-of-concept tool `PORTHOS` firstly introduced in April 2017 in the work [3] 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.

1.1 Thesis structure

The Chapter 2 gives more detailed description of the weak memory models analysis and provides description of memory models for some common architectures (x86, ARM and POWER, Sparc, ???). Chapter ...

Chapter 2

Weak Memory Models

The *weak memory model* is a set of predicative constraints on possible executions of a concurrent program. The study of formalisation the weak memory models for different architectures is being rapidly developed over last decade. Research of WMM aimes, firstly, 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.

Secondly, researchers work on extracting the formal hardware memory models 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 [4], SPARC-TSO for Sparc architecture **?!?!?!**, much more relaxed memory model ??? for Power and ARM architectures defined in ??? [5], Alpha (???). Moreover, in 2005 <who?Milner> started the work on developing the weak memory model for C++ language, which was introduced in C++11 (?) standard **???** (//todo: C11 MM [6]). The memory model for Java that is based on the *happens-before* principle was introduced in JVM??? in ??<year>.

Thirdly, important research direction targets the problem of verifying (or at least finding bugs in) existing software systems with respect to weak memory models. In this <area> the notable works <are on> defining the Linux kernel memory model <that is being actively developing these days **kernel_wmm_1**>. Distributed databases <also need the wmm, see transaction consistency [7]>

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 [8], 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 (*event-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. An execution is considered to be *valid* if the memory events follow a single global timeline, i.e., can be embedded in a single partial order allowed by the memory model restrictions [9]. 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 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 $\mathtt{id}(e_m)$ [9]. 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 $e_c = r_3 \leftarrow add(r_1, r_2)$ that writes the sum of values stored in registers r_1 and r_2 to the register r_3. The *control-flow* instructions (conditional and unconditional jumps) are encoded to the model directly, without additional events, as the *po*-relation (for *program order*; see Chapter 2.1.2 for detailed definition of relations).

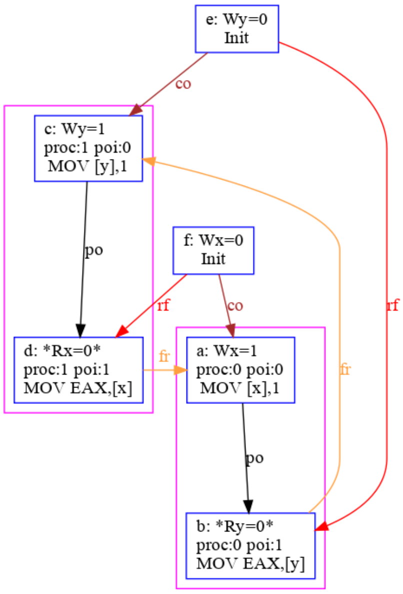
The third class of events is *barrier events*, events caused by the synchronisation instructions (called *fences*). Barrier events do not perform any computation or memory value transfer, instead, they add new relations to the model that restrict the set of allowed behaviours. Technically, a fence may either serve as a synchronisation barrier, or flush local memory caches, etc.

2.1.2 Relations

The basic relation in the event-based program model is the po-relation $\subset \mathbb{E} \times \mathbb{E}$ (*program-order*), which represents the total order of memory events *within single 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 $e_1 \xrightarrow{\mathsf{po}} e_2$ in the event-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-graph.

The data-flow of a program is encoded by the *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.

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 (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) **??**. As it was shown in **??**, it is enough for memory models to constrain the executions independently instead of constraining the program as a whole.

2.2 The CAT language

The Porthos tool uses the language `CAT` **??** for defining weak memory models.
the event representation

2.3 Some known WMM

2.3.1 x86-TSO

// exmaple with reordering // ex. with // Rev-29 Example 7-6. Stores Are Transitively Visible.

There is a barrier instruction `mfence` that may be used for flushing the buffers into the main memory.

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

Portability of Concurrent Software

As it was discussed in Chapter 1, the program may behave differently when compiled for different parallel hardware architectures. This can cause the portability bugs, the behaviour allowed under one architecture and forbidden under another. The concurrent software portability analysis may be stated as a Bounded Model Checking (BMC) problem, which in turn can be reduced to the problem of satisfiability modulo theories (SMT) [3].

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 major optimising technique was symbolic model checking with binary decision diagrams (BDDs). In this approach, a set of states is represented by a BDD instead of by listing each state individually [10]. 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 [11]. 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., [12]).

3.2 The portability as a BMC problem

A program P is called portable from the source weak memory model $\mathcal{M}_\mathcal{S}$ to the target memory model $\mathcal{M}_\mathcal{T}$ if all executions consistent under $\mathcal{M}_\mathcal{T}$ are consistent under $\mathcal{M}_\mathcal{S}$ [3]:

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*) [3].

It is possible to formulate this requirement as an SMT formula, so that the portability analysis problem becomes reduced to the BMC problem.

The full SMT formula ϕ should contain encodings of control-flow (ϕ_{CF}) and data-flow (ϕ_{DF}) of the program, and assertions of both memory models: $\phi = \phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}_{\mathcal{T}}} \wedge \phi_{\neg\mathcal{M}_{\mathcal{S}}}$. The control-flow and data-flow encodings are standard for BMC [13], they are described below. However, encoding of memory models requires additional techniques due to recursive definitions of relations, that were proposed in [3].

3.2.1 Encoding of the control-flow constraints

3.2.2 Encoding of the data-flow constraints

3.2.3 Encoding of the memory model constraints

Chapter 4

Implementation

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

4.1 Program Requirements

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

4.2 Program Components

Big view

4.3 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.

4.4 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);

4.4.1 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.

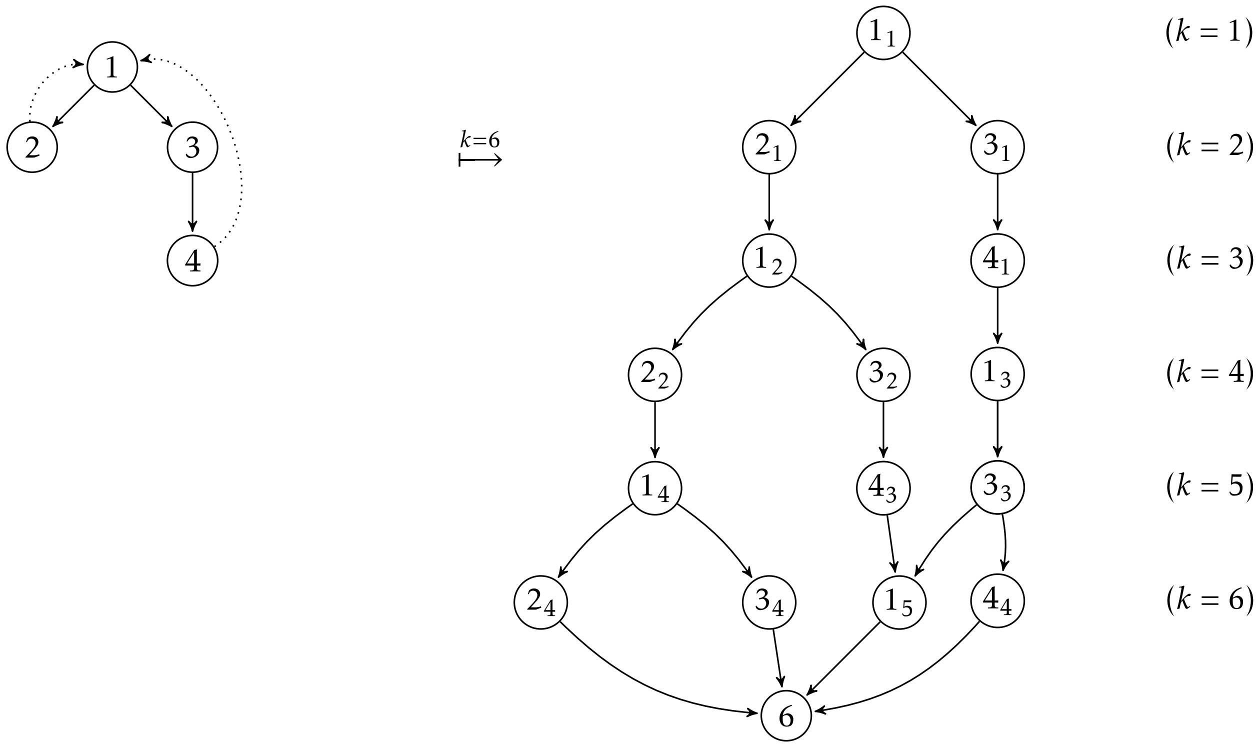


Figure 4.1: *Example of the flow graph from the Figure* **??**, *unwinded up to the bound* $k = 6$

4.5 XGraph to ZFormula (SMT) encoder

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

4.6 Optimisations

... performed on each stage

Chapter 5

Evaluation

5.1 Comparison with PORTHOS

5.1.1 Unique Features

5.1.2 Performance

5.2 Comparison with HERD

5.2.1 Unique Features

5.2.2 Performance

Chapter 6

Summary

Bibliography

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

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

[3] 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.

[4] 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.

[5] Susmit Sarkar et al. "Understanding POWER multiprocessors". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 175–186.

[6] Mark Batty et al. "Mathematizing C++ concurrency". In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 55–66.

[7] Peter Bailis et al. "Highly available transactions: Virtues and limitations". In: *Proceedings of the VLDB Endowment* 7.3 (2013), pp. 181–192.

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

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

[10] Edmund M Clarke et al. "Model checking and the state explosion problem". In: *Tools for Practical Software Verification*. Springer, 2012, pp. 1–30.

[11] Edmund Clarke et al. "Bounded model checking using satisfiability solving". In: *Formal methods in system design* 19.1 (2001), pp. 7–34.

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

[13] Hélene Collavizza and Michel Rueher. "Exploration of the capabilities of constraint programming for software verification". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2006, pp. 182–196.