AUTOMATED REASONING UNDER WEAK MEMORY CONSISTENCY

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Michail Kokologiannakis

Date of Defense: 14.12.2023

Dean:	Prof. Dr. Christoph Garth
Reviewer:	Dr. Viktor Vafeiadis
Reviewer:	Prof. Dr. Constantin Enea
Reviewer:	Prof. Dr. Mohamed Faouzi Atig

DE-386

Στην οικογένειά μου, Γιάννη, Ελένη και Αλέξη

To my family, Yannis, Eleni and Alexis

ABSTRACT

Weak memory consistency models capture the outcomes of concurrent programs that appear in practice and yet cannot be explained by thread interleavings. Such outcomes pose two major challenges to formal methods. First, establishing that a memory model satisfies its intended properties (e.g., supports a certain compilation scheme) is extremely error-prone: most proposed language models were initially broken and required multiple iterations to achieve soundness. Second, weak memory models make verification of concurrent programs much harder, as a result of which there are no scalable verification techniques beyond a few that target very simple models.

This thesis presents solutions to both of these problems. First, it shows that the relevant metatheory of weak memory models can be effectively decided (sparing years of manual proof efforts), and presents Kater, a tool that can answer metatheoretic queries in a matter of seconds. Second, it presents GenMC, the first (and only) scalable stateless model checker that is parametric in the choice of the memory model, often improving the prior state of the art by orders of magnitude.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Viktor Vafeiadis, who influenced my thinking in more ways than I could imagine. Guidance and technical skills aside, he taught me that what lies at the heart of high-quality research is the combination of abstraction, mathematical rigor, and the elegance of simplicity. A prime example of academic leadership, Viktor was always there, as a mentor and a friend. Thank you, Viktor, from the bottom of my heart.

Throughout my PhD, I was also very fortunate to be unofficially mentored by Rupak Majumdar. Rupak was an endless source of motivation through stimulating scientific discussions, continuous feedback on my research, but also encouraging words and witty remarks.

The work presented in this thesis would not have been possible without a number of wonderful collaborators: Constantin Enea, Dimitra Giannakopoulou, Vladimir Gladstein, Ilya Kaysin, Ori Lahav, Rupak Majumdar, Iason Marmanis, Evgenii Moiseenko, Azalea Raad, Xiowei Ren, and Kostis Sagonas. Most of the papers in this thesis were written in close collaboration with Iason Marmanis, Azalea Raad and Ori Lahav, with whom I immensely enjoyed working together.

I am thankful to Constantin Enea and Mohamed Faouzi Atig for reviewing the thesis; to Dimitra Giannakopoulou for hosting me in AWS and making San Francisco sound like "Agios Fragkiskos"; and to Kostis Sagonas for introducing me to the world of verification.

I am also indebted to the past and current members of the Software Analysis and Verification group for fostering a friendly and welcoming environment: Soham Chakraborty, Marko Doko, Aristotelis Koutsouridis, Iason Marmanis, Anton Podkopaev, Lovro Rožič, and particularly to Azalea Raad and Léo Stefanesco, with whom I shared an office for the first and second half of my PhD, respectively.

During my PhD journey, I was also very fortunate to be accompanied by a number of friends, old and new, in Germany and abroad. In lieu of a dedicated page in the thesis containing their names, I extend to each and every one of them a heartfelt "thank you". Special thanks to Amir Bahador, James, Ivan, Kaushik, Manuel, Maria, Nastaran, Rosa, Andriana, Vasilis and Zannis for their help in different times.

I thank my fiancée, Kyriaki, for her love, support and patience during the past few years.

Finally, I would like to thank my parents, Yannis and Eleni, and my brother, Alexis, for all the sacrifices they made so that I can pursue my studies. This thesis is dedicated to them, as a token of an otherwise ineffable gratitude.

CONTENTS

III Tools & Evaluation

IV Conclusion

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

CAS	:	Compare-and-Swap
CFG	:	Control-Flow Graph
DFS	:	Depth-first Search
DPOR	:	Dynamic Partial Order Reduction
FAI	:	Fetch-and-Increment
RA	:	Release-Acquire
SC	:	Sequential Consistency
SCC	:	Strongly Connected Component
SMC	:	Stateless Model Checking
SSA	:	Static Single Assignment
TSO	:	Total Store Ordering

1 INTRODUCTION

In the modern age of computing, the world is concurrent: from multicore CPUs all the way up to the network stack and the users themselves, operations take place concurrently. In turn, reasoning about concurrency is of utmost importance.

This thesis focuses on *automated reasoning* of concurrent programs.

Traditionally, automated reasoning techniques assume sequential consistency (SC)[1], i.e., that all the behaviors of a concurrent program can be generated by some arbitrary interleaving of its threads. This assumption, however, is wrong. Due to compiler and/or hardware optimizations, concurrent programs can exhibit a number of additional behaviors, which are referred to as "weak" behaviors.

As an example, consider the MP program below, where *data* and *flag* are shared variables (initially 0), and || separates different threads:

$$\begin{array}{l||l} \mathit{data} := 42 & \texttt{if}\ (\mathit{flag} = 1) \\ \mathit{flag} := 1 & \quad \texttt{assert}(\mathit{data} = 42) \end{array} \qquad (\textsc{mp})$$

Under SC, this program is safe: if thread II reads $flag = 1$, the write to *data* will already have happened, and the assertion will not be violated. In architectures like ARMv8 or POWER[2], however, the CPU is within its rights to *reorder* the instructions of thread I, thereby leading to a weak behavior where thread II reads $flag = 1$ and $data = 0$.

Reasoning about concurrency in the presence of such behaviors requires precise models. The formal models describing the exact behaviors that concurrent programs can exhibit are called (*weak*) *memory models*, while the field that studies such models is called *weak memory consistency* (or simply weak memory).

In the past few years, a plethora of memory models has emerged. These models do not solely concern hardware architectures like ARMv8 and POWER, but also extend to languages like C/C++ and Java[3]. Indeed, the purpose of such language models is not merely to define the behaviors of the CPU, but rather to define the concurrency semantics at the level of the programming language, and provide guarantees that carry over all the way down to the produced binary.

1.1 CHALLENGES OF WEAK MEMORY CONSISTENCY

Along with the emergence of weak memory models, new challenges to automated reasoning emerged. This thesis addresses two major challenges.

[1] *"How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs"* [Lam79]

[2] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [AMT14]

[3] *"Mathematizing C++ concurrency"* [Bat+11]; *"The Java memory model"* [MPA05]

First, how do we establish that a given memory model M satisfies its intended metatheoretic properties? A concrete example of such a property would be the correctness of a local program transformation in M (e.g., reordering of independent memory accesses). For such a transformation to be sound in M, it must not introduce any new behaviors.

Unfortunately, even though there is a long line of work trying to establish such properties for various memory models, these topics used to require lengthy human investigation, and were extremely error prone: most proposed language models were initially broken (e.g., C/C++, Java) and required multiple iterations to achieve soundness.

Second, how do we verify programs under weak memory consistency? As we are interested in automated verification, *model checking*[4] seems like the obvious solution. In a nutshell, model checking verifies a program (expressed as a finite-state machine) by exploring all its reachable states, and checking that none of them violates a provided specification.

Despite its success in the context of sequential programs, model checking is inadequate when it comes to concurrency. A first major disadvantage is the increased memory consumption caused by the state-explosion problem. Indeed, as the state space of a concurrent program grows larger, so does the memory required to keep track of all explored states, rendering verification intractable. A second (and perhaps more relevant) limitation is that model checking does not gracefully extend to weak memory models. Even though the effects of certain weak memory models can be simulated by encoding instruction reorderings as non-deterministic choice, such an encoding further blows up the state space, leaving much to be desired for concurrent program verification.

1.2 CONTRIBUTIONS

This thesis presents scalable and practical solutions to both challenges arising from weak memory models.

First, it presents Kater, a sound, complete, and automated way to prove metatheoretic properties of weak memory models. More specifically, the thesis shows that most metatheoretic queries can be solved by answering a more fundamental question: "Given two memory models, is one weaker than the other?" For a wide class of weak memory models, this basic question can be reduced to a language inclusion problem between regular languages, which is decidable. Kater can answer metatheoretic queries in a matter of seconds, effectively sparing years of manual proof efforts.

Second, it presents GenMC, a model-checking algorithm that is *parametric* in the choice of the memory model, and has *linear* memory consumption in the size of the program under test. GenMC is a based on a technique called *dynamic partial order reduction* (DPOR)[5], which, under

[4] *"Automatic verification of finite-state concurrent systems Using temporal logics specification: A practical approach"* [CES83]

[5] *"Dynamic partial-order reduction for model checking software"* [FG05]

SC, verifies a concurrent program by partitioning its interleavings into equivalence classes, and then striving to explore one interleaving per equivalence class.

Crucially, GenMC reconciles two notions that were incompatible in past DPOR approaches: linear memory consumption and optimality (i.e., exploring one interleaving per equivalence class). Prior techniques would either require memory exponential in the size of the program under verification, or explore (exponentially) many unnecessary interleavings. By contrast, GenMC is the first DPOR framework that is a) optimal, b) has linear memory consumption, and c) is memory--model-parametric.

Finally, in addition to a number of optimizations that enhance its performance for various programming patterns such as synchronization barriers, locks and spinloops, GenMC extends DPOR for *persistency models* as well. Similarly to how weak memory models describe the behaviors that concurrent programs can exhibit (i.e., the values that their loads can read), persistency models describe the behaviors that persistent storage (e.g., hard drive, non-volatile memory) can exhibit in the presence of crashes. GenMC is the first DPOR algorithm that can verify persistency properties of both sequential and concurrent programs.

A key ingredient of both solutions above was a shift in representation. Rather than following existing work and representing program executions as traces, this thesis employs *declarative semantics*[6] and represents executions as partially-ordered structures known as *execution graphs*. Execution graphs are used in the weak memory literature to formalize the concurrency semantics of modern hardware architectures.

Using declarative semantics is crucial for both contributions. For Kater, declarative semantics is key in observing that most weak memory models can be expressed using Kleene Algebra with Tests (KAT)[7], an observation that largely explains why metatheoretic queries can be automated. In the case of GenMC, execution graphs not only subsume the equivalence classes that are used by traditional DPOR approaches, but also enable support for weak memory consistency. In other words, execution graphs form a new foundation for DPOR, suitable for model checking under weak memory.

1.3 STRUCTURE

The thesis is structured in four parts.

Part I deals with weak memory metatheory. First, it describes how Kater automates weak memory metatheory (§3), and then how consistency is checked in execution graphs, and how Kater can be used to synthesize code that checks whether a given graph is consistent (§4).

Part II describes GenMC. After a brief introduction to DPOR, it describes the basic algorithm of GenMC and the requirements it sets on the underlying weak memory model (§5), and then how the algorithm

[6] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [*AMT14*]

[7] *"Kleene Algebra with Tests"* [Koz97]

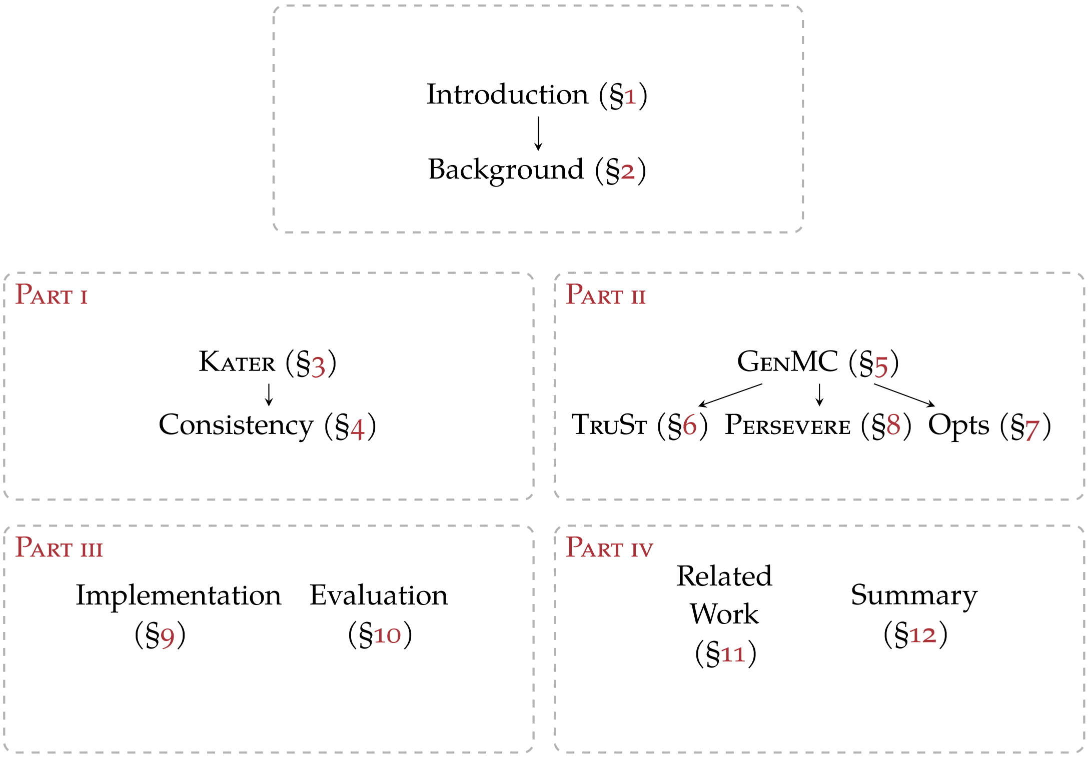


Figure 1.1: The part- and chapter dependencies of the thesis.

is modified in order to achieve linear memory consumption (§6). Subsequently, we explain how GenMC can be extended to handle persistency models (§8), and the optimizations that enhance its scalability in the presence of various programming patterns (§7).

Part III describes the implementation of Kater and GenMC (§9), and evaluates the tools on various benchmarks (§10).

Part IV concludes with a description of related work (§11), a summary of the contributions, and some future directions (§12).

These four parts are split in three layers (see Fig. 1.1). Each layer assumes knowledge of the layer above it, but not of the parts on the same layer or the layers below. Analogously, a chapter in a given part only depends on the chapters above it. Each chapter begins with a high-level description of the problem being solved, the key idea underpinning the solution, and a chapter outline.

Throughout the thesis, non-colored hyperlinks are used to connect definitions to their usages. Such hyperlinks are used for the employed notation (e.g., `sbr`), terms (e.g., maximal extension), as well as for certain exploration examples (e.g., ②). In certain viewers (e.g., `skim` in MacOS, `evince` in GNU/Linux), hovering over such hyperlinks offers a preview of the target.

1.4 PUBLICATIONS AND IMPACT

PUBLICATIONS The work making up this thesis has been published in the following papers (in reverse chronological order):

POPL 2023 KATER: Automating Weak Memory Metatheory and Consistency Checking
Michalis Kokologiannakis, Ori Lahav, Viktor Vafeiadis [KLV23b]

POPL 2022 Truly Stateless, Optimal Dynamic Partial Order Reduction
Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, Viktor Vafeiadis [Kok+22a]

FMCAD 2021 Dynamic Partial Order Reductions for Spinloops
Michalis Kokologiannakis, Xiaowei Ren, Viktor Vafeiadis [KRV21]

CAV 2021 GENMC: A Model Checker for Weak Memory Models
Michalis Kokologiannakis, Viktor Vafeiadis [KV21b]

NETYS 2021 BAM: Efficient Model Checking for Barriers
Michalis Kokologiannakis, Viktor Vafeiadis [KV21a]

POPL 2021 PerSeVerE: Persistency Semantics for Verification under Ext4
Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, Viktor Vafeiadis [Kok+21]

ASPLOS 2020 HMC: Model Checking for Hardware Memory Models
Michalis Kokologiannakis, Viktor Vafeiadis [KV20]

PLDI 2019 Model Checking for Weakly Consistent Libraries
Michalis Kokologiannakis, Azalea Raad, Viktor Vafeiadis [KRV19]

TOOLS GENMC and KATER are provided as open-source tools. The following repositories contain instructions on how to build and use the tools:

- KATER:
  `https://github.com/MPI-SWS/kater`
- GENMC:
  `https://github.com/MPI-SWS/genmc`

Among the two tools, GENMC has been more successful (perhaps due to it being more mature) and has attracted some industrial users. As an example, Huawei researchers have used it to verify and optimize the placement of fences in their synchronization library [8], while other works have found bugs in published algorithms [9] and in Microsoft's mimalloc allocator [10].

While working on extending GENMC for persistency models, we also used GENMC to reproduce a bug in the `nano` text editor, and formally verify our fix[11].

[8] *"VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models"* [*Obe+21b*]

[9] *"Verifying and Optimizing the HMCS Lock for Arm Servers"* [*Obe+21a*]

[10] Play nice with thread sanitizer #130 [*mar*]; fix memory order for weak CAS [*daa*]

[11] *"PerSeVerE: Persistency semantics for verification under ext4"* [*Kok+21*]

SUPPLEMENTARY MATERIAL All papers and supplementary material making up this thesis (including proofs and Coq development, where available) can be found at the following pages:

- Kater:
  https://plv.mpi-sws.org/kater
- GenMC:
  https://plv.mpi-sws.org/genmc
- GenMC for persistency models:
  https://plv.mpi-sws.org/persevere
- Benchmarks and tools (§10):
  doi.org/10.5281/zenodo.10575926

2 BACKGROUND

Following the standard declarative (a.k.a. axiomatic) approach of weak memory models, the semantics of a given program is given as a set of allowed outcomes. These outcomes are expressed as *execution graphs*[12].

In this chapter, we present how programs are mapped to execution graphs. First, we define a toy programming language (§2.1), then, we define execution graphs (§2.2) and demonstrate how they are used in declarative semantics (§2.3), and finally, we show how execution graphs are constructed from programs (§2.4).

RELATIONAL NOTATION We write $\emptyset$, $\mathtt{univ}$, and $\mathtt{id}$ for the empty, the full, and the identity relation, respectively. Given a relation r, we write r^{-1} for its inverse (i.e., $\{\langle a,b\rangle \mid \langle b,a\rangle \in \mathsf{r}\}$), and $\mathsf{r}^?$, r^+ and r^* for its reflexive, transitive and reflexive-transitive closures, respectively. We write $\mathit{dom}(\mathsf{r})$ and $\mathit{rng}(\mathsf{r})$ for the domain and range of r, respectively, and $\mathsf{r}|_{\mathsf{imm}}$ for the immediate edges in r, i.e., $\mathsf{r} \setminus (\mathsf{r}\,;\mathsf{r})$. Given a set S, we write $\mathsf{r}|_S$ for the restriction of r on S. Given two relations r_1 and r_2, we write $\mathsf{r}_1 \,;\, \mathsf{r}_2$ for their relational composition, i.e., $\{\langle a,b\rangle \mid \exists c.\, \langle a,c\rangle \in \mathsf{r}_1 \wedge \langle c,b\rangle \in \mathsf{r}_2\}$. Given a set A, we write $[A]$ for the identity relation on A: $\{\langle a,a\rangle \mid a \in A\}$.

We say that a relation r is *irreflexive* if $\nexists a.\, \langle a,a\rangle \in \mathsf{r}$ and *acyclic* if r^+ is irreflexive. A relation is a *strict partial order* if it is irreflexive and transitive. A relation r is *total* on a set A if $\langle a,b\rangle \in \mathsf{r} \cup \mathsf{r}^{-1} \cup [A]$ for all $a, b \in A$. A relation is a *strict total order* on a set A if it is a strict partial order that is total on A.

Given a total order r on a set A and an element $a \in A$, we write $\mathrm{succ}_{\mathsf{r}}(a)/\mathrm{pred}_{\mathsf{r}}(a)$ for the immediate successor/predecessor of a in r. Given a set $S \subseteq A$, we write $\max_{\mathsf{r}}(S)/\min_{\mathsf{r}}(S)$ for the r-maximal/minimal element in S.

2.1 PROGRAMMING LANGUAGE

We start by defining a simple untyped assembly language. Instructions in our language, $i \in \mathsf{Inst}$, are given by the following grammar:

$$i ::= r := e \mid r := [e]^{a_{\mathsf{R}}} \mid [e_1]^{a_{\mathsf{W}}} := e_2 \mid \mathtt{fence}^{a_{\mathsf{F}}} \mid \mathbf{if}\ e\ \mathbf{goto}\ n$$

where $r \in \mathsf{Reg}$ ranges over registers, $n \in \mathbb{N}$ over integers, and $e \in \mathsf{Exp}$ over simple expressions built from integers, registers, and arithmetic operators:

$$e ::= n \mid r \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots$$

[12] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [*AMT14*]

In the instructions above, $a_{\mathsf{R}} \subseteq \mathsf{Rattr}$, $a_{\mathsf{W}} \subseteq \mathsf{Wattr}$, and $a_{\mathsf{F}} \subseteq \mathsf{Fattr}$ range over read, write, and fence *attributes*, respectively, many of which are going to be introduced in subsequent chapters. Attributes include the mode of a memory access[13], which can be used to enforce synchronization, as well as the exclusivity flag, excl, denoting whether a memory access is part of an atomic read-modify-write (RMW) instruction (see below).

Returning to the instructions, $r := e$ assigns the value of e to register r (without any effect on memory); $r := [e]$ reads the value in the address pointed by e and stores it in register r; $[e_1] := e_2$ stores the value contained in e_2 in the address contained in e_1; $\texttt{fence}^{a_{\mathsf{F}}}$ is used to place global barriers; and $\texttt{\textbf{if}}\ e\ \texttt{\textbf{goto}}\ n$ jumps to n if e has a non-zero value. We assume that the special addresses `block` and `error` denote program blocking and error, respectively.

Notice that our language does not contains atomic read-modify-write (RMW) operations or functions like `assert` and `assume`, as these can be defined as syntactic sugar over a sequence of instructions:

$$
\begin{aligned}
r := \texttt{CAS}(e_1, e_2, e_3) &\triangleq r := [e_1]^{\mathsf{excl}} \\
&\quad\ \texttt{\textbf{if}}\ r \neq e_2\ \texttt{\textbf{goto}}\ pc + 2 \\
&\quad\ [e_1]^{\mathsf{excl}} := e_3 \\
r := \texttt{fetch\_add}(e_1, e_2) &\triangleq r := [e_1]^{\mathsf{excl}} \\
&\quad\ [e_1]^{\mathsf{excl}} := e_2 + r \\
\texttt{assert}(e) &\triangleq \texttt{\textbf{if}}\ \neg e\ \texttt{\textbf{goto}}\ \texttt{error} \\
\texttt{assume}(e) &\triangleq \texttt{\textbf{if}}\ \neg e\ \texttt{\textbf{goto}}\ \texttt{block}
\end{aligned}
$$

where $pc \in \mathsf{Reg}$ is the program counter.

Finally, a sequential program, S, is simply a collection of instructions (defined as a finite map from $\mathbb{N}$ to instructions), while a concurrent program, P, is a parallel composition of sequential programs (defined as a finite map from thread identifiers to sequential programs).

In all examples, we use vertical alignment to denote sequences of instructions and || for the parallel composition of threads. We use x, y, z for global (shared) variables and $a, b, c, \ldots$ for registers. We also omit the square brackets for global variables, as their address is known and does not need to be computed (as in e.g., the MP example of §1).

Remark 1. The actual implementation of GenMC does not operate on the toy language defined above. It operates on the level of LLVM-IR, and can handle the full complexity of languages like C and C++ (including dynamic thread creation, parts of the standard library, etc). As such features are orthogonal to the weak memory semantics, we omit them here. See §9.2 for more details.

[13] *The precise definition of access modes is not important for this thesis and depends on the memory model. For example, C11 [Bat+11] has non-atomic, relaxed, acquire, release, acquire-release, and SC accesses.*

2.2 EXECUTION GRAPHS

An execution graph G models a distinct behavior of a given program. It comprises (a) a set of events (nodes), modeling instructions of the program, and (b) some relations on these events (edges), modeling the various interactions between the instructions. The two kinds of relations present in all memory models are the *program order* ($\mathtt{po}$), ordering events in a given thread, and the *reads-from* relation ($\mathtt{rf}$), which relates each read event r in G to a write event w in G, from which r obtains its value.

Let us begin by defining execution graph events.

Definition 2.2.1. An *event*, $e \in \mathsf{Event}$, is either the initialization event $\mathtt{init}$, or a thread event $\langle t, n, lab\rangle$ where $t \in \mathsf{Tid}$ is a thread identifier, $n \in \mathsf{Idx} \triangleq \mathbb{N}$ is a serial number inside each thread, and $lab \in \mathsf{Lab}$ is a label that takes one of the following forms:

- Read label: $\mathtt{R}^{a_\mathtt{R}}(l)$ where $a_\mathtt{R} \subseteq \mathsf{Rattr}$ denotes any attributes the read might have, and $l \in \mathsf{Loc}$ is the location accessed.
- Write label: $\mathtt{W}^{a_\mathtt{W}}(l, v)$ where $a_\mathtt{W} \subseteq \mathsf{Wattr}$ denotes any attributes the write might have, $l \in \mathsf{Loc}$ is the location accessed, and $v \in \mathsf{Val}$ the value written.
- Fence label: $\mathtt{F}^{a_\mathtt{F}}$ where $a_\mathtt{F} \subseteq \mathsf{Fattr}$ denotes any attributes the fence might have.
- Block label: $\mathtt{B}$ denotes the blockage of a thread (due to reaching the `block` address).
- Error label: `Error` denotes a thread error (due to reaching the `error` address).

Given an event e, we use $e.\mathtt{tid}$, $e.\mathtt{idx}$ and $e.\mathtt{lab}$ to project to its components. We omit the $\emptyset$ for read/write labels with no attributes. The functions $\mathtt{tid}$, $\mathtt{idx}$, and $\mathtt{loc}$, respectively return the thread identifier, serial number, and location of an event, when applicable. We use $\mathtt{R}$, $\mathtt{W}$, $\mathtt{B}$, and $\mathtt{error}$ to denote the set of all read, write, block, and error events, respectively, and assume that $\mathtt{init} \in \mathtt{W}$. We use superscript and subscripts to further restrict those sets (e.g., $\mathtt{W}_l \triangleq \{\mathtt{init}\} \cup \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$).

Observe that event labels correspond to instructions with memory side-effects. Assignments ($r := e$) and conditionals (**`if`** e **`goto`** n) do not generate any events.

Now, we can formally define execution graphs as follows.

Definition 2.2.2. An *execution graph* $G \in \mathsf{Exec}$ consists of:

1. a sequence $G.\mathtt{E}$ of distinct events, and
2. the reads-from relation $G.\mathtt{rf} \subseteq \mathtt{W} \times \mathtt{R}$, that relates each write event to the same-location reads that read from it.

We often use $G.\mathsf{E}$ to denote the set of graph events (implicit conversion). We write $G.\mathsf{R}$ for the set $G.\mathsf{E} \cap \mathsf{R}$ and similarly for other sets. Given two events $e_1, e_2 \in G.\mathsf{E}$, we write $e_1 <_G e_2$ if e_1 precedes e_2 in $G.\mathsf{E}$, and $e_1 \leq_G e_2$ if $e_1 <_G e_2$ or $e_1 = e_2$. We write $G|_E$ for the restriction of an execution graph G to a set of events E, and $G \setminus E$ for the graph obtained by removing a set of events E. Finally, we write $G_1 \approx G_2$ if the $G_1.\mathsf{E}$ is a permutation of $G_2.\mathsf{E}$, and G_1 and G_2 agree on all other components.

Observe that the definition of graphs does not contain a *program order* ($\mathtt{po}$) as an explicit component, since $G.\mathtt{po}$ can be recovered from the representation of events: it relates the initialization event before all other events, and events in the same thread according to their n component.

$$\begin{aligned} G.\mathtt{po} \triangleq & \{\langle \mathtt{init}, e\rangle \mid e \in G.\mathsf{E} \setminus \{\mathtt{init}\}\} \cup \\ & \{\langle e, e'\rangle \in G.\mathsf{E} \times G.\mathsf{E} \mid \mathtt{tid}(e) = \mathtt{tid}(e') \land \mathtt{idx}(e) < \mathtt{idx}(e')\} \end{aligned}$$

We write $G.\mathtt{rmw} = [G.\mathsf{R}^{\mathsf{excl}}]; G.\mathtt{po}|_{\mathrm{imm}}; [G.\mathsf{W}^{\mathsf{excl}}]$ for the restriction of $\mathtt{po}$ to (immediate) pairs forming RMW instructions[14].

As far as the $\mathtt{rf}$ relation is concerned, we ensure that it only relates same-location events by requiring that G be well-formed:

Definition 2.2.3 (Well-formedness). An execution graph G is *well-formed* if the following hold for $G.\mathtt{rf}$:

1. $G.\mathtt{rf}$ only relates writes and reads with matching locations, i.e., for every $\langle w, r\rangle \in G.\mathtt{rf}$ it is $w \in G.\mathsf{W}$, $r \in G.\mathsf{R}$, $\mathtt{loc}(w) = \mathtt{loc}(r)$,
2. $G.\mathtt{rf}$ is functional on its range, i.e., if $\langle w_1, r\rangle, \langle w_2, r\rangle \in G.\mathtt{rf}$ it is $w_1 = w_2$, and
3. each read reads a value, i.e., $\forall r \in G.\mathsf{R}.\ \exists w.\ \langle w, r\rangle \in G.\mathtt{rf}$.

As $\mathtt{rf}$ is functional on its range, we sometimes write $G.\mathtt{rf}(r)$, to refer to the unique write $w \in G.\mathsf{W}$ such that $\langle w, r\rangle \in G.\mathtt{rf}$, and write $G.\mathtt{val}(e)$ for the value read/written by a read/write.

We write $G.\mathtt{porf}$ for $(G.\mathtt{po} \cup G.\mathtt{rf})^+$, and also define the *causal order* of a graph, and the *causal prefix* of an event e as follows:

$$\begin{aligned} G.\mathsf{corder} &\triangleq G.\mathtt{porf} \\ G.\mathsf{cprefix}(e) &\triangleq \{e' \in G.\mathsf{E} \mid \langle e', e\rangle \in G.\mathsf{corder}\} \end{aligned}$$

Intuitively, the causal prefix of an event e represents the minimal set of events that need to be executed before executing e.

The *same-location* relation, $\mathtt{sameloc}$, relates pairs of events that have the same location: $\mathtt{sameloc} \triangleq \{\langle e_1, e_2\rangle \in \mathsf{Event} \times \mathsf{Event} \mid \mathtt{loc}(e_1) = \mathtt{loc}(e_2)\}$. Using $\mathtt{sameloc}$, we define a per-location version of $\mathtt{po}$ as $\mathtt{poloc} \triangleq \mathtt{po} \cap \mathtt{sameloc}$.

Finally, most memory models also make use of the *coherence order* relation, $\mathtt{co}$, which totally orders the writes in each memory location. As such, we define the following augmented version of execution graphs.

[14] *For executions coming from programs (see §2.4), for every* $\langle r, w\rangle \in G.\mathtt{rmw}$, *it always is* $\mathtt{loc}(r) = \mathtt{loc}(w)$.

Definition 2.2.4. A *coherence-tracking execution graph* G is an execution graph with the following extra relation:

1. the *coherence order*, $G.\texttt{co} \subseteq \bigcup_{l \in \mathsf{Loc}} G.\mathtt{W}_l \times G.\mathtt{W}_l$, a strict partial order which is total on $G.\mathtt{W}_l$ for every location $l \in \mathsf{Loc}$.

In what follows, we write execution graph, graph, or execution to denote either a plain or a coherence-tracking execution graph. We explicitly disambiguate when necessary.

2.3 WEAK MEMORY CONSISTENCY MODELS

In declarative semantics, a memory model M is expressed using a consistency predicate, $\mathsf{consistent}_{\mathsf{M}}(\cdot)$, denoting what kind of behaviors are allowed under M. In turn, the semantics of a program P under M is given by the set of (well-formed) execution graphs corresponding to the program that satisfy the consistency predicate of M (see §2.4).

Consistency predicates generally constrain the possible choices of co and rf, which indirectly constrain the possible final values of memory locations and the values that reads can return. A non-coherence-tracking execution graph is consistent if there exists some co such that the resulting coherence-tracking execution graph is consistent.

For instance, SC[15] can be defined by making use of two auxiliary definitions.

First, we define the *reads-before* (a.k.a. from-read) relation to relate a read r and a write w if r reads from a co-earlier write than w: $\texttt{rb} \triangleq G.\texttt{rf}^{-1};\texttt{co}$.

Then, we define RMW-atomicity to disallow two RMWs to read from the same write:

Definition 2.3.1 (RMW-atomicity)**.** An execution graph G is *RMW-atomic* iff there are no two distinct exclusive reads that have corresponding exclusive writes (i.e., $r_i \in dom(G.\texttt{rmw})$ for $i \in \{1, 2\}$) and read from the same write w (i.e., $\langle w, r_i \rangle \in G.\texttt{rf}$ for $i \in \{1, 2\}$).

Definition 2.3.2 (SC)**.** An execution graph G is *sequentially consistent*, written $\mathsf{consistent}_{\mathsf{SC}}(G)$, iff G is RMW-atomic and $(G.\texttt{po}\,\texttt{rf} \cup G.\texttt{co} \cup \texttt{rb})^{+}$ is irreflexive.

As a concrete example of how consistency predicates rule out inconsistent executions, consider the MP program from §1 and its three consistent executions[16] under SC in Fig. 2.1. Intuitively, the MP program has three consistent executions under SC, because SC forbids the load of *data* to read from the initial state as the load is already aware of the *flag* := 1 store. Formally, reading $flag = 1$ and $data = 0$ creates an SC-cycle due to the rb edge from $\mathtt{R}(data)$ to $\mathtt{W}(data, 42)$[17]. Other models such as the "relaxed" fragment of RC11[18] allow this behavior.

[15] *"How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs"* [Lam79]

[16] *Throughout this thesis, we use green-circled numbers for consistent executions, and red-circled numbers for inconsistent ones.*

[17] *We generally do not depict* co *and* rb *edges to avoid cluttering the presentation*

[18] *"Repairing sequential consistency in C/C++11"* [Lah+17]

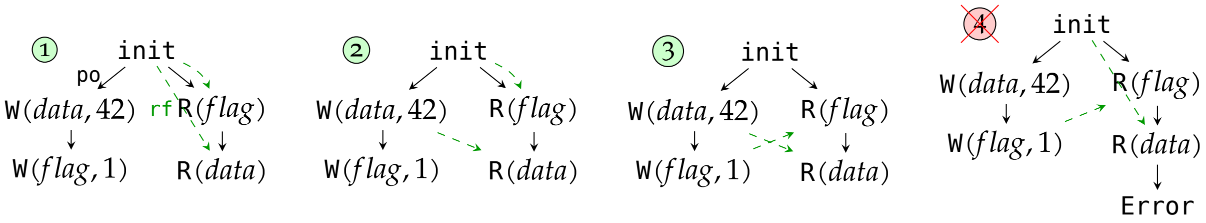


Figure 2.1: MP: three consistent execution graphs under SC.

Certain models enforce synchronization via a *happens-before* relation, $\texttt{hb} \supseteq \texttt{po}$, prescribing the order induced by synchronization mechanisms, e.g., locks[19]. Even though we do not restrict our framework to an explicit memory model, some of the algorithms presented later on (e.g., §7.1), do rely on the existence of hb. For these algorithms, we assume that hb is a parameter of the memory model, with the proviso that for all executions $G \in \mathsf{Exec}$, if $\mathsf{consistent}_\mathsf{M}(G)$ holds, then hb is a strict partial order.

Finally, models often define certain graphs as erroneous (e.g., due to data races). To account for this, we also assume an error predicate, $\textsc{IsErroneous}_\mathsf{M}(\cdot)$, denoting whether a given (consistent) graph is erroneous, and require it to be monotone: for all G and $E \subseteq G.\mathsf{E}$, if $dom(G.\mathsf{corder}; [E]) \subseteq E$ and $\textsc{IsErroneous}_\mathsf{M}(G|_E)$, then $\textsc{IsErroneous}_\mathsf{M}(G)$. We call a graph G M-erroneous if $\textsc{IsErroneous}_\mathsf{M}(G)$ holds.

2.4 FROM PROGRAMS TO EXECUTION GRAPHS

Now that we have defined execution graphs and consistency under weak memory, let us see how programs are mapped to sets of consistent execution graphs. We do so by defining the $\textsc{ExecProgram}(P, G)$ procedure (algorithm 2.1), which checks that the execution G corresponds to some run of the program P. Later, in §5.8, we extend this procedure to also generate the execution incrementally.

ExecProgram interprets the program P and checks that the memory accesses generated match those recorded in G. After all threads have been interpreted, ExecProgram ensures that G does not contain events from threads not originating from P (line 4).

The interpretation of each thread is performed by ExecThread. For each thread t of the program (line 2), ExecThread constructs a configuration of the form $\langle t, n, \Phi \rangle$, where n is the index in t that was considered, and $\Phi : \mathsf{Reg} \rightarrow \mathsf{Val}$ is the *register file* that maps registers to values. Initially, n is set to 0 signaling that no events have yet been checked for t, and every register has the value 0 (line 6).

The register set includes a special register, the *program counter* (*pc*), that points to the next instruction to be executed. The program counter is incremented by every instruction (line 8), except for conditional

[19] *For SC,* $\texttt{hb}$ *can be defined as* $\texttt{hb} \triangleq (\texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb})^+$, *as all instructions imply synchronization.*

Algorithm 2.1 Check that G is an execution of program P

```
 1: procedure ExecProgram($P, G$)
 2:     for $\langle t, sprog \rangle \in P$ do
 3:         ExecThread($t, sprog, G$)
 4:     assert($\forall e \in G.\mathsf{E}.\, P(\mathtt{tid}(e)) \neq \bot$)
 5: procedure ExecThread($t, sprog, G$)
 6:     $\langle t, n, \Phi \rangle \leftarrow \langle t, 0, \lambda r.\, 0 \rangle$
 7:     while $i \leftarrow sprog(\Phi(pc))$ do
 8:         $\Phi(pc) \leftarrow \Phi(pc) + 1$
 9:         ExecInstruction($G, \Phi, t, n, i$)
10:     assert($n = |\{e \in G.\mathsf{E} \mid \mathtt{tid}(e) = t\}|$)
11: procedure ExecInstruction($G, \Phi, t, n, i$)
12:     switch $i$ do
13:         case $i \equiv r := e$
14:             $\Phi(r) \leftarrow \Phi(e)$
15:         case $i \equiv r := [e]^{a_\mathsf{R}}$
16:             $\Phi(r) \leftarrow G.\mathtt{val}(\textsc{Gen}(G, \langle t, n+1, \mathsf{R}^{a_\mathsf{R}}(\Phi(e)) \rangle))$
17:         case $i \equiv [e_1]^{a_\mathsf{W}} := e_2$
18:             $\textsc{Gen}(G, \langle t, n+1, \mathtt{W}^{a_\mathsf{F}}(\Phi(e_1), \Phi(e_2)) \rangle)$
19:         case $i \equiv \mathtt{fence}^{a_\mathsf{F}}$
20:             $\textsc{Gen}(G, \langle t, n+1, \mathsf{F}^{a_\mathsf{F}} \rangle)$
21:         case $i \equiv$ if $e$ goto $l$
22:             if $\Phi(e) \neq 0$ then
23:                 $\Phi(pc) \leftarrow l$
24:                 if $l = \mathtt{error}$ then $\textsc{Gen}(G, \langle t, n+1, \mathtt{Error} \rangle)$
25:                 if $l = \mathtt{block}$ then $\textsc{Gen}(G, \langle t, n+1, \mathtt{B} \rangle)$
26: procedure Gen($G, a$)
27:     assert($a \in G.\mathsf{E}$)
28:     return $a$
```

branches where it is set to a specified value when the condition holds. We assume that the two special program counter values `error` and `block` do not point to valid instructions.

The interpretation of a thread proceeds in a loop as long as the program counter points to a valid instruction (line 7). In each loop iteration, ExecInstruction is called to interpret the current instruction (line 9). Under the usual assumption that programs are loop-free (or equivalently, that its loops are unrolled to some specified depth), the **while** loop is guaranteed to terminate[20]. Finally, when the loop finishes, ExecProgram checks that all events of G pertaining to thread t have been generated (line 10).

ExecInstruction does a case analysis over the type of the instruction, updating Φ as appropriate. For memory accesses, it calls the Gen

[20] *Technically, this renders algorithm 2.1 a semi-algorithm.*

helper function, which checks that the next event of the given thread recorded in G is the expected one. Whenever a read event a is generated, GEN returns the value read by looking up the value written by the write from which a reads (line 28). Whenever a branching instruction that jumps to the special `error` or `block` address is encountered, EXECINSTRUCTION ensures that the graph contains an `Error` or a `B` label, respectively.

We define the executions of a program P under a model M as the set of all M-consistent executions G generated by P; i.e., EXECPROGRAM(P, G) terminates without assertion violations and $\mathsf{consistent}_{\mathsf{M}}(G)$ holds. For example, notice how EXECPROGRAM would terminate without assertion violations for MP and the rightmost graph of Fig. 2.1, denoting that this graph is indeed an execution graph of MP. That execution graph, however, is inconsistent under e.g., SC, since it contains a $\mathtt{porf} \cup \mathtt{co} \cup \mathtt{rb}$ cycle.

Definition 2.4.1 (Program correctness)**.** A program is deemed *erroneous* under a memory model M if its executions under M contain an M-erroneous graph. A program is *correct* if it is not erroneous.

2.4.1 *Dependency-Tracking Models*

The consistency predicates of certain hardware memory models like ARMv8 or POWER require some additional components in their execution graphs. Indeed, in such architectures, we have to be able to express various instruction *dependencies* in order to reason about consistency.

To account for such models, we define dependency-tracking execution graphs.

Definition 2.4.2. A *dependency-tracking execution graph*, G, is an execution graph with the following extra relations (all functional on their range):

- the *address-dependency* relation, $G.\mathtt{addr} \subseteq \mathcal{P}(G.\mathsf{R}) \times (G.\mathsf{R} \cup G.\mathtt{W})$, that records the address dependencies of memory accesses.
- the *data-dependency* relation, $G.\mathtt{data} \subseteq \mathcal{P}(G.\mathsf{R}) \times G.\mathtt{W}$, that records the data dependencies of writes.
- the *control-dependency* relation, $G.\mathtt{ctrl} \subseteq \mathcal{P}(G.\mathsf{R}) \times G.\mathsf{E}$, that records the control dependencies of events.

We write $\mathtt{deps} \triangleq \mathtt{addr} \cup \mathtt{data} \cup \mathtt{ctrl}$, assume that all dependency edges are included in $\mathtt{po}$, i.e., $G.\mathsf{x} \subseteq \mathtt{po}$ for $\mathsf{x} \in \{\mathtt{addr}, \mathtt{data}, \mathtt{ctrl}\}$, and also assume that the memory model defines a causal order such that

$$(G.\mathtt{rf} \cup G.\mathtt{deps})^+ \subseteq G.\mathsf{corder} \subseteq G.\mathtt{porf}$$

We also modify the ExecProgram by adding a *dependency set*, $\Delta : \mathsf{Reg} \rightarrow \mathcal{P}(\mathsf{Event})$, which maps each register to the set of events used to calculate its value (see algorithm 2.2). Dependencies are recorded in Δ by ExecProgram in a straightforward manner.

Remark 2. One may wonder why the calculation of dependencies happens dynamically via algorithm 2.2 instead of statically. The reason for this is to avoid an over-approximation of dependencies: in the case of a static calculation, and especially in programs with more complicated control flow (e.g., loops, goto statements, etc), a static calculation often results in over-approximating the dependencies of instructions.

To see this, consider the following program:

```
    $a := x$
    $b := y$
    if $a = 42$ goto calcZ
    $a := b$
  calcZ :
    $z := a + 1$
```

In the program above (ignoring control dependencies), the $z := a+1$ instruction data-depends on either x or y, as a will take its value from one of the two respective read statements. Knowing from which one, however, is impossible to determine statically. Since $a := x$ is an instruction reading from shared memory, statically predicting the value of x (which will in turn determine whether $a := b$ is going to be executed) is impossible. Thus, while a static calculation of dependencies would over-approximate by making $z := a + 1$ depend on both reads, a dynamic calculation has precise knowledge of the write's dependencies.

Algorithm 2.2 Check that G is an execution of program P

```
procedure ExecProgram$(P, G)$
    for $\langle t, sprog \rangle \in P$ do
        ExecThread$(t, sprog, G)$
    assert$(\forall e \in G.\mathsf{E}.\, P(\mathtt{tid}(e)) \neq \bot)$

procedure ExecThread$(t, sprog, G)$
    $\langle t, n, \Phi, \Delta \rangle \leftarrow \langle t, 0, \lambda r.\, 0, \lambda r.\, \emptyset \rangle$
    while $i \leftarrow sprog(\Phi(pc))$ do
        $\Phi(pc) \leftarrow \Phi(pc) + 1$
        ExecInstruction$(G, \Phi, \Delta, t, n, i)$
    assert$(n = |\{e \in G.\mathsf{E} \mid \mathtt{tid}(e) = t\}|)$

procedure ExecInstruction$(G, \Phi, \Delta, t, n, i)$
    switch $i$ do
        case $i \equiv r := e$
            $\Phi(r) \leftarrow \Phi(e)$ ; $\Delta(r) \leftarrow \Delta(e)$
        case $i \equiv r := [e]^{a_\mathtt{R}}$
            $\Phi(r) \leftarrow G.\mathtt{val}(\textsc{Gen}(G, \langle t, n+1, \mathtt{R}^{a_\mathtt{R}}(\Phi(r')) \rangle, \Delta(r'), \emptyset, \Delta(pc)))$
            $\Delta(r) \leftarrow \{a\}$
        case $i \equiv [e_1]^{a_\mathtt{W}} := e_2$
            $\textsc{Gen}(G, \langle t, n+1, \mathtt{W}^{o_\mathtt{W}}(\Phi(e_1), \Phi(e_2)) \rangle, \Delta(e_1), \Delta(e_2), \Delta(pc))$
        case $i \equiv \mathtt{fence}^{a_\mathtt{F}}$
            $\textsc{Gen}(G, \langle t, n+1, \mathtt{F}^{a_\mathtt{F}}, \emptyset, \emptyset, \Delta(pc) \rangle)$
        case $i \equiv \mathtt{if}\ r\ \mathtt{goto}\ l$
            if $\Phi(r) \neq 0$ then
                $\Phi(pc) \leftarrow l$
                if $l = \mathtt{error}$ then $\textsc{Gen}(G, \langle t, n{+}1, \mathtt{Error}, \emptyset, \emptyset, \Delta(pc) \rangle)$
                if $l = \mathtt{block}$ then $\textsc{Gen}(G, \langle t, n{+}1, \mathtt{B}, \emptyset, \emptyset, \Delta(pc) \rangle)$
                $\Delta(pc) \leftarrow \Delta(pc) \cup \Delta(r)$

procedure Gen$(G, a, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl})$
    assert$(a \in G.\mathsf{E})$
    for $\mathsf{x} \in \{\mathtt{lab}, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl}\}$ do
        assert$(G.\mathsf{x}(a) = \mathsf{x})$
    return $a$
```

Part I

METATHEORY

3

KATER: AUTOMATING WEAK MEMORY MODEL METATHEORY

In this chapter, we present how Kater automates weak memory model metatheory. Before doing so, however, let us briefly discuss the kind of metatheoretic properties Kater automates.

In the past few years, there has been a large body of work on formal definitions of memory models. Along such definitions, there has also been a long line of work trying to establish basic metatheoretic properties of these definitions, answering questions such as:

- Is a given memory model *monotone* with respect to various natural strengthenings, such as inserting a memory fence, merging two threads into a single thread, or, if applicable, strengthening the mode of a memory access (e.g., from release to sequentially consistent)?
- Does a given model admit *local program transformations*, such as reordering of independent memory accesses?
- Given two memory models A and B, is A *weaker* than B? More generally, is a given *compilation scheme* from A to B (e.g., by inserting certain fences) sound?
- Does a given model rule out "out-of-thin-air" (OOTA) outcomes? That is, does it rule out dependency cycles?

Using Kater, we can automate the questions above for declarative models that are expressed as emptiness, acyclicity, and irreflexivity constraints over relational algebra terms.

The key observation that allows us to do so is that the fragment of relational algebra used in most definitions of memory models (e.g., SC, TSO, PSO, POWER, ARMv8, RC11, IMM) corresponds closely to *Kleene Algebra with Tests* (KAT)[21], an extension of regular expressions with a Boolean algebra over a collection of predicates describing a state.

Leveraging this insight, these questions can be automated as follows. First, we show that checking whether one model is weaker than another can naturally be expressed as a language inclusion problem that can be decided using finite-state automata (§3.3). While the constraints themselves are not directly encodable in vanilla KAT, memory model inclusion can be reduced to proving entailments between KAT formulae, which is decidable for simple classes of entailments. Then, using this result, we also show how to check monotonicity, correctness of program transformations, correctness of compilation mappings, lack of OOTA behaviors, prefix-closedness, and extensibility[22]

[21] *"Kleene Algebra with Tests"* [Koz97]

[22] *Properties like lack of OOTA, prefix-closedness and extensibility are useful for model checking and are explained in detail in* §5.

(Section 3.4 to 3.8).

Before explaining how Kater works, we start with some background on regular languages and KAT (Section 3.1 and 3.2). Readers familiar with these concepts may skip these sections.

3.1 REGULAR LANGUAGES AND FINITE STATE AUTOMATA

We fix an *alphabet* (i.e., a finite non-empty set) Σ. A *language* L is a set of words in Σ^*. We use $a, b, \ldots$ to range over Σ, and $u, v, w, \ldots$ to range over Σ^*.

A *non-deterministic finite automaton* (NFA) over Σ is a tuple $\langle Q, \delta, S, F\rangle$ where Q is a finite set of states, $S \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function which, given a state $q \in Q$ and a letter $a \in \Sigma$, returns the set of possible next states $\delta(q, a)$. By abuse of notation, we extend the domain of the transition function to take as parameters a set of states and a word as follows: $\delta(S, a) \triangleq \bigcup_{q \in S} \delta(q, a)$, $\delta(S, \epsilon) \triangleq S$, and $\delta(S, aw) \triangleq \delta(\delta(S, a), w)$.

The *language accepted* by an NFA contains all words for which there is a path from an initial state of the NFA to a final state: $\mathsf{L}(\langle Q, \delta, S, F\rangle) \triangleq \left\{ w \in \Sigma^* \;\middle|\; \delta(S, w) \cap F \neq \emptyset \right\}$. Two NFAs are *language-equivalent* iff they accept the same language.

A *deterministic finite automaton* (DFA) is an NFA that has exactly one initial state and where for every $q \in Q$ and $a \in \Sigma$, the set $\delta(q, a)$ contains at most one element. The *powerset construction* transforms an NFA $\langle Q, \delta, S, F\rangle$ over Σ into a language-equivalent DFA $\langle \mathcal{P}(Q), \delta_p, \{s_0\}, F'\rangle$ where $s_0 \triangleq S$, $F' \triangleq \left\{ s \subseteq Q \;\middle|\; s \cap F \neq \emptyset \right\}$, and $\delta_p(s, a) \triangleq \{\delta(s, a)\}$.

A *regular language* is one described by a regular expression or equivalently one accepted by an NFA. There are standard conversions from regular expressions to NFAs and vice versa. Regular languages are closed under:

- *union* $(L_1 \cup L_2)$;
- *concatenation* $(L_1 \,;\, L_2)$;
- *repetition* (L^*);
- *intersection* $(L_1 \cap L_2)$, by the product construction on NFAs: $\langle Q_1 \times Q_2, \delta_P, S_1 \times S_2, F_1 \times F_2\rangle$ where $\delta_P(\langle q_1, q_2\rangle) \triangleq \delta_1(q_1) \times \delta_2(q_2)$;
- *complementation* $(\overline{L})$, by conversion to DFA and complementing the set of final states;
- *reversal* (L^{-1}), by swapping initial and final states in NFA, and reversing the transitions;
- *substitution* $(L[L_1/a_1, \ldots, L_n/a_n])$, by replacing all a_i transitions of an NFA with automata accepting L_i;

- *rotational closure* ($\mathsf{ROT}(L) \triangleq \{uv \mid vu \in L\}$), which can be computed on an NFA N as $\bigcup_{q \in N.Q} \mathsf{After}_q \,;\, \mathsf{Before}_q$ where After_q is the NFA obtained from N by making q be its only initial state and Before_q is the NFA obtained from N by making q be the only final state; and

- *deduplication closure* ($\mathsf{DEDUP}(L) \triangleq \{w \in \Sigma^* \mid \exists n.\, w^n \in L\}$), which can be computed on an NFA[23].

Finally, inclusion and equivalence of regular languages are decidable (PSPACE-complete) by noting that $L_1 \subseteq L_2 \Leftrightarrow L_1 \cap \overline{L_2} = \emptyset$. Given that the expensive part of this inclusion checking is the DFA conversion as part of the complementation of L_2, there are algorithms[24] that avoid performing the DFA conversion upfront and perform it "on demand" while traversing the NFA of L_1.

3.2 KLEENE ALGEBRA WITH TESTS (KAT)

Kleene algebra with tests[25] (KAT) extends regular languages with a set of *tests*, over which there is a Boolean algebra.

Let Predicate be a finite set of primitive predicate symbols and Relation be a finite set of primitive relation symbols. KAT tests (t) and expressions (e) are given by the following grammar:

$$t ::= p \mid \mathsf{true} \mid \mathsf{false} \mid t_1 \cup t_2 \mid t_1 \cap t_2 \mid \overline{t}$$
$$e ::= [t] \mid r \mid e_1 \cup e_2 \mid e_1 \,;\, e_2 \mid e^*$$

where $p \in \mathsf{Predicate}$ ranges over primitive predicates and $r \in \mathsf{Relation}$ over primitive relations. KAT tests contain the usual Boolean operators, while KAT expressions contain tests, relations, union, sequencing, and iteration. Tests allow us to express the empty relation $\emptyset \triangleq [\mathsf{false}]$ and the identity relation $\mathtt{id} \triangleq [\mathsf{true}]$. Moreover, as usual, reflexive closure is expressed as $e^? \triangleq e \cup \mathtt{id}$ and transitive closure as $e^+ \triangleq e \,;\, e^*$.

KAT expressions are standardly interpreted as languages of *guarded* words, that is, alternating sequences of satisfiable tests and relations starting and ending with a test, $t_1 r_1 t_2 r_2 \ldots t_n r_n t_{n+1}$ for some $n \geq 0$. We write $\mathsf{L}(e)$ for the language induced by a KAT expression e.

KAT expressions can equivalently be interpreted as binary relations over a certain universe. In our context, we use execution graphs as models. Recall from §2.2 that each execution graph G consists of a set $G.\mathsf{E}$ of nodes, called *events*, and interpretations of primitive tests as subsets of events and of primitive relations as binary relations on events:

$$[\![.]\!]_G : \mathsf{Predicate} \to \mathcal{P}(G.\mathsf{E}) \qquad [\![.]\!]_G : \mathsf{Relation} \to \mathcal{P}(G.\mathsf{E} \times G.\mathsf{E})$$

[23] Is the power of a regular language regular? Is the root of a regular language regular? [Fil]

[24] *"Checking NFA equivalence with bisimulations up to congruence"* [BP13]

[25] *"Kleene Algebra with Tests"* [Koz97]

This interpretations are extended to KAT tests and expressions in the obvious way:

$$\begin{array}{lll}
[\![\mathsf{true}]\!]_G \triangleq G.\mathsf{E} & [\![\mathsf{false}]\!]_G \triangleq \emptyset & [\![\bar{t}]\!]_G \triangleq G.\mathsf{E} \setminus [\![t]\!]_G \\
[\![t_1 \cup t_2]\!]_G \triangleq [\![t_1]\!]_G \cup [\![t_2]\!]_G & [\![t_1 \cap t_2]\!]_G \triangleq [\![t_1]\!]_G \cap [\![t_2]\!]_G & [\![[t]]\!]_G \triangleq [[\![t]\!]_G] \\
[\![e_1 \cup e_2]\!]_G \triangleq [\![e_1]\!]_G \cup [\![e_2]\!]_G & [\![e_1 \,;\, e_2]\!]_G \triangleq [\![e_1]\!]_G \,;\, [\![e_2]\!]_G & [\![e^*]\!]_G \triangleq [\![e]\!]_G^*
\end{array}$$

On top of KAT expressions, KAT *formulas* are defined by the following grammar:

$$\phi ::= e_1 \subseteq e_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \Leftrightarrow \phi_2$$

KAT formulas are interpreted as sets of execution graphs in the standard way: for example, $[\![e_1 \subseteq e_2]\!] \triangleq \{G \mid [\![e_1]\!]_G \subseteq [\![e_2]\!]_G\}$ and $[\![\phi_1 \Rightarrow \phi_2]\!] \triangleq \{G \mid G \in [\![\phi_1]\!] \Rightarrow G \in [\![\phi_2]\!]\}$. The interpretation is extended to sets of KAT formulas in the obvious way: $[\![\Phi]\!] \triangleq \bigcap_{\phi \in \Phi} [\![\phi]\!]$. We say that a KAT formula ϕ *holds*, denoted by $\vdash \phi$, if $[\![\phi]\!]$ is equal to the set of all graphs. We write $\Phi \vdash \phi$ if $[\![\Phi]\!] \subseteq [\![\phi]\!]$.

Inclusion between KAT expressions (i.e., $\vdash e_1 \subseteq e_2$) is PSPACE-complete, and remains so even under basic assumptions like emptiness of a KAT expression ($e = \emptyset$) or transitivity of a primitive relation ($r \,;\, r \subseteq r$)[26]. Inclusion and equivalence can be decided either by algebraic techniques or by reduction to finite state automata. In the latter case, it is convenient to first convert the automata into a normal form that accepts only guarded words, and then apply standard ways of checking language inclusion/equivalence between automata.

Conversion into the normal form has to ensure: (1) that each automaton state has incoming edges being predicates and outgoing edges being relations (or the other way round), (2) that all outgoing edges from initial states are predicate edges, and (3) that all incoming edges to accepting states are predicate edges. To do so, any states with both kinds of incoming and outgoing transitions have to be duplicated and suitably restricted: adjacent predicate transitions of the form $[p_1] ; [p_2]$ are replaced with single composite transitions of the form $[p_1 \cap p_2]$, while adjacent transitions with relations are moved apart by adding a dummy $[\mathsf{true}]$ transition between them. Similarly, outgoing relation edges from initial states have to be prefixed with a dummy $[\mathsf{true}]$ transition, and conversely incoming relation edges to accepting states have to be postfixed with a $[\mathsf{true}]$ transition.

3.3 MEMORY MODELS AS KAT CONSTRAINTS

Axiomatic memory models can be formulated as a single emptiness constraint and a single irreflexivity constraint over KAT. For this purpose, we extend KAT formulas with a new construct $\mathsf{irreflexive}(e)$ with semantics $[\![\mathsf{irreflexive}(e)]\!] \triangleq \{G \mid \nexists a.\, \langle a, a \rangle \in [\![e]\!]_G\}$. Models with mul-

[26] *"Kleene Algebra with Tests: Completeness and Decidability"* [KS96]

tiple such constraints can be encoded because of the following basic relational algebra properties:

$$e_1 = \emptyset \land e_2 = \emptyset \Leftrightarrow e_1 \cup e_2 = \emptyset$$
$$\mathsf{irreflexive}(e_1) \land \mathsf{irreflexive}(e_2) \Leftrightarrow \mathsf{irreflexive}(e_1 \cup e_2)$$

Similarly, acyclicity constraints can be encoded as $\mathsf{acyclic}(e) \triangleq \mathsf{irreflexive}(e^+)$.

Formally, a *memory model* M is a pair of KAT expressions $\langle e_\emptyset, e_{\text{irr}} \rangle$, interpreted as a collection of execution graphs

$$[\![\langle e_\emptyset, e_{\text{irr}} \rangle]\!] \triangleq [\![e_\emptyset = \emptyset \land \mathsf{irreflexive}(e_{\text{irr}})]\!]$$

We say that a memory model M_1 is *stronger* than another model M_2 (and M_2 is *weaker* than M_1) if $[\![M_1]\!] \subseteq [\![M_2]\!]$. Two models are equivalent if they are both stronger and weaker than each other.

3.4 ADDING DOMAIN-SPECIFIC ASSUMPTIONS

To be able to prove interesting metatheoretic properties, we need to equip KATER with some domain-specific assumptions. We go over these assumptions using some rather simple examples.

3.4.1 *Extended Coherence Order*

We begin with a rather simple example. In addition to the two basic relations of §2.2, Lahav et al.[27] define the *extended coherence order* as the transitive closure of `rf`, `co` and `rb`: $\mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{co} \cup \mathtt{rb})^+$. Observe that `eco` can equivalently be expressed without the transitive closure as $\mathtt{rf} \cup (\mathtt{co} \cup \mathtt{rb})\,;\mathtt{rf}^?$.

Suppose that we want to automatically verify the latter claim. The idea is to think of the two different formulations of `eco` as regular expressions over the alphabet $\{\mathtt{rf}, \mathtt{rf}^{-1}, \mathtt{co}\}$, and then check for equivalence between them. In KATER, we would write the following[28]:

```
declare rf rf^{-1} co
let fr = rf^{-1};co
let eco_1 = (rf ∪ co ∪ fr)^+
let eco_2 = rf ∪ (co ∪ fr);rf^?
assert eco_1 = eco_2
```

With this input, KATER immediately returns a counterexample saying that $\mathtt{eco}_1$ accepts the string $\mathtt{rf}\,;\mathtt{rf}$ but $\mathtt{eco}_2$ does not.

We clearly want to dismiss this counterexample because `rf` takes us from a write to a read, and we know that an event cannot be both a read and a write. One way to do so is to tell KATER that the `rf` does not compose with itself:

[27] *"Repairing sequential consistency in C/C++11"* [Lah+17]

[28] *This a pretty-printed version of the actual input syntax, which uses ASCII (e.g., | for union and <= for inclusion).*

```
assume rf;rf = 0
```

Adding assumptions makes the language inclusion/equivalence problem more challenging. For some very simple kinds of assumptions, such as ones of the form $e = \emptyset$ (where e is a KAT expression), language inclusion remains decidable.

Proposition 1 ([KS96, Theorems 6 and 9])**.** Let e, e_1, and e_2 be KAT expressions. Then, $e = \emptyset \vdash e_1 \subseteq e_2$ if and only if $\vdash e_1 \subseteq e_2 \cup \mathsf{Relation}^* ; e ; \mathsf{Relation}^*$.

This time Kater returns $\texttt{rf} ; \texttt{co}$ as a counterexample, which we dismiss for the same reason. And since we are at it, let's also state that $\texttt{co} ; \texttt{rf}^{-1} = \emptyset$.

```
assume rf;co = 0
assume co;rf^{-1} = 0
```

Next comes a more interesting counterexample: $\texttt{co};\texttt{co}$. Here, the equivalence proof relies upon $\texttt{co}$ being transitive, but Kater has not way of knowing that. So, let's add the assumption:

```
assume co;co ⊆ co
```

Such transitivity assumptions can also be eliminated completely: to check that $\Phi \vdash \phi$ under the additional assumption that a primitive relation r is transitive, we can replace all uses of r in Φ and ϕ with r^+.

Proposition 2. Let ϕ be a KAT formula, Φ be a set of KAT formulas, and r be a primitive relation symbol. Then, $\Phi, r ; r \subseteq r \vdash \phi$ if and only if $\Phi[r^+/r] \vdash \phi[r^+/r]$.

Running Kater now reveals another interesting counterexample: $\texttt{rf}; \texttt{rf}^{-1} ; \texttt{co}$. What is missing is the knowledge that $\texttt{rf}^{-1}$ is functional: every read reads from exactly one write. Adding the missing assumption

```
assume rf;rf^{-1} ⊆ id
```

allows Kater to complete the equivalence proof and report success.

Assumptions of the form $e \subseteq \texttt{id}$ for an inclusion query $e_1 \subseteq e_2$ can be eliminated by saturating the right-hand-side. In terms of KAT expressions, we let $\mathsf{satid}(e, e_2) \triangleq e^* ; e_2'$, where e_2' is obtained from e_2 by replacing every $r \in \mathsf{Relation}$ with $r ; e^*$. This transformation can also be defined in terms of NFAs. For each state of the automaton, we can add a self loop accepting the language described by e. If e is a primitive relation r', then this construction immediately reaches a fixpoint: running the construction on a saturated automaton will not introduce any new edges. If e is a composite expression, however, the construction does not reach a fixpoint. Since it introduces new states in the automaton, for completeness, the construction needs to be repeated again (and again). In principle, this repetition can be stopped after exceeding the number of states of e_1, but we stop it after a single iteration.

Proposition 3. Let Φ be a set of KAT formulas, and e, e_1 and e_2 be KAT expressions. If $\Phi \vdash e_1 \subseteq \mathsf{satid}(e, e_2)$, then $\Phi, e \subseteq \mathtt{id} \vdash e_1 \subseteq e_2$.

We note that instead of assuming that $\mathtt{rf} ; \mathtt{rf}^{-1} \subseteq \mathtt{id}$, the proof can also be completed with the assumption $\mathtt{rf} ; \mathtt{rb} \subseteq \mathtt{co}$. This assumption can be used by saturating the right-hand-side of the inclusion query in a similar way: wherever there is a $\mathtt{co}$ transition from state a to b in its NFA, construct new states m and n, and add an $\mathtt{rf}$ transition from a to m, an $\mathtt{rf}^{-1}$ transition from m to n, and a $\mathtt{co}$ transition from n to b. Although this construction adds more new states for each substitution, it has the benefit that it applies only to states with $\mathtt{co}$ transitions as opposed to all states of the NFA of the right-hand-side.

Proposition 4. Let Φ be a set of KAT formulas, ϕ be a KAT formula, e be a KAT expression, and r be a primitive relation. If $\Phi[(r \cup e)/r] \vdash \phi[(r \cup e)/r]$, then $\Phi, e \subseteq r \vdash \phi$. Furthermore, in the case that $\vdash e[(r \cup e)/r] \subseteq r \cup e$, the converse holds as well.

To avoid users having to explicitly define assumptions as those above, we equip Kater with built-in theory Φ_{base} with primitive relations $\mathtt{rf}$, $\mathtt{co}$, and $\mathtt{rb}$ and predicates $\mathtt{R}$ and $\mathtt{W}$. It consists of the following assumptions encoding the basic properties:

- Disjoint tests: $\mathtt{R} \cap \mathtt{W} = \emptyset$.
- Domain and range restrictions: $\mathtt{rf} = [\mathtt{W}];\mathtt{rf};[\mathtt{R}]$, $\mathtt{co} = [\mathtt{W}];\mathtt{co};[\mathtt{W}]$, and $\mathtt{rb} = [\mathtt{R}] ; \mathtt{rb} ; [\mathtt{W}]$.
- Transitivity: $\mathtt{co} ; \mathtt{co} \subseteq \mathtt{co}$.
- From-read properties: $\mathtt{rf} ; \mathtt{rb} \subseteq \mathtt{co}$ and $\mathtt{rb} ; \mathtt{co}^{+} \subseteq \mathtt{rb}$.

The disjointness assumption is used to remove edges from the (normal-form) NFAs corresponding to KAT expressions: Kater removes any transitions labeled with tests containing (i.e., stronger than) $\mathtt{R}\cap\mathtt{W}$. More generally, disjointness assumptions can be eliminated using the following claim.

Proposition 5. Let Φ be a set of KAT formulas, ϕ be a KAT formula, p be a primitive predicate, and t be a test. Then, $\Phi[p \cap \bar{t}/p] \vdash \phi[p \cap \bar{t}/p]$ if and only if $\Phi, p \cap t = \emptyset \vdash \phi$.

In turn, the domain and range restrictions can easily be eliminated by replacing the left-hand-sides of the inclusions with their right-hand-sides throughout. This transformation is formally justified by the following proposition.

Proposition 6. Let Φ be a set of KAT formulas, ϕ be a KAT formula, e be a KAT expression, and r be a primitive relation. If $\Phi[e/r] \vdash \phi[e/r]$, then $\Phi, r = e \vdash \phi$. Furthermore, in the case that $\vdash e[e/r] = e$, the converse holds as well.

Finally, the transitivity assumption is eliminated using Prop. 2, and the from-read properties are eliminated using Prop. 4.

Using the converse directions of the propositions above, we also obtain completeness of this process, which entails decidability as we state next.

Proposition 7. The question whether $\Phi_{\mathtt{base}} \vdash e_1 \subseteq e_2$ given two KAT expressions e_1 and e_2 is decidable.

Proof (*sketch*). First, the assumption $\mathtt{rb} \mathbin{;} \mathtt{co}^+ \subseteq \mathtt{rb}$ can be eliminated using Prop. 4, and completeness follows since $\vdash (\mathtt{rb} \mathbin{;} \mathtt{co}^+)[(\mathtt{rb} \cup \mathtt{rb} \mathbin{;} \mathtt{co}^+)/\mathtt{rb}] \subseteq \mathtt{rb} \cup \mathtt{rb} \mathbin{;} \mathtt{co}^+$. After applying this elimination, we obtain a theory that can be shown to be equivalent to Φ_1 that consists of the disjointness assumption, the domain range restrictions, and the assumption $(\mathtt{rf} \mathbin{;} \mathtt{rb} \cup \mathtt{co})^+ \subseteq \mathtt{co}$. Then, again, $(\mathtt{rf} \mathbin{;} \mathtt{rb} \cup \mathtt{co})^+ \subseteq \mathtt{co}$ can be eliminated using Prop. 4, and completeness follows since $\vdash (\mathtt{rf} \mathbin{;} \mathtt{rb} \cup \mathtt{co})^+[(\mathtt{co} \cup (\mathtt{rf} \mathbin{;} \mathtt{rb} \cup \mathtt{co})^+)/\mathtt{co}] \subseteq \mathtt{co} \cup (\mathtt{rf} \mathbin{;} \mathtt{rb} \cup \mathtt{co})^+$. Finally the domain assumptions can be eliminated using Prop. 6 and the disjointness assumption is eliminated using Prop. 5. All in all, we obtained a sequence of substitutions S to be performed on $e_1 \subseteq e_2$, such that $\Phi_{\mathtt{base}} \vdash e_1 \subseteq e_2$ iff $\vdash e_1[S] \subseteq e_2[S]$. Decidability then follows from decidability of inclusion in KAT (without assumptions). □

3.4.2 *Release-Acquire Consistency*

In our next example, we will show equivalence between two different definitions of the release/acquire consistency model[29]. This example is, in fact, motivated by wanting to show the correctness of a program optimization, namely store-load de-ordering (e.g., $x := 1 \mathbin{;} a := y \rightsquigarrow x := 1 \parallel a := y$ under any program context). The effect of this transformation on execution graphs is to remove certain $\mathtt{po}$ edges from write events to read events. A simple way to show that a memory model allows this transformation is if its consistency condition does not depend at all on $[\mathtt{W}] \mathbin{;} \mathtt{po} \mathbin{;} [\mathtt{R}]$ edges. In other words, the model should not be affected if we substitute all instances of $\mathtt{po}$ by $\mathtt{po} \setminus [\mathtt{W}] \mathbin{;} \mathtt{po} \mathbin{;} [\mathtt{R}]$ in its definition.

The first model is the usual definition of release-acquire consistency. An execution graph is RA-consistent if $\mathtt{hb} \mathbin{;} (\mathtt{co} \cup \mathtt{rb})^?$ is irreflexive, where $\mathtt{hb}$ is the *happens-before* order, that relates two events a and b if there is a path composed of $\mathtt{po}$ and $\mathtt{rf}$ edges from a to b. In terms of relational algebra: $\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{rf})^+$.

[29] *"Taming Release-acquire Consistency"* [LGV16]

According to the second definition, an execution is release/acquire-consistent if $\mathtt{hb}_2 ; (\mathtt{co} \cup \mathtt{rb})^?$ is irreflexive and $\mathtt{rb}$ does not contradict $\mathtt{po}$ (i.e., $\mathtt{po} ; \mathtt{rb}$ is irreflexive). In this definition,

$\mathtt{hb}_2$ is a subset of happens-before which avoids using any $[\mathtt{W}];\mathtt{po};[\mathtt{R}]$ edges in its definition: $\mathtt{hb}_2 \triangleq ([\mathtt{R}] ; \mathtt{po} \cup \mathtt{po} ; [\mathtt{W}] \cup \mathtt{rfe})^+$, where $\mathtt{rfe}$ denotes all external $\mathtt{rf}$ edges (where the write and the read are not $\mathtt{po}$-related).

So, now let's try to prove equivalence between the two versions. First, we need to equip Kater with some additional built-in knowledge: (1) that $\mathtt{rf}$-edges are either internal (inside $\mathtt{po}$) or external; (2) that internal $\mathtt{rf}$-edges are included in the program order; and (3) that the program order is transitive.

$$\mathtt{rf} = \mathtt{rfi} \cup \mathtt{rfe} \qquad \mathtt{rfi} \subseteq \mathtt{po} \qquad \mathtt{po} ; \mathtt{po} \subseteq \mathtt{po} \qquad (\mathtt{po}\text{-properties})$$

Elimination of these assumptions can be done using Propositions 2, 4 and 6.

Then, we can simply formulate the following Kater query:

```
let hb = ([R ∪ W];po;[R ∪ W] ∪ rf)^+
let ra = hb;(co ∪ fr)^?
let hb_2 = ([R ∪ W];po;[W] ∪ [R];po;[W ∪ R] ∪ rfe)^+
let ra_2 = hb_2;(co ∪ fr)^? ∪ po;fr
assert ra = ra_2
```

Running Kater yields the counterexample $[\mathtt{W}] ; \mathtt{po} ; [\mathtt{R}]$. The point is that while $\mathtt{ra} = \mathtt{ra}_2$ is a sufficient condition for $\mathsf{irreflexive}(\mathtt{ra}) \Leftrightarrow \mathsf{irreflexive}(\mathtt{ra}_2)$, it is *not* a necessary one. (For example, $\vdash \mathsf{irreflexive}(r_1 ; r_2) \Leftrightarrow \mathsf{irreflexive}(r_2 ; r_1)$ but $\nvdash r_1;r_2 = r_2;r_1$.) Here, as a standard assumption, we know that the program order is always irreflexive. So, to check for equivalence between the two models, it suffices to check the following equality:

```
assert (ra ∪ po) = (ra_2 ∪ po)
```

which Kater can easily prove.

3.5 IRREFLEXIVITY IMPLICATIONS

There is, in fact, another way to prove the equivalence between the two release-acquire models without assuming that $\mathtt{po}$ is irreflexive. Under our assumption that writes and reads are disjoint, there can never be a cycle of form $[\mathtt{W}] ; \ldots ; [\mathtt{R}]$.

In reality, what we want to show is that $\mathtt{ra} \cap \mathtt{id} = \mathtt{ra}_2 \cap \mathtt{id}$ but this falls outside of the known decidable fragments. (Although regular languages are closed under intersection, they do not support a concept like the identity relation. We cannot simply treat $\mathtt{id}$ as an uninterpreted symbol because we need it to denote the identity relation.) We can, however, express a somewhat weaker constraint in KAT, which Kater can easily prove.

```
assert sameEnds(ra) = sameEnds(ra2)
```

where $\mathsf{sameEnds}(e)$ restricts e to enforce that its endpoints are compatible. In the fragment we have seen so far, that would be that $\mathsf{sameEnds}(e)$ returns $[\mathtt{R}] ; e ; [\mathtt{R}] \cup [\mathtt{W}] ; e ; [\mathtt{W}] \cup [\mathtt{F}] ; e ; [\mathtt{F}]$. Soundness easily follows from the following proposition.

Proposition 8 (Same-Ends Closure)**.** For every KAT expression e, $\vdash \mathsf{irreflexive}(e) \Leftrightarrow \mathsf{irreflexive}(\mathsf{sameEnds}(e))$.

Consider now a third version of release-acquire consistency defined as $\mathsf{irreflexive}(\mathtt{ra}_3)$ where $\mathtt{ra}_3 \triangleq (\mathtt{co} \cup \mathtt{rb})^? ; \mathtt{hb}$, which we would like to show equivalent to the first version. If we just ask Kater to show $\mathtt{ra} = \mathtt{ra}_3$, we will get counterexamples such as $\mathtt{po} ; \mathtt{co}$ and $\mathtt{co} ; \mathtt{po}$. The issue is that $\mathtt{ra}_3$ is not equal to $\mathtt{ra}$, but to a *rotation* of it.

Therefore, to prove the equivalence between the two models, we employ the *rotational closure* operator $\mathsf{ROT}(L) \triangleq \left\{uv \,\middle|\, vu \in L\right\}$. Recall from §3.1 that regular languages are closed under rotational closure. By extension, KAT expressions are closed under rotational closure as well (so we can freely use $\mathsf{ROT}(e)$ for a KAT expression e). We now show that employing rotational closure is sound for proving implications between irreflexivity constraints.

Proposition 9 (Rotational Closure)**.** For every KAT expression e, $\vdash \mathsf{irreflexive}(e) \Leftrightarrow \mathsf{irreflexive}(\mathsf{ROT}(e))$.

Proof. For the right-to-left direction, it suffices to note that $[\![e]\!]_G \subseteq [\![\mathsf{ROT}(e)]\!]_G$. For the converse, consider a loop in $[\![\mathsf{ROT}(e)]\!]_G$, i.e., there exists a such that $\langle a, a\rangle \in [\![\mathsf{ROT}(e)]\!]_G$. From the definition of $\mathsf{ROT}(.)$, we get $\langle a, a\rangle \in [\![uv]\!]_G$ for some $vu \in \mathsf{L}(e)$. The definition of $[\![.]\!]_G$ ensures that there exists b such that $\langle a, b\rangle \in [\![u]\!]_G$ and $\langle b, a\rangle \in [\![v]\!]_G$, from which we can obtain that $\langle b, b\rangle \in [\![e]\!]_G$, which means that $[\![e]\!]_G$ is not irreflexive. □

Putting the two together, to prove an implication $\mathsf{irreflexive}(e_1) \Rightarrow \mathsf{irreflexive}(e_2)$, we ask Kater to prove $\mathsf{sameEnds}(e_2) \subseteq \mathsf{ROT}(e_1)$.

While this method is sound, it is not complete. Indeed, we have $\mathsf{irreflexive}(r ; r) \Rightarrow \mathsf{irreflexive}(r)$, but $\mathsf{sameEnds}(r) \not\subseteq \mathsf{ROT}(r ; r)$. To recover completeness, we need the *deduplication closure* operator $\mathsf{DEDUP}(L) \triangleq \{w \mid \exists n.\, w^n \in L\}$. Recall from §3.1 that regular languages are closed under deduplication closure, and by extension, so are KAT expressions.

Proposition 10. For every KAT expression e, $\vdash \mathsf{irreflexive}(e) \Leftrightarrow \mathsf{irreflexive}(\mathsf{DEDUP}(e))$.

Proof. For the right-to-left direction, it is sufficient to note that $[\![e]\!]_G \subseteq [\![\mathsf{DEDUP}(e)]\!]_G$. For the converse, consider a loop in $[\![\mathsf{DEDUP}(e)]\!]_G$, i.e., there exists a such that $\langle a, a\rangle \in [\![\mathsf{DEDUP}(e)]\!]_G$. From the definition of $\mathsf{DEDUP}(.)$, there is some n and w such that $\langle a, a\rangle \in [\![w]\!]_G$ and $w^n \in \mathsf{L}(e)$. Since $\langle a, a\rangle \in [\![w]\!]_G$, we also have $\langle a, a\rangle \in [\![w^n]\!]_G$, and so $\langle a, a\rangle \in [\![e]\!]_G$, which means that $[\![e]\!]_G$ is not irreflexive. □

With same-ends, rotation, and deduplication together, we can rephrase irreflexivity entailment queries as inclusion queries in a sound and complete way:

Proposition 11 (Soundness and Completeness)**.** For every two KAT expressions e_1 and e_2, $\vdash \mathsf{sameEnds}(e_1) \subseteq \mathsf{DEDUP}(\mathsf{ROT}(e_2))$ if and only if $\vdash \mathsf{irreflexive}(e_2) \Rightarrow \mathsf{irreflexive}(e_1)$.

Proof. For the left-to-right direction, suppose that $\vdash \mathsf{sameEnds}(e_1) \subseteq \mathsf{DEDUP}(\mathsf{ROT}(e_2))$. It follows that $\vdash \mathsf{irreflexive}(\mathsf{DEDUP}(\mathsf{ROT}(e_2))) \Rightarrow \mathsf{irreflexive}(\mathsf{sameEnds}(e_1))$. By Propositions 9 and 10, we know that $\vdash \mathsf{irreflexive}(e_2) \Leftrightarrow \mathsf{irreflexive}(\mathsf{DEDUP}(\mathsf{ROT}(e_2)))$. By Prop. 8, we have $\vdash \mathsf{irreflexive}(e_1) \Leftrightarrow \mathsf{sameEnds}(e_1)$. Hence, it follows that $\vdash \mathsf{irreflexive}(e_2) \subseteq \mathsf{irreflexive}(e_1)$.

For the converse, suppose that $\vdash \mathsf{irreflexive}(e_2) \Rightarrow \mathsf{irreflexive}(e_1)$. We show that $\mathsf{L}(\mathsf{sameEnds}(e_1)) \subseteq \mathsf{L}(\mathsf{DEDUP}(\mathsf{ROT}(e_2)))$. Let us assume that $t_1 r_1 t_2 r_2 \ldots t_n r_n t_{n+1} \in \mathsf{L}(\mathsf{sameEnds}(e_1))$. Let G be an execution graph with: 1. n events, $a_1, \ldots, a_n$, such that a_i satisfies t_i for every $1 \le i \le n$ and a_n satisfies t_1 (this is possible due to $\mathsf{sameEnds}(.)$ closure); 2. the relations of G are constructed such that $\langle a_i, a_{i+1}\rangle \in r_i$ for every $1 \le i \le n-1$ and $\langle a_n, a_1\rangle \in r_n$. This construction ensures that $\langle a_1, a_1\rangle \in [\![e_1]\!]_G$. Then, the assumption that $\vdash \mathsf{irreflexive}(e_2) \Rightarrow \mathsf{irreflexive}(e_1)$ entails that $\langle a_i, a_i\rangle \in [\![e_2]\!]_G$ for some $1 \le i \le n$. By the construction of G, there exists $m \ge 0$ such that $t_i r_i \ldots r_n t_{n+1} (t_1 r_1 \ldots r_n t_{n+1})^m t_1 r_1 \ldots t_{i-1} r_{i-1} \in \mathsf{L}(e_2)$. Hence, $(t_1 r_1 \ldots r_n t_{n+1})^{m+1} \in \mathsf{L}(\mathsf{ROT}(e_2))$, which means that $t_1 r_1 \ldots r_n t_{n+1} \in \mathsf{L}(\mathsf{DEDUP}(\mathsf{ROT}(e_2)))$. □

This leads to a decision procedure for queries of the form $\Phi_{\mathsf{base}} \vdash \mathsf{irreflexive}(e_1) \Rightarrow \mathsf{irreflexive}(e_2)$ (Φ_{base} can be extended with the additional po-properties assumptions mentioned in § 3.4.2). Indeed, one can apply the elimination of assumptions as in the proof of Prop. 7, and finally apply Prop. 11.

3.6 proving memory-model equivalence

Although Prop. 11 provides a sound and complete way to check equivalence between models without emptiness constraints, certain equivalences only hold under additional assumptions. In what follows, we demonstrate how Kater can deal with such assumptions in a heuristic fashion.

3.6.1 *Coherence*

Memory models often contain the following axiom, which is known as "Coherence" or "SC-per-location".

$$\texttt{poloc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb} \text{ is acyclic, where } \texttt{poloc} \triangleq \texttt{po} \cap \texttt{sameloc}$$

We would like to show that this axiom is equivalent to $\texttt{po} \mathbin{;} \texttt{eco}$ being irreflexive.

A first obvious problem is that KATER cannot support the term "$\texttt{po} \cap \texttt{sameloc}$" for the same reason it could not support the term "$\texttt{ra} \cap \texttt{id}$". We can work around this problem by making KATER treat $\texttt{poloc}$ as an uninterpreted relation, and adding two basic assumptions about $\texttt{poloc}$: that it is transitive and that it is included in $\texttt{po}$.

Simply doing so, however, is not sufficient. KATER will return us a counterexample: $\texttt{po};\texttt{rf}$ is included in $\texttt{po};\texttt{eco}$ but not in any rotation of $(\texttt{poloc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb})^+$. The problem lies in the initial $\texttt{po}$ edge. KATER should not really be considering arbitrary paths of $\texttt{po} \mathbin{;} \texttt{eco}$, but only ones that start and end with the *same* event. Following this principle, we have so far ruled out paths starting with a read event and ending with a write event. Now, we additionally want to rule out paths that start and end with events of different locations. Specifically, we can extend KATER's built-in knowledge with the $\texttt{sameloc}$ relation and its basic properties:

$$\texttt{rf} \cup \texttt{co} \cup \texttt{rb} \cup \texttt{id} \cup (\texttt{sameloc} \mathbin{;} \texttt{sameloc}) \subseteq \texttt{sameloc}$$

Thus, as part of $\mathsf{sameEnds}(e)$, we will intersect e with $\texttt{sameloc}$ and try to distribute the intersection to the primitive relations with rules such as $(r_1 \mathbin{;} r_2) \cap \texttt{sameloc} = (r_1 \cap \texttt{sameloc}) \mathbin{;} r_2$ provided $r_2 \subseteq \texttt{sameloc}$. While this procedure is generally incomplete (it will not always succeed in pushing the $\_ \cap \texttt{sameloc}$ to primitive relations), when applied to $\texttt{po} \mathbin{;} \texttt{eco}$, it will yield the term $\texttt{poloc} \mathbin{;} \texttt{eco}$, and so will rule out the counterexample.

Still, however, this is not enough. KATER will now return us another counterexample: $[\texttt{W}];\texttt{poloc};[\texttt{W}];\texttt{co};[\texttt{W}];\texttt{poloc};[\texttt{W}];\texttt{co}$, which is clearly not included in any rotation of $\texttt{poloc} \mathbin{;} \texttt{eco}$.

The problem is that KATER does not (yet) know that $\texttt{co}$ is total over all writes to the same location. From totality and $\texttt{poloc};\texttt{co}$ irreflexivity, it follows that $[\texttt{W}];\texttt{poloc};[\texttt{W}] \subseteq \texttt{co}^?$. Adding this inclusion as an assumption, lets KATER proceed further and generate another counterexample, which can be resolved by adding the assumption $[\texttt{W}];\texttt{poloc};\texttt{rb};[\texttt{W}] \subseteq \texttt{co}^?$. This assumption, however, is still not enough. With a few more iterations, we can arrive at the constraint: $[\texttt{W}];\texttt{rf}^?;\texttt{poloc};\texttt{rb}^?;[\texttt{W}] \subseteq \texttt{co}^?$, which lets KATER complete the proof.

The question is how can we arrive at such constraints without the manual trial-and-error loop. The solution is again by a saturation procedure on the right-hand-side of an inclusion query.

$$\mathsf{TOT}(r, L) \triangleq L \cup (r \cup \{w \mid rw \in L, w \neq \epsilon\})^+$$

Proposition 12. Let Φ be a set of KAT formulas, e_1 and e_2 be KAT expressions, and r be a primitive relation. If $\Phi \vdash e_1 \subseteq \mathsf{TOT}(r, e_2)$, then Φ, r is a strict total order, $\mathsf{irreflexive}(e_2) \vdash \mathsf{irreflexive}(e_1)$.

Proof. Let $e' \triangleq \{w \mid rw \in \mathsf{L}(e_2), w \neq \epsilon\}$. By means of contradiction, consider an execution graph $G \in [\![\Phi, r$ is a strict total order, $\mathsf{irreflexive}(e_2)]\!]$ and a loop in $[\![e_1]\!]_G$, i.e., a such that $\langle a, a \rangle \in [\![e_1]\!]_G$. From our assumptions, we have $\langle a, a \rangle \in [\![\mathsf{TOT}(r, e_2)]\!]_G$. From the definition of $\mathsf{TOT}(.)$, we either get a loop in $[\![e_2]\!]_G$, which contradicts our hypothesis, or a cyclic path $\langle a, a \rangle \in [\![r \cup e']\!]^+_G$. Let $n \geq 1$ and $a_1, \ldots, a_n$ such that $\langle a_i, a_{i+1} \rangle \in [\![r \cup e']\!]_G$ for every $1 \leq i \leq n$ (to simplify the notation here we work modulo n, so $n + 1 = 1$). We claim that for every $1 \leq i \leq n$, we must have $\langle a_i, a_{i+1} \rangle \in [\![r]\!]_G$. From this claim we obtain $\langle a_1, a_1 \rangle \in [\![r^+]\!]_G$, which contradicts our hypothesis that $[\![r]\!]_G$ is a strict order. To prove this claim, let $1 \leq i \leq n$. The totality of $[\![r]\!]_G$ ensures that either $\langle a_i, a_{i+1} \rangle \in [\![r]\!]_G$ or $\langle a_{i+1}, a_i \rangle \in [\![r]\!]_G$. By means of contradiction, suppose that $\langle a_{i+1}, a_i \rangle \in [\![r]\!]_G$. Since $\langle a_i, a_{i+1} \rangle \in [\![r \cup e']\!]_G$ and r is a strict order, we must have $\langle a_i, a_{i+1} \rangle \in [\![e']\!]_G$, and so we have $\langle a_i, a_{i+1} \rangle \in [\![w]\!]_G$ for some w such that $rw \in \mathsf{L}(e_2)$. Then, we obtain $\langle a_{i+1}, a_{i+1} \rangle \in [\![r\,;w]\!]_G \subseteq [\![e_2]\!]_G$, which is a loop in $[\![e_2]\!]_G$, and contradicts our hypothesis. □

3.6.2 *Total Store Ordering (TSO)*

Our next example concerns the SPARC/x86 TSO memory model[30]. As with the previous example, the goal is to prove equivalence between two different definitions of TSO.

The first model is the standard one. It defines the *preserved program order*, $\mathtt{ppo} \triangleq [\mathtt{R} \cup \mathtt{F}]\,;\mathtt{po} \cup \mathtt{po}\,;[\mathtt{W} \cup \mathtt{F}]$, to include all program order edges except for edges from writes to reads, and requires that $\mathtt{tso} \triangleq \mathtt{ppo} \cup \mathtt{rfe} \cup \mathtt{co} \cup \mathtt{rb}$ be acyclic and the coherence property hold.

The second model, due to Lahav and Vafeiadis[31], requires that $\mathtt{hb};\mathtt{rb}^?$ be irreflexive and that there be a strict total order *mo* over *all* write and fence events such that $mo;\mathtt{tso}_2$ is irreflexive where

$$\mathtt{hb} \triangleq (\mathtt{po}{\cup}\mathtt{rf})^+ \quad \text{and} \quad \mathtt{tso}_2 \triangleq (\mathtt{co}{\cup}[\mathtt{F}{\cup}\mathtt{W}];\mathtt{hb};[\mathtt{F}{\cup}\mathtt{W}]{\cup}([\mathtt{F}]{\cup}\mathtt{rfe});\mathtt{po};\mathtt{rb})\,.$$

First, note that irreflexivity of $\mathtt{tso}_2$ holds from irreflexivity of $\mathtt{hb}\,;\mathtt{rb}^?$ and $\mathtt{co}$.

```
assert sameEnds(tso_2) ⊆ hb;fr^? ∪ co
```

[30] The SPARC architecture manual (version 9) [SPA94]; *"A better x86 memory model: x86-TSO"* [OSS09]

[31] *"Explaining Relaxed Memory Models with Program Transformations"* [LV16]

Then, we can prove the following lemma:

Lemma 1. A relation R is acyclic if and only if R is irreflexive and there exists a strict total order T on $dom(R) \cup rng(R)$ such that $T ; R$ is irreflexive.

Proof. In the forward direction, take T to be any total order extending R^+. In the backward direction, by means of contradiction, consider a cycle in R. Since R is irreflexive, the cycle will contain at least two distinct nodes. Because T is total, all pairs of adjacent nodes will be ordered by $T \cup T^{-1} \cup \mathtt{id}$. However, it cannot be the case that all pairs of adjacent nodes will be ordered by $T \cup \mathtt{id}$, or else we would get a cycle in T. So, there has to be a pair of R-adjacent nodes ordered by T^{-1}, contradicting the assumption that $T ; R$ is irreflexive. □

Finally, by applying Lemma 1, the two models are equivalent provided that $\mathtt{tso}^+ \cup \mathtt{po};\mathtt{eco}$ is irreflexive iff $\mathtt{hb} ; \mathtt{rb}^? \cup \mathtt{tso}_2^+$ is irreflexive, which Kater proves with the following queries:

```
assert sameEnds(tso_2^+ ∪ hb;fr^?) ⊆ rot (tso^+)
assert sameEnds(tso^+) ⊆ rot (tso_2^+ ∪ hb;fr^?)
```

3.7 C11 COMPILATION RESULTS

Let us now see how Kater can establish some more substantial results about the revised C11 memory model ofLahav et al.[32] without their $(\mathtt{po} \cup \mathtt{rf})$-acyclicity constraint.

First, correctness of local transformations can be achieved in a similar way as in § 3.4.2 concerning the release/acquire memory model. The idea is to prove equivalence with respect to a variant of the C11 definition obtained by replacing all instances of po with the following subset of po

$$\mathtt{ppo}_{C11} \triangleq [\mathtt{ACQ}] ; \mathtt{po} \cup \mathtt{po} ; [\mathtt{REL}] \cup [\mathtt{SC}] ; \mathtt{po} ; [\mathtt{SC}]$$

that is guaranteed to be preserved by the transformations, and adding the usual coherence axiom asserting acyclicity of $(\mathtt{poloc} \cup \mathtt{eco})$. Kater easily proves the equivalence.

Next, we examine the correctness of C11's default compilation mappings to the various hardware architecture models.

C11 TO ARMV8 We start with compilation to the ARMv8 model. Although the ARMv model is more complicated than some other hardware memory models, compilation from C11 to ARMv8 is actually easier to establish than to some other memory models because the compilation mapping is the identity. That is, every primitive C11 access or fence maps to exactly one access or fence at the architecture level.

Therefore, to prove compilation correctness, we have to show that C11 is weaker than ARMv8. C11 consistency checks three properties:

[32] "Repairing sequential consistency in C/C++11" [Lah+17]

- Coherence with respect to happens-before (i.e., $\mathsf{irreflexive}(\mathtt{hb} \,;\, \mathtt{eco})$);
- RMW-atomicity; and
- $\mathtt{psc}$ acyclicity.

ARMv8 consistency has three other properties:

- Coherence with respect to the program order (i.e., acyclicity of $\mathtt{poloc} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{rb}$, which, as we have seen, is equivalent to $\mathsf{irreflexive}(\mathtt{po} \,;\, \mathtt{eco})$);
- RMW atomicity; and
- Acyclicity of its $\mathtt{ob}$ relation.

Therefore, to prove correctness of compilation, it suffices to call Kater with the following input:

```
include "C11.kat"
include "Arm8.kat"
assert C11::hb;eco ⊆ po;eco ∪ Arm8::ob+
assert C11::psc ⊆ Arm8::ob+
```

This code snippet demonstrates two small features of Kater: (1) it allows one to include files containing additional definitions, and (2) it provides a simple name resolution mechanism to refer to definitions from other files. Kater easily proves these assertions.

C11 to x86-TSO Our next compilation result concerns the mapping from C11 to the x86 model. There are actually two mappings of interest: one which inserts TSO fences right after SC-atomic stores, and one which inserts TSO fences right before SC-atomic loads, In both cases, all remaining accesses are mapped to plain TSO accesses, C11's SC fences are mapped to TSO fences and all remaining fences to NOPs.

Our general approach for handling such mappings is to define the architecture model in terms of the C11 access modes (e.g., only treat $\mathtt{F} \cap \mathtt{sc}$ as a TSO fence) and add additional assertions about the presence of additional fences induced by the mapping. In particular, we let Kater prove the following:

```
assume [W;SC];po;[R;SC] ⊆ po;[F;SC];po
assert C11::hb;eco ⊆ po;eco ∪ TSO::tso+
assert C11::psc+ ⊆ TSO::tso+
```

The assumption states that the mapping always introduces an SC fence between an SC write and a subsequent SC read from the same thread, and allows Kater to complete the proof, establishing the correctness of both mappings at once.

Kater uses such assumptions in a heuristic fashion whenever it is asked to prove an inclusion assertion. Given an assumption $A \subseteq B$, it searches for pairs of states $\langle x, y \rangle$ in the NFA representing the right-hand-side of the inclusion such that there is a B path from x to y. Whenever this is the case, it adds an A path from x to y, which may introduce further states if A is a composite expression.

C11 TO POWER Next, we consider the compilation to POWER, which is substantially more complex than the compilations to TSO and ARMv8, and has led to incorrect claims about the compilation of the original C11 model to it. Here, we will follow the axiomatic POWER model ofAlglave, Maranget, and Tautschnig[33], which consists of the following axioms:

- Coherence: (`poloc` ∪ `rf` ∪ `co` ∪ `rb`) is acyclic.
- No-thin-air: A certain `hb` relation containing preserved program order edges (due to dependencies or fences) and `rfe` edges is acyclic.
- Propagation: `co` ∪ `prop` is acyclic, where `prop` is POWER's propagation order.
- Observation: `obs` $\triangleq$ `rb` ; `prop` ; `hb`* is irreflexive.

There exist multiple correct compilation mapping schemes from RC11 to POWER. For concreteness, we will present the "leading-sync with lwsyncs" scheme. This scheme maps C11's SC fences to POWER's global synchronization fence (`sync`), C11's other fences to POWER's lightweight synchronization fence (`lwsync`), introduces an `lwsync` fence before every release write and after every acquire read, and a `sync` fence before every SC access (read or write). We therefore model POWER's `sync` as C11's SC-fence, `lwsync` as any C11's fence, and formulate the following assumptions about the presence of additional fences.

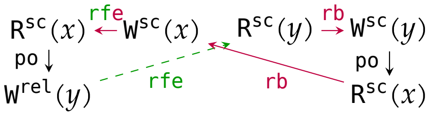
```
assume [R;ACQ];po ∪ po;[W;REL] ⊆ po;[F];po
assume po;[R;SC]  ∪ po;[W;SC]  ⊆ po;[F;SC];po
```

To establish the correctness of C11's coherence axiom, we ask Kater the following:

```
let r = eco;po?;eco* ∪ Pow::hb+ ∪ (co ∪ prop)+ ∪ obs
assert sameEnds(eco;C11::hb) ⊆ r
```

which it proves in less than a minute. Note that to assist Kater's inclusion check, we have incorporated a small 'optimization' in this query. We have used a reformulation of the coherence axiom that is equivalent to POWER's acyclic(`poloc` ∪ `rf` ∪ `co` ∪ `rb`) axiom, as already shown in §3.4.1, and already incorporates a rotation.

[33] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [AMT14]


$\mathtt{R}^{\mathtt{sc}}(x)$
rfe
$\mathtt{W}^{\mathtt{sc}}(x)$
$\mathtt{R}^{\mathtt{sc}}(y)$
rb
$\mathtt{W}^{\mathtt{sc}}(y)$
po
po
$\mathtt{W}^{\mathtt{rel}}(y)$
rfe
rb
$\mathtt{R}^{\mathtt{sc}}(x)$

$$\begin{array}{l|l|l|l} a := x^{\mathtt{sc}} & & & y^{\mathtt{sc}} := 2 \\ & x^{\mathtt{sc}} := 1 & b := y^{\mathtt{sc}} & \\ y^{\mathtt{rel}} := 1 & & & c := x^{\mathtt{sc}} \end{array}$$

Outcome: $x = a = b = 1 \wedge y = 2 \wedge c = 0$

Figure 3.1: A counterexample produced by KATER

Next, to show that C11's `psc` relation is acyclic, ideally one would ask KATER the following query.

```
assert sameEnds(C₁₁::psc⁺) ⊆ rot(r)
```

While KATER can in principle prove this inclusion, in practice it takes forever for KATER to return. The issue is that applying the rotational closure and the implication assumptions to `r` generates a huge automaton, and so the various simplification passes and the inclusion checking takes too long.

By performing these transformations manually to the POWER model, KATER is able to establish the inclusion (without the rotation). Specifically, to avoid the explicit assumptions, we adapt the definition of the POWER relations to include two additional disjuncts, which are shown in comments below.

```
let sync = po;[F;SC];po //∪ po;[R;SC] ∪ po;[W;SC]
let lwsync = po;[F];po //∪ [R;ACQ];po ∪ po;[W;REL]
```

Let us now consider a simplified version of the original C11 model, whose compilation to POWER turned out to be incorrect. The goal is to (dis)prove the following inclusion:

```
assert sameEnds(([SC];(hb ∪ co ∪ fr);[SC])⁺) ⊆ rot(r)
```

Again, rewriting the POWER model to avoid the scalability issues, we get the counterexample:

$$[\mathtt{R}^{\mathtt{sc}}];\mathtt{po};[\mathtt{W}^{\mathtt{rel}}];\mathtt{rfe};[\mathtt{R}^{\mathtt{sc}}];\mathtt{rb};[\mathtt{W}^{\mathtt{sc}}];\mathtt{po};[\mathtt{R}^{\mathtt{sc}}];\mathtt{rb};[\mathtt{W}^{\mathtt{sc}}];\mathtt{rfe}$$

which is allowed by POWER but not by C11. We depict it also as an execution graph and a litmus test in Fig. 3.1.

3.8 OTHER METATHEORETIC PROPERTIES

In addition to comparing memory models, as we discussed so far, we can use KAT queries to check for prefix-closedness, extensibility, and monotonicity. Key to establishing these properties is the observation that all primitive relations are used only positively in KAT expressions, while KAT expressions are used negatively in the model definitions (since $x = \emptyset \Leftrightarrow x \subseteq \emptyset$).

- *Prefix-closedness* (§5.2) holds by construction for every expressible memory model: removing edges from an execution graph cannot create any additional paths that cannot exist or cannot be cyclic.

- *Extensibility* (§5.4) holds as long as the model is defined purely in terms of the built-in relations (`po`, `rf`, `rfe`, `rfi`, `co`, and `rb`) and does not contain emptiness checks. Adding an event e maximally to a consistent execution graph does not create any outgoing edges from e, and so it cannot create any new cycles.

- For *monotonicity* with respect to the merging of the threads, it suffices for the memory model to be defined purely in terms of the relations like `po`, `rf`, `co`, and `rb` and not in terms of relations like `rfi` and `rfe` (internal and external reads-from, respectively), which distinguish between events originating from the same thread or not. This holds, for example, for the SC and RC11 models, but not for TSO and Arm8.

- For *monotonicity* with respect to access mode strengthenings, e.g., from acquire to SC, it suffices for the "acquire" predicate of the memory model to also include SC accesses ($[\![\texttt{SC}]\!]_G \subseteq [\![\texttt{ACQ}]\!]_G$ for all G), and to never use predicates in a negative context, i.e., never take the complement of a predicate or the set difference between two predicates.

Moreover, given a way to prove that a model is weaker than another, we can leverage it to answer the remaining two meta-theoretical questions from the introduction.

- For *local program transformations*, it suffices to prove equivalence with a model where the correctness of the transformation is evident (see example in §3.4.2).

- For the absence of *out-of-thin-air behaviors* (§5.1), it suffices to show that the model is stronger than $\langle \emptyset, (\texttt{deps} \cup \texttt{rf})^+ \rangle$, where `deps` represents the set of program-induced dependencies between `po`-ordered events, i.e., the union of the address, data, and control dependencies (see §2.4.1).

4 CHECKING EXECUTION GRAPH CONSISTENCY

In the previous chapter we saw how KATER can prove implications between memory-model consistency predicates. In this chapter, we discuss how such predicates can be calculated in a given execution graph.

But how do we check consistency of a graph given an arbitrary memory model M to begin with? Since M is expressed as emptiness and irreflexivity constraints over some relations, a simple solution is to calculate all relations of M in a fixpoint, and then check for emptiness/irreflexivity.

This solution, however, is computationally expensive. Consider a coherence-tracking graph G and suppose we want to check its consistency under SC. Following the outlined approach, we have to calculate the transitive closure of porf ∪ mo ∪ rb in a fixpoint, which, assuming that po, rf, mo and rb are represented as $n \times n$ matrices (where n is the number of graph nodes), is at least as hard as matrix multiplication[34]. Adding insult to the injury, if G does not track coherence, then we have to enumerate all possible co choices, leading to exponential complexity.

Fortunately, we can deal with this complexity issue by making two key observations. First, if we are interested in acyclicity constraints, then we simply have to find cycles in the execution graph that correspond to violations according to M. Finding such cycles can be done using a depth-first-search (DFS), which yields an $O(n)$ complexity (assuming a sparse graph). In fact, as we show in §4.2, the generation of such consistency checking code can be automated by using KATER: treating the graph as another automaton, finding violating cycles boils down to finding words that the intersection of M and the graph accepts.

Second, as far as checking consistency in non-coherence-tracking graphs is concerned, even though finding an appropriate co might have an exponential complexity under certain models[35], in many cases we can adequately *approximate* it. The key idea is to only consider co edges that are forced in a particular way by the memory model's irreflexivity constraints. We present such an approximation that only partially orders stores at each memory location (instead of totally ordering them).

In what follows, we first provide some intuition on how the naive solution can be improved assuming SC (§4.1). Then, we show how KATER can synthesize code that checks consistency of a given graph under arbitrary models (§4.2). We end the chapter by presenting how co can be approximated, and the repercussions this approximation has in consistency checking (§4.3).

[34] *For SC, one could of course use Floyd–Warshall or DFS to calculate the transitive closure, but the fixpoint construction is more general.*

[35] *"Testing shared memories"* [GK97]

4.1 OPTIMIZED CONSISTENCY CHECKS FOR SC

Let us begin by showing how consistency checking under SC can be optimized. Recall that an execution graph is SC-consistent if $\mathtt{sc} \triangleq \mathtt{porf} \cup \mathtt{co} \cup \mathtt{rb}$ is acyclic.

To check whether a graph is cyclic, one can simply perform a plain depth-first search through the graph, recording at each node whether it has been visited and whether the recursive visits of its children have been completed[36]. The depth-first search has complexity $O(n + m)$ where n is the number of graph nodes and m the number of sc edges. Since a graph with n nodes can have $O(n^2)$ sc edges, the overall complexity is quadratic.

We can, however, do even better and bring down the DFS complexity to $O(n)$ by making the graph sparse. The idea is to observe that:

$$\mathsf{acyclic}(\mathtt{sc}) \Leftrightarrow \mathsf{acyclic}(\mathtt{po}|_{\mathrm{imm}} \cup \mathtt{rf} \cup \mathtt{co}|_{\mathrm{imm}} \cup \mathtt{rf}^{-1};\mathtt{co}|_{\mathrm{imm}})$$

and to use the immediate counterparts of po, co, and rb to make the graph sparse, thus bringing down the DFS complexity to $O(n)$.

4.2 ARBITRARY ACYCLICITY CHECKS WITH KATER

The fact that SC admits such fast consistency checks begs the question of whether such efficient consistency checks can be generalized for an arbitrary memory model. At a first glance, this does not seem obvious. Even though SC is expressed as a single transitive closure of some primitive relations, this is not the case for memory models in general: a model may require the acyclicity of relations defined in terms of other (potentially complex) relations, and thus merely performing a depth-first search is insufficient.

Fortunately, we can utilize KATER and automata intersection to produce efficient consistency checking routines for any model expressed using acyclicity constraints, in the following way.

First, given a model M, observe that we can build an automaton $\textsc{Nfa}_{\mathsf{M}}$ corresponding to the acyclicity constraints of M[37]. In addition, we can enforce that $\textsc{Nfa}_{\mathsf{M}}$ has a single initial/final state merely by taking its reflexive-transitive closure: as long as we only consider non-empty words, the two automata accept the same set of words. By expressing M's acyclicity constraints as an NFA, we also get to express its acyclicity constraints in terms of primitive relations (i.e., $\mathtt{po}|_{\mathrm{imm}}$, $\mathtt{co}|_{\mathrm{imm}}$, $\mathtt{rf}|_{\mathrm{imm}}$ and $\mathtt{rb}|_{\mathrm{imm}}$), which are already calculated in an execution graph.

Similarly, observe that a (transitively reduced) execution graph G can also be considered as an automaton $\textsc{Nfa}_G$ with states corresponding to the graph nodes and transitions corresponding to the graph edges. As a $\textsc{Nfa}_G$ does not have any obvious initial states, a given graph G actually corresponds to N automata $\textsc{Nfa}_{G_1}, \ldots, \textsc{Nfa}_{G_N}$ (where

[36] Introduction to Algorithms, 3rd Edition [Cor+09]

[37] *Recall from §3 that this is a single NFA corresponding to the union of the constraints.*

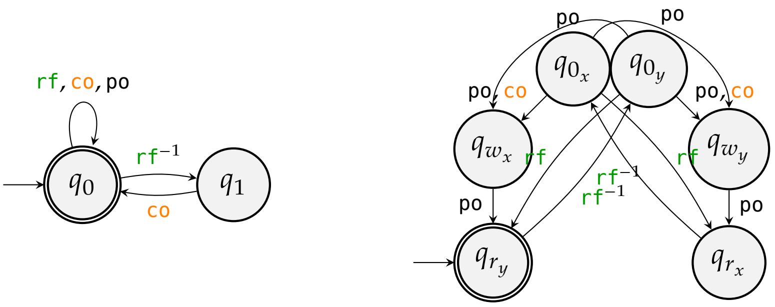


Figure 4.1: Consistency checks with Kater [38]

N is the number of G's nodes), each of which has a distinct node as an initial/final state. By expressing G as a collection of automata, we get the benefit of also expressing all possible graph cycles via these automata (as the set of non-empty words accepted by the NFAs), which are also expressed in terms of primitive relations.

Now, we can check for acyclicity violations simply by taking the intersection of the NFA$_{\mathsf{M}}$ with each of the graph automata. Concretely, utilizing the product construction between NFA$_{\mathsf{M}}$ and NFA$_{G_i}$ (for $1 \leq i \leq N$), and "running" the two automata in parallel, we can detect whether any (non-empty) cycle in G violates M's acyclicity constraint. These intersections yield all the graph cycles that satisfy M's acyclicity constraints.

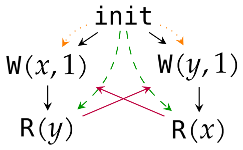


Figure 4.2: An inconsistent execution under SC

For illustration purposes, consider the SC model again where $\mathtt{rb}$ is decomposed into its constituent parts $\mathtt{rb} \triangleq \mathtt{rf}^{-1}; \mathtt{co}|_{\mathrm{imm}}{}^{+}$ so that NFA$_{SC}$ is not completely trivial. An example of how this procedure rules out the inconsistent behavior of Fig. 4.2 can be seen in Fig. 4.1. There are two things to notice in Fig. 4.1. First, NFA$_G$ has two states corresponding to the graph's initial node. That is because the initial node corresponds to the initializing writes to all memory locations and therefore needs to be decomposed. (Besides, it would be wrong to add a $\mathtt{co}$ transition from the same state to both states q_{w_x} and q_{w_y}.) Second, we have picked q_{r_y} as the initial state for NFA$_G$, although Kater will examine all graph states as initial.

Let us now see how we detect the violation of Fig. 4.2. Starting from NFA$_G$'s initial state q_{r_y}, perform a depth-first search on NFA$_G$ while at the same time maintaining NFA$_{SC}$'s state. Whenever a cycle on NFA$_G$ is detected, check whether NFA$_{\mathsf{M}}$ is in a final state. If so, a violation is detected; otherwise, the exploration proceeds normally. In the case of Fig. 4.1, we explore the following pairs of states before detecting a violation (we use overline notation to denote the product's final state):

$$\langle \overline{q_0, q_{r_y}} \rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_y} \rangle \xrightarrow{\mathtt{co}} \langle q_0, q_{w_y} \rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{r_x} \rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_x} \rangle \xrightarrow{\mathtt{co}} \langle q_0, q_{w_x} \rangle \xrightarrow{\mathtt{po}} \langle \overline{q_0, q_{r_y}} \rangle$$

[38] Non-immediate relations are drawn to not clutter the presentation

On the other hand, when we take q_{0_y} as the initial state of $\textsc{Nfa}_G$, the violation is not detected. One cycle starting and ending at q_{0_y} is the following.

$$\langle \overline{q_0, q_{0_y}} \rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{w_y} \rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{r_x} \rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_x} \rangle \xrightarrow{\mathtt{co}} \langle q_0, q_{w_x} \rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{r_y} \rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_y} \rangle$$

In this case, even though a cycle from/to q_{0_y} was detected in $\textsc{Nfa}_G$, no violation is reported as $\textsc{Nfa}_{SC}$ is not in a final state at that point. In fact, this is the case for all cycles starting and ending at q_{0_y}: since they will have to end with an $\mathtt{rf}^{-1}$ edge (the only incoming edge to q_{0_y}), $\textsc{Nfa}_{SC}$ will be in a non-final state. This also explains why one has to try *all* states of $\textsc{Nfa}_G$ as its initial states: we have to find a proper starting point in a cycle of G so that it also becomes a word accepted by $\textsc{Nfa}_{SC}$.

4.2.1 *Checking Consistency in Linear Time*

The consistency checking procedure above works reasonably well, but has a complexity of $O(n^2m)$ (where n, m are the number of states of $\textsc{Nfa}_G$ and $\textsc{Nfa}_{\mathsf{M}}$, respectively), since each DFS on a given intersection requires $O(nm)$ time. This begs the question whether we can come up with a more efficient procedure to check consistency of a given graph.

It turns out we can in fact bring the total complexity of consistency checking graph down to $O(nm)$ by making the following two observations. First, the final states of $\textsc{Nfa}_G$ are irrelevant when calculating the intersection of $\textsc{Nfa}_G$ and $\textsc{Nfa}_{\mathsf{M}}$. Indeed, since $\textsc{Nfa}_G$ does not contain any self-loops, any cycle found in a given intersection should be accepted if $\textsc{Nfa}_{\mathsf{M}}$ accepts the corresponding word. Second, we can avoid repeating work by finding cycles in a *single* intersection of $\textsc{Nfa}_G$ and $\textsc{Nfa}_{\mathsf{M}}$ where *all* the states of $\textsc{Nfa}_G$ are deemed as initial.

More precisely, what we are interested in is finding *strongly connected components* (SCCs) in the intersection of $\textsc{Nfa}_G$ and $\textsc{Nfa}_{\mathsf{M}}$ that also contain the final state of $\textsc{Nfa}_{\mathsf{M}}$[39]. Such SCCs are by construction guaranteed to contain cycles in G that are forbidden under M, and they are non-singletons (as G's relations do not contain any self-loops, and $\textsc{Nfa}_{\mathsf{M}}$'s transitions are based on said relations).

In fact, we can calculate all SCCs in the intersection of $\textsc{Nfa}_G$ and $\textsc{Nfa}_{\mathsf{M}}$ without explicitly computing it. An algorithm for doing so can be seen in algorithm 4.1. Similarly to Tarjan's SCC algorithm[40], the algorithm maintains *nextIndex* representing the current "timestamp"; *Worklist*, a stack maintaining the node exploration order; *status*, a map that tracks whether a given node is unexplored/currently being explored/ fully explored; *index*, a map that tracks the timestamp at which a given node was discovered; and *sccIndex*, a map that tracks the timestamp of the SCC's root a given node belongs in. Observe that all these variables operate on pairs of the form $\langle e, s \rangle$, where e is an event of G, and s a state in $\textsc{Nfa}_{\mathsf{M}}$; we use q_0 for the initial state of $\textsc{Nfa}_{\mathsf{M}}$. Also

[39] *Recall that $\textsc{Nfa}_M$ has a single initial state that is also final.*

[40] Introduction to Algorithms, 3rd Edition [Cor+09]

observe that these variables are all declared as global in the consistent procedure, so that they are not always passed around as arguments.

consistent visits every node of the intersection by iterating over each graph event e, and calling HasAcceptingSCC on the pair $\langle e, q_0 \rangle$ (line 8). If any of the HasAcceptingSCC calls returns true (denoting that an SCC containing an accepting state of M has been found), then consistent returns false.

HasAcceptingSCC recursively visits a given node and its descendants, and then checks whether the node is the root of an SCC that representing an acyclicity violation. Its structure closely follows Tarjan's algorithm, so we will not discuss it in detail here. Instead, let us only focus on two key components: getting the descendants of a given node, and finding accepting SCCs.

Descendants of a given node $\langle e, s \rangle$ are obtained via GetSuccessors. Given a graph event e, GetSuccessors will return all pairs $\langle e', s' \rangle$ where e' can be reached from e in G using some primitive relation (i.e., $\mathtt{po}|_{\mathrm{imm}}$, $\mathtt{co}|_{\mathrm{imm}}$, $\mathtt{rf}|_{\mathrm{imm}}$ and $\mathtt{rb}|_{\mathrm{imm}}$), and s' can be reached from s in NFA_M if the respective transition is taken[41]. Notice that the implicit intersection construction (courtesy of GetSuccessors) is rendered possible because both G and NFA_M are expressed using only primitive relations.

Finally, accepting SCCs are detected using the *acceptingSCC*, which is set if the SCC being examined contains the single initial/final state of NFA_M line 29.

Since NFA_M is typically small and (in contrast to NFA_G) does not depend on the size of the input program, consistent is much more efficient than the previous one: its time complexity is $\mathcal{O}(nm)$, and, since the size NFA_M does not change when verifying a given program P, deciding consistency is effectively linear on the size of P.

4.2.2 *Checking Consistency Incrementally*

A possible use of algorithm 4.1 is to automate consistency checking in the context of model checking and DPOR (see Chapter 5 and 9). However, when it comes to that specific setting, we can adjust the consistency checking routine and obtain an even more optimized algorithm.

The key observation that allows us to do so is that that DPOR does not check for consistency of an arbitrary graph. Rather, given an execution graph G and an event e for which $\mathsf{consistent}_\mathsf{M}(G \setminus e)$ holds, it *incrementally* checks whether $\mathsf{consistent}_\mathsf{M}(G)$ holds[42].

Exploiting this very fact, we can make our consistency checking algorithm more efficient with a simple trick. The key insight here is that instead having all of NFA_G's states as initial, we now know which state of NFA_G we should use as the initial one: the state q_e, where e is a newly added event. Since G is consistent, any cycle that exists in $\mathsf{add}(G, e)$ must involve e. Therefore, we do not have to iterate over events other than e in line 6 of algorithm 4.1 (as the cycles detected in these explo-

[41] *If NFA_M cannot take the same step, such a pair is not returned.*

[42] *DPOR incrementally constructs execution graphs by taking consistent steps; see §5 for more details.*

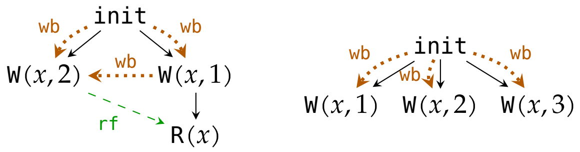


Figure 4.3: Writes-before relation: Two cases of induced edges

rations will not involve e), and only calling HasAcceptingSCC($\langle e, q_0 \rangle$) suffices: because Nfa_M has a single initial/final state, any accepting SCC detected is an actual violation, and some rotation of its constituent primitive relations is accepted by Nfa_M.

4.3 APPROXIMATING COHERENCE WITH WRITES-BEFORE

Suppose we are given a non-coherence-tracking execution graph G, and we want to check whether G is consistent under a model M that involves co. How can we check consistency of G besides naively enumerating all of its co possibilities?

The idea is to compute the *writes-before* (wb) relation, which records the set of co-edges whose direction is forced by the memory model irreflexivity axioms. Computing wb for an arbitrary model M can be done with the fixpoint construction presented in algorithm 4.2. For each axiom of M implying that co; r be irreflexive, we simply make wb "agree" with r.

To see what this means, let us see an example. Consider a model M that enforces coherence, i.e., that hb; eco be irreflexive, or, equivalently, that hb; (rf ∪ (co ∪ rb) ; $\text{rf}^?$) be irreflexive. As the definition implies that co; $\text{rf}^?$; hb; rf^{-1} should be irreflexive, algorithm 4.2 will consider r = $\text{rf}^?$; hb; rf^{-1}.

Fig. 4.3 shows two example graphs where wb edges are induced under such a model. In both execution graphs, init is wb-before the writes of both threads, as po; co needs to be irreflexive. In the left execution graph, W(x, 1) is wb-before W(x, 2), since po; rb = po; rf^{-1}; co should be irreflexive[43]. By contrast, in the right execution graph, the writes of the three threads are not wb-ordered, as there is no kind of ordering (e.g., po, rf) among them.

Calculating wb with the construction of algorithm 4.2 always yields wb ⊆ co (i.e., it is sound, but incomplete). Specifically, by monotonicity of KAT, if there is a consistency violation that uses wb instead of co, then the corresponding graph is indeed inconsistent. If, however, there is no violation with wb, it is still possible that the graph is inconsistent for all possible co ⊇ wb.

As such, when checking for consistency, in principle one has to consider all co ⊇ wb. In practice, however, a given graph is almost always consistent for all co-extensions of wb, and there are many efficient

[43] *Recall that* po ⊆ hb. *Intuitively, if* wb *was the other way, the read should have read 1.*

(model-specific) ways for checking the consistency of a given (non-coherence-tracking) graph[44].

DEFINING WRITES-BEFORE FOR RA As a further example, let us give a formal definition of $\mathtt{wb}$ for RA[45]. RA provides an $\mathtt{hb}$ definition and requires that $\mathtt{hb};\mathtt{eco}$ be irreflexive (see §3.4.2). In turn, $\mathtt{wb}$ can be defined as follows:

$$\begin{aligned}\mathtt{wb} \triangleq\;& (\mathtt{rmw}^{-1};\mathtt{rf}^{-1})^*;\\ & [G.\mathtt{W}];\mathtt{rf}^?;\mathtt{hbloc};(\mathtt{rf}^{-1})^?;[G.\mathtt{W}];(\mathtt{rf};\mathtt{rmw})^* \setminus (\mathtt{rmw}^{-1};\mathtt{rf}^{-1})^*\\ & \textbf{where } \mathtt{hbloc} \triangleq \mathtt{hb} \cap \mathtt{sameloc}\end{aligned}$$

Let us now go over the above $\mathtt{wb}$ definition. First, ignoring all $\mathtt{rmw}$ parts, $\mathtt{wb}$ orders a write w_1 before a write w_2 (a) if w_1 (or some of its readers) happens-before w_2 (i.e., $\langle w_1, w_2\rangle \in \mathtt{rf}^?;\mathtt{hbloc}$), or (b) if w_1 (or some of its readers) happens-before some reader of w_2 (i.e., $\langle w_1, w_2\rangle \in \mathtt{rf}^?;\mathtt{hbloc};\mathtt{rf}^{-1}$).

Second, given two same-location writes w_1 and w_2 s.t. $\langle w_1, w_2\rangle \in \mathtt{wb}$, the $\mathtt{rmw}$ parts ensure that (a) if w_1 is part of an RMW chain, then all writes before it in the chain are $\mathtt{wb}$-before w_2 ($(\mathtt{rmw}^{-1};\mathtt{rf}^{-1})^*$ part), and (b) if w_2 is part of an RMW chain, then w_1 is $\mathtt{wb}$-before all writes after w_2 in the chain ($(\mathtt{rf};\mathtt{rmw})^*$ part). The definition also ensures that $\mathtt{wb}$ will not relate writes in the same RMW chain in the opposite direction of the chain (set difference part).

[44] *"Optimal stateless model checking for reads-from equivalence under sequential consistency" [Abd+19]; "On the Complexity of Checking Transactional Consistency" [BE19]; "The Reads-from Equivalence for the TSO and PSO Memory Models" [Bui+21]*

[45] *The same $\mathtt{wb}$ definition carries over to RC11.*

Algorithm 4.1 Kater: Checking consistency in linear time

```
procedure consistent$_M(G)$
    global $nextIndex \leftarrow 0$
    global $Worklist \leftarrow \emptyset$
    global $status \leftarrow \lambda\langle e, q\rangle.\ \mathsf{unseen}$
    global $index \leftarrow sccIndex \leftarrow \lambda\langle e, q\rangle.\ 0$
    for $e \in G.\mathsf{E}$ do
        if $status[\langle e, q_0\rangle] = \mathsf{unseen}$ then
            if HasAcceptingSCC$_M(G, \langle e, q_0\rangle)$ then return false
    return true

procedure HasAcceptingSCC$_M(G, \langle e, q\rangle)$
    $index[\langle e, q\rangle] \leftarrow sccIndex[\langle e, q\rangle] \leftarrow nextIndex$
    $nextIndex \leftarrow nextIndex + 1$
    Push$(Worklist, \langle e, q\rangle)$
    $status[\langle e, q\rangle] \leftarrow \mathsf{onstack}$
    for $\langle e', q'\rangle \leftarrow$ GetSuccessors$_M(G, \langle e, q\rangle)$ do
        if $status[\langle e', q'\rangle] = \mathsf{unseen}$ then
            if HasAcceptingSCC$_M(G, \langle e', q'\rangle)$ then
                return false
            $sccIndex[\langle e, q\rangle] \leftarrow \min(sccIndex[\langle e, q\rangle], sccIndex[\langle e', q'\rangle])$
        else if $status[\langle e', q'\rangle] = \mathsf{onstack}$ then
            $sccIndex[\langle e, q\rangle] \leftarrow \min(sccIndex[\langle e, q\rangle], index[\langle e', q'\rangle])$
    if $index[\langle e, q\rangle] = sccIndex[\langle e, q\rangle]$ then
        $scc \leftarrow \emptyset$
        $acceptingSCC \leftarrow$ false
        do
            $\langle e', q'\rangle \leftarrow$ Pop$(Worklist)$
            $status[e', q'] \leftarrow \mathsf{left}$
            $scc \leftarrow scc \cup \{\langle e', q'\rangle\}$
            if $q' = q_0$ then $acceptingSCC \leftarrow$ true
        while $\langle e', q'\rangle \neq \langle e, q\rangle$
        if $acceptingSCC \land |scc| > 1$ then
            print "Cycle detected: " $scc$
            return false
    return true
```

Algorithm 4.2 Fixpoint for approximating $\mathtt{co}$ in a model M

```
procedure CalculateWB($\mathsf{M}$)
    $\mathtt{wb} \leftarrow \emptyset$
    do
        $\mathsf{new} \leftarrow \emptyset$
        for each $\mathsf{r}$ such that $\mathsf{irreflexive}(\mathtt{co}\,;\mathsf{r}) \in \mathsf{M}$ do
            $\mathsf{new} \leftarrow \mathsf{new} \cup [\mathtt{W}];(\mathsf{r} \cap \mathtt{sameloc});[\mathtt{W}]$
        $\mathsf{new} \leftarrow \mathsf{new} \setminus \mathtt{wb}$
        $\mathtt{wb} \leftarrow \mathtt{wb} \cup \mathsf{new} \cup (\mathsf{new};\mathtt{wb}) \cup (\mathtt{wb};\mathsf{new})$
    while $\mathsf{new} \neq \emptyset$
```

Part II

VERIFICATION

5 GENMC: MODEL CHECKING UNDER WEAK MEMORY CONSISTENCY

In this chapter, we describe GenMC, a model-checking algorithm that is parametric in the choice of the (weak) memory model and has linear memory requirements. Before diving into the details of GenMC, however, let us briefly dive into the challenges that concurrency poses for verification.

Traditionally, model checkers explore all program states that are reachable from an initial configuration, and ensure that none of them violates a given specification. To avoid exploring the same state over and over, model checkers would simply record the set of visited states, leading to exponential memory consumption.

This led to *stateless model checking* (SMC)[46], that aims to visit all reachable program states without actually recording them. Specifically, SMC verifies a concurrent program merely by enumerating all of its thread interleavings. In turn, because the number of thread interleavings grows exponentially with the program size, other techniques were developed to tackle this problem.

Arguably, the most prominent among these techniques is *dynamic partial order reduction* (DPOR)[47]. Instead of exploring all program interleavings, DPOR partitions the interleavings into equivalence classes, and then strives to explore one interleaving per equivalence class.

To make this idea concrete, let us consider the following example.

$$y := 1 \;\Big\|\; x := 1 \;\Big\|\; a := x \qquad (\textsc{w+w+r})$$

The w+w+r program has 3! interleavings. As can be seen in Fig. 5.1, however, these interleavings can be partitioned into 2 equivalence classes, as the only thing that matters in this program is whether $a := x$ is executed before $x := 1$ (or vice versa).

As such, a DPOR algorithm verifies the program by exploring one interleaving from each equivalence class, effectively observing that the order in which $y := 1$ is executed w.r.t. to the other instructions is irrelevant.

Unfortunately, previous DPOR approaches have two major limitations: (1) they typically assume SC, and (2) they suffer from a memory/optimality trade-off — they either require memory exponential in the size of the program under verification, or can explore many unnecessary program executions.

GenMC solves both these two challenges in a unified framework, and the key idea that allows it to do so is the usage of declarative semantics. Rather than following existing work and representing program

[46] *"Software Model Checking: The VeriSoft Approach"* [God05]

[47] *"Dynamic partial-order reduction for model checking software"* [FG05]

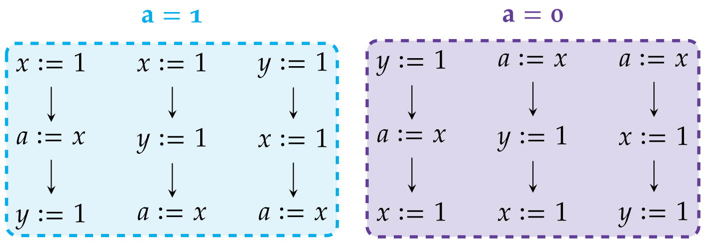


Figure 5.1: W+W+R: interleavings and equivalences classes

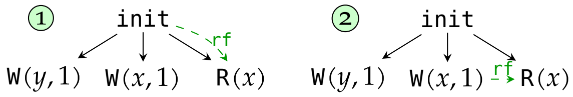


Figure 5.2: The graphs of W+W+R subsume its equivalence classes

executions as traces, GenMC represents them as execution graphs (cf. Fig. 5.2). In turn, instead of verifying programs by enumerating interleavings from equivalence classes, GenMC simply verifies a program by enumerating its execution graphs. As we are going to shortly see, it is this change of representation that enables exploration algorithms that are parametric in the choice of the memory model.

In the rest of this chapter, we present an intuitive account of GenMC and the requirements it sets the underlying memory model. In subsequent chapters we show how it obtains linear memory consumption (§6), how it can be extended for persistency models (§8), and how it can be improved for various programming patterns (§7).

5.1 REQUIREMENT #1: NO "OUT OF THIN AIR"

Given a program like W+W+R and its execution graphs under a model M, our goal is to *enumerate* such executions systematically. A simple approach taken, e.g., by Herd[48] and CPPMEM[49], is to enumerate *all* possible executions and filter them according to the consistency predicate of the memory model.

Unfortunately, however, even such a simple approach is not possible for an arbitrary model M. To see why, consider the following example under the (arguably useless) memory model that deems every execution graph consistent:

$$\begin{array}{c||c} a := y & b := x \\ x := a & y := b \end{array} \qquad \text{(LB+DEP)}$$

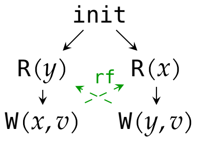


Figure 5.3: LB+DEP: $x = y = v$

Under such a model, the program can return $x = y = v$, for *any* value v, by having both threads read v and write v in a circular fashion as shown in Fig. 5.3.

In the weak memory literature, such executions are considered problematic because they generate values "out of thin air" (OOTA)[50] and inhibit compositional reasoning.

[48] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [AMT14]

[49] *"Mathematizing C++ concurrency"* [Bat+11]

[50] *"The Java memory model"* [MPA05]; *"Outlawing ghosts: Avoiding out-of-thin-air results"* [BD14]

To preclude OOTA values and thereby enable the enumeration of the program execution graphs, we require that the underlying memory model be well-formed.

Definition 5.1.1 (Well-formedness)**.** A memory model M is *well-formed* iff for all G, if $\mathsf{consistent}_{\mathsf{M}}(G)$, then $\mathsf{consistent}_{\mathsf{M}}(G')$ for all $G' \approx G$, and $G.\mathsf{corder}$ is irreflexive.

This requirement is satisfied by several models (e.g., SC, TSO, PSO, and RC11, among others), and ensures that loop-free programs have finitely many executions. In what follows, we mostly use examples from non-dependency-tracking models to simplify the presentation. We provide run of GenMC under a dependency-tracking model in §5.7.

Remark 3. While restricting OOTA behaviors, well-formedness also forbids the outcome $a = b = 1$ for the following litmus test:

$$\begin{array}{l||l} a := y; & b := x; \\ x := 1 + a - a & y := 1 + b - b \end{array} \qquad \text{(LB+FAKEDEP)}$$

This outcome is not an OOTA one since, even though there are syntactic dependencies in both threads, a compiler is within its right to transform this program to one where both threads simply write the value 1 (in which case a syntactic dependency does not exist anymore).

As such, certain models like Promising[51] and WeakestMO[52] try to distinguish between LB+DEP and LB+FAKEDEP. However, these models are not even defined in terms of execution graphs, and are thus beyond the scope of this thesis[53].

5.2 REQUIREMENT #2: PREFIX-CLOSEDNESS

Even without OOTA executions, generating *all* executions and then checking consistency does not scale (see §10.2.1), as there is an exponential number of `rf`/`co` options that have to be examined to check whether a given execution is consistent.

A much better approach, followed by most tools[54], is to construct executions *incrementally* by adding events one at a time and checking for consistency at each step, thereby avoiding the exploration of inconsistent graphs.

For this approach to work, the underlying memory model must satisfy the following condition:

> Every non-empty consistent graph has a causally maximal event that, if removed, yields a consistent graph.

This condition ensures that each execution can be generated by adding its events in *some* total extension of the corder order, and checking for

[51] *"A promising semantics for relaxed-memory concurrency"* [Kan+17]

[52] *"Grounding thin-air reads with event structures"* [CV19]

[53] *Moiseenko, Kokologiannakis, and Vafeiadis [MKV22] have devised a DPOR algorithm that can operate under certain models similar to Promising/WeakestMO.*

[54] *Examples include CDSChecker [ND13], Nidhugg [Abd+15; Abd+14], Tracer [Abd+18], and RCMC [Kok+17].*

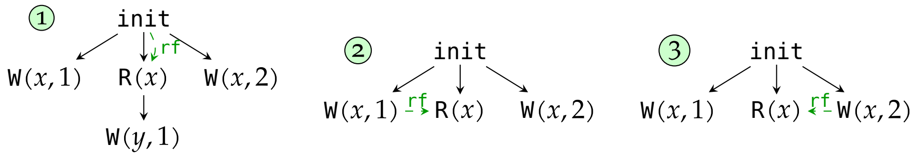


Figure 5.4: Execution graphs of w+rw+w under SC

consistency after each step. For instance, execution ② in Fig. 5.2 can be generated by adding its events in the following order: init, $\mathtt{W}(x,1)$, $\mathtt{W}(y,1)$, and $\mathtt{R}(x)$.

The problem with such a condition is that, in order to generate all executions of a program, one must in principle consider *all* possible extensions of corder. This, however, very often leads to duplicate explorations.

Therefore, we would like to generate all executions without considering all possible extensions of corder, regardless of the memory model. In fact, we can do this for all well-known memory models. In particular, models such as SC, TSO, PSO, and RC11 all satisfy an even stronger guarantee, namely prefix-closedness:

Definition 5.2.1 (Prefix-closedness)**.** A memory model M is *prefix-closed* iff for all G and $E \subseteq G.\mathsf{E}$, if $\mathit{dom}(G.\mathsf{corder};[E]) \subseteq E$ and $\mathsf{consistent}_{\mathsf{M}}(G)$, then $\mathsf{consistent}_{\mathsf{M}}(G|_E)$.

Prefix-closedness ensures that to generate a particular execution, it is sufficient to consider *any* total extension of corder, and construct the execution by adding its events following that total order.

As we demonstrate in the next section, we can leverage this fact and *fix* an order in which we add execution events one at a time, thus generating all executions of a program systematically.

5.3 A FIRST EXAMPLE

Let us now explain how GenMC can generate the execution graphs of the w+rw+w program (shown below) by adding its events in a *fixed* order given by thread identifiers: first the events of (the leftmost) thread I, then the events of thread II, and so forth. The execution graphs of w+rw+w under SC are depicted in Fig. 5.4[55].

$$x := 1 \;\left\|\; \begin{array}{l} a := x; \\ \textbf{if } a = 0 \textbf{ then } y := 1 \end{array} \right\|\; x := 2 \qquad (\text{W+RW+W})$$

As depicted in Fig. 5.4, the read in thread II may read either 0 (from the initialization write), 1 (from the write in thread I), or 2 (from thread III).

[55] *Observe that these graphs are not coherence-tracking.We come back to this in §5.6.*

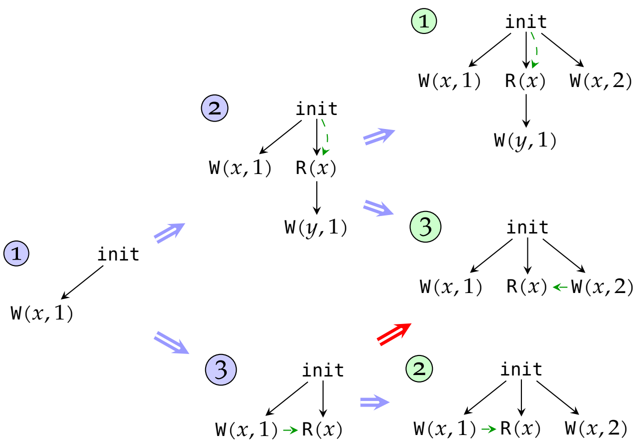


Figure 5.5: GenMC: Enumerating the execution graphs of w+rw+w

GenMC enumerates these graphs in a depth-first manner (cf. Fig. 5.5). We start with an initial graph $G_\emptyset$ containing only the initialization write.First, we add the $\mathtt{W}(x,1)$ write of thread I to $G_\emptyset$, simultaneously adding the appropriate po edge between the events, leading to graph ①[56].

Continuing in thread order, we next add the $\mathtt{R}(x)$ read of thread II, which may read from either of the writes in the graph, yielding the distinct graphs ② and ③ (one for each case)[57]. As such, each time we add a read that can read from more than one place, we examine all options recursively (denoted by ⇒).

We refer to such alternative exploration options such as $\mathtt{W}(x,1)$ as *forward revisits* since they are already in the graph when the read ($\mathtt{R}(x)$) is added to the graph.

Let us suppose that GenMC continues with the subexploration of graph ②. Since the value read is 0, we next add the $\mathtt{W}(y,1)$ write of thread II, and finally the $\mathtt{W}(x,2)$ of thread III, which yields graph ①. This first subexploration is now completed.

Notice, however, that if GenMC takes no action when $\mathtt{W}(x,2)$ is added, then graph ③ where $\mathtt{R}(x)$ reads 2 will not be explored. Indeed, as $\mathtt{W}(x,2)$ was not present when $\mathtt{R}(x)$ was added, it was not considered as a possible rf-option at that point.

Thus, to guarantee completeness (i.e., that all execution graphs of a given program are explored), whenever a write w is added to the graph, GenMC checks whether it is consistent for any existing read r to add from the newly-added write. If so, it initiates another subexploration where r reads from w. In our example, it is consistent for $\mathtt{R}(x)$ to read 2 from the newly-added $\mathtt{W}(x,2)$, and we therefore initiate a recursive subexploration where $\mathtt{R}(x)$ reads 2 (graph ②).

We refer to such alternative exploration options such as $\mathtt{W}(x,2)$ as *backward revisits* since they are added to the graph after the corresponding read ($\mathtt{R}(x)$).

[56] *Recall from §2, that we use green-circled numbers to denote complete, consistent executions and red-circled numbers to denote inconsistent executions. We also use blue-circled numbers to denote incomplete (consistent) graphs during some exploration.*

[57] *Note that recording both graphs is unnecessary: in the general case, we need to record one graph for each of the reads-from options of each read. The two graphs are identical up to the read, which is the point of divergence.*

Let us now continue with the backward revisit of $\texttt{R}(x)$. Observe that graph ③ (where $\texttt{R}(x)$ is backward-revisited by $\texttt{W}(x, 2)$), is restricted so that it contains the events added *prior* to the read. Restricting the graph is important because events added after the read may depend on its value. For instance, it is crucial to remove $\texttt{W}(y, 1)$ as it is only present when 0 is read from x.

In a similar manner, graph ③ also contains the events that $\texttt{W}(x, 2)$ depends on (its causal prefix). Keeping these events is important as they are necessary to "trigger" $\texttt{W}(x, 2)$. For instance, if $x := 2$ in W+RW+W is wrapped in the conditional **if** $y = 1$ **then**, the presence of the $\texttt{W}(x, 2)$ event in the graph depends on the value read for y, i.e., the events before $\texttt{W}(x, 2)$ in causal order[58].

Since graph ③ is complete (there are no more events to add), let us now proceed with the last remaining subexploration, namely the forward-revisit of $\texttt{R}(x)$ (graph ③). Continuing from graph ③, we add the $\texttt{W}(x, 2)$ and obtain the last (complete) execution graph of W+RW+W (graph ②).

At this point, however, we have to be careful and not revisit $\texttt{R}(x)$ again. Indeed, if we do this, we will end up initiating the same subexploration twice (denoted by ⇒). A simple way to avoid exploring duplicate executions is to store the graphs that have occurred as a result of a backward revisit[59]. Finally, as all the execution graphs of W+RW+W have been explored, the algorithm terminates.

In §6, we show that storing the graphs that have occurred as a result of a backward revisit is unnecessary, and present a new core algorithm for GenMC that has linear memory consumption.

5.4 REQUIREMENT #3: EXTENSIBILITY

As described in §5.3, GenMC generated all execution graphs of W+RW+W, even though it did not add events according to corder. The reason it managed to do this is because of backward revisiting: in cases where a read is added before the write it should read from (e.g., reading from $\texttt{W}(x, 2)$ in ③), it is then later backward-revisited when the write is added.

This then leads to the question: could events added after a read r affect the consistency of the execution in a way that the write from which r should read from is never added?

Perhaps surprisingly, the answer is yes. For example, consider the following program under a (contrived) memory model that dictates "if a read of y reads 0, then there cannot be a read of x that also reads 0":

$$a := x \;\Big\|\; b := y \;\Big\|\; x := 42 \tag{R+R+W}$$

In this case, adding the events in thread order results in a graph where both x and y read 0, which is then dropped as inconsistent, and

[58] Another way to think of the restricted graph G obtained by the backward revisit of r from w is that G emulates the scenario where w (and the events it depends on) were present when r was added.

[59] One does not actually have to store the entirety of these graphs. Given a read r it suffices to store all prefixes of the writes that have backward-revisited it, for as long as r remains in the exploration. See [KRV19] for a full treatment.

thus we cannot generate the execution where thread I reads 42. This brings us to our third requirement on memory models, extensibility.

Intuitively, a memory model M is extensible if, given a consistent execution graph G and a new event to be added to G, there is some way to add e to G such that the resulting graph is also consistent.

Before formally defining extensibility, let us present a couple of useful helper definition. First, given a graph G, we define $\mathsf{SetRF}(G, r, w)$, which returns a graph G' that is identical to G except for its rf component, which has r reading from w:

$$G'.\mathtt{rf} = G.\mathtt{rf} \setminus (G.\mathsf{E} \times \{r\}) \cup \{\langle w, r\rangle\}$$

We also define $\mathsf{SetCO}(G, w, w_p)$ that returns a graph G' that is identical to G except for its co component, which has w after w_p in co:

$$\begin{aligned} G'.\mathtt{co} =& G.\mathtt{co} \setminus (G.\mathsf{E} \times \{w\}) \cup \{\langle w_p, w\rangle\} \cup \\ & \left\{\langle w', w\rangle \;\middle|\; \langle w', w_p\rangle \in G.\mathtt{co}\right\} \cup \left\{\langle w, w'\rangle \;\middle|\; \langle w_p, w'\rangle \in G.\mathtt{co}\right\} \end{aligned}$$

(For non-coherence-tracking graphs, $\mathsf{SetCO}(G, \_, \_) = G$.)

Now, we can define extensibility as follows:

Definition 5.4.1 (Extensibility)**.** A memory model M is *extensible* if there is a function $\mathsf{f}_{\mathsf{ext}} : \mathsf{Exec} \times (\mathsf{R} \cup \mathtt{W} \setminus \mathtt{W}^{\mathsf{excl}}) \to \mathsf{Event}$ such that, for all $G \in \mathsf{Exec}$ and $e \in G.\mathsf{E}$ such that e is causally maximal and $\mathsf{consistent}_{\mathsf{M}}(G \setminus \{e\})$, exactly one of the following holds:

- if $e \in G.\mathsf{R}$, then $\mathsf{consistent}_{\mathsf{M}}(\mathsf{SetRF}(G, e, \mathsf{f}_{\mathsf{ext}}(G, e)))$
- if $e \in G.\mathtt{W} \setminus \mathtt{W}^{\mathsf{excl}}$, then $\mathsf{consistent}_{\mathsf{M}}(\mathsf{SetCO}(G, \mathsf{f}_{\mathsf{ext}}(G, e), e))$
- if $e \in G.\mathtt{W}^{\mathsf{excl}}$ and $\langle \mathsf{f}_{\mathsf{ext}}(G \setminus e, e_p), e_p\rangle \in G.\mathtt{rf}$, then $\mathsf{consistent}_{\mathsf{M}}(\mathsf{SetCO}(G, \mathsf{f}_{\mathsf{ext}}(G \setminus e, e_p), e))$, where $e_p = \mathrm{pred}_{G.\mathsf{po}}(e)$
- if $e \notin G.\mathsf{R} \cup G.\mathtt{W}$, then $\mathsf{consistent}_{\mathsf{M}}(G)$

In essence, $\mathsf{f}_{\mathsf{ext}}$ is an oracle prescribing how a particular event e can be added to rf or co to preserve consistency. In case e is a read, $\mathsf{f}_{\mathsf{ext}}$ returns an rf-option for it that preserves consistency, while if e is a write, it returns its (immediate) co-predecessor. Observe that $\mathsf{f}_{\mathsf{ext}}$ only provides a co-predecessor for an exclusive write, if its corresponding exclusive read is reading from the place $\mathsf{f}_{\mathsf{ext}}$ is prescribing. We get back to the extensibility of exclusive writes in §5.5.

Extensibility holds for all well-known memory models, and excludes "nonsensical" memory models such as that above. In particular, the model above is not extensible, as the consistent execution of R+R+W comprising the initialization events and $\mathtt{R}(x)$ of thread I reading 0 cannot be extended by adding $\mathtt{R}(y)$ for any choice of rf.

5.4.1 *Defining the Extensibility Oracle* $\mathsf{f}_{\mathsf{ext}}$

Given that extensibility holds for all well-known models, it is natural to wonder how $\mathsf{f}_{\mathsf{ext}}$can be defined.

Fortunately, doing so for all well-known non-dependency tracking models is trivial: $\mathsf{f}_{\mathsf{ext}}$ can simply be defined as $\mathsf{f}_{\mathsf{ext}}(G, e) \triangleq \max_{G.\mathtt{co}} G.\mathtt{W}_{\mathtt{loc}(e)}$, i.e., to always return the co-latest write.

For dependency-tracking models, on the other hand, defining $\mathsf{f}_{\mathsf{ext}}$ is not so straightforward, as demonstrated by the LB-EXT program below.

$$\begin{array}{r||l} a := x & b := y \\ y := 1 & c := y \\ & x := c \end{array} \qquad (\text{LB-EXT})$$

Assuming POWER, one possible partial (consistent) graph of LB-EXT can be seen in Fig. 5.6. Observe that the causal prefixes of all events are present in the graph, even though the first read of thread II is missing. In this case, if we define $\mathsf{f}_{\mathsf{ext}}$ as above then we cannot add the missing read reading 1 (the co-maximal write), as that would violate consistency[60].

One way of defining $\mathsf{f}_{\mathsf{ext}}(G, e)$ that would avoid the above issue in cases where e is a non-po-maximal read, is as follows:

$$\mathsf{f}_{\mathsf{ext}}(G, e) \triangleq \begin{cases} \max_{G.\mathtt{co}} G.\mathtt{W}_{\mathtt{loc}(e)} & e = \max_{\mathtt{poloc}_{\mathtt{tid}(e)}}(G.\mathtt{E}) \\ G.\mathtt{rf}(\mathsf{succ}_{\mathtt{poloc}_{\mathtt{tid}(e)}}(e)) & \text{otherwise} \end{cases}$$

Note that while such a definition might guarantee extensibility for a given formulation of e.g., the POWER model, it might not work for a different formulation, or a different model. For instance, consider the MP-EXT program below, assuming a formulation of POWER that defines $\mathtt{ppo} \triangleq (G.\mathtt{rf} \cup G.\mathtt{deps})^+$, but also requires that $\mathtt{ppo} \cup \mathtt{po};[\mathtt{F}] \cup [\mathtt{F}];\mathtt{po} \cup \mathtt{rb}$ be acyclic (in a separate axiom).

$$\begin{array}{r||l} a := x & y := 1 \\ \mathtt{fence} & \mathtt{fence} \\ b := y & x := 1 \end{array} \qquad (\text{MP-EXT})$$

For such a formulation, given the partial graph of Fig. 5.7, even though the read of y in thread I can be added reading 0, after doing so, there is no way to add the events of thread II so that consistency is maintained: there will be a cycle caused by rb edge of the read of y in thread I.

All in all, for dependency-tracking models, $\mathsf{f}_{\mathsf{ext}}$needs to be defined on a per-case basis, and with $(G.\mathtt{rf} \cup G.\mathtt{deps})^+ \subset \mathtt{ppo}$.

Now what about non-coherence-tracking graphs, where co is not explicitly tracked? In such cases, one typically approximates co with some other relation. Indeed, in §4.3, we define wb, an approximation

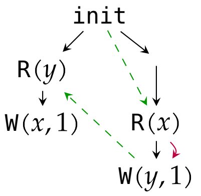


Figure 5.6: A prefix-closed execution of LB-EXT under POWER

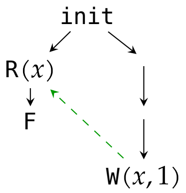


Figure 5.7: A prefix-closed execution of MP-EXT under POWER

[60] *POWER dictates that if a read in a given thread reads from a certain write,* po*-earlier reads cannot read from more recent writes.*

[61] *Even though the precise definition of* wb *is not important for this discussion, interested readers are encouraged to read §4.3.*

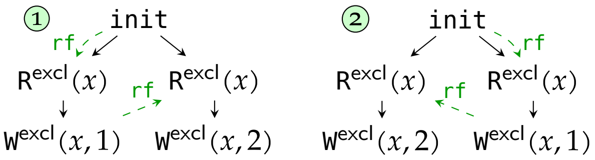


Figure 5.8: The executions of the FAI/2 program.

of co that partially orders writes in each memory location, instead of totally ordering them[61].

That said, we cannot define f_{ext} simply by replacing co with wb in the definition of maximally added events, because there may be multiple wb-maximal writes in a consistent execution graph.

To maintain optimality, we need a tiebraker between these wb-maximal events. A simple solution is to pick an arbitrary tiebraker (e.g., the write that was inserted last to the graph, i.e., the $<_G$-maximal event among the wb-maximal ones). This solution works but it requires a stronger extensibility property of the underlying memory model: extending a consistent execution graph with a read event that reads from any wb-maximal event should result in a consistent graph. While this property holds of certain simple models, such as release-acquire consistency, it does not hold of other models, including SC. The problem is that while a consistent graph must have its co be some location-total order extending wb, it is not the case that all location-total orders extending wb satisfy the consistency predicate of the model.

A better solution is to only assume that there is a way (given by an oracle function GetConsCO), to calculate a co relation according to which a graph is consistent. A naive implementation of this function is to enumerate all location-total orders extending wb in a systematic fashion, and return the first one that satisfies the consistency predicate of the model. Better implementations can be derived from the more efficient ways of checking consistency of an execution graph that does not contain a co component[62].

In any case, however, given such an oracle, the definition of f_{ext} can remain the same as in the coherence-tracking case.

5.5 READ-MODIFY-WRITE OPERATIONS

Let us now get back to the extensibility of exclusive writes. Indeed, many memory models prescribe rf-functionality constraints requiring that certain writes be read by at most one read. For instance, in case of the RMW (read-modify-write) instructions, e.g., CAS (compare-and-swap) or FAI (fetch-and-increment), to ensure their atomicity, two RMW events may not read from the same write[63].

[62] *"Optimal stateless model checking for reads-from equivalence under sequential consistency"* [Abd+19]; *"On the Complexity of Checking Transactional Consistency"* [BE19]; *"The Reads-from Equivalence for the TSO and PSO Memory Models"* [Bui+21]

[63] *These constraints are not exclusive to RMWs. For instance, as discussed in* [KRV19], *a lock library may require rf-functionality to ensure mutual exclusion.*

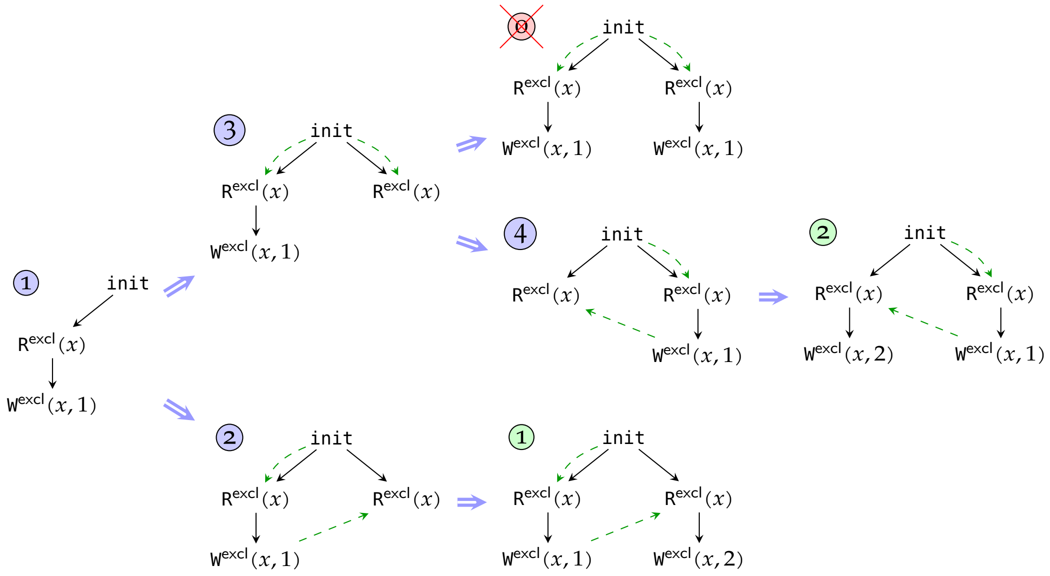

0
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
3
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
4
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
2
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,2)$
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
1
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
2
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
1
init
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,1)$
$\mathsf{R}^{\mathsf{excl}}(x)$
$\mathsf{W}^{\mathsf{excl}}(x,2)$

Figure 5.9: GenMC: Enumerating the execution graphs of FAI/2

Handling such constraints requires additional care. Consider the program below and its SC-executions depicted in Fig. 5.8.

$$a : \texttt{fetch\_add}(x) \;\Big\|\; b : \texttt{fetch\_add}(x) \qquad (\text{FAI}/2)$$

Execution ① captures the case where thread I increments x first, while ② captures the case where thread II increments x first.

Let us run GenMC on this example (cf. Fig. 5.9). We proceed by adding the RMW instruction of thread I (a) which reads from the initialization write (graph ①). We then add the read of the RMW instruction of thread II (b), for which it is consistent[64] to read both 0 and 1.

The exploration where b reads 1 (graph ②) is straightforward, and leads to the full execution graph ①. Observe, however, that when the write of b is added, it cannot backward-revisit the read of a, as that would lead to a causal cycle, thereby violating well-formedness (Def. 5.1.1).

In the exploration where both a and b read 0 (graph ③), if we simply try to add the write of b, then the graph becomes inconsistent (graph ⓪). On the other hand, if we immediately discard the graph, we will fail to generate execution graph ②.

To remedy this, we allow for single-step, temporary inconsistency in the graph for RMW writes. More specifically, when the write of b is added, we do not immediately discard the execution graph, but rather we first backward-revisit existing reads, and then discard the execution graph.

Using the above procedure, not only do we manage to reverse the order between conflicting RMWs, but we also do not embark into fruit-

[64] *RMW-atomicity does not rule out the execution graph until the write of b is added as well.*

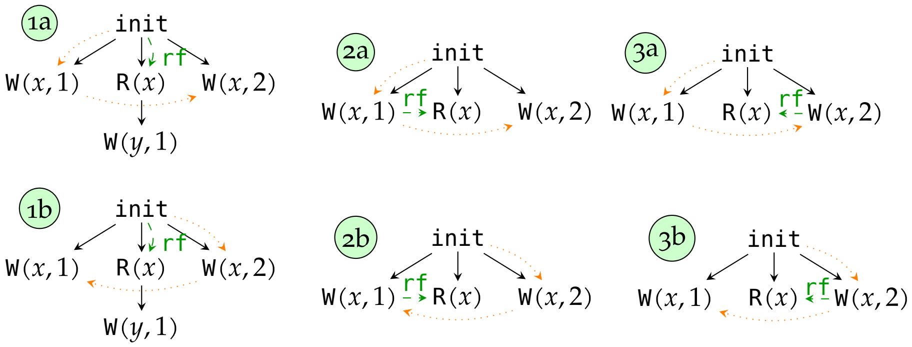


Figure 5.10: Execution graphs of W+RW+W under SC (with co)

less explorations (which would be the case if we were to keep exploring inconsistent executions).

Observe, however, that extensibility could not guarantee that the write of the RMW of thread II could be added in ③, as the read of thread II was not reading from the write prescribed by $\mathsf{f}_{\mathsf{ext}}$.

As such, one may wonder: "For an arbitrary program with events in-between the RMWs, what if it were inconsistent for the exclusive read of the second RMW to read from the same place as the first?" At a first glance, in such cases the algorithm seems unable to produce the execution where the second RMW is executed before the first.

Fortunately, however, GenMC will always explore *some* execution where it is indeed consistent for the read of the second RMW to have the same rf as the read of the first: the execution where all the events added between the RMWs were added in the way prescribed by $\mathsf{f}_{\mathsf{ext}}$.

To see why such an execution is consistent, let us assume that $\mathsf{f}_{\mathsf{ext}} \triangleq \max_{G.\mathtt{co}} G.\mathsf{W}_{\mathtt{loc}(e)}$. Further, let us consider a run of the algorithm where the read of the first RMW is added first, then the read of the second RMW, and then the write of the first (and possibly all other events between the RMWs). If all the events are added in the prescribed by $\mathsf{f}_{\mathsf{ext}}$, then the resulting graph is consistent.

Formally, we straightforwardly get the following corollary from the extensibility definition:

Corollary 1. Given an extensible model M, for all G, r_1, r_2, if $r_1, r_2 \in G.\mathsf{R}^{\mathsf{excl}}$, $r_1 \in dom(G.\mathtt{rmw})$, r_2 is causally maximal, $\mathsf{consistent}_{\mathsf{M}}(G \setminus \{r_2\})$, and all $r_1 <_G e <_G r_2$ were added in the way prescribed by $\mathsf{f}_{\mathsf{ext}}$, then $\mathsf{consistent}_{\mathsf{M}}(\mathsf{SetRF}(G, r_2, G.\mathtt{rf}(r_1)))$.

From this corollary, it is also easy to see that such single-step inconsistencies always lead to a consistent execution (where the order between two RMWs is reversed).

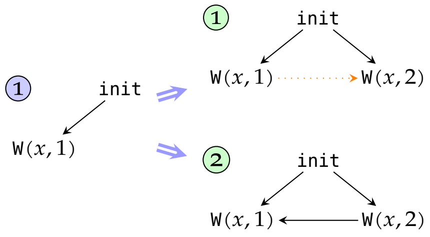


Figure 5.11: GenMC: Enumerating co-tracking graphs of w+w

5.6 Shasha-Snir and Reads-From Equivalence

So far we have used GenMC to generate all three SC-consistent execution graphs of w+rw+w (shown in Fig. 5.4). These executions, however, do not exactly correspond to the actual execution graphs of w+rw+w under SC. As discussed in §2.3, SC also requires us to record the coherence order co, which totally orders all writes to a given memory location.

As such, the three (non-coherence-tracking) execution graphs in Fig. 5.4 correspond to the six (coherence-tracking) execution graphs depicted in Fig. 5.10. In this program, each behavior corresponds to two execution graphs representing the two ways W$(x, 1)$ and W$(x, 2)$ could be ordered by co.

In fact, the graphs in Fig. 5.4 and Fig. 5.10 represent the executions of w+rw+w under different equivalence partitionings. The graphs of Fig. 5.10 represent the executions of the program under the standard Shasha-Snir equivalence[65] (or Mazurkiewicz equivalence[66], in the context of SC), while the graphs of Fig. 5.4 represent the executions of w+rw+w under the coarser reads-from equivalence[67].

In terms of verification, it would be certainly desirable to be able to verify a program under either partitioning[68]. In what follows, we show how GenMC can be adapted to operate under the Shasha-Snir partitioning.

The key idea is to treat alternative co-placings similar to forward revisits. As an example, consider the w+w program below and the exploration shown in Fig. 5.11.

$$x := 1 \;\Big\|\; x := 2 \qquad \text{(w+w)}$$

GenMC first adds W$(x, 1)$ (graph 1), and then recursively explores both co-placings for W$(x, 2)$ (graphs 1 and 2).

As far as backward revisiting and co-placings are concerned, GenMC first backward-revisits a given read, and then tries all possible co-placings of the revisiting write in the resulting graph. A full description is provided in §5.8.

[65] *"Efficient and correct execution of parallel programs that share memory"* [SS88]

[66] *"Trace Theory"* [Maz87]

[67] *"Data-centric dynamic partial order reduction"* [Cha+17]

[68] *Always verifying under reads-from might seem tempting. However, apart from the potentially more expensive consistency checks, real-world benchmarks typically have the same number of executions under both partitionings.*

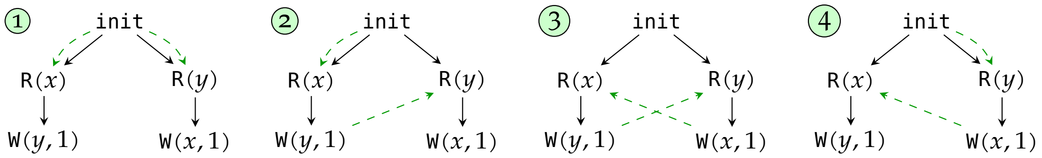


Figure 5.12: Execution graphs of LB

5.7 DEPENDENCY-TRACKING MODELS

Finally, let us now examine how GenMC can be extended to handle dependency-tracking models like POWER and ARM, using the LB program below as an example.

$$\begin{array}{l||l} a := y; & b := x; \\ x := 1 & y := 1 \end{array} \qquad (\textsc{lb})$$

The execution graphs of LB under POWER/ARM can be seen in Fig. 5.12. As it can be seen, even though $G.\mathtt{po}\mathtt{rf}$ might contain cycles in such models (e.g., graph ③), the models still satisfy well-formedness: no event is included in its causal prefix (using the definition of §2.4.1).

As such, the algorithm presented so far works out-of-the-box for LB, with the only exception being the definition of the causal prefix used when cutting the graph as a result of a backward revisit.

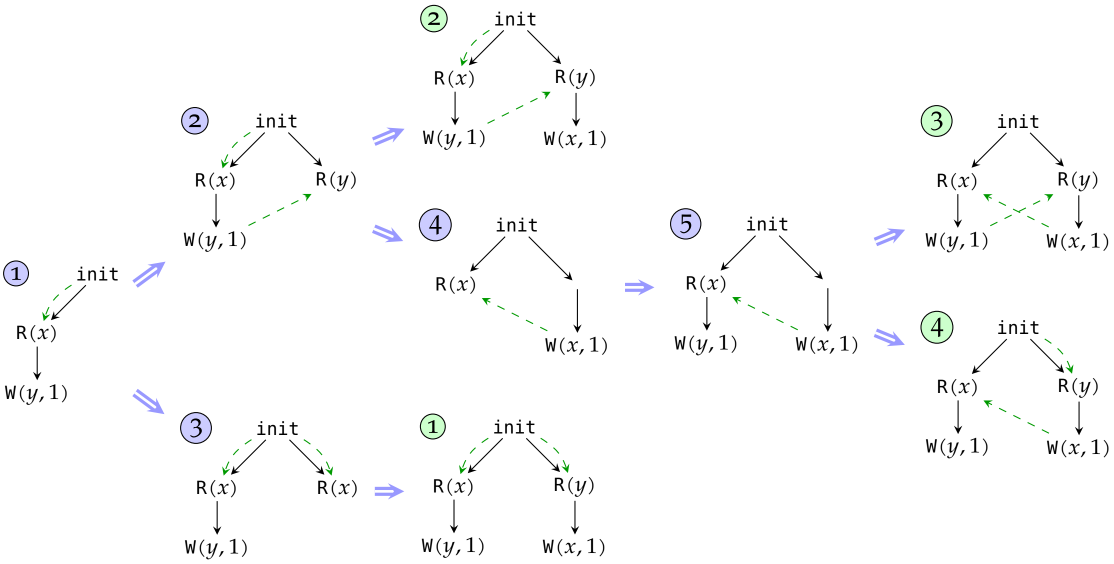


Figure 5.13: GenMC: Enumerating the execution graphs of LB

The exploration procedure for LB can be seen in Fig. 5.13. GenMC starts from the empty graph and adds the events of thread I (graph ①). These can only be added in a single way, as only the initializer event is present when $\mathtt{R}(x)$ is added, and no read of y is present when $\mathtt{W}(y, 1)$ is added.

Next, GenMC will add the $\mathtt{R}(y)$ event corresponding to $b := y$. Since this read can read both from the initializer and from $\mathtt{W}(y, 1)$, GenMC will initiate two subexplorations (graphs ② and ③).

Let us assume that GenMC first explores graph ②. In the next step, it adds the $\mathtt{W}(x, 1)$ of thread II, which can potentially backward-revisit $\mathtt{R}(x)$[69]. (If it does not, we obtain execution ②.)

In the case where $\mathtt{W}(x, 1)$ does backward-revisit $\mathtt{R}(x)$ we obtain graph ④, which contains a "hole" in the place of $\mathtt{R}(y)$. Indeed, as the causal prefix of $\mathtt{W}(x, 1)$ is empty, the cut graph only contains $\mathtt{W}(x, 1)$ itself, as well as $\mathtt{R}(x)$ (and the event added before it).

Removing $\mathtt{R}(y)$ in the cut graph is crucial. When it is re-added in graph ⑤, it will be able to read both 0 and 1, and both of these options will be explored by GenMC (leading to graphs ③ and ④). If it were not removed, however, the obtained graph would be inconsistent, as $\mathtt{R}(y)$ would be reading from (the non-existent) $\mathtt{W}(y, 1)$, thereby violating graph well-formedness.

Finally, when GenMC explores the forward revisit for $\mathtt{R}(x)$ (graph ③), it will add $\mathtt{W}(x, 1)$ again. This time, however, $\mathtt{W}(x, 1)$ will not backward-revisit $\mathtt{R}(x)$ (even though it is *not* in its causal prefix), as graph ④ (where the backward revisit occurs) has been explored before.

5.8 ALGORITHM

In this section, we present GenMC in its entirety. We start with a variant of GenMC for the Mazurkiewicz/Shasha-Snir equivalence (which fully tracks co), and then, in §5.8.2, we adapt GenMC to work for the coarser reads-from equivalence.

5.8.1 *Overview*

GenMC uses a consistent execution graph to drive the exploration. Starting from an empty graph, GenMC repeatedly interprets the program to find the next event to be added to the graph. Whenever a read is added and more than one rf options exist, the algorithm explores them recursively, in a depth-first manner.

As explained in §5.3, whenever a read is added to the graph, GenMC detects the available places it can read from by scanning the graph, and not the program. And as not all possible writes may have been added to the graph when a read is added, whenever GenMC adds a write, it checks whether any of the existing reads in the graph can be backward-revisited and made to read from the newly added write.

The algorithm consists of two main components: (1) the interpreter, which executes the program and produces the next event to be added to an execution graph; and (2) the exploration algorithm, which repeatedly calls the interpreter to generate every execution of the pro-

[69] *In the non-dependency tracking case, writes do not revisit reads in their* porf*-prefix, since* corder $\triangleq$ porf*. In the dependency-tracking case, revisiting such reads is possible.*

Algorithm 5.1 Generating events incrementally

```
procedure Gen($G, a$)
    if $a \notin G.\mathsf{E}$ then
        produce $a$
    return $a$
```

Algorithm 5.2 Choosing non-blocked threads

```
procedure NextEvent$_P(G)$
    $S \leftarrow \{\text{ExecThread}(t, sprog, G) \mid \langle t, sprog\rangle \in P\}$
    if $S = \emptyset$ then return $\bot$
    $a \leftarrow \min_{<_{\mathsf{next}}} S$
    $G.\mathsf{E} \leftarrow G.\mathsf{E} +\!\!+ a$
    return $a$
```

gram exactly once. These two components can be thought of as as "coroutines" calling each other.

In what follows, we describe these two components in turn.

THE PROGRAM INTERPRETER The interpreter defines the NextEvent$_P(G)$ routine, which continues interpreting the program from where it had previously stopped until the next event is added to the graph, after which it returns the newly added event. If no further event can be added (i.e., the program has terminated), it returns $\bot$.

Technically, NextEvent$_P(G)$ is an incremental version of ExecProgram (algorithm 2.1), which instead of checking that all the events resulting from P belong to G, simply returns the next event to be added to G.

Concretely, this is done in two steps. First, we replace the Gen procedure of algorithm 2.1 with that of algorithm 5.1. Whenever the event a passed to Gen does not already belong to $G.\mathsf{E}$, the new Gen returns it with a **produce** statement so that it can be added to the graph. The **produce** statement means that ExecThread halts at that point and returns a to its caller.Equivalently, we can also think of the **produce** as throwing an exception that NextEvent catches in order to find the next event to be added to the graph[70].

Second, as far as choosing which thread to execute next, even though ExecProgram explores each thread to completion before moving on to the next one, this does not need to be the case for NextEvent. Indeed, we simply assume that there is some total order $<_{\mathsf{next}}$ on events such that a) it respects the program order (i.e., $\mathsf{po} \subseteq <_{\mathsf{next}}$), and b) any read and write events that correspond to the same RMW instruction are adjacent in $<_{\mathsf{next}}$ (i.e., $\mathsf{rmw} \subseteq <_{\mathsf{next}}|_{\mathsf{imm}}$)[71]. Given a set $\mathsf{avail}(G)$ of available events that could be added to G, (i.e., namely, the set of next events of each non-terminated, non-blocked thread of the program), NextEvent$_P(G)$ adds the $<_{\mathsf{next}}$-minimal such event w.r.t. to $<_{\mathsf{next}}$. In

[70] *Observe that the interpreter re-runs the program at every call in order to determine which event to add. This can be avoided if we make NextEvent decrement pc for the non-selected threads, and ExecThread to resume execution right after* **produce**.

[71] *This is to ensure a single-step temporary inconsistency as described in §5.5.*

particular, this means that if G contains an event corresponding to the read-exclusive event of a successful RMW instruction without its matching write-exclusive event, then $\textsc{NextEvent}_P(G)$ will return that write-exclusive event (which is anyway immediately `po`-after it).

Putting everything together, NextEvent can be seen in algorithm 5.2. NextEvent obtains a list S of the next available event of each thread (line 2), and returns the $<_{\mathsf{next}}$-minimal event of S after adding it to the graph (line 6), or $\bot$ if S is empty (line 3).

Note that algorithm 5.2 does not take any special care for blocked threads. As soon as a thread blocks (and the corresponding `B` is added to the graph), no further instructions from that thread can be executed, as the special labels `block` and `error` do not point to valid instructions (see §2). When Visit is later called with some graph that does not contain a `B` label (e.g., due to some different exploration choice before the blocked event), the thread will be again schedulable, and other options that might not lead to the blocking will be considered.

THE EXPLORATION PROCEDURE GenMC's algorithm can be seen in algorithm 5.3. Given an input program P, Verify verifies P by calling Visit with an execution graph $G_\emptyset$ containing only the initialization event. Subsequently, Visit will enumerate all execution graphs of P in a depth-first manner, and ensure that none of them contains an `Error`, denoting a safety violation.

Let us now take a closer look at the Visit function, lying at the heart of the verification procedure. At each step, so long as the current execution graph G remains consistent according to the underlying memory model (line 4) and is not erroneous (line 5)[72], Visit extends the current graph G by calling $\textsc{NextEvent}_P(G)$.

As explained, the $\textsc{NextEvent}_P(G)$ function locates a thread that is not blocked nor finished, adds the corresponding event to G.`E`, and returns it via a (line 6), without updating G.`rf` or G.`co`. If no such thread exists, it returns $\bot$.

The next action that Visit takes, depends on a itself.

- If a is $\bot$, Visit returns (line 7).
- If a is a read, Visit needs to calculate all possible `rf` options for the newly added event. To that end, for each write w to the same-location as a (line 10), it recursively calls Visit on the graph that results if r reads from w. Any inconsistent choices will be subsequently eliminated by the consistency check on line 4 of the corresponding recursive call.
- If a is a write, as explained in the previous sections, Visit needs to examine both the case where a does not revisit any of the graph reads, and the case where a revisits some read in G.

To take care of the first case, Visit calls VisitCOs (line 13). For each possible `co`-predecessor w_p of a in G, VisitCOs will insert a immediately

[72] *Recall from §2.4 that apart from searching for* `Error` *labels, error checks might also be memory-model-specific. We give an example of such memory-model-specific error checks in §7.1.*

Algorithm 5.3 GenMC: Generic Model Checking

```
1:  procedure Verify($P$)
2:      Visit$_P(G_\emptyset)$
3:  procedure Visit$_P(G)$
4:      if $\neg\mathsf{consistent}_\mathsf{M}(G)$ then return
5:      if IsErroneous$_\mathsf{M}(G)$ then exit(“error”)
6:      switch $a \leftarrow$ NextEvent$_P(G)$ do
7:          case $a = \bot$
8:              return “Visited full execution graph $G$”
9:          case $a \in \mathsf{R}$
10:             for $w \in G.\mathsf{W}_{\mathtt{loc}(a)}$ do
11:                 Visit$_P(\mathsf{SetRF}(G, a, w))$
12:         case $a \in \mathtt{W}$
13:             VisitCOs$_P(G, a)$
14:             for $r \in G.\mathsf{R}_{\mathtt{loc}(a)}$ such that $r \notin G.\mathsf{cprefix}(a)$ do
15:                 $\mathit{Deleted} \leftarrow \{e \in G.\mathsf{E} \mid r <_G e \wedge e \notin G.\mathsf{cprefix}(a)\}$
16:                 if ShouldRevisit$(G, r, a)$ then
17:                     VisitCOs$_P(\mathsf{SetRF}(G|_{G.\mathsf{E}\setminus \mathit{Deleted}}, r, a), a)$
18:         case _
19:             Visit$_P(G)$
20: procedure VisitCOs$_P(G, a)$
21:     for $w_p \in G.\mathsf{W}_{\mathtt{loc}(a)}$ do Visit$_P(\mathsf{SetCO}(G, w_p, a))$
22: procedure ShouldRevisit$(G, r, w)$
23:     static $S \leftarrow \emptyset$
24:     $\mathit{Deleted} \leftarrow \{e \in G.\mathsf{E} \mid r <_G e \wedge e \notin G.\mathsf{cprefix}(a)\}$
25:     returnSetRF$(G|_{G.\mathsf{E}\setminus \mathit{Deleted}}, r, w) \in S$
```

after w_p in co via $\mathsf{SetCO}(G, w_p, a)$, and then call Visit on the resulting graph (line 21).

To deal with the case where a backward-revisits some read in G, Visit iterates over all same-location reads as a that do not causally precede a as candidates for a backward revisit (line 14). (Reads that causally precede a are excluded because revisiting them would create a causal cycle, which is forbidden by memory-model well-formedness.) For each candidate read r, Visit calculates the set of events that will be deleted from G if a backward-revisits r (line 15), and checks whether the graph resulting by removing the deleted events has been encountered before by calling ShouldRevisit (line 16). If so, Visit appropriately restricts G by removing the deleted events, makes r read from a, and then calls VisitCOs to explore all possible coherence positions for a in the new graph (line 17).

ShouldRevisit(G, r, w) simply checks whether the graph resulting from the backward revisit has been seen before. To that end, it maintains a set of "seen" backward revisits across explorations S (line 23). Then, after obtaining a restriction G' of G that contains the events that were added before r, as well as the causal prefix of w, it checks whether G' is in S, and if so returns false.

Finally, coming back to the switch-statement of Visit, for all other cases of events (e.g., memory fences), Visit simply initiates a recursive call (line 18), with no special care taken.

Remark 4. A more space-efficient implementation of ShouldRevisit that does not save full graphs (but rather the prefixes of backward-revisiting writes) can be found in the respective paper[73]. In §6 we present an implementation of ShouldRevisit that does not require saving.

5.8.2 *Adaptation for a Reads-From Equivalence*

Adapting algorithm 5.3 for a reads-from equivalence partitioning is straightforward. Indeed, one has to simply check consistency (line 4) using `wb` as opposed to `co`, and make VisitCOs$_P(G, w)$ boil down to a single Visit$_P(G)$ call.

[73] *"Model checking for weakly consistent libraries"* [KRV19]

6

TRUST: POLYNOMIAL MEMORY REQUIREMENTS FOR GENMC

As described in §5, GenMC is optimal: it does not explore the same graph twice, nor does it embark on fruitless explorations that are doomed to never yield a consistent execution graph. While this is satisfactory in terms of time complexity, it might lead to exponential memory consumption.

In this chapter, we present TruSt[74], a modification for GenMC's core algorithm that provides linear memory requirements in the size of the program under test. TruSt achieves that first by observing that duplicate exploration can only arise due to backward-revisiting, and then by imposing a condition on the subexplorations that perform the same backward revisit, so that a given backward revisit is only performed once across all subexplorations. The only question to be answered is what condition would be both sufficient (i.e., guarantees uniqueness of a backward revisit) and necessary (i.e., guarantees existence of the subexploration performing the backward revisit).

Uniqueness is easily obtained by observing that, given multiple subexplorations which can perform the same backward revisit, only the events affected by the backward revisit (i.e., the deleted events and the read being revisited) are different among them. Indeed, recall the w+rw+w example from §5.3. Graph ③ can occur as a result of a backward revisit from both subexploration ② and subexploration ③, and the only difference between ② and ③ are the events of thread II and $\mathtt{R}(x)$. As such, any condition differentiating the affected events among different subexplorations performing the same backward revisit would be a sufficient one.

Existence, on the other hand, is much more challenging, as it effectively boils down to ensuring that the execution that produces a given backward revisit will be explored. TruSt guarantees existence by intertwining the definition of the revisiting condition with extensibility. Specifically, TruSt's revisiting condition (called maximal extension[75]) only allows a given backward revisit if the events affected by it were added in the way prescribed by $\mathsf{f}_{\mathsf{ext}}$. In the case of w+rw+w (and assuming that $\mathsf{f}_{\mathsf{ext}}(G,e) \triangleq \max_{G.\mathtt{co}}(G.\mathtt{W}_{\mathtt{loc}(e)})$) TruSt will backward-revisit $\mathtt{R}(x)$ from graph ③ (as the $\mathtt{R}(x)$ is reading co-maximally), and not from graph ②.

In what follows, we formally describe maximal extensions and the changes they induce in GenMC's algorithm. In §6.4 we show how using maximal extensions leads to linear memory consumption.

[74] *"Truly stateless, optimal dynamic partial order reduction"* [Kok+22a]

[75] *The name stems from the fact that, for non-dependency-tracking models, adding events* co*-maximally guarantees extensibility.*

6.1 MAXIMAL EXTENSIONS

The key idea behind maximal extensions is simple. Consider the (consistent) graph G' that may occur as a result of a backward revisit of a read event r by a write event w. From prefix-closedness (see §5.2), we also know that $G'' \triangleq G' \setminus \{r, w\}$ is consistent. Starting from G'' and assuming a fixed construction order, if we add all the remaining events of the program, we can in general arrive to multiple graphs $G_1, G_2, \ldots$, depending on the way we add the remaining events (e.g., the $\mathtt{rf}$ edges of reads, etc). In principle, these are all the graphs which can lead to a backward revisit of r from w. Our goal is to both allow such revisiting in only one of these graphs, G, and also ensure that such a graph exists. We achieve this by requiring that (1) all additional events be added in the way prescribed by $\mathsf{f}_{\mathsf{ext}}$ (to guarantee consistency), and (2) no revisiting takes place while constructing G. Uniqueness follows because $\mathsf{f}_{\mathsf{ext}}$ prescribes a single way to add all events if no revisiting takes place, while existence follows from extensibility: since G'' is consistent, it is always consistent to add events in the way prescribed by $\mathsf{f}_{\mathsf{ext}}$.

Before formalizing maximal extensions, let us provide some auxiliary definitions. Given a set of events E, we say that a write event $w \in G.\mathtt{W}$ is *maximal w.r.t. E* if $w \in E$ and $\mathsf{pred}_{G.\mathtt{co}|_E}(w) = \mathsf{f}_{\mathsf{ext}}(G|_E, w)$. Similarly, a read event $r \in G.\mathtt{R}$ is *maximal w.r.t. E* if $r \in E$ and $G.\mathtt{rf}(r) = \mathsf{f}_{\mathsf{ext}}(G|_E, r)$. An event $e \in G.\mathsf{E}$ is *maximally added* before a write $w \in G.\mathtt{W}$ if e is *maximal* w.r.t. $\mathit{Prevs}_G(e, w) \triangleq \{e' \in G.\mathsf{E} \mid e' \leq_G e \vee e' \in G.\mathsf{cprefix}(w)\}$, and there does not exist $r \in \mathit{Prevs}_G(e, w)$ such that $G.\mathtt{rf}(r) = e$.

Given the above, maximal extensions can be defined as follows.

Definition 6.1.1 (Maximal Extension)**.** An execution graph G is a *maximal extension* of a potential backward revisit from $w \in G.\mathtt{W}$ to $r \in G.\mathtt{R}$ if every $e \in G.\mathsf{E}$ such that $r \leq_G e$ and $e \notin G.\mathsf{cprefix}(w)$ is added maximally before w.

The above definition closely follows the intuitive description above, so let us go through it in detail, while keeping the above explanation in mind. First, notice that maximality of an event e is checked w.r.t. the set $\mathit{Prevs}_G(e, w)$, which contains e and all events added before it, as well as those events that are causally preceding w, since the latter events will be included in the resulting graph G'. Second, notice that maximality is only required for r and all the events added after it, excluding those events that in the causal prefix of w. The reason why the causal predecessors of w are excluded is because, as explained previously, this prefix will be included in the resulting graph G'. Observe that, even though the definition technically requires maximality for w, in fact this is unimportant: since backward revisits are calculated when a write is encountered, the write has not been added to $\mathtt{co}$ yet[76]. Finally, notice that the definition of $\mathit{Prevs}_G(e, w)$ forbids backward revisits from deleted events.

[76] *The same definition would work if backward revisits were calculated after adding a write to* $\mathtt{co}$.

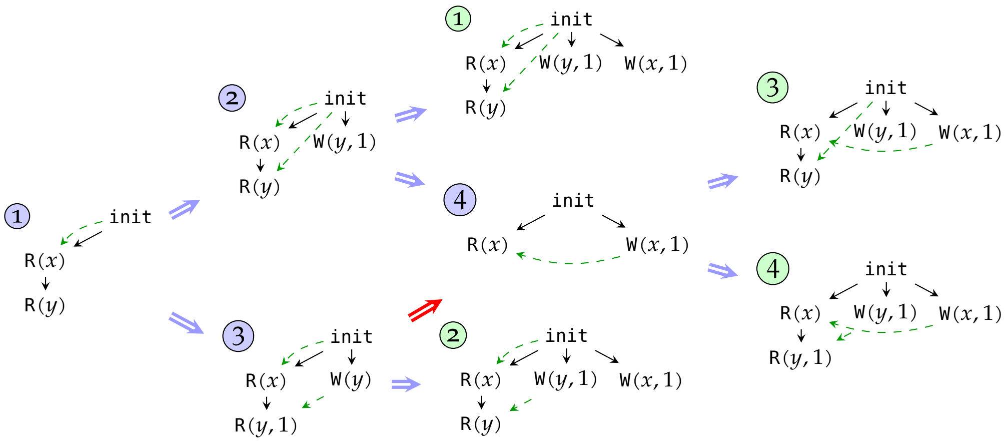


Figure 6.1: TruSt: Enumerating the execution graphs of rr+w+w

6.2 EXAMPLES

We begin with an example of how TruSt avoids performing the same backward revisit twice with the rr+w+w example below.

$$\begin{array}{l||l||l} a := x & & \\ & y := 1 & x := 1 \\ b := y & & \end{array} \qquad \text{(rr+w+w)}$$

An TruSt exploration of rr+w+w can be seen in Fig. 6.1. For convenience, let us also assume that $\mathsf{f}_{\mathsf{ext}}(G, e) \triangleq \max_{G.\mathtt{co}}(G.\mathtt{W}_{\mathtt{loc}(e)})$. As it can be seen, the backward revisit of $\mathtt{R}(x)$ from $\mathtt{W}(x, 1)$ is considered twice, in executions ② and ③. However, according to Def. 6.1.1, the backward revisit will only be performed from execution ②, since $\mathtt{R}(x)$, $\mathtt{R}(x)$ and $\mathtt{W}(y, 1)$ were all added in a maximal manner[77]. (Note that $\mathtt{R}(y)$ is not reading from $\mathtt{W}(y, 1)$, which is the co-maximal write in execution ①, but that is OK, since we only want events to be maximal when they are added.) By contrast, graph ③ is not a maximal extension of the same revisit because $\mathtt{R}(y)$ was not maximally added: it is reading from $\mathtt{W}(y, 1)$, which was added to the graph after it and does not belong in the causal prefix of $\mathtt{W}(x, 1)$.

Another way of seeing why TruSt should *not* do the revisit of $\mathtt{R}(x)$ in execution ③, is that, by doing it, it would "undo" the previous revisit of $\mathtt{R}(y)$ by $\mathtt{W}(y, 1)$. This is indeed the case: the maximal extension condition ensures that a backward revisit cannot be contained among the events that will be deleted by a subsequent backward revisit.

As we will shortly see (§6.4), the fact that TruSt avoids undoing work that it has already done, along with maximal extensions, are crucial in achieving polynomial memory requirements.

[77] *Recall that all writes are* co*-after the initializer.*

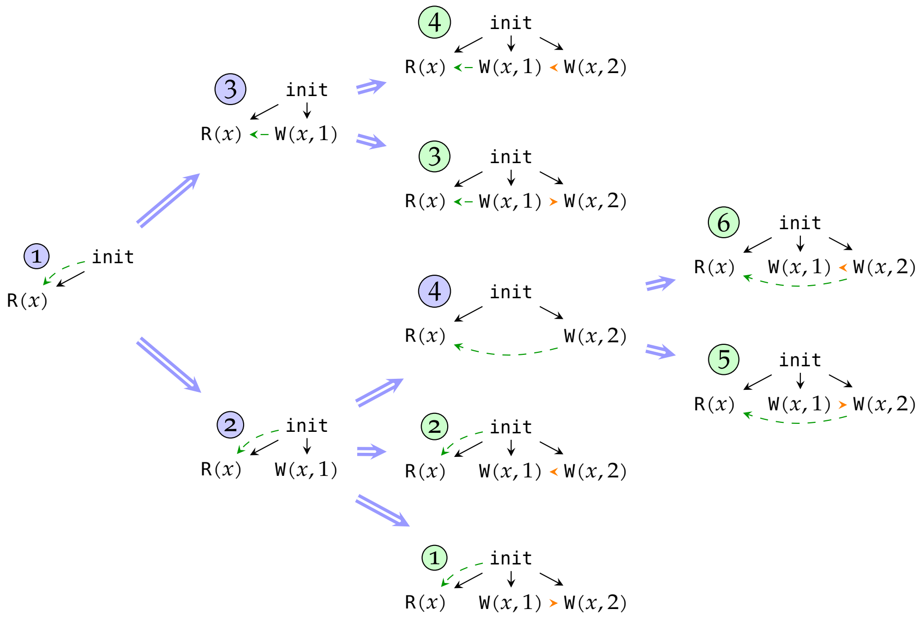


Figure 6.2: TruSt: Enumerating the execution graphs of R+W+W

Let us now move on to a different example demonstrating how TruSt enumerates the coherence-tracking graphs of the R+W+W program below:

$$a := x \;\Big\|\; x := 1 \;\Big\|\; x := 2 \qquad\qquad (\text{R+W+W})$$

This program has 6 executions under SC, which can be seen at the leaf nodes of the exploration procedure of Fig. 6.2 (assuming a left-to-right exploration by TruSt). In what follows, we again assume that $\mathsf{f}_{\mathsf{ext}}(G, e) \triangleq \max_{G.\mathsf{co}}(G.\mathsf{W}_{\mathtt{loc}(e)})$.

Focusing on the exploration, when TruSt first encounters $\mathtt{W}(x, 1)$, it can either revisit $\mathtt{R}(x)$ or not. For each of these scenarios, TruSt will initiate a recursive subexploration (graphs ② and ③). Revisiting $\mathtt{R}(x)$ is possible, as the current graph is a maximal extension of the graph resulting if $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$.

Assuming that TruSt first explores the non-revisiting case (graph ②), it will next add $\mathtt{W}(x, 2)$, in all possible co positions. When adding $\mathtt{W}(x, 2)$, TruSt also has the option of backward-revisiting $\mathtt{R}(x)$, and, since the graph forms a maximal extension, recursively explores that option too (graph ④)[78]. In that case, $\mathtt{W}(x, 1)$ is re-added to the graph, but now it cannot revisit $\mathtt{R}(x)$ because the latter is not maximally added (it has been backward-revisited by a write not in the causal prefix of $\mathtt{W}(x, 1)$).

Finally, TruSt explores the second top-level recursive call where $\mathtt{W}(x, 1)$ backward-revisits $\mathtt{R}(x)$ (graph ③). When $\mathtt{W}(x, 2)$ is re-added, it cannot revisit $\mathtt{R}(x)$, since again it is not maximally added before $\mathtt{W}(x, 2)$. TruSt

[78] *Recall that the maximality of* $\mathtt{W}(x, 2)$ *is unimportant, because backward revisits are considered before calculating its* co *placings.*

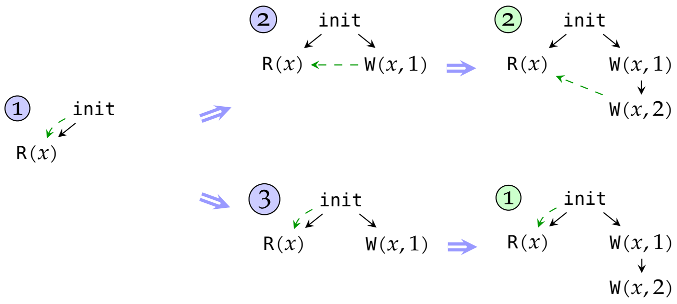


Figure 6.3: Revisiting a read multiple times is often necessary

will, however, explore all possible co placings for W$(x,2)$, thus concluding the verification of this program.

We conclude this section with a last example demonstrating an important point, namely that TruSt can backward-revisit a given read multiple times, even though backward revisits are generally preserved. Doing so is frequently necessary to obtain some outcomes.

One such case can be seen with the R+WW program below, and the exploration of Fig. 6.3.

$$a := x \;\Bigg\|\; \begin{array}{l} x := 1 \\ x := 2 \end{array} \qquad \text{(R+WW)}$$

In this example, graph ② where $a := x$ reads 2 can only be obtained if R(x) is revisited twice: R(x) is not maximally added before W$(x,2)$ in graph ③ where it is not revisited by W$(x,1)$ and keeps reading 0, thereby precluding the revisit of R(x) by W$(x,2)$.

More generally, even though maximal extensions forbid backward revisits from deleted events, they do allow backward revisits from causally related writes.

6.3 ALGORITHM

Let us now present TruSt in detail. Similarly to the algorithm of §5.8, TruSt works for both the Snasha-Snir and the reads-from equivalence (with no further changes required), as well as for dependency-tracking models. In this section, we present an overview of TruSt (§ 6.3.1), while in §6.4 we show a version of TruSt with linear memory requirements.

6.3.1 *Overview*

TruSt shares the algorithmic structure of GenMC. In fact, TruSt can be expressed in terms of algorithm 5.3, simply by redefining ShouldRevisit (see algorithm 6.1). As it can be seen, TruSt only performs a backward

Algorithm 6.1 TruSt: Backward-revisiting condition

```
1: procedure ShouldRevisit$(G, r, w)$
2:     $Affected \leftarrow \{e \in G.\mathsf{E} \mid r \leq_G e \wedge e \notin G.\mathsf{cprefix}(a)\}$
3:     return $\forall e \in Affected.$ IsMaximallyAdded$(G, e, w)$
4: procedure IsMaximallyAdded$(G, e, w)$
5:     $Prevs \leftarrow \{w' \in G.\mathsf{E} \mid w' \leq_G e \vee w' \in G.\mathsf{cprefix}(w)\}$
6:     if $\exists r \in Prevs$ such that $G.\mathtt{rf}(r) = e$ then return false
7:     if $e \in G.\mathsf{R}$ then return $G.\mathtt{rf}(e) \in Prevs \wedge \mathsf{f}_{\mathsf{ext}}(G|_{Prevs}, e) = G.\mathtt{rf}(e)$
8:     if $e \in G.\mathtt{W} \setminus \mathtt{W}^{\mathsf{excl}}$ then return $\mathrm{pred}_{G.\mathtt{co}|_{Prevs}}(e) = \mathsf{f}_{\mathsf{ext}}(G|_{Prevs}, e)$
9:     return true
```

revisit from w to r when the current graph G forms a maximal extension: all the events affected by the revisit have been added maximally (line 3).

Accordingly, IsMaximallyAdded(G, e, w) closely follows the definition of the event e being maximally added before w in G (cf. §6.1). First, it calculates the set *Prevs* of previous events (i.e., those that were added before e or that causally precede a). Next, it checks whether some other event r that has been backward-revisited by e and, if so, returns false. Then, if e is a read event, it checks that e reads from the event prescribed by $\mathsf{f}_{\mathsf{ext}}$ when it was added. If e is a write event, it checks that e itself was added in the way prescribed by $\mathsf{f}_{\mathsf{ext}}$. Note that if e is neither a read nor a write (e.g., a fence event), then the maximality check trivially succeeds.

6.3.2 *Memory Requirements*

Let us now examine why TruSt has polynomial memory requirements. As already mentioned, the key in achieving this is the fact that in the maximal extension of a given backward revisit there cannot be a read r that is going to be deleted that was backward-revisited by a write r that is also going to be deleted.

As such, since an already performed backward revisit will not be "deleted" from the graph by subsequent backward revisits, the number of events that will never be removed from the graph increases. In turn, as the number of events that can be added in a graph is bounded by the program size, so is the number of recursive calls that can be performed from a given graph.

A simple calculation gives us a space complexity bound of $O(n^3)$, where n is the size of the program: the recursion depth is at most n^2 (there are at most n backward revisits, between any pair of which up to n events may have been added) and each recursive call uses $O(n)$ space to store the execution graph.

Algorithm 6.2 TruSt: Iterative version with linear memory

```
procedure Visit_P(G)
    while true do
        $a \leftarrow$ if $\mathsf{consistent}_\mathsf{M}(G)$ then NextEvent_P(G) else $\bot$
        if $a \in \mathtt{error}$ then exit("error")
        else if $a \in \mathsf{R}$ then $\mathsf{SetRF}(G, a, \max_{<_G} G.\mathtt{W}_{\mathtt{loc}(a)})$
        else if $a \in \mathtt{W}$ then $B[a] \leftarrow a$
        else if $a = \bot$ then Backtrack(G)
```

With clever data structures and a more careful calculation, we can bring down TruSt's memory requirements to $O(n)$. Such an adaptation is presented in §6.4.

6.3.3 *Parallelization*

Another major benefit of TruSt is that it is inherently parallelizable. Indeed, as shown above, different revisits proceed in a completely disjoint manner, and they can thus be explored concurrently. In other words, TruSt's revisiting condition does not require any information not present in the current exploration.

Even though optimal DPOR algorithms have been parallelized[79], such parallelizations concerned the implementation of those algorithms, and not the algorithms themselves, as data sharing was required for revisiting. TruSt is the first optimal, memory-model-agnostic DPOR that requires absolutely no sharing among different threads.

As we show in §10.2.7, the parallelizable nature of TruSt enables it to scale extremely well in architectures offering a large number of cores.

6.4 LINEAR MEMORY REQUIREMENTS

The key idea in achieving linear memory consumption is to store a single execution graph, and to have all recursive calls update the graph *in place* when they are called, and to *roll back* their updates when they return.

Rolling back forward revisits is easy as they update only one rf or co edge. Further, by executing forward revisits for a given read in a fixed order (e.g., examining rfs in reverse-addition order), we do not need to remember any information to return to the previous state.

Rolling back backward revisits is somewhat more difficult, but can still be achieved by keeping only a constant amount of information per backward revisit. Specifically, as a read may be backward-revisited by a write from a unique configuration (the graph being a maximal extension), to get to that configuration it suffices to remove the revisited read-write pair from the graph and to keep adding events maximally until the revisiting write is reached.

[79] *"Parallel Graph-Based Stateless Model Checking"* [LS20]

Algorithm algorithm 6.2 presents an iterative version of TruSt that has linear memory consumption. For simplicity, in algorithm 6.2, we show the version that does not record co and works for the reads-from equivalence.

The algorithm clearly has linear space complexity, as it keeps only one copy of the execution graph G together with an auxiliary array B for tracking backward revisits, and does not call itself recursively[80].

Algorithm 6.2 operates in an iterative fashion in one of two modes: 1. the *forward mode*, which keeps adding events to the graph while possible; and 2. the *backtracking mode*, which changes rf edges of graph when alternative exploration options are possible, removes events (e.g., to perform a backward revisit or when all revisit options of an event have been explored), and/or restores events that were removed by a backward revisit that needs to be undone.

The forward mode corresponds to the outer **while** loop of algorithm 6.2 and is quite straightforward. As long as the graph is consistent, the graph is extended with the next available event a (line 3). If that event signifies an error, verification fails with an error message (line 4). If a is a read, its rf is set to the maximal write of the same location according to insertion order. If a is a write, we initialize its index in the B array. Otherwise, if the execution is complete (or inconsistent), we enter into the backtracking mode (line 2).

The backtracking mode corresponds to the Backtrack procedure and is a bit more subtle. It starts by selecting the maximal event a from G (line 3). If no such event exists (i.e., the graph contains only the initialization event), backtracking is complete, and so verification finishes (line 4). Now if a exists and is a read event, we have to examine whether a has any remaining forward revisiting options that were not considered. If there are further possible writes where a can read from, earlier in insertion order than the write a is currently reading (line 5), then we set a to read from the maximal such write (line 6), and go back into the forward mode (line 7).

Similarly, if a is a write event, we have to examine whether there are any (further) reads that need to be backward-revisited by a. If there are such reads (line 8), we select the latest according to insertion order, and store it in $B[a]$ (line 9). Then, if the selected read satisfies the maximal extension condition (line 11), we update its rf to read from a (line 12), restrict the graph (line 13), and go back into the forward mode (line 14).

Finally, if a does not have any remaining revisit options, we call Prev(G) (algorithm 6.4) that returns the previous execution step (line 15)[81]. If that is not a backward-revisit step, we simply remove the maximal event from G (line 16). If, however, it was a backward revisit from a graph G_p (line 18), we need to do some more work to reconstruct the correct sequence of events in G_p. For this, we follow the order of events in G for events prior to a, and the insertion order for the deleted

[80] *For algorithm 6.2, we assume that* SetRF *operates in place.*

[81] Prev(G) *performs a case analysis on the maximal event of G, and returns a tuple of the graph G_p at the previous step, along with the type of the step (revisit/non-revisit).*

Algorithm 6.3 TruSt: Iterative version (backtracking)

```
procedure Backtrack_P(G)
    while true do
        $a \leftarrow \max_{<_G} G.\mathsf{E}$
        if $a = \bot$ then exit(“Verification complete”)
        if $a \in \mathsf{R} \land \exists w \in G.\mathsf{W}_{\mathtt{loc}(a)}.\ w <_G G.\mathtt{rf}(a)$ then
            $\mathsf{SetRF}(G, a, \max_{<_G}\{w \in G.\mathsf{W}_{\mathtt{loc}(a)} \mid w <_G G.\mathtt{rf}(a)\})$
            break
        else if $a \in \mathsf{W} \land \exists r \in G.\mathsf{R}_{\mathtt{loc}(a)}.\ r <_G B[a] \land r \notin G.\mathsf{cprefix}(a)$ then
            $B[a] \leftarrow \max_{<_G}\{r \in G.\mathsf{R}_{\mathtt{loc}(a)} \mid r <_G B[a] \land r \notin G.\mathsf{cprefix}(a)\}$
            $Deleted \leftarrow \{e \in G.\mathsf{E} \mid B[a] <_G e \land e \notin G.\mathsf{cprefix}(a)\}$
            if ShouldRevisit$(G, r, a)$ then
                $\mathsf{SetRF}(G, B[a], a)$
                $G \leftarrow G \setminus Deleted$
                break
        else  switch Prev$(G)$ do
            case $\langle \_, $“non-revisit $e$”$\rangle$
                $G \leftarrow G \setminus \{e\}$
            case $\langle G_p, $“$e$ back-revisits $a$”$\rangle$
                $E_{\leq} \leftarrow G.\mathsf{E}|_{\{e \in G.\mathsf{E} \mid e <_G a\}}$
                while $d \leftarrow \min_{<_{\mathsf{next}}} (G_p.\mathsf{E} \setminus E_{\leq})$ do
                    $P \leftarrow G.\mathsf{E}|_{\{b \in G.\mathsf{E} \cap G_p.\mathsf{E} \setminus E_{\leq} \mid b \leq_G G_p.\mathtt{rf}(d)\}}$
                    $E_{\leq} \leftarrow E_{\leq} \mathbin{++} d \mathbin{++} P$
                $G \leftarrow G_p\ ;\ G.\mathsf{E} \leftarrow E_{\leq}$
```

events. Whenever a deleted event d reads from a later (in insertion order) event of G, this means that d had been backward-revisited; thus, we also add all prior (not yet added) events of G immediately after d[82].

6.5 correctness proofs

Assuming that the input program P has executions only of a bounded size, we show that the TruSt algorithm (algorithm 5.3) always terminates, and is sound, complete and optimal. Soundness ensures that if Verify(P) generates G, then G is a consistent full program execution. Completeness ensures that if G is a consistent full execution of P, then Verify(P) will generate G. Optimality ensures that TruSt generates each execution exactly once and never engages in wasteful explorations. Proofs of these results are given in full in a technical appendix[83]; we proceed with an overview[84].

[82] *We assume that $b \leq_G G.\mathtt{rf}(d)$ is false if $d \notin G.\mathsf{R}$.*

[83] *“Truly Stateless, Optimal Dynamic Partial Order Reduction (supplementary material)”* [Kok+22b]

[84] *The proofs of correctness for TruSt should be attributed to Iason Marmanis.*

Algorithm 6.4 Prev: Backward step from G to G_p

```
procedure Prev_P(G)
    $a \leftarrow \max_{<_{\mathsf{next}}} \left\{ e \in G.\mathsf{E} \;\middle|\; \nexists e'.\, e \in G.\mathsf{cprefix}(e') \right\}$
    if $a \in \mathsf{R} \wedge \langle a, G.\mathtt{rf}(a) \rangle \in <_{\mathsf{next}} \wedge \nexists b \neq a.\, G.\mathtt{rf}(a) \in G.\mathsf{cprefix}(b)$ then
        return $\langle$MaxCompletion$_P(G \backslash \{a, w\}, w)$, "$w$ back-revisits $a$"$\rangle$
    else
        return $\langle G \setminus \{a\}$, "non-revisit $a$"$\rangle$
procedure MaxCompletion_P(G, e)
    while $a \leftarrow$ NextEvent$_P(G)$ do
        if $a = e$ then return $G \setminus \{e\}$
        else if $a \in \mathtt{R}$ then SetRF$(G, a, \mathsf{f}_{\mathsf{ext}}(G, a))$
        else if $a \in \mathtt{W}$ then SetCO$(G, a, \mathsf{f}_{\mathsf{ext}}(G, a))$
```

6.5.1 *Termination*

We first show that once any write w backward-revisits some read r, it cannot be deleted in any subsequent subexploration. Suppose, by contradiction, that w gets deleted by a backward revisit of some previous read r' by a write w'. If it is $r' \leq r$, then r must be in the set of deleted events for the revisit of r', or else w would not get deleted. But r itself cannot be deleted in such a scenario because it has not been added maximally before w': it reads from the deleted event w which was added after it. Otherwise, it must be $r' > r$, but in that case w cannot be deleted because a non-deleted event (r) is reading from it.

Termination of algorithm 5.3 follows from the assumption that all executions of P are of bounded size. Since all algorithm steps except for backward revisits increase the graph size, and since writes initiating a backward revisit cannot be removed, there can only be a bounded number of backward revisits, and therefore a bounded number of algorithm steps.

6.5.2 *Soundness*

TruSt is trivially sound because events are added to the graph following the program semantics, while inconsistent executions are dropped as soon as they are reached.

6.5.3 *Completeness*

Completeness states that Verify(P) visits every consistent full graph of P.

Theorem 1 (Completeness)**.** Let G_f be a consistent full execution graph of P. Then Verify(P) calls Visit$_P(G'_f)$ for some graph $G'_f \approx G_f$.

The key idea behind the proof is that, given an execution G reached by the algorithm, we can infer the execution that immediately precedes it in the (unique) production sequence that leads to G. This observation enables us to define a procedure Prev (algorithm 6.4) that maps every non-empty consistent execution to its "previous" execution. Prev lets us take a backward step; from G to the unique execution G_p such that $\text{Visit}_P(G_p)$ immediately leads to a $\text{Visit}_P(G)$ call.

We show that repeatedly taking Prev-steps from G_f will eventually lead to the initial graph: at each point G_f's maximal event is removed or made to read from an earlier event. Next, we show that whenever a graph G is Prev-reachable from a consistent full execution and G_p is a reachable algorithm configuration such that $\text{Prev}(G) = \langle G'_p, \_\rangle$ with $G'_p \approx G_p$, then $\text{Visit}_P(G_p)$ calls $\text{Visit}_P(G')$ for some $G' \approx G$. Then, Theorem 1 follows by induction on the sequence of Prev-steps from G_f.

6.5.4 *Optimality*

Optimality consists of showing two properties: (1) that there are no duplicate explorations, and (2) that there are no fruitless explorations that are doomed to be blocked and can never lead to a full execution.

To establish the former, we first show that for every reachable algorithm configuration G, if G performs an algorithm step t and reaches configuration G', then $\text{Prev}(G') = \langle G, t\rangle$. This follows because t will either be adding the maximal event to G (non-revisiting case) or the write read by it (backward-revisit case). In either case, $\text{Prev}(G')$ will identify that step and "undo" it.

We can then easily prove that there are no duplicate explorations, in that each configuration G is reached at most once. (Assume by contradiction there are two production sequences that reach the same configuration. However, we have just shown that they must have the exact same last step, and now we have two shorter production sequences reaching the same configuration, which by induction should also agree.)

Theorem 2 (No duplicate exploration)**.** Given a graph G, $\text{Verify}(P)$ goes through at most one sequence of nested $\text{Visit}_P(\_)$ calls before calling $\text{Visit}_P(G')$ for some $G' \approx G$.

To establish the latter property, we can show that if a reachable algorithm configuration is fruitless, then it is immediately blocked. In fact, the only way a fruitless configuration could arise is by adding the read-exclusive part of an RMW event reading from a write already read by another RMW. The immediate next step will add its write-exclusive part, thereby making the graph inconsistent.

7

OPTIMIZING GENMC FOR PROGRAMMING PATTERNS

In this chapter, we present how GenMC (with or without TruSt) can be optimized for programming patterns commonly occurring in concurrent programs. Indeed, even though GenMC is optimal w.r.t. the number of explored execution graphs, this number can significantly drop if we model certain common constructs in a special manner.

Here we focus on three patterns: synchronization barriers, zero-net-effect spinloops, and `assume` statements. First (§7.1), we present BAM, a DPOR extension that achieves an exponential speedup in programs with barriers by observing that the order in which different threads meet at a barrier is irrelevant to most user programs. Next (§7.2), we present SAVer, a DPOR extension that automatically transforms spinloops to `assume` statements, as long as such loops can be (statically or dynamically) proven to be side-effect-free. We conclude this chapter (§7.3) by presenting certain heuristics than can reduce the number of blocked executions caused by blocked `assume` statements.

A high-level overview of the key ideas is presented in the beginning of the respective sections.

7.1 BAM: DPOR FOR SYNCHRONIZATION BARRIERS

Synchronization barriers[85] (as in e.g., `pthread_barrier`[86]) are synchronization primitives used to ensure that the execution of a program will continue only after all threads have reached a certain point ("a barrier").

Their usage is best understood with an example:

$$\texttt{barrier\_init}(b,N)$$
$$\begin{array}{l||c||l} m[1] := \dots & & m[N] := \dots \\ \texttt{barrier\_wait}(b) & \dots & \texttt{barrier\_wait}(b) \\ n[1] := \dots & & n[N] := \dots \end{array} \qquad (\textsc{Barrier-}N\textsc{-Sync})$$

In this program, the main thread first initializes a barrier object to N, indicating that N threads will meet together ("rendezvous") at the barrier. Each thread calculates a part of the array m, and waits for all the other threads using a `barrier_wait` call: no thread gets past `barrier_wait` until all threads have executed their respective call to `barrier_wait`. After all threads have met at the barrier, each thread continues and calculates a part of the array n, which (potentially) uses the array m that was calculated in the previous step. Such iterative

[85] *Not to be confused with memory barriers.*

[86] pthread.h man page [17]

parallel computations are common in scientific applications, e.g., simulations.

More generally, barriers are useful when we want to wait for the threads to perform some calculations before continuing. Upon continuation, all calculations performed by one thread will be visible to all other threads. In contrast to joining the threads, using barriers does not cause the threads to be terminated, but rather blocked; this can be crucial for performance reasons.

But while the usage of barriers is straightforward, verifying programs with barriers is not always so. Suppose that we want to verify the BARRIER-N-SYNC program from above using GENMC. Alas, GENMC will explore an exponential number of executions for this program, as it examines all possible orderings in which different threads arrive at the barrier[87] . Even worse, GENMC does so even though the order in which the threads rendezvous is irrelevant.

In fact, the order in which threads reach the barrier is not even observable by the user program; the only thing that *is* observable according to the `pthread_barrier` documentation[88], is whether a thread was the last one to reach the barrier. However, for the programs we are aware of, even that condition is never used.

Leveraging this insight, we will now discuss BAM (Barrier-Aware Model-checker), a DPOR extension that reconciles GENMC with barriers. BAM avoid the exploration of executions that only differ in the order in which threads execute `barrier_wait`, by treating such calls as no-ops. Since the order in which threads arrive at the barrier is unimportant, BAM can correctly model the program semantics simply by waiting for all threads to arrive at the barrier (in some order), and then synchronizing them so that all instructions executed after a rendezvous at a barrier will see the effects of all instructions executed before the rendezvous.

In what follows, we first review the interaction of DPOR and barriers (§7.1.1), and then present BAM in detail (Section 7.1.2 and 7.1.3). As our experiments confirm (see §10.2.3), BAM is exponentially faster than vanilla GENMC in programs with barriers.

7.1.1 *Barriers and DPOR*

The reason why barriers and DPOR do not work well together is that barriers *inhibit* DPOR. Existing DPOR algorithms consider `barrier_wait` calls conflicting, and thus explore an exponential number of interleavings, even for a barrier program doing the bare minimum:

$$\begin{array}{c} \texttt{barrier\_init}(b, N) \\ \texttt{barrier\_wait}(b) \parallel \ldots \parallel \texttt{barrier\_wait}(b) \end{array} \qquad (\textsc{Barrier-}N)$$

For BARRIER-N, a DPOR algorithm would explore $N!$ executions, effectively rendering DPOR a useless addition to SMC.

[87] *Barriers are typically implemented using fetch-and-increment instructions; see § 7.1.1 for more details.*

[88] pthread.h man page [17]

```
barrier_init($b$, 0) :          barrier_wait($b$) :
    $[b] := N$                      $r_N := [b+1]$
    $[b+1] := N$                    $r_0 :=$ fetch_add($b$, 1)
                                    $r_1 := [b]$
                                    assume($\frac{r_1}{r_N} > \frac{r_0}{r_N}$)
```

Figure 7.1: A toy implementation of synchronization barriers

To understand why barriers are considered conflicting operations by DPOR, however, we have to examine how barriers are implemented. Typically, barriers are implemented using condition variables or futexes: a thread executing `barrier_wait` acquires a lock, manipulates a variable indicating the number of threads that have reached the barrier, and then waits on a futex/condition variable. Such implementations, however, while standard for barrier libraries, are suboptimal for model checking: each `barrier_wait` call would boil down to many different instructions, thus unnecessarily increasing the number of different events a model checker would have to generate.

Since we are only interested in verifying programs that *use* barriers, we can get away with a much more abstract barrier implementation, such as the one in Fig. 7.1. We model each `barrier_init`(b, N) as a plain write that initializes the barrier b to 0, followed by another plain write that stores N to the (read-only) location[89] $b+1$. In turn, we model `barrier_wait`(b) as an atomic fetch-and-add instruction followed by a read and an `assume` statement. For the `barrier_wait` call, the `assume` blocks the calling thread if $\frac{r_1}{N} \leq \frac{r_0}{N}$, denoting that fewer than N threads have executed `barrier_wait`(b).

Given this implementation, it becomes clear that programs like Barrier-N lead to an exponential blowup in the state space. Since the RMW instructions all write to the same location b, they are considered conflicting, and so the model checker will examine all their $N!$ possible orderings.

What's more, in addition to these $N!$ executions, state-of-the-art DPOR implementations like GenMC will also consider an exponential number of blocked executions. To see this, consider the executions of Barrier-N for $N = 2$ under SC, using the conventional modeling of barriers of Fig. 7.1 (see Fig. 7.2). Execution graphs ① and ③ are blocked because one `assume` condition is violated. By contrast, graphs ② and ④ satisfy the `assume` conditions and are thus non-blocked. DPOR algorithms will thus have to generate at least the two non-blocked executions, though actual implementations typically generate all four (blocked and non-blocked) executions.

[89] *We assume this location is inaccessible to non-barrier operations.*

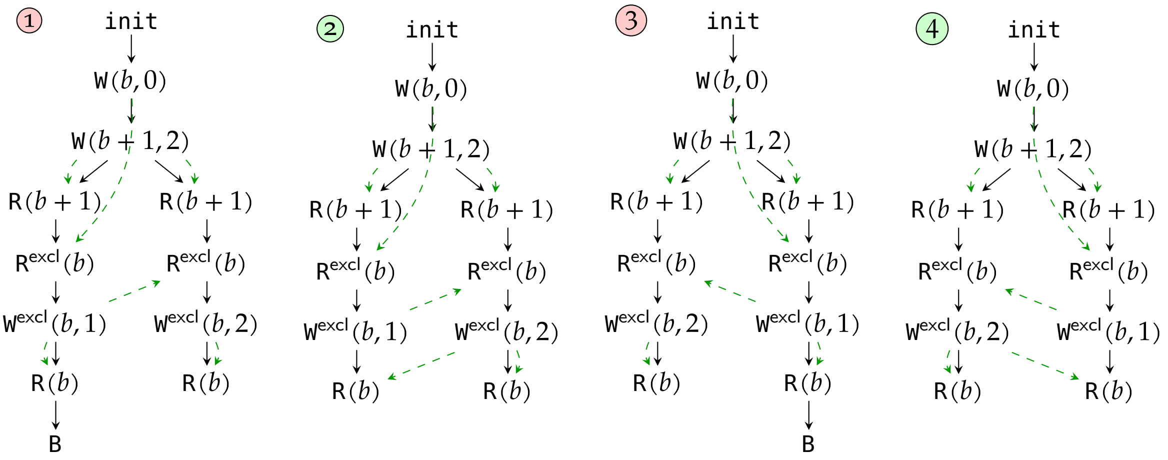


Figure 7.2: Execution graphs of Barrier-N for $N = 2$.

7.1.2 *Keeping Barriers Unordered*

We note that, although the barrier implementation of Fig. 7.1 effectively records the order in which different thread call `barrier_wait` by counting the number of threads that have joined the barrier, programs that use barriers do not care about this order. In fact, even though barrier implementations typically provide a distinct value returned by the `barrier_wait` call that resets the barrier to its initial value, the user programs we are aware of do not make use of that.

We further observe that programs using barriers typically initialize the barrier to the number of threads in the system, and so there is never a case with more parallel calls to `barrier_wait` than the barrier's initial value. Intuitively, this is because the standard scenario for barrier synchronization is to arrange a rendezvous between all threads participating in a parallel computation. With that in mind, it does not really make sense to initialize a barrier with a value smaller than the number of threads calling `barrier_wait`, as that would imply that only some threads will be unblocked after reaching the barrier, while the others will remain blocked.

The key insight behind BAM is that, for programs satisfying the two conditions described above, tracking the order between `barrier_wait` calls is unnecessary. BAM models `barrier_wait` calls as *dummy events* that are not considered conflicting, thus enabling the underlying DPOR algorithm to consider fewer executions. More specifically, when a thread executes `barrier_wait` it simply checks how many threads have reached the barrier: if not all threads have arrived, the thread blocks; otherwise all program threads unblock and continue their execution. Notice that, when all threads unblock, all the instructions before the respective `barrier_wait` statements will have been executed, thereby satisfying the fundamental guarantee provided by barriers i.e., instructions executed after the threads have rendezvoused will see the effects of the instructions executed before the rendezvous.

Let us now make the above idea formal in the framework of §2.

First of all, we assume that the underlying model defines an `hb` relation (see §2.3), that encompasses barrier-induced synchronization.

Then, to model barriers, we extend the definition of events (Def. 2.2.1) to allow for a new kind of label modeling calls to the `barrier_wait` operation:

- Barrier-wait label: $\texttt{BW}(l)$ where $l \in \mathsf{Loc}$ is the barrier location accessed.

We write $G.\texttt{BW}$ for all the barrier events of an execution graph G. Barrier events do not participate in the `rf` relation of execution graphs.

Keeping barriers unordered by `rf` achieves an exponential reduction in the number of execution graphs of programs like Barrier-N, as all four graphs of Fig. 7.2 would correspond to the single execution graph of Fig. 7.3.

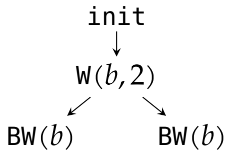


Figure 7.3: Unordered barriers: a single graph for Barrier-N

Treating barrier events as dummy events is inadequate because the `barrier_wait` calls also provide some synchronization guarantees. Specifically, every event `po`-before a barrier call is guaranteed to happen before every event `po`-after a barrier call in the same rendezvous. Recall the Barrier-N-Sync program from §7.1:

$$\begin{array}{l||c||l} m[1] := \ldots & & m[N] := \ldots \\ \texttt{barrier\_wait}(b) & \ldots & \texttt{barrier\_wait}(b) \\ n[1] := \ldots & & n[N] := \ldots \end{array} \qquad (\textsc{Barrier-}N\textsc{-Sync})$$

Here, merely treating `BW` events as dummy events is unsound. As `BW` events do not contribute to `hb` between different threads, each thread will only see its own calculation of a single part of m. By contrast, had we used the conventional barrier representation, the `rf` edges across threads would ensure that the calculation of m is visible when n is calculated.

To solve this problem, we extend the definition of execution graphs (Def. 2.2.2) with a new component:

- a partial equivalence relation $G.$`sbr`, called *same-barrier-round*, that relates barrier events that synchronize with each other in a rendezvous. Events related by $G.$`sbr` act on the same (barrier) location.

We will use the `sbr` relation to enforce synchronization between the events executed before the threads meet at the barrier, and the events executed after the rendezvous at the barrier. But before presenting how barrier synchronization works, we assume two basic conditions about the `sbr` relation.

Given a graph G and a barrier location b initialized with value N (i.e., there is a unique write $w \in G.\mathsf{E}$ such that $\texttt{lab}(w) = \texttt{W}(b, N)$, and that

$\langle w, n\rangle \in G.\texttt{hb}$, for all $n \in G.\texttt{BW}_b$), we further require that $G.\texttt{sbr}$ satisfy the following conditions:

$$|G.\texttt{BW}_b \setminus \mathit{dom}(G.\texttt{sbr})| < N \qquad (\textsc{sbr-must-meet})$$

$$\forall e \in G.\texttt{BW}_b.\ |\mathsf{succ}_{G.\texttt{sbr}}(e)| = N \ \vee\ \mathsf{succ}_{G.\texttt{sbr}}(e) = \mathsf{succ}_{G.\texttt{po}}(e) = \emptyset \qquad (\textsc{sbr-block})$$

The SBR-MUST-MEET condition captures the basic guarantee provided by the barrier implementation that once N `barrier_wait` calls are issued, then they will meet in a rendezvous round. A consistent graph can therefore contain at most $N-1$ barrier calls that do not belong to any barrier round.

The purpose of the SBR-BLOCK condition is twofold. First, it dictates that exactly N calls to `barrier_wait` participate in the same barrier round. That is, each event e either belongs in the same round with N events or does not have any events in the same round. Second, it dictates that no thread is allowed past a `barrier_wait` call before all threads rendezvous at the barrier. In other words, if an event does not participate in a (full) barrier round, it is blocked and has no po-successors in the graph. This condition renders graphs like the in Fig. 7.4 for BARRIER-N-SYNC and $N = 2$ invalid.

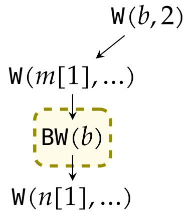


Figure 7.4: An invalid graph for BARRIER-N-SYNC

As soon as all threads reach the barrier, all corresponding barrier events become part of `sbr`, and events past the barrier may be added.

We next discuss how barrier synchronization contributes to the happens-before (`hb`) relation. We extend the (model-specific) definition of `hb` with `sbr`; `po` and `po`; `sbr`. That is, a barrier happens before the po-successors of any barriers it synchronizes with and after their po-predecessors. Since `hb` is transitive, this means that all events that are po-before a given barrier round happen before all events that are po-after the same barrier round. For example, for the BARRIER-N-SYNC program (cf. Fig. 7.5), all events po-after the highlighted barrier round will also be hb-after the events that are po-before the highlighted barrier round.

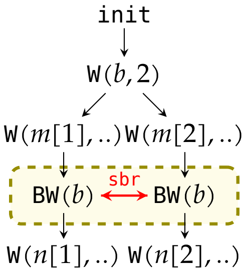


Figure 7.5: BAM: Execution graph of BARRIER-N-SYNC for $N = 2$.

Synchronization ensures that the `barrier_wait` events related by `sbr` belong to the same barrier round. To see how this is achieved, consider the program below where two threads rendezvous at a barrier twice:

$$\begin{array}{c|c} \multicolumn{2}{c}{\texttt{barrier\_init}(b, N)} \\ \texttt{barrier\_wait}(b) & \texttt{barrier\_wait}(b) \\ \texttt{barrier\_wait}(b) & \texttt{barrier\_wait}(b) \end{array} \qquad (\textsc{Barrier2-}N)$$

For this example, graphs like the one in Fig. 7.6, where `sbr` includes `barrier_wait` events from different rounds of the same barrier acquisition, are invalid.

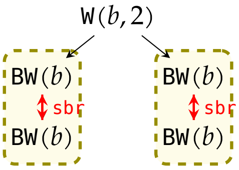


Figure 7.6: An invalid `sbr` relation for BARRIER2-N

The reason why this graph is invalid, is that $G.\texttt{sbr}; G.\texttt{po}$ is included in $G.\texttt{hb}$. This condition implies that, e.g., the second barrier event of thread I is hb-before itself (since we can take an `sbr`; `po` step), which contradicts the fact that `hb` is a strict partial order.

Algorithm 7.1 Adaptation of NextEvent for BAM

```
procedure NextEvent_P(G)
    ...
    $a \leftarrow \min_{<_{\mathsf{next}}} \{a \in S \mid \max_{G.\mathsf{po}}(G.\mathtt{E}_{\mathtt{tid}(a)}) \notin G.\mathtt{BW} \setminus G.\mathtt{sbr}\}$
    ...
procedure ExecInstruction$(G, \Phi, t, n, i)$
    switch $i$ do
        case $i \equiv \mathtt{barrier\_wait}(e)$
            Gen$(G, \langle t, n+1, \mathtt{BW}(\Phi(e)) \rangle)$
        ...
```

Finally, let us end this section by formalizing the conditions under which BAM can be used (see §7.1.2). These are expressed by the notion of barrier well-formedness, as described below.

Definition 7.1.1 (Barrier Well-formedness)**.** An execution graph G is *barrier-well-formed* on a barrier location b if $G.\mathtt{BW}_b = \emptyset$ or if the following hold.

1. There is a unique plain write event $w_0 \in G.\mathtt{W} \setminus G.\mathtt{W}^{\mathsf{excl}} \setminus \{\mathtt{init}\}$ with $\mathtt{loc}(w_0) = b$.
2. w_0 is $\mathtt{hb}$-before all $\mathtt{BW}_b$ events: $\langle w_0, e \rangle \in G.\mathtt{hb}$ for all $e \in G.\mathtt{BW}_b$.
3. For all $S \subseteq G.\mathtt{BW}_b$ with $|S| > G.\mathtt{val}(w_0)$, there exist $e, e' \in S$ s.t. $\langle e, e' \rangle \in G.\mathtt{hb}$.

Barrier well-formedness[90] ensures that there is a unique initializing write for each barrier location, and that no more threads than the barrier's initializing value call `barrier_wait` concurrently. Note that the latter precludes the usage of BAM in programs like the following:

$$\begin{array}{c} \mathtt{barrier\_init}(b,2) \\ \mathtt{barrier\_wait}(b) \parallel \mathtt{barrier\_wait}(b) \parallel \mathtt{barrier\_wait}(b) \end{array}$$

That said, as already mentioned, we do not expect such programs to show up often in practice, as they are built on the (not very useful) premise that some subset of the threads meeting at the barrier will continue past the barriers, while the rest will remain blocked.

7.1.3 *Algorithm*

We now explain how GenMC can be extended to accommodate BAM. The changes required are shown in algorithm 7.2.

A first modification that we need to perform is to change NextEvent. As can be seen in algorithm 7.1, NextEvent$_P(G)$ considers a thread

[90] *Checked as part of IsErroneous; see §7.1.3.*

Algorithm 7.2 Adaptation of algorithm 5.3 for BAM

```
1:  procedure $\textsc{Visit}_P(P, G)$
2:      ...
19:     case $a \in G.\mathtt{BW}$
20:         $N \leftarrow G.\mathtt{val}(w)$ where $w \in G.\mathtt{W}_{\mathtt{loc}(a)} \setminus \{\mathtt{init}\}$
21:         $S \leftarrow G.\mathtt{BW}_{\mathtt{loc}(a)} \setminus dom(G.\mathtt{sbr})$
22:         if $|S| = N$ then $G.\mathtt{sbr} \leftarrow G.\mathtt{sbr} \cup \{\langle e, e' \rangle \mid e \in S, e' \in S\}$
23:         $\textsc{Visit}_P(G)$
24:     ...
```

blocked if it contains a barrier event that is not in the domain of $G.\mathtt{sbr}$. (By construction, such events are po-maximal.)

A second modification is that IsErroneous now needs to report an error if the graph is not barrier-well-formed.

All other barrier-related changes are done when handling BW events (i.e., a new case is added). If a is a barrier-wait event, BAM-specific code takes over. First, BAM finds this barrier initializing value N (line 20). Well-formed programs contain a unique initialization of barrier, and so their execution graphs have a unique write event w to each barrier location. Then, BAM collects in the set S all barrier events to the same location as a that are not related by $G.\mathtt{sbr}$ (line 21). This set contains a as well as all blocked events to the same location. If the number of such events is N, then they form a rendezvous and are thus added to $G.\mathtt{sbr}$, which has the effect of unblocking the waiting threads (line 22). Subsequently, Visit recursively calls itself.

As can be seen, BAM can be seamlessly integrated into GenMC. The additional work performed—a linear scan over the graph—does not incur any overhead as it is dominated by the memory-model consistency checks.

7.2 SAVER: DPOR FOR SPINLOOPS

As explained in §5, the key design choice that makes DPOR scalable is that it does not record the set of all visited program states. The downside of this choice, however, is that DPOR struggles with spinloops, i.e., loops that continuously read a shared variable until some condition holds: with no set of visited program states, DPOR cannot distinguish loop iterations that make progress from those that return to the same state. To make matters even worse, such loops are ubiquitous in real-world concurrent programs, whether lock-based or lock-free.

Consequently, spinloops typically have to be *bounded*. Since bounding generally sacrifices the soundness of the verification, one would like to use fairly large loop bounds to be confident enough that the program verified is correct. Doing so, however, is practically infeasible. A loop bound of $N \geq 2$ typically leads to an exponential blowup in the state space, since the model checker explores the possibility of each spinloop failing 0, 1, ..., $N-1$ times and, for each failure, all possible stores from which the spinloop load(s) can read.

To avoid the blowup, the solution is to use a bound of $N = 1$. So far, this is typically done manually by rewriting the program to use `assume` statements (a.k.a. `await`), special verifier commands that block the execution of the relevant thread when the condition of the `assume` is violated.

We now discuss SAVer, a DPOR extension that determines *conditions* under which it is sound to do such conversions automatically, and reduces spinloops that satisfy these conditions to a single iteration. The key idea behind SAVer is to operate at the level of reduced control flow graphs, obtained by merging bisimilar nodes. Whenever a spinloop cannot be shown to be side-effect-free statically, SAVer checks that the reduced spinloop iterations have a zero net effect (in particular, that the context does not observe any of their effects) *dynamically*, and if the check fails, it rolls back the transformation.

In what follows, we review the challenges that spinloops pose for DPOR (§7.2.1), and then how representing programs as control flow graphs (§2.1) helps SAVer in exponentially improving DPOR (Sections 7.2.3 to 7.2.7).

7.2.1 *Spinloops and DPOR*

Automatically bounding loops to a single iteration is challenging for a couple of reasons.

First, spinloops cannot be adequately detected by a simple syntactic criterion. Since programming languages have many ways of creating spinloops (e.g., while loops, repeat-until loops, for-loops, goto statements), their detection is best done after converting each program thread into a *control-flow graph* (CFG). However, even there, simply removing the CFG backedges for side-effect-free loops (i.e., loops with no stores to global variables or to local variables that are live at the loop header) is insufficient, as illustrated by the program below.

$$\begin{array}{l||l} \texttt{do}\quad a := x & b := x \\ \texttt{while}\ (a \neq 0) & \texttt{while}\ (b \neq 0)\ b := x \end{array} \qquad \text{(LOOP-PEEL)}$$

While the loop in thread I can be easily bounded by converting it into $a := x$;`assume`$(a = 0)$, the one in thread II cannot because b is "live" at the header of the loop (its value is used in the loop).

Second, some spinloops may have side-effects, but these either do not occur on all their iterations or are never observed by the other threads (e.g., writing to a global variable that is not concurrently read) or cancel each other out (e.g., incrementing and then decrementing a variable, acquiring and releasing a lock). As an example of the latter kind, consider the following *zero-net-effect* (ZNE) spinloops extracted from a lock implementation.

```
while ($true$)                  ‖ while ($true$)
    $a$ := fetch_add($x$, 1)    ‖     $b$ := fetch_add($x$, 1)
    if ($a = 0$) break          ‖     if ($b = 0$) break        (INC-DEC-SPIN)
    fetch_add($x$, $-1$)        ‖     fetch_add($x$, $-1$)
// critical section             ‖ // critical section
fetch_add($x$, $-1$)            ‖ fetch_add($x$, $-1$)
```

Each thread tries to acquire the lock by incrementing x. If the lock was already taken, it decrements x and tries again. The lock is finally released by decrementing x. Since each decrement cancels out the previous increment, we would like to avoid considering loop iterations with a decrement, i.e., unsuccessful lock acquisition attempts. The soundness of doing so depends on the context. If, for instance, there is another thread repeatedly reading x, it may observe the value of x flickering, which cannot happen if we bound the ZNE loops to a single iteration. Similarly, if another thread writes to x concurrently, the loop may no longer have a zero net effect, rendering the transformation unsound.

As we will shortly see, SAVER addresses these challenges through a combination of some static and dynamic checks.

7.2.2 *Control Flow Graphs*

The static transformations that SAVER performs heavily rely on a control-flow-graph (CFG) representation.

A program P in written in the language of §2.1 can be alternatively expressed as a top-level parallel composition of threads, each of which is modeled as a control-flow graph (CFG). A CFG is a directed graph whose nodes are program labels and whose edges are labeled with instructions of the grammar presented in §2.1[91].

We assume that input programs are deterministic in that each node n either has at most one successor (for standard program statements), or it has two successors labeled with assume(e) and assume($\neg e$) respectively (for conditionals and loops). To ease the presentation, we also consider RMWs as single instructions (instead of splitting them to their constituent parts).

As an example, Fig. 7.7 shows the CFGs for the two threads of the LOOP-PEEL program from §7.2.1. The loops generate cycles in the CFGs, and the conditional tests (whether to execute another loop iteration or to exit the loop) generate the edges labeled with assume statements.

Let us now provide some more definitions on CFGs.

A *path* π in a CFG is an alternating sequence of nodes and instructions corresponding to edges in the CFG, starting and ending with a

[91] *Technically, these directed graphs are control flow automata but in this thesis we use the term CFG.*

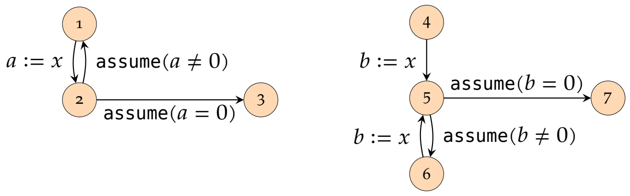


Figure 7.7: CFGs for the two threads of LOOP-PEEL.

node. That is, π is of the form $n_1 i_1 n_2 i_2 n_3 \dots n_{k-1} i_{k-1} n_k$ where (n_j, i_j, n_{j+1}) is an edge in the CFG for all $1 \leq j < k$. As it is common in the literature, we are primarily interested in *simple paths*, which do not visit the same node twice, except possibly by their last node. A (simple) path is *cyclic* if it starts and ends with the same node, while a *lasso* path is one whose end node is one of its intermediate nodes. We write $|\pi|$ to denote the length of the path (i.e., the number of edges it contains), and $\pi(k)$ to project the k^{th} node and/or instruction of the path.

We say that node *a dominates b* if all paths from the entry node of the CFG to b contain a. Given a path π in a CFG, we say that a node h of π is its *header* if it dominates all nodes in π. By definition, paths can have at most one header; in the case of reducible graphs, every cyclic path has a header. For example, in Fig. 7.7, nodes 1 and 5 are the headers of the two cyclic paths, respectively.

A loopy path is a simple path that starts and ends at its header. Formally, a simple path π is called a *loopy path* of an edge $n \rightarrow h$ if $\pi(1) = \pi(|\pi|) = h$ and $\pi(|\pi| - 1) = n$ and h dominates all nodes in π (i.e., h is a header of π).

7.2.3 *Effect-Free Spinloops*

Effect-free loop iterations that do not exit the loop are almost unobservable: they do not affect the set of reachable program states, and so can be ignored when verifying safety properties of a program. (We note that for liveness properties, effect-free loop iterations cannot be discarded that simply. An infinite sequence of such effect-free iterations, unless prevented by some fairness assumption about the program's semantics, yields a non-terminating run of the program.)

What remains to be clarified is what exactly constitutes an effect-free loop iteration. Clearly, the iteration should not be writing to a global variable, as otherwise other threads may be able to observe whether the iteration took place or not. Similarly, it should also not be assigning to any local registers that could affect the subsequent execution of the thread itself, i.e., to any variables that are *live* at the header of the loop. Assigning to a dead variable is harmless because, by definition, it does not affect the subsequent execution of the thread, even if technically it might reach a slightly different local state (differing only in the values of dead variables).

```
while ($true$)
    $h := head$
    $t := tail$
    $n := next[h]$
    $h' := head$
    if ($h \neq h'$) continue
    if ($h = t$)
        if ($n$) break
        CAS($tail, t, n$)
    else
        $b :=$ CAS($head, h, n$)
        if ($b$) break
```

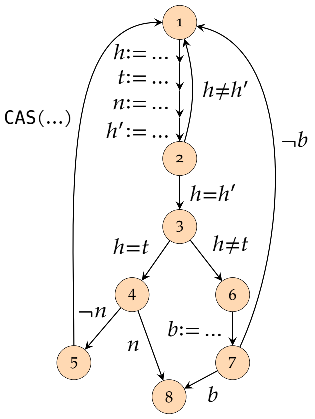


Figure 7.8: Simplified dequeu from ms-queue and its CFG[92]

We note that spinloops need to be effect-free only along looping paths—they may well have side-effects on paths exiting the loop. This is frequently the case for CAS-loops, such as the following implementation of an atomic increment:

```
do
    $a := x$
    $success :=$ CAS($x, a, a + 1$)                    (CAS-LOOP)
while ($\neg success$)
```

Here, even though the loop contains a CAS, which is generally an effectful instruction, along the looping path, the CAS fails, and so the path is effect-free.

We also note that loops often have multiple looping paths, only some of which are effect-free. Consider, for instance, the **while** loop in Fig. 7.8, which is extracted from the ms-queue benchmark of §10.2.4. It contains three loopy paths. The first (through the **continue** statement) is trivially effect-free because it contains only loads and assignments to dead variables. (All local variables are dead at the loop header.) The second path (when $h = t$) can have side-effects—the CAS to *tail*. The third path (when $h \neq t$) is again effect-free because whenever its CAS succeeds, the function returns.

Let us now make these intuitions more formal. A path π is *pure* if it either contains no store instructions or, if it contains any, all of them are failed CASes. That is, whenever $\pi(i)$ is a store instruction, then it is of the form $r :=$ CAS(x, e_1, e_2) and there is $i < j < |\pi|$ such that $\pi(j) =$ assume$(\neg r)$ and for all $i < k < j$, $\pi(k)$ does not assign to r.

Pure paths do not affect the global state, but can affect the local state. A loopy path does not affect the local state if it always reaches the same local state it started from. A simple approximation to reaching the same state is for the path to not assign to any variable that is live at its header. Putting these conditions together, an *effect-free spinloop* is a pure loopy path that does not assign to any variable live at its header. Formally:

Definition 7.2.1. A CFG edge $n \to h$ is an *effect-free spinloop backedge* if every loopy path of $n \to h$ is pure and assigns only to registers dead at h.

The *spin-assume transformation* removes all effect-free spinloop backedges from the CFG. Returning to the example in Fig. 7.7, the edge $2 \to 1$ is an effect-free spinloop backedge; removing it transforms thread I of LOOP-PEEL into $a := x$;`assume`$(a = 0)$. By contrast, the backedge of thread II ($6 \to 5$) is not effect-free and so the spin-assume transformation does not affect thread II.

7.2.4 *Transforming Loops into Effect-Free Spinloops*

While the spin-assume transformation defined in the previous section can detect typical cases of **do-while** spinloops, it does not apply to **while** loops that have a non-trivial condition.

The main problem is that the registers used to evaluate the condition are live at the loop header, and so any loop iterations that update these registers are deemed effectful. As a simple example, consider the spinloop of thread II of LOOP-PEEL from §7.2.1: register b is live at the beginning of the loop, and so the body of the loop ($b := x$) is effectful. (Formally, in the CFG of Fig. 7.7, register b is live at node 5—the loop header.)

One simple way to resolve this problem is to apply a compiler transformation called *loop rotation*, which moves the loop exit checks to the end of the loop. Applying loop rotation transforms thread II of LOOP-PEEL as follows:

$$\begin{array}{lcl} b := x & & b := x \\ \textbf{while}\ (b \neq 0) & \rightsquigarrow & \textbf{if}\ (b \neq 0) \\ \quad b := x & & \quad \textbf{do}\ \ b := x\ \textbf{while}\ (b \neq 0) \end{array}$$

The transformed loop can be bounded with the spin-assume transformation yielding executions with at most two loads of x. We note that this bounding outcome is suboptimal, since thread I of LOOP-PEEL is bounded with a single load of x.

A better approach for this example is to exploit *bisimilarity* among CFG nodes. Two nodes are bisimilar if they produce the exact same computations, i.e., if their outgoing edges can be matched 1-to-1 in a way that every two matched edges are labeled with the same instruction and lead to bisimilar nodes. Bisimilarity can be computed as a

[92] *head, next, and tail are global variables, while b, h, h', n, and t are local registers*

greatest fixed point, starting with the identity relation (i.e., each node being bisimilar to itself) and adding pairs of nodes whenever they have matching outgoing edges to nodes already calculated to be bisimilar. For example, in Fig. 7.7, nodes 4 and 6 are bisimilar because they both have only one outgoing edge labeled with the same instruction ($b := x$) and leading to the same node (5).

Having detected that two (distinct) nodes a and b are bisimilar, we can then merge them into one node by redirecting b's incoming edges to a and deleting node b. For example, merging nodes 4 and 6 of Fig. 7.7 would add an edge from 5 to 4 with label $\texttt{assume}(b \neq 0)$, and remove node 6. Effectively, this transformation converts thread II of LOOP-PEEL to a **do-while** loop analogous to that in its first thread, which makes the spin-assume transformation applicable.

We note that merging bisimilar nodes is not always strictly better than loop rotation. There are cases where loop rotation (or a similar transformation called *jump threading*) can transform a loop into the **do-while** form, but no two distinct bisimilar nodes exist. Such cases frequently arise with **CAS** loops like the following.

```
success := false
while (¬success)                          (CAS-LOOP2)
    a := x
    success := CAS(x, a, a + 1)
```

Here, the spin-assume transformation is not directly applicable to CAS-LOOP2 because *success* is live at the loop header and is updated by the loop body. Loop rotation and/or jump threading, followed by dead assignment elimination, convert this program to CAS-LOOP, which can by handled by the spin-assume transformation. By contrast, merging bisimilar nodes does not change the program, since the program does not contain the same instruction twice.

7.2.5 *Potentially Effect-Free Spinloops*

The spin-assume transformation as described in § 7.2.3 uses a completely static definition of purity. If a CAS along a CFG path cannot be determined to always fail, the path is deemed effectful. This is, however, suboptimal for two reasons.

First, using a static purity definition prevents us from transforming paths that are pure only under certain contexts. For instance, consider the thread in Fig. 7.9, and assume that it is running as part of a program that only writes the value 0 to z (this might not be inferable statically).

In this case, the (only) loopy path of this thread will not be deemed pure (as the CAS is not followed by an $\texttt{assume}(\neg b)$ statement), even though it will never produce observable effects in its running context as a will always be 0.

```
do
    $a := z$
    $b :=$ CAS$(x, 0, 1)$
while $(a = b)$
```

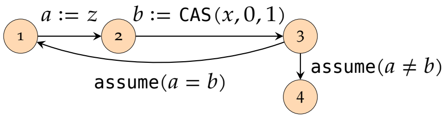


Figure 7.9: Example where static purity inference is impossible

Second, in cases where a loopy path contains a CAS that *does* have observable effects, it is wasteful to explore executions where such a CAS fails. To see this, consider again the dequeue operation of the `ms-queue` example in Fig. 7.8. As explained in §7.2.3, the second loopy path of this operation is not pure, as it potentially has side-effects. Still, it does not make sense to consider iterations where the CAS of this path fails, as they both do not contribute to the loop exiting, and they produce no observable side-effects.

Leveraging the insights above, we say that a CFG backedge $n \rightarrow h$ is a *potentially effect-free spinloop backedge* if every loopy path of $n \rightarrow h$ assigns only to registers dead at h. The *dynamic-spin-assume transformation* marks all potentially effect-free spinloop backedges with a dynamic purity check. Whenever the NextEvent$_P(G)$ function of algorithm 5.3 encounters such a check, it validates whether G contains any write event originating from the respective loop iteration and, if not, it returns a `B` event, thereby blocking the execution of the respective thread. Otherwise, if the loop iteration did generate a write event, NextEvent$_P(G)$ proceeds with the next event.

In fact, the dynamic purity check described above can be relaxed even further: SAVer allows loop iterations to contain write events, as long as these only affect memory locations that are not reachable by other threads[93]. In turn, this proves very useful in cases where some initialization writes need to take place as part of a loop.

```
$n :=$ new-node()
$n.value := 42$
do
    $s := stack$
    $n.next := s$
    $b :=$ CAS$(stack, s, n)$
while $(\neg b)$
```

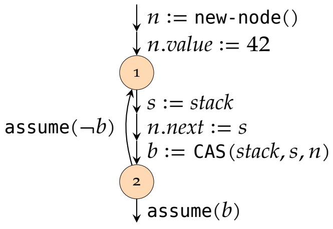


Figure 7.10: Simplified push from `treiber-stack` and its CFG[94]

To see an example of this, consider the push operation of the `treiber-stack` benchmark (cf. Fig. 7.10). First, a node to be inserted to the stack is created, but this node cannot be initialized fully: its *next* field needs to point to the existing top of the stack, but the stack top might change

[93] *That is, these writes do not have any* `rf` *edges toward other threads.*

[94] *stack is a global variable, while b, n, and s are registers*

between the time it is read, and the time the node is created. Thus, the push operation first reads the stack, sets it as the node's *next*, and then tries to atomically replace the stack with the newly created node. If the replacement succeeds, the operation exits; otherwise, it tries again. Notice, however, that, as long as the replacement CAS does not succeed, the store to the node's *next* remains unobserved by the other threads. Thus, it is safe to consider failed CAS loop iterations as effect-free, and block their exploration.

Formally, we assume that the program is annotated such that each header loop first executes a `spin_begin`(n), where $n \in \mathsf{Lid} \triangleq \mathbb{N}$ is a unique identifier for each loop. We also extend the definition of events (Def. 2.2.1) with a new label type `spin` that corresponds to the beginning of a spinloop iteration:

- Spin-begin label: `spin`(n), where $n \in \mathsf{Lid}$ is loop's unique identifier

When $\textsc{NextEvent}_P(G)$ encounters the next instruction after a `spin_begin`(n) statement, this means that a spinloop iteration has been completed[95]. If the iteration was effect free, then $\textsc{NextEvent}_P(G)$ can safely block this thread, until some other thread reads from some write in the blocked loop iteration. The full algorithm is provided in §7.2.7.

As a final remark, we observe that validating effect-free loops dynamically makes SAVer resilient to more aggressive loop rotation passes that convert loops to a canonical form containing a single backedge (see §9.2.2.2).

7.2.6 *Zero-Net-Effect Spinloops*

Let us now consider the more challenging case of *zero-net-effect* (ZNE) loops. Recall that these are spinloop iterations that do have side-effects but (1) whose side-effects cancel each other out, and (2) whose intermediate effects are not observed by other threads. While condition (1) can be checked pretty well statically, condition (2) has to be checked dynamically. In the discussion below, we focus on ZNE loops that arise because of an atomic increment being followed by an atomic decrement of the same location and value.

A decrement instruction at node k is a *canceling decrement* in a loop h if all of h's loopy paths that contain node k also contain a prior opposite increment instruction, and the paths are effect-free modulo two instructions. More formally:

Definition 7.2.2. A node k in a (minimal) CFG cycle with header h is a *canceling decrement* if it has a (unique) outgoing edge of the form r_1 := `fetch_add`$(x, -n)$, and for every loopy path π of h such that $\pi(i) = k$ for some $1 < i < |\pi|$, there exists $j < i$ such that $\pi(j) = r_2$:= `fetch_add`(x, n) for some r_2, and replacing the instructions at

[95] *Assuming this is not the first* `spin_begin` *of loop* n.

$\pi(i)$ and $\pi(j)$ with plain assignments to r_1 and r_2 yields an effect-free path.

SAVer's *spin-zne* transformation annotates all canceling decrements so that when NextEvent$_p(G)$ encounters them for the first time, it generates a special `zne` event and blocks the thread instead of generating a read event followed by a write event.

Formally, we extend the definition of events (Def. 2.2.1) to allow for a new kind of label modeling canceling decrements:

- ZNE label: `zne`(l), where l is the location accessed by the canceling decrement.

In effect, the `zne`(x) event serves as a marker for SAVer to validate that the transformation is sound. We also use the fai and zne attributes to denote events from fetch-and-add and annotated fetch-and-add instructions, respectively[96].

Let us now examine how this validation is performed. SAVer validates ZNE loops every time a new event e is added to the graph. If we use the pair $\langle w, z\rangle$ to represent a blocked ZNE loop iteration with w being the event corresponding to the increment of the ZNE loop and z being the `zne` event, the addition of e can render the reduction of the $\langle w, z\rangle$ loop unsound in one of the following two ways.

First, if e writes to the same location as w, it can be ordered (in coherence) between w and the blocked decrement (after z), and so, unless e is also an atomic increment, w and its corresponding decrement will no longer cancel each other out.

Second, if e reads from w and there is already some other read event reading from w, then, in an alternate execution, it is possible for e to read from the canceling decrement instead of w, thereby observing the value of the shared variable flickering. To see this, consider the example below.

```
while ($true$)                  ||  $b := x$
  $a :=$ fetch_add($x$, 1)      ||  if ($b$)
  if ($a = 42$) break           ||    $c := x$
  fetch_add($x$, −1)            ||    assert($c$)
```

(zne-obs)

Note that the loop of thread I fulfills the conditions of a ZNE loop, and so the second `fetch_add()` will be annotated by the spin-zne transformation.

Figure 7.11 shows the execution graph arising from adding the events of thread I and then adding the read event corresponding to the $b := x$ instruction of thread II in the case it reads the incremented value of x. Next, we have to add the event corresponding to $c := x$. In this graph, the only consistent option for this event is to also read the incremented value of x, which satisfies the subsequent assertion. Yet, if we had the decrement of x instead of the `zne` event in the graph, c could also have

[96] *Annotated fetch-and-add instructions have both attributes, but we omit writing* fai *in graphs for brevity.*

read the value 0 from the decrement, and the `assert` would have failed. Thus, it is clear that concurrent reads can render the transformation of ZNE spinloops unsound.

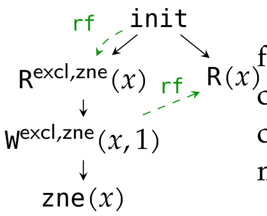

rf
init
$\mathtt{R}^{\mathsf{excl,zne}}(x)$
rf
$\mathtt{R}(x)$
$\mathtt{W}^{\mathsf{excl,zne}}(x, 1)$
$\mathtt{zne}(x)$

Figure 7.11: Graph encountered during the exploration of ZNE-OBS.

Therefore, SAVer checks whether either of these two conditions holds for any existing $\mathtt{zne}(x)$ event in the graph (where x is the location accessed by e), and if so, it removes the `zne` event(s) and unblocks the corresponding thread(s), which will eventually add the missing decrement event(s) and restore soundness.

Other cases of ZNE loops can be handled in a similar manner. For example, consider spinloops containing matching lock acquisitions and releases. In such a case, acquiring the lock acts as the increment operation and releasing the lock as the matching decrement. Statically, it therefore suffices to check that each lock release in the spinloop has its corresponding lock acquisition earlier in the same spinloop iteration. Dynamically, we simply check that no other thread accesses the lock besides by calling the acquire and release methods.

7.2.7 *Algorithm*

Algorithm 7.3 ZNE Spinloop Validity Check

```
procedure Visit_p(G)
    ...
    EnsureZNEValid(G)
    switch a ← NextEvent_p(G) do
        ...
        case $a \in \mathsf{R}$
            if $a \in \mathsf{R}^{\mathsf{zne}} \land$ IsZNEValid$(G, a)$ then
                Visit_p(SetZNE$(G, a)$)
                break
        ...

procedure IsZNEValid$(G, e)$
    return $G.\mathsf{W}_{\mathtt{loc}(e)} \subseteq G.\mathsf{W}^{\mathsf{fai}} \land \nexists w \in G.\mathsf{W}_{\mathtt{loc}(e)}.\ |[w]; G.\mathtt{rf}| > 1$

procedure EnsureZNEValid$(G)$
    let $e$ be the last event in sequence $G.\mathsf{E}$
    if $e \in G.\mathsf{W} \setminus G.\mathsf{W}^{\mathsf{fai}} \lor e \in G.\mathsf{R} \land \exists e' \neq e.\ G.\mathtt{rf}(e') = G.\mathtt{rf}(e)$ then
        $G.\mathsf{E} \leftarrow G.\mathsf{E} \setminus \mathtt{zne}_{\mathtt{loc}(e)}$
```

The algorithmic changes induced by SAVer can be seen in algorithm 7.4 and algorithm 7.3. Similarly to BAM, SAVer modifies both the NextEvent procedure, as well as Visit and the main exploration procedure.

Let us begin with the changes to Visit. These changes only concern ZNE loops, as potentially effect-free loops only require changes in NextEvent [97]. the first thing that SAVer when a new event a is added is to check whether a causes the rollback of a previously per-

[97] `spin` *events are handled by* Visit *recursively calling itself.*

Algorithm 7.4 Adaptation of NextEvent for SAVer

```
1:  procedure $\textsc{NextEvent}_P(G)$
2:      ...
4:      $a \leftarrow \min_{<_{\text{next}}}\{a \in S \mid (m \notin G.\mathtt{zne} \cup G.\mathtt{spin}) \lor$
            $(m \in G.\mathtt{spin} \lor \neg\textsc{IsEffectFree}(G, m))\}$
             where $m = \max_{G.\mathtt{po}}(G.\mathsf{E}_{\mathtt{tid}(a)})$
5:      ...
6:  procedure $\textsc{IsEffectFree}(G, a)$
7:      return $\exists b \in G.\mathtt{spin}_{\mathtt{tid}(a)} \setminus \{a\}.\ [b]; G.\mathtt{po}; [\mathtt{W}]; G.\mathtt{po}; [a] = \emptyset$
8:  procedure $\textsc{ExecInstruction}(G, \Phi, t, n, i)$
9:      switch $i$ do
10:         case $i \equiv \mathtt{spin\_begin}(e)$
11:             $\textsc{Gen}(G, \langle t, n+1, \mathtt{spin}(e)\rangle)$
12:         ...
```

formed ZNE-assume (lines 7 and 21). This check is performed by the EnsureZNEValid and IsZNEValid procedures, which closely follow the description of §7.2.6: if there is a non-FAI write or two reads on a ZNE location, the transformation is rolled back (line 21).

In the case where a is the read of a canceling decrement, and ZNE-assume is valid (line 12), SAVer used the SetZNE(G, a) function to add a ZNE label to the graph instead of a read (line 13), and calls Visit on the new graph. Formally, SetZNE(G, a) returns a new graph G' that agrees with G on all components apart from its events:

$$G'.\mathsf{E} = (G.\mathsf{E} \setminus \{a\}) +\!\!+ \langle \mathtt{tid}(a), \mathtt{idx}(a), \mathtt{zne}(\mathtt{loc}(a))\rangle$$

Now let us turn our attention to NextEvent (algorithm 7.4). SAVer has to ensure that thread blocked on ZNE or potentially effect-free loops (with no side effects) are not scheduled. To that end, SAVer only selects events in threads where the last event[98] is not a `zne` event or a `spin` event of an effect-free iteration (line 4)[99]. Since `zne` events replace reads of canceling decrements, whenever the ZNE-assume transformation is rolled back, the interpreter needs to be reset, however this is already taken care of by NextEvent, which re-interprets the program at every visit call (see §5.8).

Similarly to BAM, SAVer imposes negligible overhead over GenMC, as its transformations take place statically, before the verification procedure starts, and the dynamic conditions for purity and ZNE loops can be checked in $O(n)$ time (where n is the size of the graph), which is dominated by GenMC's existing consistency checks.

Remark 5. For these transformations to be correct (i.e., for bounding to a single iteration to be sound), $\textsc{IsErroneous}_{\mathsf{M}}$ must not contain a condition involving only one thread. For instance, if $\textsc{IsErroneous}_{\mathsf{M}}$ demands that a thread not contain four reads of x in a row, an error will

[98] *In line 4, the last event of each thread has to be checked (instead of the* `po`*-predecessor of a), because as soon as the interpreter encounters a canceling decrement it will stop the execution (see algorithm 5.1), because the read produced by the interpreter is not contained in the graph (it has been replaced by a* `zne` *event).*

[99] *For effect-free loops, writes that were allocated in within a given loop iteration can be excluded from the check in line 7.*

not be reported for a program with a single effect-free spinloop reading x.

7.3 PREVENTING BLOCKING IN DPOR

Now suppose that we have applied the transformations of the previous sections and the program contains a number of `assume` statements. Whenever GenMC encounters an `assume(e)` statement and the condition e does not hold, it blocks the current thread and continues executing the other threads, just in case one of the other threads produces a write that will revisit some read before the `assume` statement, in which case the relevant thread will be unblocked. Naturally, this can lead to exploring a large number of blocked executions.

In this section, we show how we can reduce the number of blocked executions induced by `assume` statements. There are two ideas that we can leverage. First, we can (automatically) *annotate* the loads whose results are used in an `assume` statement, and only consider values that will make the `assume` succeed. Second, we can detect that a given execution will not produce any new behaviors (e.g., because a read before a block label was not added maximally, and eagerly stop the corresponding subexploration.

We note that the ideas above are just heuristics. Even though they dramatically reduce blocking in certain cases (see §10.2.5), they are not guaranteed to eliminate it[100].

7.3.1 *Assume Annotations*

Consider the following program, where the `assume` statement only succeeds if it reads $a = 42$.

$$\begin{array}{l||l} \texttt{for}\ (i = 1;\ i \le 42;\ i{+}{+}) & a := x; \\ \quad x := i & b := a - 42 \\ & \texttt{assume}(b = 0) \end{array} \qquad (\text{ASM-ANNOT})$$

To avoid exploring the blocked executions where $1 \le a \le 41$, we can annotate the load of x with the expression $val - 42 \neq 0$, where val is a symbolic variable tied to the result of the reading x. Then, when GenMC adds the event corresponding to $a := x$, it will calculate the annotated expression (see §7.3.1.1) and discard all values that satisfy it, since they would subsequently block at the `assume` statement.

Just doing this simple check can prevent exploring a number of blocked executions, but is unfortunately generally unsound. At the very least, some care is required whenever all reads-from choices lead to blocking. In that case, GenMC cannot simply discard the execution, because it will miss any case where the read is revisited by later threads. Rather, it has to continue exploring one of the choices leading to blocking, so as to allow later writes to revisit the load.

[100] *See the approach of Kokologiannakis, Marmanis, and Vafeiadis*[KMV23] *for a better alternative when it comes to completely eliminating blocking.*

But even that modification is not enough, as we will shortly see. Consider the following program and its annotated behavior, which is possible under SC simply by running thread II before thread I, and suppose that the load of y is annotated to say that it can read only 0.

```
a := z; //reads 1       ||  y := 1;
if (a = 0)              ||  c := x; //reads 0
    x := 1;             ||  if (c = 0)
    b := y;             ||      z := 1;
    assume(b = 0);      ||
```

(ASM-SB-CYCLE)

Suppose further that GenMC considers the instructions in a different order, by first executing $y := 1$ from thread II and then the instructions in thread I. Under such a scheduling, GenMC will never produce the annotated execution.

Let us consider what happens in detail. When considering the load $a := z$, the only available reads-from value is 0 from the implicit initialization write to z. Thus, a read event is added to the current execution graph, and next a write event for $x := 1$ is added. Next comes the $b := y$ instruction, which can read two values: 0 (from the implicit initialization write) and 1 (from thread II). Reading 1 satisfies the load annotation, so, according to the description so far, we discard it, and we are left with only the execution where the value 0 is read. This satisfies the `assume` check and so thread II is complete. Next, GenMC moves to the next instruction of thread II, namely $c := x$. This has two writes it could potentially read from (the initial one and the write of thread I). Reading from the initial write, however, is inconsistent according to SC because it results in a store buffering pattern: there is no interleaving of the threads that yields $a = b = c = 0$. So, we are left with reading $c = 1$, which completes the exploration procedure.

It is worth pondering for a moment: What went wrong? Had the $b := y$ load not been annotated, or had GenMC explored the (blocking) option of reading 1, it would have explored all expected interleavings. Specifically, although getting $b = 1$ blocks thread I, the algorithm continues exploring thread II, and now getting $c = 0$ is a consistent option for the $c := x$ load (it's obtained by ordering all memory accesses of thread I before those of thread II). The algorithm then continues and generates an event for the write of z, which revisits the read of z in thread I and completes the exploration.

It is no accident that making thread I read from the $y := 1$ store allows all options for the subsequent $c := x$ read to be consistent (whereas reading from the initialization write does not). The key difference between the two writes is that $\mathtt{W}(y, 1)$ is the $\mathtt{rf}$ prescribed by $\mathsf{f}_{\mathsf{ext}}$[101], and so reading from it guarantees consistency of subsequent graph extensions, and will enable possible future backward revisits.

[101] *Assuming* $\mathsf{f}_{\mathsf{ext}}(G, e) \triangleq max_{G.\mathtt{co}} G.\mathtt{W}_{\mathtt{loc}(e)}$ *for SC.*

Put differently, if all the choices for a read lead to blocking, GenMC needs to continue exploring one option that preserves consistency, and this option is the one prescribed by $\mathsf{f}_{\mathsf{ext}}$[102].

7.3.1.1 *Annotatable Loads*

A second challenge is deciding *which* loads to annotate, especially in cases where the results of multiple loads are used in an `assume` statement. To see why this can be challenging, consider the following example:

$$a := x$$
$$b := y$$
$$\texttt{assume}(a + b = 42)$$

If we try to annotate the first load, GenMC will not be able to evaluate the annotation, as the value of b is unknown at the time when the event corresponding to $a := x$ is added. By contrast, we can annotate the second load, $b := y$ because by the time that it is executed the value of a will be known.

More generally, we define the notion of an *annotatable load* as follows.

Definition 7.3.1. A load instruction $r := x$ is *annotatable* along a simple path π if $\pi(1) = r := x$, $\pi(|\pi| - 1) = \texttt{assume}(e)$, and for all $1 < i < |\pi| - 1$, $\pi(i)$ is neither a memory access nor an error.

Note that the definition above states that only plain loads are annotatable. Indeed, this is because FAIs and CASes have side-effects, and these side-effects might affect different threads. For instance, in the case of a CAS, an `assume` can depend on the CAS failing, but other threads might crash if the CAS succeeds; not considering the options where the CAS succeeds due to not satisfying the `assume` leads to not identifying the error. Therefore, GenMC needs to explore all possible reads-from edges for such loads, which explains why they are not annotatable.

Algorithm 7.5 Calculate the blocking condition at a given node

```
procedure BlockCondition($n$)
    if $n$ is a loop header, an error, or a memory instruction then
        return $false$
    if CFG has edges $(n, \texttt{assume}(e), m), (n, \texttt{assume}(\neg e), m')$ then
        return $e$ ? BlockCondition($m$) : BlockCondition($n$)
    else if $n$ has a single outgoing of the form $(n, \texttt{assume}(e), m)$ then
        return $\neg e$
    else if $n$ has a single outgoing of the form $(n, r := e, m)$ then
        return BlockCondition($n$)$[e/r]$
```

[102] *Another way to see why* $\mathsf{f}_{\mathsf{ext}}(G, e) \triangleq max_{G.\mathtt{co}} G.\mathtt{W}_{\mathtt{loc}(e)}$ *preserves consistency for SC is that (1) the read can be interleaved immediately after all the events that were added before it in the execution graph and (2) the read cannot be observed by other threads and so does not directly affect the consistency of their actions.*

We annotate a load $r := x$ by computing the blocking condition of its successor and replacing any occurrences of r in it with *val*, which is a symbolic value that will stand for the value returned by the load. The blocking condition is defined recursively by algorithm 7.5 and calculates when it is possible to reach an `assume(e)` instruction with e being false without having executed any other memory accesses or error instructions along the way. This ensures that the load is annotatable along the path to the `assume`.

For instance, for the program below, the load in the then-branch is going to be annotated with $val + 1 \leq 0$, while the load in the else-branch is going to be annotated with $val \leq 0$, where *val* is a symbolic value. The first load is not annotated (technically, it is annotated with *false*).

```
$a := x;$            //no annotation
if ($a$)
    $b := y;$        //annotation: $val + 1 \leq 0$
    $b := b + 1;$
else
    $b := x;$        //annotation: $val \leq 0$
assume($b > 0$);
```

In terms of execution graphs, it is more convenient to represent blocking conditions using special read attributes encompassing all the values that make symbolic expressions succeed. As such, we introduce new attributes $\mathsf{bcond}(vs) \in \mathsf{Rattr}$, for $vs \subseteq \mathcal{P}(\mathsf{Val})$, and we write $\mathtt{annot}(r)$ to get the values vs on which a given $r \in R^{\mathsf{bcond}(vs)}$ blocks.

7.3.2 *Futile Explorations*

Now that we have seen how loads affect subsequent `assume` statements, let us present a more general way to reduce the number of blocked executions in DPOR.

The idea is to detect (as early as possible) that a given execution is futile, i.e., will not expose any program errors, and some thread will always remain blocked.

One way of doing this is by leveraging *fixed reads*, i.e., reads that cannot be revisited or removed in any subsequent subexploration.

Formally, we define fixed reads as follows:

Definition 7.3.2. An event $r \in \mathsf{R}$ is fixed in a graph G if it was added reading from a previously added write w (i.e., $G.\mathtt{rf}(r) <_G r$) such that $w \neq \mathsf{f}_{\mathsf{ext}}(G|_{\{e \in G.\mathsf{E} \mid e \leq_G r\}}, r)$.

It is easy to see that maximal extensions ensure that reads satisfying Def. 7.3.2 in a graph G are never going to be removed. To see why, let us argue by contradiction. In order for a write w to backward-revisit r in a graph G', r would have to be maximally added before w (i.e., be maximal w.r.t. $\{e \in G'.\mathtt{E} \mid e <_{G'} r \vee \langle e, w\rangle \in G'.\mathsf{corder}\}$), which contradicts the definition of fixed reads.

Another way of detecting futile explorations is by exploiting *freezing writes*, i.e., writes that are guaranteed to never be overwritten (as is often the case with insertions into queues and stacks).

Formally, we assume that certain writes are annotated[103] with the freezing write attribute, freeze $\in$ Wattr. (It is trivial to check the correctness of such annotations as part of GenMC's error checks; see §7.3.3.)

We illustrate how freezing writes can be leveraged in DPOR with the program below and its partially explored execution in Fig. 7.12.

$$\begin{array}{l\|l\|l} a := x & x := 1 \;\textit{//freezing} & \\ \mathtt{assume}(x \neq 1) & \text{[lots of code]} & \text{[lots of code]} \end{array}$$

By default, DPOR continues to explore the graph of Fig. 7.12 in the hope that another write to x is added, which will cause a revisit of the read of thread I. Since, however, the $x := 1$ write is annotated as a freezing store, such a write will never be added, and thus thread I will never become unblocked.

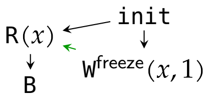


Figure 7.12: Freezing writes example

Generalizing a bit, we call a read r *frozen* in G if it is reading from a freezing write (i.e., $G.\mathtt{rf}(r) \in G.\mathtt{W}^{\mathsf{freeze}}$) or there exists a same-location freezing write $w \in G$ whose po-predecessor is a fixed read[104]. Note that a frozen read is not necessarily fixed: while it will never directly be revisited, it may be removed from G if an (insertion-order) earlier read is revisited.

Given fixed and frozen reads, we can define futile graphs as follows.

Definition 7.3.3. An execution graph G is *futile*, written $\mathsf{futile}(G)$, if there is a blocked event $b \in G.\mathtt{B}$ and a po-prior event s, which is either a fixed read or the initialization event init, and all events po-between s and b are either frozen reads or non-memory-access events[105].

7.3.3 *Algorithm*

The changes to GenMC induced by blocking annotations and futile explorations can be seen in algorithm 7.6.

First, whenever GenMC encounters an annotatable load (line 13), it only considers values not satisfying the blocking condition, as well as the rf prescribed by $\mathsf{f}_{\mathsf{ext}}$ (which is always considered).

Then, whenever the current graph is futile, Visit returns (line 2). In a similar fashion, if a write has been incorrectly annotated as freezing (i.e., there is $w_1 \in G.\mathtt{W}^{\mathsf{freeze}}$ and $w_2 \in G.\mathtt{W}_{\mathtt{loc}(w_1)} \setminus \{\mathtt{init}, w_1\}$), GenMC reports an error via IsErroneous (line 3)[106].

Note that such annotation errors are caught, even if executions are dropped as futile (i.e., it cannot be the case that the only graph where

[103] Certain languages even provide such annotations for free; e.g., Java final fields.

[104] In the latter case, r is reading from init.

[105] It is easy to extend the definition to also allow for local accesses (i.e., accesses to non-leaked allocated locations) in addition to frozen reads and non-memory accesses.

[106] We can relax this check and also allow a series of hb-ordered "initialization" writes performed hb-before a freezing write.

Algorithm 7.6 Preventing blocking in DPOR

```
procedure $\textsc{Visit}_P(G)$
    if $\neg\mathsf{consistent}_{\mathsf{M}}(G) \lor \mathsf{futile}(G)$ then return
    if $\textsc{IsErroneous}_{\mathsf{M}}(G)$ then exit("error")
    switch $a \leftarrow \textsc{NextEvent}_P(G)$ do
        ...
        case $a \in \mathsf{R}$
            for $w \in G.\mathtt{W}_{\mathtt{loc}(a)}$ do
                if $G.\mathtt{val}(w) \notin \mathtt{annot}(a) \lor w = \mathsf{f}_{\mathsf{ext}}(G, a)$ then
                    $\textsc{Visit}_P(\mathsf{SetRF}(G, a, w))$
        ...
```

an annotation error is generated by a futile one). Since the causal prefixes of w_1 and w_2 cannot contain the frozen reads before the block event of a futile execution (because there is no write to read from), this means that the (erroneous) graph G constructed by taking the causal prefixes of w_1 and w_2 and adding maximally the rest of the events, will be considered.

8

PERSEVERE: MODEL CHECKING FOR PERSISTENCY

In chapters 5 to 7, we saw how GenMC verifies safety properties of concurrent programs, assuming these were executed correctly by the underlying machine (i.e., according to the given memory consistency model). We did not take into account the possibility of the machine crashing during the execution of a program and what portion of the state would be preserved after the crash.

In this chapter, we present Persevere, an algorithm that can verify *persistency* properties, i.e., whether certain invariants hold in the presence of a crash. In essence, given a consistent execution of a given program, Persevere enumerates all its possible post-crash persisted states, and checks whether the supplied assertions/invariants hold. The novel major challenge in doing so is to combat the state space explosion arising from the persistency semantics[107].

To see this, consider the following sequential program under x86, with N writes to different memory locations:

$$x_1 := 1; \ldots ; x_N := 1 \tag{NW}$$

Even though these writes are executed in-order by x86[108], they might persist to durable storage out of order. If NW crashes, $x_i \in \{0,1\}$ for all $1 \leq i \leq N$.

Now suppose we want to enumerate all possible crash states for this program. The writes may persist to durable storage in any order (i.e., $N!$ ways) and any prefix of such orders may have completed before a crash (i.e., $N \times N!$ possible states). However, this naive enumeration of persistency ordering is far from optimal.

A much better way is not to enumerate the orders in which operations persist, but rather to consider whether each of the N operations persisted before the crash (i.e., 2^N states). Moreover, it is typically the case that only $M \ll N$ of operations are relevant for the invariant in question, so it suffices to enumerate 2^M states. When there are synchronization calls (e.g., flushes), persistency of one write implies persistency of (all) prior (same-location) writes that are separated by a synchronization call, which further reduces the number of states.

Persevere's key idea for exploring this vast state space efficiently is to model the assertions about the persisted state as a *recovery observer* that runs in parallel to the main program P and whose accesses are subject to different consistency axioms from those of P. By ensuring that these axioms do not require a total persistency order, Persevere never enumerates this order explicitly and thus significantly reduces

[107] *These are determined by the durable setting assumed (e.g., filesystem, non-volatile memory, etc).*

[108] *Recall that x86 does not reorder writes; see "A better x86 memory model: x86-TSO" [OSS09].*

the number of states to explore. Finally, following a declarative semantics enables us to integrate Persevere into GenMC, thereby leveraging its implementations.

We begin by discussing persistency models and their semantics (§8.1), we then present how a naive enumerating approach would work (§8.2), and finally present Persevere (§8.3).

8.1 PERSISTENCY SEMANTICS

In the past few years, a lot of work has been devoted to persistency models[109]. Analogously to memory consistency models (see §2.3), persistency models describe the exact state the durable storage of a system (e.g., disk, non-volatile memory) might have if a program crashes. Such models are useful to be able to reason whether a particular invariant holds in the presence of crashes.

We do not provide a full framework for specifying persistency models in this thesis. Instead, we extend the framework of §2.3 so that it can be used to reason about persistency (assuming single-crash scenarios) as follows.

First, we assume that memory locations are partitioned to *durable* and *volatile*, $\mathsf{Loc} \triangleq \mathsf{Dloc} \uplus \mathsf{Vloc}$, depending on whether they may propagate to durable storage or not. This location partitioning naturally induces a label partitioning: $\mathsf{Lab} \triangleq \mathsf{Dlab} \uplus \mathsf{Vlab}$ (labels that not associated with a durable location are volatile). Given an execution graph, we write $G.\mathtt{D}$ to refer to the durable events of G (i.e., $G.\mathtt{D} \triangleq \{e \in G.\mathtt{E} \mid \mathtt{loc}(e) \in \mathsf{Dloc}\}$).

Second, we assume that the underlying model provides a *persists-before* relation, $\mathtt{pb} \supseteq \mathtt{co}$, capturing the order in which durable writes persist. Similarly to $\mathtt{hb}$, $\mathtt{pb}$ is a parameter of the memory model, and we assume that for all $G \in \mathsf{Exec}$, if $\mathsf{consistent}_{\mathsf{M}}(G)$, then $\mathtt{pb}$ is a strict partial order.

As a couple of examples on how $\mathtt{pb}$ can be defined, assuming SC, we define *strict persistency* as $\mathtt{pb} \triangleq (\mathtt{po}\mathtt{rf} \cup \mathtt{co} \cup \mathtt{rb})^+$, prescribing that writes persist in the order they are executed, and *weakest persistency* as $\mathtt{pb} \triangleq \mathtt{co}$, prescribing that only $\mathtt{co}$-related writes persist in order.

Let us now go on to define the persistency semantics of a given program. We employ the notation $P{\lightning}T$ to specify persistency invariants, where P is a (concurrent) program and T is an invariant expressed as a sequential program that consists of a series of reads of durable locations and an `assert`.

We first define a *snapshot* as follows:

Definition 8.1.1 (Snapshot)**.** Given an execution graph G, a *snapshot* $S \subseteq G.\mathtt{D}$ is a set of durable events that (1) includes the initializer event, and (2) is downward-closed with respect to $\mathtt{pb}$, i.e., $\mathit{dom}([\mathtt{W}];\mathtt{pb};[S]) \subseteq S$.

Intuitively, a snapshot prescribes a set of durable events whose effects have reached the durable storage prior to a crash.

[109] *"Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model"* [RV18]; *"Weak persistency semantics from the ground up"* [RWV19]; *"Persistency semantics of the Intel-x86 architecture"* [Raa+19]

Given a snapshot S, we define its *frontier* as follows:

Definition 8.1.2 (Snapshot frontier)**.** Given an execution graph G and a snapshot S, the *frontier* of S, is $\mathsf{frontier}(S) \triangleq \left\{ \max_{G.\mathsf{pb}}(S_{dl}) \;\middle|\; dl \in \mathsf{Dloc} \right\}$.

Intuitively, the frontier of a snapshot S contains exactly one write for each durable location, corresponding to the pb-maximal write in S in that location.

Definition 8.1.3 (Persistency Correctness)**.** A (concurrent) program $P\lightning T$ contains a persistency error under a model M, if, there exists a consistent execution G of P and a snapshot S of G, such that T is erroneous (Def. 2.4.1) when run from the initial state $\mathsf{frontier}(S)$. A program is *p-correct* if it does not contain a persistency error.

8.2 A NAIVE APPROACH

Let us now see how we can enumerate the snapshots of a given execution, and their frontiers. To avoid confusion, we use the term *p-ordering* to refer to persistency orderings induced (due to pb).

The first partitioning mechanism we employ is representing p-ordering as a *partial* rather than a *total* relation: our definition of consistency (§8.1) does not require the p-ordering relation pb to be total and admits partial pb orders. This is in contrast to the existing literature on persistency [110], which requires p-ordering to be a *total* order.

Modeling pb as a partial order is highly effective in that it significantly reduces the number of executions to explore. To see this, assume a representation that requires a total p-ordering, and consider the 3W-PB program below comprising three parallel writes to x, y and z:

$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; z := 1 \qquad (\text{3W-PB})$$

Under such a representations, there are six (3!) possible p-orderings, i.e., the number of $[w_x, w_y, w_z]$ permutations. Moreover, recall from our notion of snapshots that a *prefix* of each p-ordering may have persisted prior to the crash. As such, since each p-ordering has three prefixes, the total number of explorations is 19 $(3 \times 3! + 1)$.

On the other hand, a representation with a partial p-ordering definition does not order the three writes, since they are in different locations. Thus, when constructing our p-snapshot S, it suffices to consider whether (the effect of) each write has persisted, i.e., $S \in \mathscr{P}(\{w_x, w_y, w_z\})$, thus yielding eight (2^3) explorations corresponding to eight different snapshots. In the general case of 3W-PB with N parallel writes, this amounts to reducing the number of explorations from $N \times N! + 1$ to 2^N.

[110] *"Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model"* [RV18]; *"Weak persistency semantics from the ground up"* [RWV19]; *"Persistency semantics of the Intel-x86 architecture"* [Raa+19]

8.3 RECOVERY OBSERVER

Although keeping pb partial eliminates a significant number of explorations, it nevertheless includes redundancies and can be further improved.

Given a program P↯T, our key idea is to not treat T as a program that runs after P, but rather as a thread *running in parallel* with P.

Let us demonstrate this idea with an example. Consider the REC-OB example below where 3W-PB from §8.2 runs in parallel with an independent write to w and crashes thereafter (↯); upon recovery (to the right of ↯) we check whether the write to w has persisted:

$$\text{3W-PB} \;\Big\|\; w := 1 \quad ↯ \quad \texttt{assert}(w = 1) \qquad \text{(REC-OB)}$$

Since there are eight (2^3) possible snapshots for the locations x, y, z and two (2^1) snapshots for w, this amounts to 16 (8×2) possible explorations.

However, note that although x, y and z are all written by 3W-PB, they are never read upon recovery. By contrast, w is observed (read from) upon recovery, and we must therefore consider two explorations: one in which the pre-crash write on w persists and one in which it does not. In other words, it suffices to consider p-orderings only on those locations that are *read from* upon recovery.

In fact, we use this intuition to reduce the number of explorations further, as long as the recovery routine only inspects the durable storage, and does not modify it. More concretely, we model observable snapshots through a *recovery observer*, a designated thread that runs in parallel with the original program. This way, we can model observability by reads-from (rf) edges between the events of the original program and those of the recovery observer. To this end, we *instrument* our executions (Def. 2.2.2) to include recovery events, as described below.

Definition 8.3.1 (Instrumented execution)**.** An *instrumented execution graph* is an execution graph G such that the event set is partitioned, $G.\mathsf{E} = G.\mathsf{NREC} \uplus G.\mathsf{REC}$, into *non-recovery events*, $G.\mathsf{NREC}$, and *recovery events*, $G.\mathsf{REC}$, comprising durable reads by a designated thread t_{r}: $G.\mathsf{REC} \overset{\text{def}}{=} \left\{ e \in G.\mathsf{E} \cap G.\mathsf{DR} \;\middle|\; G.\mathtt{tid}(e) = t_{\mathsf{r}} \right\}$.

Note that instrumented executions are executions (Def. 2.2.2) that additionally include recovery events comprising durable reads by the designated recovery thread t_{r}. That is, we model programs such as REC-OB, by having t_{r} issuing *recovery read instructions* to inspect the durable storage after the crash, e.g., the read to the right of ↯in REC-OB.

With instrumented executions in place, we no longer need a snapshot and its frontier to determine the observable values upon recovery. Instead, we simply constrain the set of observable values by requiring that the resulting instrumented execution be consistent, as defined below.

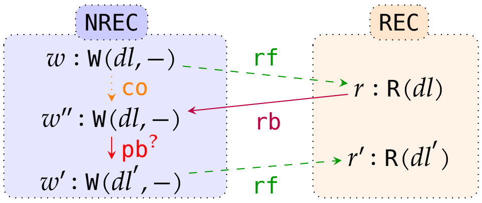


Figure 8.1: An instrumented execution precluded by rec

Definition 8.3.2 (Instrumented consistency)**.** An instrumented execution G is *consistent* iff:

- $G|_{G.\mathtt{NREC}}$ is consistent according to the underlying memory model (see §2.3); and (con)
- $[G.\mathtt{REC}]; \mathtt{rb}; \mathtt{pb}^?; \mathtt{rf}; [G.\mathtt{REC}] = \emptyset$ (rec)

The rec axiom *equivalently* enforces the conditions imposed by snapshots and frontiers.

To see an example how, let us consider an arbitrary persistency model M_P that among others requires that $\mathtt{co} \subseteq \mathtt{pb}$. rec preempts scenarios such as the one in Fig. 8.1, where a recovery read r on dl reads from w which is later overwritten by w'': $\langle w, w'' \rangle \in \mathtt{co}$. As $\mathtt{co}$ and $\mathtt{pb}$ agree for each location, we also have $\langle w, w'' \rangle \in \mathtt{pb}$. Moreover, as r and r' respectively read from w and w', then w, w' must have persisted prior to the crash, i.e., they are in the snapshot. As w'' is $\mathtt{pb}$-before w', for the snapshot to be $\mathtt{pb}$-downward-closed, w'' must also be in the snapshot. As such, since $\langle w, w'' \rangle \in \mathtt{pb}_{dl}$ and w'' is in the snapshot, then w'' is the $\mathtt{pb}$-maximal write in the snapshot for the location dl and not w, violating the $\mathtt{pb}_{dl}$-maximality condition of the snapshot frontier.

In turn, we can define correctness for instrumented executions as follows.

Definition 8.3.3 (Persistency correctness)**.** A program $P\lightning T$ contains a persistency error if any of its consistent execution graphs (according to Def. 8.3.2) contains an error event in $G.\mathtt{REC}$. A program is *p-correct* if it does not contain a persistency error.

Finally, we can now prove that recovery observers are sound and complete.

Theorem 3 (Equivalence)**.** A program $P\lightning T$ is erroneous under Def. 8.1.3 if and only if $P \uplus \{\langle t_{\mathsf{r}}, T \rangle\}$ is erroneous according under Def. 8.3.3.

Proof. $(\Rightarrow)$
Let: Let G be an erroneous consistent instrumented execution
Suffices: $S \triangleq \mathtt{init} \cup dom([\mathtt{W}]; \mathtt{pb}^?; \mathtt{rf}; [G.\mathtt{REC}])$ is a snapshot of $G|_{G.\mathtt{NREC}}$
⟨1⟩1. $\mathtt{init} \in S$
Proof: By construction.
⟨1⟩2. $dom([\mathtt{W}]; \mathtt{pb}; [S]) \subseteq S$

⟨2⟩1. $\mathit{dom}([\mathtt{W}];\mathtt{pb};[\mathtt{init}]) \subseteq S$

PROOF: Trivially, as $\mathit{dom}([\mathtt{W}];\mathtt{pb};[\mathtt{init}]) = \{\mathtt{init}\}$.

⟨2⟩2. $\mathit{dom}([\mathtt{W}];\mathtt{pb};[\mathit{dom}([\mathtt{W}];\mathtt{pb}^?;\mathtt{rf};[G.\mathtt{REC}])]) \subseteq S$

PROOF: $\mathit{dom}([\mathtt{W}];\mathtt{pb};[\mathit{dom}([\mathtt{W}];\mathtt{pb}^?;\mathtt{rf};[G.\mathtt{REC}])]) =$
$\mathit{dom}([\mathtt{W}];\mathtt{pb};[\mathtt{W}];\mathtt{pb}^?;\mathtt{rf};[G.\mathtt{REC}]) =$
$\mathit{dom}([\mathtt{W}];\mathtt{pb};\mathtt{rf};[G.\mathtt{REC}]) \subseteq S$

⟨2⟩3. Q.E.D.

PROOF: Steps ⟨2⟩1 and ⟨2⟩2 suffice; S is a union of two sets.

⟨1⟩3. Q.E.D.

PROOF: Steps ⟨1⟩1 and ⟨1⟩2 render S a snapshot by definition.

($\Leftarrow$)

LET: 1. G be an erroneous consistent execution of P and S a snapshot of G

2. G' be the corresponding instrumented execution of G where $\mathit{dom}(\mathtt{rf};[G'.\mathtt{REC}]) \subseteq \mathsf{frontier}(S)$

SUFFICES: G' is consistent according to Def. 8.3.2

⟨1⟩1. $G'|_{G'.\mathtt{NREC}} = G$ is consistent

PROOF: By construction.

⟨1⟩2. $[G'.\mathtt{REC}];\mathtt{rb};\mathtt{pb}^?;\mathtt{rf};[G'.\mathtt{REC}] = \emptyset$

LET: Let $r = [G'.\mathtt{REC}];\mathtt{rb};\mathtt{pb}^?;\mathtt{rf};[G'.\mathtt{REC}]$

⟨2⟩1. $r = [G'.\mathtt{REC}];\mathtt{rf}^{-1};\mathtt{co};\mathtt{pb}^?;\mathtt{rf};[G'.\mathtt{REC}]$

PROOF: By definition of $\mathtt{rb}$.

⟨2⟩2. $r \subseteq [G'.\mathtt{REC}];\mathtt{rf}^{-1};[\mathsf{frontier}(S)];\mathtt{co};\mathtt{pb}^?;[\mathsf{frontier}(S)];\mathtt{rf};[G'.\mathtt{REC}]$

PROOF: From ⟨2⟩1 and by construction ($\mathit{dom}(\mathtt{rf};[G'.\mathtt{REC}]) \subseteq \mathsf{frontier}(S)$).

⟨2⟩3. $r \subseteq [G'.\mathtt{REC}];\mathtt{rf}^{-1};[\mathsf{frontier}(S)];\mathtt{co};\mathtt{pb}^?;[S];\mathtt{rf};[G'.\mathtt{REC}]$

PROOF: From ⟨2⟩2 and $\mathsf{frontier}(S) \subseteq S$.

⟨2⟩4. $r \subseteq [G'.\mathtt{REC}];\mathtt{rf}^{-1};[\mathsf{frontier}(S)];\mathtt{co};[S];\mathtt{pb}^?;\mathtt{rf};[G'.\mathtt{REC}]$

PROOF: From ⟨2⟩3 and S being $\mathtt{pb}$-downward-closed.

⟨2⟩5. Q.E.D.

PROOF: From ⟨2⟩4 and $\mathsf{frontier}(S);\mathtt{co};[S] \subseteq \mathsf{frontier}(S);\mathtt{pb};[S] = \emptyset$

⟨1⟩3. Q.E.D.

PROOF: Steps ⟨1⟩1 and ⟨1⟩2 guarantee consistency of G' by definition.

□

8.4 EXAMPLE

Let us now see how PERSEVERE works through an example. Consider the program below manipulating x and y under SC with weakest persistency:

$$\begin{array}{lcl} x := 1 & & a := y \\ & ↯ & b := x \\ y := 1 & & \mathtt{assert}(\neg(a = 1 \land b = 0)) \end{array} \qquad \text{(REC-WW+RR)}$$

As before, the code to the right of ↯ denotes the recovery observer, inquiring whether it is possible upon recovery to see the second write

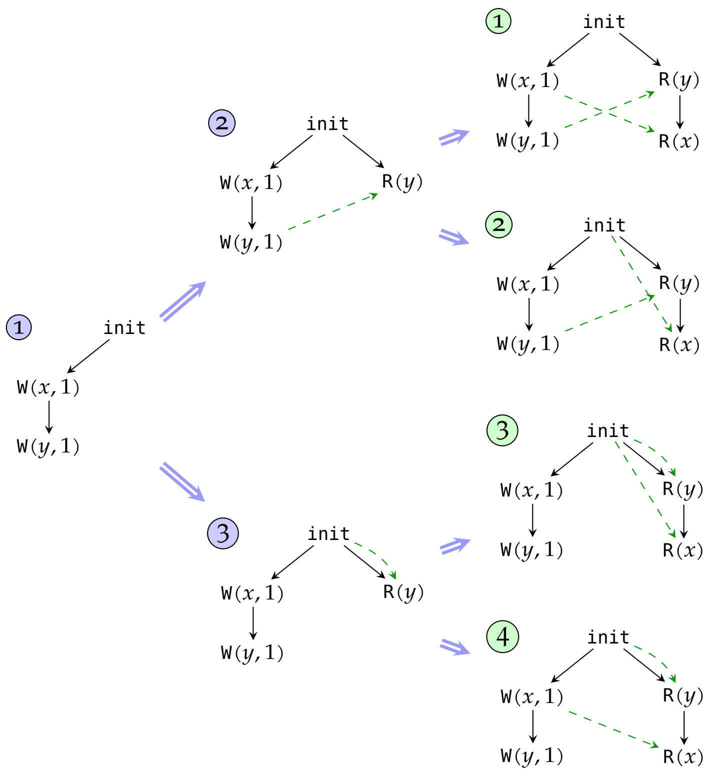


Figure 8.2: Persevere: Enumerating post-crash states of rec-ww+rr

but not the first; i.e., $a = 1 \land b = 0$. Under weakest persistency, this is indeed possible[111]; we next show how Persevere generates all possible outcomes of rec-ww+rr including one where $a = 1 \land b = 0$.

Assuming that the recovery observer runs after the main program, Persevere takes over after all consistent executions of a given program have been enumerated (see Fig. 8.2). In the case of rec-ww+rr, the program has a single execution (execution ①), and there is no pb edge between the two W events as they write to different locations. This will play in important role in the next steps, when Persevere starts exploring the recovery thread.

Persevere adds the events of the recovery observer and checks for instrumented consistency. When R(y) is added, it is consistent for it to read either the value of the main program or the initial value, and hence Persevere recursively explores both scenarios (graphs ② and ③). (Note that consistency is checked in the instrumented execution; see Def. 8.3.2.)

Let us assume that Persevere continues with ②. Next, Persevere adds the R event corresponding to $b := x$, which can similarly read from two values without breaking consistency: the value 1 written by the main program or the initial value. As before, Persevere explores both options, leading to executions ① and ②.

[111] *Recall that only same-location writes are* pb*-ordered.*

Finally, Persevere backtracks to graph ③ and, after adding the R(x) event again, explores executions ③ and ④, thereby concluding the exploration.

Suppose now we used strict persistency as the model instead. Then, the exploration would proceed as before except that ② would not be generated because it would be inconsistent.

Part III

TOOLS & EVALUATION

9 TOOLS

Kater and GenMC as described in §3 and §5 are available as open-source tools[112]. Kater can prove metatheoretical properties of memory models written in the `kat` language of §3, while GenMC can currently verify C/C++ programs, but can be easily extended to verify programs in any language that compiles down to LLVM-IR.

In this chapter, we describe the implementation of each tool, as well as their interaction: as we saw in §4, Kater can also be used as a metaprogramming tool to produce consistency-checking code that can be integrated to GenMC. This makes integrating new memory models to GenMC trivial.

9.1 KATER

The most important design decision we have to take as far as language inclusion is concerned is the inclusion algorithm itself. We opt for a breadth-first version of the Hopcroft-Karp algorithm that constructs DFAs on the fly, instead of constructing them a priori. Even though one can use a more sophisticated algorithm for inclusion checking (e.g., the ones described by Bonchi and Pous[113]), the Hopcroft-Karp algorithm seems to perform well enough for the tests we consider (see §10.1.1). Breadth-first traversal naturally leads to minimal counterexamples, which are easier to understand by humans.

To reduce the size of the automata used in the inclusions, we perform some of the saturations described in §3 implicitly. Instead of replacing `rf` with `rfe` ∪ `rfi` on the right-hand side of an inclusion, we simply modify our inclusion algorithm to allow the right-hand side to take an `rfe`/`rfi` step whenever the left-hand side takes an `rf` step. More generally, given an inclusion $a \subseteq b$, we do allow b to take a transition t_b when a takes a transition t_a, as long as $t_a \subseteq t_b$. Finally, in order for us to avoid empty assumptions like `rf;co` $= \emptyset$ we equip Kater with some domain knowledge so that it can automatically understand when two transitions do not compose.

9.2 GENMC

GenMC is a push-button verification tool that accepts as input a C/C++ program using C/C++11 atomics and/or the concurrency primitives from the `pthread` library, and reports any data races, assertion violations, or other errors encountered. By default, verification is performed with respect to the RC11 memory model[114], but there are com-

[112] Kater: Automating Weak Memory Model Metatheory and Consistency Checking (Project page) [KLV23a]; GenMC: Generic model checking for C programs [Kok]

[113] *"Checking NFA equivalence with bisimulations up to congruence"* [BP13]

[114] *"Repairing sequential consistency in C/C++11"* [Lah+17]

mand line options for selecting other models, such as IMM[115], LKMM[116], or any other model that can be encoded in KAT.

The main design goals of its implementation were:

GENERALITY: The tool should be able to verify programs written in a variety of programming languages with respect to a variety of memory models.

EFFICIENCY: The tool should implement a state-of-the-art SMC algorithm and incorporate further optimizations for common programming patterns.

USABILITY: The tool should provide useful and readable error messages.

EXTENSIBILITY: The tool should be easily adaptable to support additional models and synchronization primitives, and to tweak its performance. Extensibility is key to achieving the other goals, since it allows gradual improvements to the tool in terms of coverage, performance, and error detection/reporting.

These goals are achieved by a combination of techniques:

- GenMC's core DPOR algorithm is parametric in the choice of the memory model—subject to a few minimal constraints (§5)
- The implementation is based on LLVM, a versatile intermediate language for multiple programming languages.
- GenMC follows a modular architecture minimizing dependencies across components (see § 9.2.3.1), which makes it easy to extend with support for additional memory models (§9.3) and synchronization primitives (§9.2.1).
- GenMC contains a number of optimizations that provide noticeable performance benefits on common workloads (§9.2.3.2).
- GenMC keeps additional metadata so as to present error messages in terms of variables names appearing in the source code (§9.2.3.4).

GenMC has been applied to a few industrial settings, where it has found bugs and/or verified bounded correctness of concurrent libraries[117].

Verification with GenMC comprises three stages (Fig. 9.1).

The first stage invokes `clang` to compile the source C/C++ program to LLVM-IR. To accommodate programs written in different languages, GenMC also accepts LLVM-IR as its input, provided that it adheres to certain conventions about thread creation.

The second stage transforms the LLVM-IR code to make verification more effective by replacing spinloops by `assume` statements[118], bounding infinite loops, and performing sound optimizations, such as dead

[115] *"Bridging the gap between programming languages and hardware weak memory models"* [PLV19]

[116] *"Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel"* [Alg+18]

[117] *"VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models"* [Obe+21b]

[118] *See §7.*

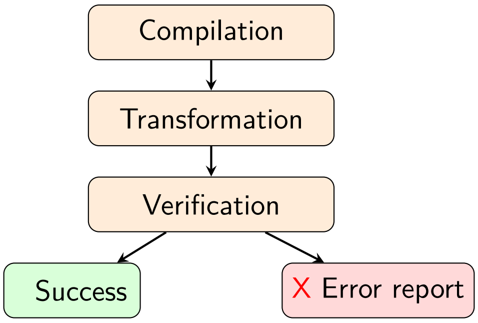


Figure 9.1: GenMC's overall architecture

allocation elimination. It also collects additional debugging information to enable better error reporting.

The third stage invokes the verification procedure, which explores all the executions of the program. If an error is found during this stage, the execution is halted and an error report is produced[119].

In what follows, we describe these stages in detail.

9.2.1 *Compilation and Supported Libraries*

As already mentioned, even though GenMC operates on LLVM-IR, it also accepts input programs written in high-level languages like C or C++. In principle, GenMC can be extended to handle any input language that compiles to LLVM-IR, as GenMC only relies on a working compiler installation for a given language.

Crucially, however, GenMC requires that all concurrency aspects of a program (i.e., atomic accesses, memory allocation and thread creation) are compiled to specific LLVM-IR instructions that the runtime can intercept (see §9.2.3). GenMC achieves this by providing its own concurrency API, which is then used by language headers. Concretely, in the case of C and C++, GenMC provides stubs for commonly used headers (e.g., `pthread.h`, `threads.h`, `stdatomic.h`, etc) that compile to its own concurrency API, so that user code that uses such headers can be properly analyzed by the tool.

Providing such an API is also beneficial when extending GenMC for different memory models and languages. For instance, adding support for LKMM[120] required that GenMC support LKMM-specific primitives for atomic accesses, thread creation, etc. Adding such support was trivial, as it merely entailed having all LKMM-specific primitives compile to GenMC's API. In a similar manner, adding support for a new language merely entails providing certain stubs, as opposed to adding extra runtime support for the language's concurrency primitives.

In addition to providing stubs for all concurrency-related headers, GenMC also used to provide stubs for certain system calls[121] like `open()`, `read()`, `write()`, etc, and modeled both their consistency and their persistency effect, assuming an `ext4` filesystem[122].

[119] *See §9.2.3.4 for an error report produced by GenMC.*

[120] *"Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel"* [*Alg+18*]

[121] *As these features seemed to be largely ignored by the tool users, recent GenMC versions no longer support them.*

[122] *"PerSeVerE: Persistency semantics for verification under ext4"* [*Kok+21*]

There are two ways one could implement these system calls: either by providing an actual implementation (which would then be compiled to LLVM-IR) or by adding support in the interpreter to internally implement those calls and communicating multiple times with the driver.

GenMC employed the latter solution because it is more portable and more efficient. An external implementation would have to be manually ported whenever support for more languages is added. By contrast, the internal implementation needs no change. Further, even if a new interpreter for a different runtime system is added, it should be simple to decouple the system calls from the interpreter, and have the different runtime systems share the infrastructure that handles system calls[123].

Finally, GenMC also exposes some header files that contain functions that can be used by user programs to aid verification. Examples of such functions include `assume` statements[124], functions that can be used to declare that a write to a (static) variable cannot be seen by other threads, functions that can be used to denote zero-effect spinloops, etc.

9.2.2 *Static Transformations*

As soon as the program code is compiled to LLVM-IR, verification can, in principle, begin. Instead of immediately trying to verify the program, however, it is beneficial to first employ some static transformations so as to make verification faster (often, exponentially faster).

In this section, we describe the implementation of some key static transformations employed by GenMC, including the ones of SAVer (see §7.2). These transformation are implemented as LLVM passes[125].

9.2.2.1 *Dead Code Elimination & Function Inlining*

A first transformation performed by GenMC is an SMC-safe form of *dead code elimination* (DCE). The reason an SMC-safe form of DCE is required is that we cannot simply remove e.g., unused load/store instructions, as these can potentially access unallocated/freed memory, thereby revealing memory errors.

As such, GenMC employs a conservative worklist-based algorithm to eliminate dead or redundant instructions. By dead, we mean (non-memory) instructions the result of which is never used, while by redundant we mean certain LLVM-specific instructions that can be safely removed (e.g., LLVM intrinsics, typecasts to the same type).

Even though DCE is purely an engineering improvement (and does not aid verification), it can lead to a significant performance improvement. As the performance of GenMC's interpreter greatly depends on the LLVM code size, reducing that size has a large cumulative effect when the interpreter re-runs the program.

[123] *Unfortunately, this design choice turned out to be an erroneous one. The design of the interpreter changed more often than anticipated, thereby making the maintenance of the system call implementations arduous.*

[124] *GenMC generally tries to be SV-COMP-compatible, and prefixes such function names with `__VERIFIER_`.*

[125] Writing an LLVM Pass [03b]

To further aid DCE, GenMC additionally applies a function inlining pass before performing DCE.

9.2.2.2 *Loop Rotation*

Let us now move to some transformations related to SAVer, and the transformations of loops to effect-free spinloops (see §7.2.4). The first such transformation is loop rotation.

Although LLVM already contains an implementation of a loop rotation pass, GenMC employs a custom pass that is applied to loops whose rotation is deemed worthwhile. The problem with the LLVM loop rotation implementation is that it performs a more aggressive transformation by converting loops to a canonical form containing a single backedge. That is, if the loop contains multiple backedges, it constructs a new node with a backedge to the loop header and redirects all the existing backedges to the new node. This latter transformation is detrimental to the static detection of effect-free paths because it would, for example, conflate the three loopy paths of ms-queue's dequeue operation (Fig. 7.8), thereby disabling the spin-assume transformation for the two that are effect-free.

To avoid this unintended consequence, one would then have to undo this transformation (e.g., by invoking a form of jump threading) or rely on dynamic purity checks (see §7.2.5).

Instead, and to be able to statically transform as many loops as possible, we opted for implementing a custom loop rotation pass, that transforms simple loops like CAS-LOOP2; loops that are not captured by the custom pass are handled dynamically.

9.2.2.3 *Bisimilarity*

Moving on to the merging of bisimilar nodes, there are also a couple of points worth mentioning regarding the implementation. First, detecting bisimilar nodes on LLVM is more complicated than what was discussed in §7.2.4 because LLVM represents programs in *static single assignment* (SSA) form. The effect of this design choice is that there are never two nodes with identical assignments on their outgoing edges, since by the SSA definition each assignment is to a different register. Therefore, the standard bisimilarity algorithm outlined earlier in this section will not detect any nodes as being bisimilar!

As an example, consider the "SSA-CFG" of thread II of the LOOP-PEEL program from § 7.2.4, which is shown below. The SSA-CFG is an enriched kind of CFG whose nodes may have ϕ-guards that define a variable differently depending on the incoming control flow path. For instance, in the SSA-CFG above, at node 2, b_1 is defined to be equal to b_0 if node 2 is reached from node 1, or to b_2 if it is reached from node 4.

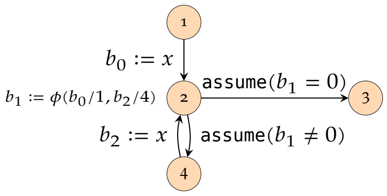


Figure 9.2: The "SSA-CFG" of thread II of LOOP-PEEL

```
a := 0
b := x
while (true)
    a := a + 1
    b := x
```

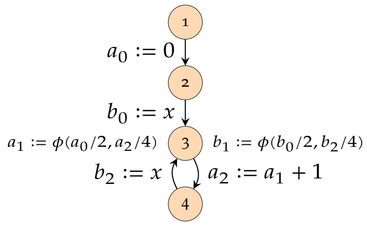


Figure 9.3: Merging bisimilar nodes in SSA

In order to match nodes 1 and 4, GenMC's bisimilarity implementation has to not only account for ϕ-nodes, but also unify the variables b_0 and b_2. It does so by collecting equality constraints and solving them by unification. For each node with more than one incoming edge, the algorithm starts iterating backwards for each pair of predecessors, and collects the constraints under which these predecessors are equal, simplifying them along the way. The iteration stops when some nodes cannot be equal under any constraints, or the entry node has been reached. At that point, any pair of nodes whose constraints can be trivially solved (namely, nodes 1 and 4 above) are deemed bisimilar.

Besides making bisimilarity detection more complex, SSA also affects the merging of bisimilar nodes. Consider the program of Fig. 9.3 along with its SSA-CFG. As can be seen, each of the assignments is to a different register, and node 3 contains two ϕ-guards (one for a and one for b) selecting the appropriate register to use depending on the incoming branch.

With the algorithm outlined above one can detect that nodes 2 and 4 are bisimilar. However, one cannot simply add an edge $a_2 := a_1 + 1$ from node 3 to node 2 because that would violate the SSA form. To ensure that the resulting CFG is well-formed we also have to introduce a ϕ-guard at node 2 to say which version of a should be used for node 2.

GenMC's implementation achieves this by *moving* ϕ-guards the incoming values of which have not been deemed bisimilar (e.g., the ϕ-guard for a here) to the new loop header, along with any other incoming edges these ϕ-guards have.

9.2.2.4 *Escape Analysis*

Next, to increase the applicability of the (static) spin-assume transformation, GenMC employs an escape analysis pass. Whenever a loop contains store instructions which write to variables that have not yet been made visible to other threads[126], the spin-assume transformation should still apply.

GenMC's escape analysis pass uses a simple worklist-based algorithm to find possible escape points of allocation instructions. Such escape points include call, return, and store instructions. If a loop contains a store to a variable that escapes after the loop, then the spin-assume transformation can still be applied.

9.2.2.5 *Spin-Assume*

Of course, there is also the spin-assume transformation itself (see §7.2), that handles effect-free loops (§7.2.4), as well as the static part of the ZNE transformation (§7.2.6).

9.2.2.6 *Load Annotation*

To mitigate the state-space explosion created by the assume statements in the code (e.g., introduced by the spin-assume pass), GenMC employs a load annotation pass, as described in §7.3.1.1. In terms of representation, GenMC represents annotations as expressions with a distinguished register corresponding to the value of the load. This register will then be replaced by possible values that the load can read at runtime, from the verification driver.

9.2.2.7 *Loop Bounding*

Finally, a last transformation that helps in dealing with infinite loops not handled by the spin-assume pass, is GenMC's loop-bounding pass. This pass bounds all loops in the program[127] by modifying their CFGs so that they decrement some dedicated variables (representing the loop bound) every time a backedge is taken. When such a dedicated variable reaches 0, the respective thread is "killed".

9.2.3 *Verification Infrastructure*

Let us now take a closer look at the verification infrastructure of GenMC. Below, we describe its architecture, as well as certain optimizations the tool performs, and its parallelization.

[126] *An example of such a loop is the push method of* `treiber-stack` *as described in* §7.2.5.

[127] *Possibly excluding certain loops indicated by the user.*

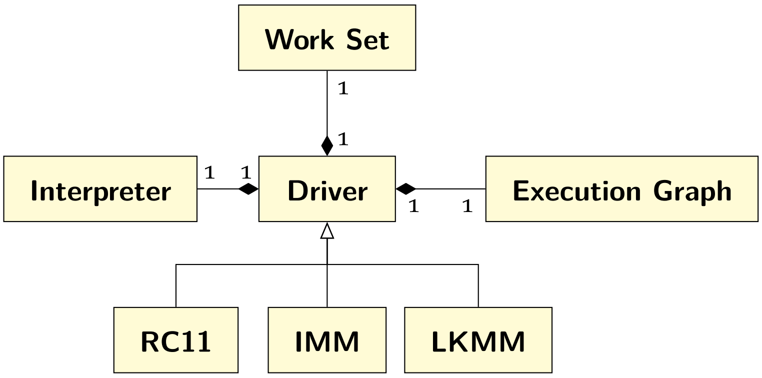


Figure 9.4: GenMC's verification components

9.2.3.1 *Architecture*

The architecture of GenMC's dynamic components are depicted in Fig. 9.4. At the center lies the verification *driver*, which owns three independent components: an interpreter, an execution graph, and a work set.

The execution graph records the visited execution trace, and has routines for calculating some basic relations on the graph, such as `rf` and `po`. As each memory model comprises different relations, the execution graph contains intrusive data structures that are dynamically populated during the exploration. These data structures are then used during consistency calculation (which is performed by the driver)[128].

The work set records revisit options for later exploration, the precise definition of which can depend on the memory model.

The interpreter merely executes the user program, notifying the driver each time a "visible" action (e.g., a load/store to shared memory) is encountered. It is directly based on the LLVM interpreter `lli`[129], and is the only part of GenMC's code base that heavily depends on LLVM.

In turn, the driver modifies accordingly the execution graph, possibly pushes some items to the work set, and returns control back to the interpreter, along with a value that will be used by the interpreter, if necessary (e.g., in the case of a load).

In effect, the driver and the interpreter can be thought of as *coroutines*[130]. The interpreter calls the driver whenever it encounters a visible action or finishes running a thread, while the driver monitors execution consistency, schedules the program threads, and discovers alternative exploration options, which are pushed to the work set.

As far as the verification algorithm itself is concerned, the driver mostly follows the recursive version of TruSt (using an explicit stack — the work set), but performs forward revisits in place, and only copies the graph for backward revisits. Although this means that the implementation consumes quadratic space, it is most likely faster than the purely iterative version of TruSt that needs to rebuild the execution graph after each backward revisit by re-interpreting the program.

[128] *See §9.3 for more information how these data structures can be controlled by the users.*

[129] lli - directly execute programs from LLVM bitcode [*03a*]

[130] *"Design of a separable transition-diagram compiler"* [*Con63*]

Moreover, as our experiments confirm (§10.2), the memory consumption of GenMC's implementation is never an issue.

Finally, the aforementioned components are all parameterized by user configuration options. The most important of these options is the memory model, which also determines whether dependencies between instructions should be tracked by the interpreter and stored in the execution graph. Concretely, the driver is overridden for each available memory model, and each instance provides memory-model-specific consistency checks and methods for crucial verification components[131].

9.2.3.2 *Optimizations*

To further enhance the performance of its core algorithm (TruSt), GenMC employs certain algorithmic and engineering optimizations.

As a first example of an algorithmic optimization, GenMC tries to reduce the verification problem to an easier one, using a simple syntactic robustness criterion. More specifically, if the user requests verification under a model M_1, but only uses a fragment of M_1 that coincides with a model M_2, then GenMC will verify the program under M_2. For instance, suppose the user requests verification under RC11, but the input program only uses SC accesses and/or locking primitives. In such a case, it is safe to verify the program under SC, which implies significantly less complex consistency checking.

As a further example, leveraging TruSt's maximal extensions, GenMC does not perform any consistency checks whenever an event e is added maximally in the current exploration. As guaranteed by extensibility, e does not violate consistency when added maximally, and hence consistency checks can be skipped.

As far as engineering optimizations are concerned, two prime examples include the scheduling policy used and the caching of instructions. Specifically, in order to reduce the number of backward revisits performed[132], GenMC prioritizes the scheduling of writes over reads. In order to reduce reliance on the interpreter even more, GenMC caches the instructions that follow each read (for each encountered value) in a trie, so that the interpreter does not have to be re-run every time a read is revisited.

9.2.3.3 *Parallelization*

As mentioned in §6.3.3, different subexplorations in GenMC can proceed completely in parallel, with no sharing required among them.

When parallelizing the implementation of GenMC, there were two major challenges that had to be surmounted. The first challenge was that GenMC uses several LLVM intrinsics that are not thread-safe. Thus, to preclude concurrency bugs due to the internal LLVM libraries on which GenMC relies, a significant portion of GenMC's infrastructure was redesigned to reduce its reliance on LLVM intrinsics[133], so that dif-

[131] *Generated by Kater; see §9.3.*

[132] *These delete a larger part of the graph and hence the interpreter has more work to do than to simply "replay" an already existing part.*

[133] *As one example, each GenMC thread compiles the user program separately, so that the representation of the compiled instances do not share any LLVM code.*

ferent threads can communicate in a thread-safe manner.

Another issue was how to pinpoint the correct design for inter-thread communication. For that, GenMC employs the following simple solution: it generates parallel tasks only for backward revisits. Indeed, as GenMC copies the current execution graph for backward revisits anyway, in a multicore setting, the copied graph can simply be passed to another worker thread, so that the two explorations can proceed in parallel.

Granted, such a design relies on there being enough backward revisits in a given program, but it nevertheless seems provides good scalability in practice[134]. Even though we also tried providing each thread with its own work-stealing queue, we found that this approach did not yield any benefits over the current implementation.

Putting everything together, GenMC's implementation has polynomial memory requirements and takes full advantage of the underlying machine's parallelism: as we will see in § 10.2.7, GenMC achieves an almost linear speedup when scaling to the number of physical cores available.

9.2.3.4 *Error Detection & Reporting*

GenMC detects a number of different kinds of errors: violations of user-supplied regular and persistency assertions, data races, memory errors and simple cases of termination errors. It reports errors by printing an offending execution graph and highlighting the event(s) that caused the violation. Upon request, GenMC can also print a total ordering of the instructions that lead to the violation, or produce the offending execution in the DOT language[135].

MEMORY ERRORS Memory errors refers to accessing uninitialized, unallocated or deallocated memory. In models like RC11, reasoning about memory safety can be tricky at times, as demonstrated by the example below:

$$\begin{array}{l||l} r := \texttt{malloc}() & a := y^{\texttt{rlx}} \\ [r]^{\texttt{rlx}} := 42 & \textbf{if}\ (a = 1) \\ y^{\texttt{rlx}} := 1 & \quad b := [r]^{\texttt{rlx}} \end{array}$$

This example is erroneous under RC11 because the allocation stored in r is not guaranteed to have propagated to thread II by the time it is dereferenced. (Since all accesses are relaxed, there is no synchronization between the threads.)

GenMC also accounts for more complicated scenarios such as r's location being concurrently freed when accessed, r's location being freed twice, or r containing the address of a local (stack) variable that might not be alive when accessed.

[134] *See §10.2.*

[135] DOT (graph description language) [23]

LIVENESS ERRORS In order to report liveness errors, GenMC follows the approach of Oberhauser et al.[136]. Given a potentially effect-free spinloop L that blocks in a full execution G, if all the reads in L's last iterations are reading co-maximally in G, then G represents a termination error and is reported to the user.

As a trivial example, consider the following (single-threaded) program and its execution in Fig. 9.5:

$$x := 1 \;\Bigg\|\; \begin{array}{l} a := x \\ \texttt{while}\ (a = 1)\ \{\} \end{array} \qquad \text{(W+R-LOOP)}$$

As the read of x is reading co-maximally (the only value available to it, in this case) and the loop blocks, the graph corresponds to a liveness violation.

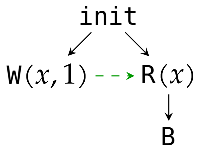


Figure 9.5: A liveness violation for W+R-LOOP

REFINING ERROR REPORTS It is often useful to refine the error reporting. For example, in memory models that treat data races as errors (such as RC11), GenMC by default detects data races and reports them as errors. This, however, can be costly in terms of verification time or even prohibit the verification of programs that use compiler/custom primitives to access shared memory, as such programs would almost certainly be considered racy.

To deal with such cases, GenMC provides switches that disable race detection and refine the range of errors that will be reported to the user. Switches of the latter kind are especially useful when dealing with programs that contain system calls. By default, when such system calls fail, GenMC reports an error, which is inconvenient for programs that contain proper error handling, as some system errors are rather benign (e.g., a file not existing). With the appropriate switch, in case of system errors, an appropriate value is written in `errno`, as dictated by the POSIX standard.

CASE STUDY We demonstrate the error reporting capabilities of GenMC with a real use case. We consider a flat-combining queue[137] that has been proposed to be ported in Rust's `crossbeam` library.

This queue serves as a nice case study for a couple of reasons. First, it contains loops that can diverge, and so its verification requires loop bounding, which GenMC can do automatically. Second, it is implemented using compiler primitives for concurrent accesses, and so its verification requires disabling race detection. Third, while experimenting with it, we found it to be buggy.

The error report produced by GenMC can be seen in Fig. 9.6. The error is quite intricate: it requires three threads to manifest, each of which executes a large number of instructions. The error is due to an ordering bug (relaxed accesses are used instead of release/acquire), which demonstrates the need for model checking tools that handle weak memory models.

[136] *"VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models"* [*Obe+21b*]

[137] Crossbeam: Flat combining #63 [*Sch16*]

```
Error detected: Attempt to read from uninitialized memory!
Event (3, 63) in graph:
<-1, 0> main:
<0, 1> thread_n:
  (1, 18): Urel (cmb.queue, 0) [(0, 36)] L.169: combiner.c
  (1, 19): Urel (cmb.queue, 2565579352) L.169: combiner.c
  (1, 96): Racq (m.msg._meta.next, 2565579416) [(2, 26)] L.228: combiner.c
  (1, 112): Wrlx (cmb.takeover, 2565579416) L.158: combiner.c
<0, 2> thread_n:
  (2, 26): Wrel (m.msg._meta.next, 94798317999592) L.167: combiner.c
<0, 3> thread_n:
  (3, 18): Urel (cmb.queue, 2565579352) [(1, 19)] L.169: combiner.c
  (3, 19): Urel (cmb.queue, 2565579480) L.169: combiner.c
  (3, 50): Rrlx (cmb.takeover, 2565579416) [(1, 112)] L.87: combiner.c
  (3, 63): Racq (m.msg._meta.next, 0) [BOTTOM] L.187: combiner.c
Number of complete executions explored: 2795
Number of blocked executions seen: 6001
Total wall-clock time: 2.12s
```

Figure 9.6: GenMC error report after removing irrelevant lines

Note that the error report contains helpful debugging information, such as the names of variables accessed (e.g., `m.msg._meta.next`) and the values read/written. To display this information, GenMC maintains a mapping from addresses to program variables using the additional debugging information collected in the "Transformation" phase.

9.3 THE INTERACTION BETWEEN KATER AND GENMC

We end this chapter by presenting some details on the integration of the consistency checks produced by Kater (see §4.2) into GenMC. After showing how Kater-generated checks are integrated into GenMC GenMC (§9.3.1) , we show how these routines can be optimized (§9.3.2), and then how Kater can validate GenMC's memory-model requirements (§9.3.3).

9.3.1 *Integrating Kater with GenMC*

Integrating the consistency checking procedure of §4.2 into GenMC consists of merely generating `C++` code that GenMC can use in order to check graph consistency.

An example consistency checker for SC (see Fig. 4.2) is shown in Fig. 9.7. In order to check consistency upon the addition of a new event e in a graph G, `isConsistent(G,e)` initiates a single DFS exploration by calling `visit0` and `visit1`, essentially modeling that $\mathrm{NFA}_{\mathrm{M}}$ can be in any state (i.e., q_0 or q_1) when a G-cycle accepted by $\mathrm{NFA}_{\mathrm{M}}$ passes through e. Whenever the DFS algorithm detects a cycle (i.e., whenever it encounters a back edge; e.g., in line 5), it checks whether $\mathrm{NFA}_{\mathrm{M}}$

```
    bool visit0(const ExecutionGraph &g, Event e)
    {
      setStatusAt0(e, ENTERED);
4     for (auto &p : po_imm_succs(g, e)) {
        if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
        else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
      }
      for (auto &p : rf_succs(g, e)) {
9       if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
        else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
      }
      for (auto &p : co_imm_succs(g, e)) {
        if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
14      else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
      }
      for (auto &p : rf_inv_succs(g, e)) {
        if (getStatusAt1(p) == UNSEEN && !visit1(p)) return false;
        else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
19    }
      setStatusAt0(e, LEFT);
      return true;
    }

24  bool visit1(const ExecutionGraph &g, Event e)
    {
      setStatusAt1(e, ENTERED);
      for (auto &p : co_imm_succs(g, e)) {
        if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
29      else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
      }
      setStatusAt1(e, LEFT);
      return true;
    }
34
    bool isConsistent(const ExecutionGraph &G, Event e)
    {
      return visit0(G,e) && visit1(G,e);
    }
```

Figure 9.7: KATER-generated code for SC-consistency checking

passed through an accepting state[1], and if so, returns `false` to denote a consistency violation.

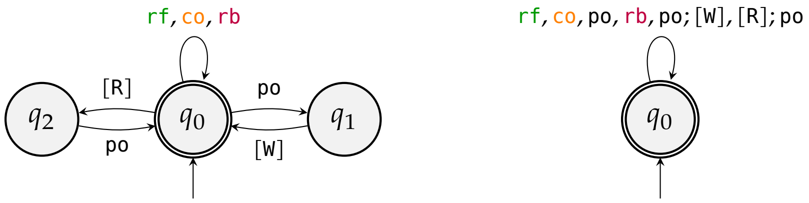
9.3.2 *Optimizing Consistency Checking for GENMC*

Even though the above procedure is linear in the size of the product of NFA_{M} and NFA_{G}, there are still a couple of ways we can improve it in the context of GENMC.

First, we can make NFA_{M} even smaller by merging its transitions. Take NFA_{SC} (cf. Fig. 4.2), for instance. If we merge $\mathtt{rf}^{-1};\mathtt{co}|_{\mathrm{imm}}{}^{+}$ into a single $\mathtt{rb}|_{\mathrm{imm}}$ transition, we can get rid of `visit1` and end up with a single-state automaton for SC, yielding a twofold complexity improvement. The only difference in the generated DFS code is that we will have to iterate over the $\mathtt{rb}|_{\mathrm{imm}}$ successor instead of the $\mathtt{rf}^{-1}$ successor in line 16.

Deciding whether to merge two transitions or not is largely a matter of engineering and tuning (automaton size vs transition complexity). In our experience, however, it is almost always worth merging predicate (guard) transitions with their successors. Such transitions boil

[1] The bookkeeping code for checking whether a cycle passed through an accepting state is omitted here for brevity.



```
   bool visit0(const ExecutionGraph &g, Event e)
2  {
     setStatus0(e, ENTERED);
     [...]
     for (auto &p : po_imm_succs(g, e)) {
       if (getStatus0(p) == UNSEEN && !visit1(p))
          return false;
7      else if (getStatus0(p) == ENTERED &&
          cycleHasAccepting()) return false;
     }
     if (isRead(g, e)) {
       if (getStatus0(e) == UNSEEN && !visit2(e))
          return false;
       else if (getStatus0(e) == ENTERED &&
          cycleHasAccepting()) return false;
12   }
     setStatus0(e, LEFT);
     return true;
   }

17 bool visit1(const ExecutionGraph &g, Event e)
   {
     setStatus1(e, ENTERED);
     if (isWrite(g, e)) {
       if (getStatus0(e) == UNSEEN && !visit0(p))
          return false;
22     else if (getStatus0(p) == ENTERED &&
          cycleHasAccepting()) return false;
     }
     setStatus1(e, LEFT);
     return true;
   }
27
   bool visit2(const ExecutionGraph &g, Event e)
   {
     setStatus2(e, ENTERED);
     for (auto &p : po_imm_succs(g, e)) {
32     if (getStatus0(p) == UNSEEN && !visit0(p))
          return false;
       else if (getStatus0(p) == ENTERED &&
          cycleHasAccepting()) return false;
     }
     setStatus2(e, LEFT);
     return true;
37 }
```

```
bool visit0(const ExecutionGraph &g, Event e)
{
  setStatus0(e, ENTERED);
  [...]
  for (auto &p : po_imm_succs(g, e))
    if (isWrite(p)) {
      if (getStatus0(p) == UNSEEN && !visit1(p))
       return false;
      else if (getStatus0(p) == ENTERED &&
       cycleHasAccepting()) return false;
    }
  if (isRead(g, e))
    for (auto &p : po_imm_succs(g, e)) {
      if (getStatus0(e) == UNSEEN && !visit2(e))
       return false;
      else if (getStatus0(e) == ENTERED &&
       cycleHasAccepting()) return false;
    }
  setStatus0(e, LEFT);
  return true;
}
```

Figure 9.8: $\text{N}\textsc{fa}_{TSO}$ before and after merging predicate transitions

down to if-statements in the generated DFS code and can thus be very efficiently merged with their successors.

To see this, consider the automaton corresponding to the TSO memory model before and after merging predicate transitions with their successors Fig. 9.8. If no transitions are merged (cf. Fig. 9.8, left), $\text{N}\textsc{fa}_{TSO}$ has three states, and each predicate transition will lead to a separate state, thereby unnecessarily enlarging the state space. If we do merge predicate transitions and their successors, on the other hand, then $\text{N}\textsc{fa}_{TSO}$ has a single-state (cf. Fig. 9.8, right), and the merged transitions will generate the same code as before, with the only difference being that the if-statements will be used as guards before/after iterating the successors of an event (e.g., line 6).

```
let eco = (rf ∪ mo ∪ fr)+
let sw = [REL] ; ([F] ; po)? ; (rf ; rmw)* ; rf ; (po ; [F])? ; [ACQ]
save hb = (po ∪ sw)+
let psc = [SC] ; po ; hb ; po ; [SC]
        ∪ [SC] ; ([F]; hb)? ; (po ∪ rf ∪ mo ∪ fr) scb ; (hb; [F])? ; [SC]
        ∪ [F]; [SC] ; hb ; [F]; [SC]
        ∪ [F]; [SC] ; hb ; eco ; hb ; [F]; [SC]
acyclic psc
```

Figure 9.9: The psc acyclicity axiom of RC11 in kat

The second way we can optimize our consistency checking routine is inspired by GenMC's existing infrastructure, and consists of saving and reusing (parts of) relations. In many memory models, various intermediate relations are defined and then used multiple times in subsequent relation definitions. Take RC11's psc, for example (cf. Fig. 9.9): eco and hb are used multiple times in psc's definition. Recalculating these relations every time they are used is quite costly, as it may redo the same computation for a given graph event, for different states of NFA$_{\mathsf{M}}$.

To alleviate this problem, Kater provides a save keyword that can be used to store the respective relation's predecessors for a given event, so that they do not have to ever be recalculated. If a user declares a given relation r as "saved", Kater will generate code that calculates an event's r-predecessors when the event is first added, and then store these predecessors in the graph so that they do not have to be recalculated. In addition, for relations that are transitive, it is sufficient to only calculate the immediate predecessors of the event, so as to reduce the memory and time complexity of the calculation.

9.3.3 *Checking GenMC's Memory-Model Requirements*

Allowing users to specify memory models and to optimize their consistency checks (e.g., by saving relations) makes porting new models to GenMC much easier.

However, there are still some subtleties one has to take care of before adding support for a new model. First, GenMC requires memory models to provide a ppo ⊆ po relation such that pporf ≜ (ppo ∪ rf)$^+$ is irreflexive in consistent executions because by design it generates only pporf-acyclic graphs[138]. Second, GenMC cannot usefully save arbitrary relations. Since GenMC continuously modifies the current execution graph (e.g., when trying a different rf edge for a read), we can only save information about predecessors of an event e that will never be removed from the graph for as long as e remains in the graph, namely its pporf; ppo predecessors.

[138] *See §5; this relation corresponds to* corder.

To preclude nonsensical GenMC behaviors when such requirements are violated, Kater checks that GenMC's memory model requirements are satisfied before generating consistency checks. Concretely, Kater will statically ensure that (a) the memory model acyclicity constraints imply irreflexivity of `pporf`, and (b) for each saved relation r, we have $r \subseteq$ `pporf;ppo`, and if r has, moreover, been declared as transitive, then $r;r \subseteq r$.

10 EVALUATION

Having discussed the implementation of Kater and GenMC, in this section we evaluate their performance.

For Kater, we evaluate its performance in proving various metatheoretic properties.

For GenMC, we empirically demonstrate the various aspects of its core algorithm (e.g., parametricity, polynomial memory consumption) on certain synthetic benchmarks, and then show how the tool performs on more realistic workloads. Where appropriate, we also compare against other similar (bounded) verification tools[139].

For their interaction, we compare the efficiency of Kater-generated consistency checks in GenMC with that of GenMC's previously employed handwritten checks.

experimental setup We conducted all experiments on a Dell PowerEdge M620 blade system, running a custom Debian-based distribution, with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz), and 256GB of RAM. We used LLVM 13.0.1 for GenMC. Unless explicitly noted otherwise, all reported times are in seconds, and the timeout limit is set to 30 minutes.

data availability statement All benchmarks used in this thesis are available online, as part of the thesis supplementary material[140].

10.1 kater

We begin by summarizing the different metatheoretic properties we were able to prove with Kater. These properties range over

- correctness of compilation,
- equivalence between different axiomatizations of the same model, and
- soundness of local reorderings.

Arguably, the most interesting result of this part of our evaluation is that Kater is able to reproduce some negative results from the literature, such as to show that the proposed compilation mapping from the original version of the C11 model to Power is unsound[141].

[139] *Though not strictly necessary, readers are encouraged to first read §11 to get a better overview of the tools presented in this section.*

[140] *"Automated Reasoning under Weak Memory Consistency (replication package)"* [Kok23]

[141] *"Repairing sequential consistency in C/C++11"* [Lah+17]

Table 10.1: Proving correctness of queries with KATER

Compilation	Time	Result
RC11 → IMM	0.05	✓
IMM → TSO	0.01	✓
RC11 → TSO	0.04	✓
$\mathrm{C11}_{\mathrm{strong}} \rightarrow \mathrm{TSO}$	0.04	✓
IMM → ARMv8	3.36	✓
RC11 → ARMv8	3.48	✓
$\mathrm{C11}_{\mathrm{strong}} \rightarrow \mathrm{ARMv8}$	1.93	✓
RC11/RA → POWER	3.11	✓
$\mathrm{RC11/SC} \rightarrow \mathrm{POWER}_{\mathrm{weak}}$	14.32	✓
$\mathrm{RC11/SC}_{\mathrm{RW}} \rightarrow \mathrm{POWER}$	138.35	✓
$\mathrm{RC11/SC}_{\mathrm{F}} \rightarrow \mathrm{POWER}$	56.67	✓
$\mathrm{C11}_{\mathrm{orig}} \rightarrow \mathrm{POWER}_{\mathrm{weak}}$	4.09	✗
$\mathrm{C11}_{\mathrm{orig}} \rightarrow \mathrm{POWER}$	16.00	✗
$\mathrm{POWER} \rightarrow \mathrm{POWER}_{\mathrm{simpl}}$	2.48	✓

Transformation	Time	Result
$\mathrm{RA} \leftrightarrow \mathrm{RA}_2$	0.01	✓
$\mathrm{RA} \leftrightarrow \mathrm{RA}_3$	0.01	✓
$\mathrm{RC11} \leftrightarrow \mathrm{RC11}_{\mathrm{alt}}$	0.39	✓

Equivalence	Time	Result
$\mathrm{TSO} \leftrightarrow \mathrm{TSO}_{\mathrm{FM}}$	0.15	✓
$\mathrm{SC} \leftrightarrow \mathrm{SC}_{\mathrm{FM}}$	0.06	✓
$\mathrm{eco} \leftrightarrow \mathrm{eco}_2$	0.01	✓
$\mathrm{Coh} \leftrightarrow \mathrm{Coh}_2$	8.82	✓

10.1.1 *Metatheoretic Properties*

An overview with some mapping-correctness and equivalence results we were able to prove with KATER can be seen in table 10.1. A ✓ entry denotes a successful proof, while an ✗ entry denotes that KATER (correctly) identified a counterexample while trying to complete a proof. Apart from the hard-coded assumptions on the primitive relations (see §3.4), each of the mapping tests requires the encoding of the compilation scheme as a KATER assumption[142].

As is evident from the table, the time required to complete a proof is proportional to the complexity of the memory models involved, and ranges from a few seconds to a few minutes. As expected, KATER requires less time to produce a counterexample than to prove compilation for models of similar complexity, and the counterexamples it produces are minimal.

10.2 GENMC

We now proceed to evaluate the performance and features of GENMC on a collection of synthetic and real-world benchmarks. Concretely, the evaluation aims to answer the following questions:

- How does generality affect GENMC's performance? Is it competitive against memory-model-specific DPOR approaches?

[142] *Some tests also required manual rotations. See §3.7 for more information and for the produced counterexamples.*

- How strongly does the optimality/memory tradeoff manifest? Does it have an impact on realistic workloads?

- How important are GenMC's optimizations (BAM and SAVer) for its scalability in real-world scenarios?

We can draw the following conclusions from the evaluation.

First, GenMC is the only DPOR tool that is optimal under weak memory models such as RC11, and optimality does play a major role when verifying code under such models (§10.2.1).

Second, although exponential memory consumption is not typically an issue in real-world benchmarks, there do exist certain workloads which cause it (§10.2.2). Moreover, even with such workloads aside, having different explorations being completely disjoint (with TruSt) has another major benefit: the ability to parallelize the DPOR algorithm with no sharing (§10.2.7).

Third, the optimizations employed by GenMC are crucial in its scaling to larger test cases and multiple cores (Section 10.2.3 to 10.2.5).

Before evaluating the above features of GenMC, however, we attempt to answer a more fundamental question:

- When is DPOR useful?

There are a plethora of other automated verification techniques and tools available, ranging from memory-model simulators to SMT-based approaches. It is not immediately evident, however, which techniques are better suited for each scenario/workload.

The evaluation of §10.2.1 suggests the following.

As expected, memory-model simulators are better suited for experimenting with memory-model features (typically expressed as litmus tests), as they can handle a larger variety of models compared to SMT and DPOR approaches.

SMT-based approaches typically scale better than DPOR for smaller program sizes, and are more effective in leveraging symmetries in the program code. However, they do not scale as well as DPOR does for larger thread sizes or larger number of threads (as the whole program code needs to be encoded as a SAT/SMT query), and do not handle certain dynamic aspects of the code (such as aliasing) well.

Having said the above, DPOR and SMT-based approaches are very different and cannot be fairly compared with one another. Whereas DPOR explores all program executions up to equivalence, SMT approaches require the programs to be given a safety specification and tries to explore only the part of the program pertinent to that specification. SMT-based tools encode the memory model together with the program semantics and the specification into a SAT/SMT formula, and query an external solver to determine whether the specification is met. To encode the program, SMT-based tools require an *a priori* loop bound for programs with loops. By contrast, DPOR tools only require that all loops terminate but do not need to unroll them.

BENCHMARK SELECTION We harvested tests from GenMC's benchmarking suite, the Competition on Software Verification (SV-COMP)[143] (mostly from the `pthread` category), as well as benchmarks proposed in the literature.

All in all, our suite comprises about 150 litmus tests, 120 synthetic benchmarks, and 30 data-structure benchmarks (including lock-free queues, stacks, and mutex algorithms).

TOOL SELECTION To keep the evaluation short yet representative, we select tools based on the following criteria.

TECHNIQUE: To obtain a comprehensive overview of the (bounded) verification landscape, we included tools based on different core techniques, including memory-model simulators, SMT-based approaches, and DPOR algorithms.

PARAMETRICITY: As tools aiming to be memory-model-parametric are likely to face somewhat similar challenges, we preferred parametric tools when facing the choice to select among tools using similar techniques. For non-parametric tools, we selected tools that can handle weak models like RC11.

INPUT LANGUAGE: To be able to make sense of the resulting runtimes, we opted for tools operating on C/C++ (i.e., the tool that GenMC supports).

As such, the evaluation comprises the following tools.

MEMORY-MODEL SIMULATORS Herd[144] is a memory model simulator that allows users to experiment with different axiomatic memory models on small "litmus test" programs written in a toy language. It supports a wide range of models, but explores executions in a naive fashion and scales rather poorly. For our tests, we compared against Herd-ARMv8, due to technical issues when trying to use RC11.

rmem[145] is a memory model simulator that, among others, supports operational definitions of ARMv8 and RISC-V. Pulte et al. claim that rmem's current operational definitions are suitable for model checking, as they are much faster than the previous ones[146], and tools like Herd. rmem operates on the ARMv8/RISC-V ISA, but does not support dynamic thread creation. Also, unlike SMC tools, rmem does not employ any DPOR techniques, and so does not scale very well. For our tests, we ran rmem under ARMv8, as rmem does not provide RC11 semantics.

DPOR TOOLS Nidhugg[147] is a stateless model checker for C programs that supports SC, TSO, PSO, and also offers limited support for POWER and ARMv7. Even though Nidhugg supports multiple memory models, it is not parametric in the choice of the model. For SC, Nidhugg implements the optimal-DPOR algorithm (which may have exponential memory consumption), while under POWER it implements

[143] Competition on Software Verification (SV-COMP) [SV-19]

[144] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [AMT14]

[145] rmem: Executable concurrency models for ARMv8, RISC-V, Power, and x86 [rme09]

[146] *"Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8"* [Pul+18]

[147] *"Stateless model checking for TSO and PSO"* [Abd+15]; *"Stateless model checking for POWER"* [Abd+16]

the RSMC algorithm which is non-optimal, and does not support features like RMW instructions. For our tests, we ran Nidhugg under SC and POWER, under a Shasha-Snir equivalence partitioning.

CDSChecker [148] is a stateless model checker for C/C++ programs that supports a certain strengthening of the (original) C11 memory model[149] that forbids "out-of-thin- air" outcomes (see §5.1). Unlike GenMC, it does not track dependencies, but rather uses a notion of promises to support load-store reorderings. This often leads to *infeasible* explorations in programs with C11 "relaxed" memory accesses. In addition, although CDSChecker operates under the reads-from equivalence, it is non-optimal with respect to its partitioning, which leads to *duplicate* explorations.

SMT-BASED MODEL CHECKERS Among the SMT-based tools, the only one that is parametric in the choice of the memory model is Dartagnan [150]. Dartagnan works for programs written in a toy language similar—but not identical—to that of Herd, and also provides a C frontend. For our tests, we ran Dartagnan under RC11.

10.2.1 *DPOR vs Other Approaches*

We start by evaluating GenMC against other state-of-the-art model checking tools. Our evaluation revolves around the following points:

1. GenMC outperforms all other SMC tools that support weak memory models like RC11—i.e., Nidhugg/POWER and CDSChecker—as well as memory model simulators—i.e., Herd and rmem.

2. Similarly to other SMC tools, GenMC cannot be fairly compared against the state-of-the-art SMT-based model checkers. GenMC tends to be faster on benchmarks with relatively few executions (i.e., polynomial in the size of the benchmark), whereas SMT-based tools are typically better on parametric benchmarks with an exponential number of executions (though SMT-based approaches explode as the program size becomes larger).

In what follows, we compare various tools on both synthetic and data-structure benchmarks[151].

10.2.1.1 *Synthetic Benchmarks*

Table 10.2 highlights the differences among different model checking tools on a set of benchmarks with only loads and stores.

For the `readers(N)` benchmark, GenMC and CDSChecker perform much better than the other stateless model checkers even though all tools explore the same number of executions (2^N). On the other hand, Herd, rmem and Nidhugg do not scale so well for $N = 15$. For Herd this is because it follows a very naive approach and uses inefficient

[148] *"CDSChecker: Checking concurrent data structures written with C/C++ atomics"* [ND13]

[149] *"Mathematizing C++ concurrency"* [Bat+11]

[150] *"BMC for weak memory models: Relation analysis for compact SMT encodings"* [Gav+19]

[151] *We ran GenMC under RC11 and a Shasha-Snir equivalence partitioning.*

Table 10.2: Synthetic benchmarks with only loads and stores

	Herd	rmem	Nidhugg/POWER	CDSChecker	GenMC	Dartagnan
`readers(5)`	0.03	0.13	0.12	0.01	0.04	1.93
`readers(10)`	1.81	2.72	4.11	0.19	0.19	127.53
`readers(15)`	96.36	207.66	295.58	11.07	2.61	oom
`nwrites-loc(5)`	0.05	0.40	0.10	0.02	0.04	1.79
`nwrites-loc(10)`	🕒	🕒	1177.34	0.02	33.26	241.05
`nwrites-loc(15)`	🕒	🕒	🕒	0.02	🕒	oom
`nwrites(5)`	0.02	0.42	0.08	0.01	0.04	2.25
`nwrites(10)`	0.03	🕒	0.08	0.01	0.04	37.86
`nwrites(15)`	0.03	🕒	0.09	0.02	0.04	oom
`mp(10)`	2.08	0.13	0.09	0.01	0.04	1.54
`mp(50)`	🕒	0.60	0.10	0.02	0.06	1.79
`mp(100)`	🕒	1.88	0.16	0.03	0.23	2.32

`readers(N)`: A writer thread and N reader threads accessing the same memory location (by Abdulla et al. [Abd+17]).

`nwrites-loc(N)`: N threads writing the same memory location (by Abdulla et al. [Abd+18]).

`nwrites(N)`: N threads writing different memory locations.

`mp(L)`: Two threads showcasing the Message Passing idiom, with the reader reading the first variable in a loop until the value true has been read. L is the loop bound.

data structures (e.g., linked lists), while for rmem and Nidhugg this is because, for every event that they add to the constructed trace, they have to check which of its possible parameters are consistent according to the memory model. As can be seen, this is a costly procedure that dominates the running time. We also note that Dartagnan performs poorly, and consumes all memory available to the Java Virtual Machine for $N = 15$[152]. This is in contrast to previously reported results for the same benchmark and Dartagnan [153], where the SMT solver was able to verify this benchmark really fast.

Similarly, for `nwrites-loc(N)`, CDSChecker outperforms the other stateless model checkers. In this case, however, this is due to the used equivalence partitionings. More specifically, while rmem, Herd, Nidhugg and GenMC run under the Shasha-Snir equivalence, CDSChecker only runs under the reads-from equivalence, and does not order unread, same-location writes. Thus, rmem, Herd, Nidhugg and GenMC explore $N!$ executions, while GenMC and CDSChecker explore only one execution.

Concluding table 10.2, the `nwrites(N)` and `mp(L)` benchmarks demonstrate interesting aspects of rmem and Dartagnan, respectively. For `nwrites(N)`, while all other SMC tools explore only one execution by observing that the threads write to different memory locations, rmem explores $N!$ executions because it imposes a single total ordering across the writes of *all* memory locations. For `mp(L)`, while Dartagnan needed exponential time and memory for the previous benchmarks where the

[152] *About 32 GB in the machine we used.*

[153] *"HMC: Model checking for hardware memory models"* [KV20]

Table 10.3: Synthetic benchmarks taken from SV-COMP [SV-19]

	Herd	rmem	Nidhugg/POWER	CDSChecker	GenMC	Dartagnan
`peterson(10)`	⏲	5.16	0.14	0.05	0.02	2.72
`peterson(20)`	⏲	31.93	0.24	0.17	0.02	4.54
`peterson(30)`	⏲	97.47	0.39	0.40	0.03	7.84
`parker(3)`	⏲	79.35	1.71	0.28	0.17	2.99
`parker(4)`	⏲	224.39	5.01	0.89	0.27	3.89
`parker(5)`	⏲	498.57	12.84	2.30	0.55	5.02
`szymanski(1)`	⏲	1.10	0.13	0.45	0.05	4.85
`szymanski(2)`	⏲	98.94	3.79	⏲	0.18	4.17
`szymanski(3)`	⏲	⏲	178.73	⏲	3.34	17.24
`lamport(2)`	⏲	9.87	0.12	0.02	0.02	1.69
`lamport(3)`	⏲	⏲	313.57	24.41	2.76	1.93
`lamport(4)`	⏲	⏲	⏲	⏲	⏲	2.48

`peterson(L)`: Two threads performing Peterson's mutual-exclusion algorithm, with L being the loop bound.

`parker(L)`: A recreation of a bug in JDK, adapted from [Abd+15]. L is the loop bound.

`szymanski(N)`: Two threads perform Szymanski's mutual-exclusion algorithm N times.

`lamport(N)`: N threads perform Lamport's fast mutex algorithm.

number of threads increases, it only needs time that increases linearly with L for `mp(L)`, where the number of threads remains constant.

Next, we move on to table 10.3, which contains somewhat more challenging benchmarks, where Herd consistently times out. While the observations for Nidhugg and rmem are similar to those for table 10.2 above, this table provides us with some useful insight regarding the differences between CDSChecker and GenMC, but also between Dartagnan and stateless model checkers in general.

For the first two benchmarks, both GenMC and CDSChecker outperform Dartagnan. Indeed, in this case our observations agree with those of Gavrilenko et al.[154]: when the executions do not grow exponentially as the loop bound gets larger, stateless model checking tools are faster than bounded model checkers. This is not only because the SMT solver has a worst case exponential complexity, but also because even the encoding fed to the SMT solver may be larger than the state space of the program.

In the last two benchmarks, the situation is reversed: Dartagnan scales more nicely than both CDSChecker and GenMC. As explained before, this is because the number of executions grows exponentially, much faster compared to the encoding fed to the SMT solver by Dartagnan [155].

Overall, observe that GenMC generally outperforms CDSChecker by a large factor in table 10.3, mostly because CDSChecker explores duplicate/infeasible executions. For instance, in the case of `szymanski(N)`, CDSChecker quickly times out, as the notion of promises it

[154] *"BMC for weak memory models: Relation analysis for compact SMT encodings"* [Gav+19]

[155] *Even though Dartagnan needs more time to verify `szymanski(3)`, in contrast to SMC tools, the time required does not increase exponentially with the program size.*

Table 10.4: Synthetic benchmarks with RMW instructions

	Herd	rmem	CDSChecker	GenMC	Dartagnan
ainc(3)	0.28	0.17	0.01	0.05	2.21
ainc(4)	7.58	0.89	0.34	0.05	2.84
ainc(5)	365.56	8.01	25.46	0.05	2.01
binc(3)	71.81	42.86	0.21	0.05	1.94
binc(4)	🕒	🕒	125.73	0.06	2.09
binc(5)	🕒	🕒	🕒	0.87	2.42
indexer(12)		🕒	0.89	0.05	
indexer(13)		🕒	131.75	0.11	
indexer(14)		🕒	🕒	0.56	
indexer(15)		🕒	🕒	4.40	

ainc(n): N threads perform fetch_add$^{\mathtt{rlx}}(x)$.

binc(n): N threads perform fetch_add$^{\mathtt{rlx}}(x)$; fetch_add$^{\mathtt{rlx}}(y)$.

indexer(n): A classic benchmark by Flanagan and Godefroid [FG05] designed to demonstrate the benefits of DPOR compared to classic POR techniques. N threads and each thread adds four entries in a shared hash tables. Collisions occur for $N \geq 12$.

employs make it explore many infeasible executions, which cripple its performance.

Finally, in table 10.4, we present some synthetic benchmarks that contain RMW accesses. We exclude Nidhugg from that table because it does not support RMW accesses under POWER and ARMv7. As can be seen, GenMC scales fairly well for these benchmarks.

CDSChecker, by contrast, scales poorly because it explores a very large number of infeasible executions. In some of the benchmarks, they are even four orders of magnitude more than the consistent ones. Similarly to to szymanski(N) in the previous table, infeasible executions arise due to the way CDSChecker handles porf cycles and release sequences. The problem manifests itself especially when relaxed accesses are involved. When, for example, we change all accesses of ainc(5) to acquire/release accesses, CDSChecker terminates in 0.07 seconds, and explores much fewer infeasible executions.

Dartagnan, on the other hand, scales nicely both for the ainc(N) and binc(N) benchmarks. The SMT solver manages to establish the supplied assertion without exploring the entire exponential search space, presumably by leveraging the fact that addition is commutative.

For the indexer(N) benchmark, we had to exclude Herd and Dartagnan from the table because their input language does not support all the necessary constructs (e.g., multiplication operations or proper con-

Table 10.5: Benchmarks adapted from Pulte et al. [Pul+19]

	rmem	Dartagnan	GenMC
`DQ/211-2-1`	162.94	1.85	0.12
`DQ-opt/211-2-1`	732.04	1.90	0.14
`STC/210-011-000`	1081.44	240.98	0.14
`STC-opt/210-011-000`	1105.03	253.68	0.15
`QU/100-100-010`	1077.86	207.08	0.14
`QU-opt/100-100-010`	⏲	214.41	0.14

`dq`: An implementation of the Chase-Lev deque.
`stc`: An implementation of the Treiber stack.
`qu`: An implementation of the Michael-Scott queue.

ditional branching). In addition, rmem times out even for $N = 12$, presumably because of the many accesses to different memory locations.

10.2.1.2 *"Real World" Benchmarks*

We proceed by evaluating GenMC on more realistic workloads. For the rest of this section, we exclude Nidhugg and Herd from our comparisons. Nidhugg does not handle RMW instructions (which are required for these benchmarks), while Herd operates on a toy language that is insufficient for these benchmarks, and thus requires a complete rewrite of the code in its input format, which is tedious and very restricting for larger benchmarks (e.g., no support for loops). Thus, we only compare GenMC against rmem, CDSChecker and Dartagnan.

That said, since rmem and CDSChecker/Dartagnan require a different format as input, we compare against each tool on two separate sets of benchmarks (published along with rmem and CDSChecker, respectively). Unfortunately, we cannot use CDSChecker on rmem's benchmarks since they contain structs with atomic pointer fields (which are not supported by CDSChecker), nor can we use rmem on CDSChecker's benchmarks, since these benchmarks are too large to be translated to rmem's input language.

We begin by comparing against the most intensive benchmarks for which Pulte et al.[156] published results for rmem (cf. table 10.5). While rmem operates on litmus tests, Pulte et al. provide a C++ version of these benchmarks, which we converted to C. We note that the "opt" version of each benchmark differs from the non-opt version in that it has "less" synchronization, i.e., the opt version uses weaker access modes for some accesses.

As can be seen, GenMC outperforms rmem by a very large margin, which is consistent with the outcomes of § 10.2.1.1. Since rmem does not leverage DPOR techniques, it embarks on many redundant explo-

[156] *"Promising-ARM/RISC-V: A simpler and faster operational concurrency model"* [Pul+19]

Table 10.6: Benchmarks adapted from Norris and Demsky [ND13]

	CDSChecker	Dartagnan	GenMC
`linuxrwlocks`	13.63	43.47	0.05
`linuxrwlocks-bnd`	🕒	2.05	0.58
`mpmc-queue`	10.33	🕒	0.12
`mpmc-queue-bnd`	18.24	150.66	2.39

`linuxrwlocks`: A reader-writer lock ported from the Linux kernel. Three threads use the lock to read and/or write a shared variable.

`mpmc-queue`: A multiple-producer multiple-consumer queue. Two threads are enqueueing and then dequeueing an item.

rations. In addition, the time for GenMC and GenMC does not change much for the variations of each benchmark; by contrast, the time for rmem changes dramatically, even for `DQ`, where the number of consistent executions (according to GenMC's equivalence partitioning) remains the same across the opt and the non-opt version.

As it can also be seen, Dartagnan only performs well for `DQ`. This is because both `STC` and `QU` employ dynamic memory allocation, and thus Dartagnan spends a lot of time on various analyses (e.g., alias analysis) to make verification tractable.

We continue with the most intensive benchmarks published by Norris and Demsky[157] for CDSChecker, which are shown in table 10.6.

Let us first focus on the two SMC tools, namely GenMC and CDSChecker. Before going into details about individual benchmarks, let us briefly discuss the different mechanisms these tools use to handle infinite loops. GenMC uses a combination of loop bounding along with the transformation of infinite loops with no side-effects into `assume()` statements. CDSChecker, on the other hand, uses a combination of control over the memory liveness and a CHESS-like yield-based fairness system[158]. As we will see, these different mechanisms force the two tools to explore a different number of executions in tests with infinite loops.

For `linuxrwlocks`, GenMC outperforms CDSChecker. That said, GenMC also explores a different number of executions compared to CDSChecker. More specifically, this test case contains infinite loops that GenMC manages to transform into `assume()` statements, while CDSChecker terminates only if we use an upper bound on the times a thread is allowed to see the same value. Naturally, CDSChecker is sensitive to that upper bound, and as this grows larger the verification becomes even slower. In addition, CDSChecker explores more than 40 times more infeasible executions than consistent executions, presumably due to its handling of relaxed accesses.

Given the above, in an effort to alleviate these discrepancies between the explored executions and perform a more precise comparison, we

[157] *"CDSChecker: Checking concurrent data structures written with C/C++ atomics"* [ND13]

[158] *"Finding and reproducing Heisenbugs in concurrent programs"* [Mus+08]

also manually bounded the test case, thus rendering the mechanisms of all tools that handle infinite loops useless. We also simplified the client code (the threads perform less operations on the lock), since the manual bounding increases the state space of the program.

As can be seen in the respective entry for the first benchmark (namely, `linuxrwlocks-bnd`), although the test case is simplified, CDSChecker times out, while both GenMC finishes almost instantly. While this may be surprising at a first glance, it is again due to the way CDSChecker handles relaxed accesses and release sequences[159]. More specifically, the definition CDSChecker uses for release sequences leads the tool to explore more executions compared to GenMC, and the relaxed accesses lead the tool to also explore many infeasible executions (plus a few duplicates).

Interestingly enough, however, even though CDSChecker does not verify `linuxrwlocks-bnd` after 8 hours, if all accesses are changed into acquire/release accesses, it manages to verify it in only 4 seconds. Assuming this is not to an implementation issue, this fact highlights the importance of optimality and the way it affect theperformance of a tool.

For `mpmc-queue`, the observations are similar to `linuxrwlocks`. For the original version of the test case, GenMC outperforms CDSChecker, as it transforms infinite loops into `assume()` statements, while CDSChecker is sensitive to the liveness bound. For the bounded version, CDSChecker explores a few more consistent executions (due to its definition of release sequences) and many infeasible ones.

Finally, focusing on Dartagnan, we see that we can make observations similar to the ones for table 10.5. Dartagnan performs much better for `linuxrwlocks`, where no dynamic allocation takes place, but also for the bounded versions of the two benchmarks compared to the unbounded ones.

10.2.2 *Optimality and Memory Consumption*

To measure how strongly the memory/optimality tradeoff manifests and its impact, we perform an evaluation comprising two parts.

In the first part, we compare GenMC against Nidhugg/SC[160]. We chose Nidhugg/SC because it provides both an optimal and a nonoptimal (sleepset-based) DPOR algorithm, thus nicely highlighting the tradeoff between time and memory that DPOR algorithms have to pay. For GenMC, we run it in two modes: TruSt and naive, with the latter achieving optimality by recording the set of backward revisits performed (see §5).

In the second part, we evaluate the two modes of GenMC on a set of realistic, weak-memory benchmarks. As Nidhugg does not support a memory model similar to GenMC's RC11[161], we had to exclude it from this comparison. Note, however, that it is easy to construct clients of

[159] *Or due to some bug in the corresponding code.*

[160] *"Stateless model checking for TSO and PSO" [Abd+15]; "Optimal dynamic partial order reduction" [Abd+14]*

[161] *Recall that Nidhugg's support for POWER and ARM is based on a different (nonoptimal) algorithm, and spans over a limited subset of these models.*

Table 10.7: Synthetic benchmarks (24h timeout)

	Nidhugg/nonoptimal			Nidhugg/optimal			GenMC$_{\text{naive}}$		GenMC	
	Executions	*Mem.*	*Time*	*Executions*	*Mem.*	*Time*	*Mem.*	*Time*	*Mem.*	*Time*
`lastzero(10)`	20 195	84	5.55	3328	84	1.19	74	0.14	84	0.22
`lastzero(15)`	4 799 353	84	2025.86	147 456	274	89.99	74	7.69	84	8.62
`lastzero(20)`	⏱	⏱	⏱	OOM	OOM	OOM	75	361.76	84	398.28
`exp-mem(7)`	10 080	83	1.78	10 080	98	1.99	227	0.80	84	1.17
`exp-mem(8)`	80 640	83	15.04	OOM	OOM	OOM	OOM	OOM	84	10.19
`exp-mem(9)`	725 760	83	152.81	OOM	OOM	OOM	OOM	OOM	84	101.69
`exp-mem2(4)`	21 386	83	3.78	20 736	84	3.94	86	1.09	84	1.55
`exp-mem2(5)`	746 378	83	164.01	705 600	84	157.82	418	45.46	84	59.88
`exp-mem2(6)`	36 044 140	83	8673.37	33 177 600	84	9298.98	OOM	OOM	84	3204.28

such weak memory benchmarks where Nidhugg/SC consumes an exponential amount of memory as well.

As we show in § 10.2.2.1, GenMC (with TruSt) is always exponentially faster than nonoptimal DPOR implementations, and exponentially "lighter" (in terms of memory consumption) than optimal DPOR implementations. We also observe that the overhead that GenMC faces due to TruSt is insignificant[162]. In addition, in cases where optimal DPORs consume an exponential amount of memory, GenMC with TruSt can be exponentially faster, since its computations do not become memory-bound.

10.2.2.1 *TruSt vs State-of-the-Art*

Let us begin with some synthetic benchmarks that highlight the differences between the different DPOR algorithms (cf. table 10.7).

The first benchmark in table 10.7, `lastzero` from Abdulla et al.[163], is a prime example of (a) the memory/time tradeoff that DPOR algorithms have to face, and (b) the different backtracking strategies that DPOR algorithms employ. As can be seen in table 10.7, Nidhugg/optimal is exponentially faster than Nidhugg/source for both `lastzero(10)` and `lastzero(15)`, as it explores exponentially fewer executions than Nidhugg/source. That speed however, comes at a price: Nidhugg/optimal also consumes an exponential amount of memory, which makes it exceed the memory limit when the number of threads is increased to 20. GenMC$_{\text{naive}}$ and GenMC, on the other hand, both consume much less memory. (As already explained, GenMC is somewhat slower than GenMC$_{\text{naive}}$ due to it being based on a different tool version that the naive implementation.)

As the next two benchmarks show, however, it is not at all hard for GenMC$_{\text{naive}}$ to exceed the memory limit as well. The `exp-mem` benchmark (adapted from Abdulla et al.[164]) is another example where optimal DPORs explore an exponential amount of memory. Here, both GenMC$_{\text{naive}}$ and Nidhugg/optimal quickly exceed the memory limit, while GenMC verifies all variants of this program with essentially the

[162] *GenMC is somewhat slower than its naive version in some benchmarks due to the two implementations being based on different versions of the tool.*

[163] *"Source sets: A foundation for optimal dynamic partial order reduction" [Abd+17]*

[164] *"Source sets: A foundation for optimal dynamic partial order reduction" [Abd+17]*

Table 10.8: Weak memory benchmarks (24h timeout)

		GenMC$_{\text{naive}}$		GenMC	
	Executions	*Mem.*	*Time*	*Mem.*	*Time*
`lamport(3)`	6690	75	0.36	84	0.40
`lamport(4)`	12 163 630	433	1121.41	84	719.75
`mcs_spinlock(4)`	15 264	107	6.80	84	0.27
`mcs_spinlock(5)`	964 320	OOM	OOM	85	5.52
`ttaslock(3)`	162	73	0.02	85	0.04
`ttaslock(4)`	20 760	82	2.41	85	0.16
`ttaslock(5)`	14 457 720	OOM	OOM	84	4.08
`mpmc_queue(3)`	143	75	0.14	85	0.20
`mpmc_queue(4)`	31 880	86	216.39	85	152.96
`mpmc_queue(5)`	1 270 584	OOM	OOM	85	16 377.38
`seqlock-atomic(5)`	1500	135	16.68	85	8.02
`seqlock-atomic(6)`	16 200	OOM	OOM	85	140.25
`seqlock-atomic(7)`	185 220	OOM	OOM	85	2437.31

same memory consumption. For `exp-mem2` (a variant of `exp-mem` with no RMW operations), GenMC$_{\text{naive}}$ consumes an exponential amount of memory for 6 threads and above, while GenMC maintains the same memory consumption, as expected.The reason why Nidhugg/optimal also maintains low memory consumption could be that the particular exploration order the tool chooses, does not lead it to consume exponential memory for this particular benchmark.

Moving on to the second part of this evaluation (cf. table 10.8), we observe that the same high-level claims and trends we made for table 10.7 also extend to realistic benchmarks. As can be seen in table 10.8, as the number of threads increases, GenMC$_{\text{naive}}$ quickly exceeds the memory limit, while GenMC's memory consumption basically remains constant. In addition, even though GenMC$_{\text{naive}}$ is usually slightly faster than GenMC, whenever an exploration becomes memory-bound (e.g., `seqlock`), GenMC outperforms GenMC$_{\text{naive}}$ in terms of both memory and time.

In order to stress the importance of polynomial memory consumption, we also mention in passing that there are cases[165] where GenMC$_{\text{naive}}$ consumed all available RAM, while GenMC managed to terminate with more or less constant memory consumption.

[165] *See e.g., the* `mutex_musl` *benchmark in* [*Kok+22a*].

10.2.3 *Synchronization Barriers Optimization*

We now change gears and we compare the optimizations of §7 against a baseline version of GenMC without them.

We do not include other tools in our comparison as 1) a comprehensive comparison has been presented in the previous sections, and 2) most other tools do not offer built-in support for such optimizations anyway (and would thus yield similar results to the baseline GenMC encoding).

Instead, as mentioned in §10.2, we set out to show that GenMC's optimizations yield exponential benefits compared to the baseline GenMC implementation, while at the same time impose zero overhead.

We do so by evaluating the effectiveness of BAM on a variety of synthetic benchmarks, ranging from simple benchmarks containing a single rendezvous round with no additional computation to benchmarks that involve multiple rendezvous rounds.

The results are reported in tables 10.9 and 10.10. As expected, BAM achieves exponential gains over GenMC for all these benchmarks, and scales very well to larger programs. By contrast, the baseline GenMC implementation frequently times out, especially on benchmarks with multiple rendezvous rounds.

Let us first focus on table 10.9. Starting with `barrier`, we see that GenMC explores exponentially more executions than BAM, most of which correspond to blocked executions. Indeed, as explained in §7.1.1, since the `barrier_wait` operations are considered conflicting, GenMC explores an exponential number of executions for this benchmark. In fact, GenMC explores $(N!)^2$ executions for `barrier(N)`, of which $(N!)^2 - N!$ are blocked.

These numbers might come off as a surprise at first, since it would suffice for GenMC to explore precisely $(N!)$ executions, and no blocked executions. The discrepancy is due to the modeling of `barrier_wait` calls. As described in §7.1.1 and Fig. 7.1, each `barrier_wait` comprises an RMW operation followed by a read of the barrier value, which is later used in an `assume` statement. This second read, however, has another $N!$ consistent `rf` options, which GenMC subsequently has to explore. And at this point, one may wonder: isn't it possible to pack the `assume` statement into the atomic block, and use the value already read for b for the `assume`? Unfortunately, the answer is no. The second read statement is necessary under most weak memory models to ensure synchronization between the events before and after the barrier rendezvous.

The differences between the unoptimized and the BAM version are magnified once we consider benchmarks with multiple rendezvous rounds. Starting with 4 threads, The unoptimized explores 5 orders of magnitude more executions than BAM for `barrier2`, and 6 orders of magnitude more for `barrier3`. As the number of threads increases,

Table 10.9: Synthetic benchmarks with only barrier operations

	Unoptimized Barriers			BAM		
	Executions	*Blocked*	*Time*	*Executions*	*Blocked*	*Time*
`barrier(4)`	24	552	0.06	1	0	0.04
`barrier(5)`	120	14 280	0.29	1	0	0.04
`barrier(6)`	720	517 680	9.32	1	0	0.05
`barrier2(4)`	576	36 816	0.65	1	0	0.04
`barrier2(5)`	14 400	5 156 880	87.43	1	0	0.05
`barrier2(6)`	🕒	🕒	🕒	1	0	0.06
`barrier3(4)`	13 824	907 152	18.99	1	0	0.04
`barrier3(5)`	🕒	🕒	🕒	1	0	0.04
`barrier3(6)`	🕒	🕒	🕒	1	0	0.04

`barrier(n)`: N threads rendezvous at a barrier.

`barrier2(n)`: N threads rendezvous twice at a barrier.

`barrier3(n)`: N threads rendezvous thrice at a barrier.

the performance gap between unoptimized and BAM increases even more, despite the fact that most of the executions that the unoptimized version explores are blocked; as it turns out, the cost of enumerating blocked executions quickly becomes exorbitant.

We move on to table 10.10, which contains some typical use cases of barriers. The observations here are similar to the ones made for table 10.9. The simplest case is that of `barrier-det` that includes a single rendezvous round and only local computations. The unoptimized version scales similarly to the `barrier` benchmark, but takes much more time because of the higher cost per execution. By contrast, the number of threads has a negligible effect to BAM's execution time.

The other three benchmarks use multiple rendezvous rounds to synchronize some computations, while still maintaining a high cost per execution. As expected, this makes the unoptimized version quickly time out. In addition, observe that in the case of `barrier-lock` and `barrier-count` barriers are used to synchronize computations that have additional sources for an exponential number of executions. As the state space of these benchmarks is large to begin with (even disregarding barriers), The unoptimized version quickly exceeds the time limit, while BAM is able to scale to a larger number of threads. We note that the blocked executions that BAM explores in `barrier-lock` are not due to barriers, but rather due to spinloops that can block in the lock implementation under test.

We end this section with a remark on scalability. While it can be argued that scaling up to a large number of threads is unimportant (since e.g., these benchmarks are symmetric), this is not always the case. Often, concurrent implementations tune their behavior depend-

Table 10.10: Benchmarks with realistic barrier use cases

	Unoptimized Barriers			BAM		
	Executions	*Blocked*	*Time*	*Executions*	*Blocked*	*Time*
`barrier-det(3)`	6	30	1.88	1	0	0.81
`barrier-det(4)`	24	552	8.46	1	0	0.81
`barrier-det(5)`	120	14 280	97.46	1	0	0.81
`barrier-transc(3)`	⏱	⏱	⏱	1	0	0.13
`barrier-transc(4)`	⏱	⏱	⏱	1	0	0.13
`barrier-transc(5)`	⏱	⏱	⏱	1	0	0.13
`barrier-lock(3)`	1296	4383	0.94	36	54	0.10
`barrier-lock(4)`	331 776	2 165 299	479.07	576	966	3.00
`barrier-lock(5)`	⏱	⏱	⏱	14 400	29 342	55.79
`barrier-count(3)`	55 296	715 878	100.08	64	0	0.08
`barrier-count(4)`	⏱	⏱	⏱	4992	0	2.57
`barrier-count(5)`	⏱	⏱	⏱	2 276 352	0	28min

`barrier-det(n)`: Given a matrix M, calculates the determinant of M^4. The calculation of M^4 is split among N threads, which rendezvous after calculating M^2.

`barrier-transc(n)`: N threads calculate the transitive closure of a matrix via a fixpoint. They rendezvous twice per fixpoint iteration.

`barrier-lock(n)`: N threads test a simple lock implementation: after they rendezvous at a barrier, the threads concurrently attempt to enter their critical section, and mutual exclusion is checked.

`barrier-count(n)`: Contains N threads, with each thread i waiting at barriers b_k, where $i \leq k \leq N$. Counts the number of threads getting through at each round.

ing on the number of threads spawned, and concurrency bugs cannot be manifested with a few threads. Being able to verify programs that employ a large number of threads can therefore be crucial.

10.2.4 *Spinloop Optimization*

We continue by evaluating the effectiveness of SAVER's optimizations on a variety of benchmarks[166]. The evaluation comprises two distinct parts, with the first part concerning the overall performance of SAVER in a real-world setting, and the second part evaluating the effectiveness of employing individual transformations.

We observe that enabling SAVER typically leads to *exponential gains* in real-world benchmarks with spinloops. Key to these gains are SAVER's dynamic checks for spinloop purity and/or validity of ZNE spinloops, as well as the bisimilarity-based reduction of CFGs, which enables more spinloops to be bounded.

OVERALL PERFORMANCE We start by applying SAVER on some challenging data structures utilizing weak-memory atomics that we har-

[166] *As with BAM, we only compare against a baseline GENMC implementation; see §10.2.3.*

vested from the literature, including all data-structure benchmarks from GenMC's test suite. The results can be seen in table 10.11.

Table 10.11: Real-world benchmarks

	Unoptimized Spinloops			SAVer			
	Executions	*Blocked*	*Time*	*Executions*	*Blocked*	*Time*	*Trans*
`mcslock(3)`	7128	0	0.56	66	124	0.06	`S`
`mcslock(4)`	4 055 612	0	411.60	1080	2261	0.32	`S`
`qspinlock(3)`	12	0	0.05	6	0	0.04	`S`
`qspinlock(3)`	13 764	0	2.18	258	427	0.17	`S`
`seqlock(3)`	430	456	0.22	9	54	0.06	`S`
`seqlock(4)`	⏱	⏱	⏱	88	1393	0.21	`S`
`mpmc-queue(3)`	1 232 884	268 476	234.31	166	679	0.21	`S, D`
`mpmc-queue(4)`	⏱	⏱	⏱	39 706	795 718	104.78	`S, D`
`linuxrwlock(3)`	⏱	⏱	⏱	24	59	0.06	`B, S, Z`
`linuxrwlock(4)`	⏱	⏱	⏱	1060	5518	0.81	`B, S, Z`
`chase-lev(5)`	41 816	0	1.86	12 916	10 264	0.88	`S`
`chase-lev(6)`	1 341 250	0	66.03	115 134	219 640	12.03	`S`
`treiber-stack(3)`	22	0	0.05	8	8	0.05	`S, D`
`treiber-stack(4)`	34 968	0	5.98	236	1463	0.26	`S, D`
`mutex(2)`	18	0	0.07	12	1	0.04	`S, D`
`mutex(3)`	59 760	0	6.57	7086	483	0.79	`S, D`
`ttaslock(3)`	11 031	0	0.81	36	58	0.06	`S, D`
`ttaslock(4)`	⏱	⏱	⏱	576	1590	0.16	`S, D`
`twalock(3)`	1338	0	0.43	96	31	0.34	`S`
`twalock(4)`	1 018 872	0	130.74	6144	2877	1.29	`S`
`ms-queue(3)`	1389	0	0.50	75	274	0.13	`L, S, D`
`ms-queue(4)`	⏱	⏱	⏱	10 662	159 374	36.73	`L, S, D`
`sc-gather(3`	7560	11 340	4.71	90	181	0.08	`Z`
`sc-gather(4)`	1 247 400	1 995 840	1210.08	2520	4845	0.77	`Z`

The benchmarks of table 10.11 demonstrate that SAVer is effective in a real-world setting, and that SAVer's extensions combined lead to exponential gains. For all these benchmarks, we have used an unroll value of $N + 1$ (where N is the number of threads, shown in parentheses) for both GenMC versions to avoid manually unrolling any loops that spawn threads or initialize thread-local variables. The transformations that SAVer applies are shown on the rightmost column, where `S`, `D`, `Z`, `L`, and `B` stand for spin-assume, dynamic-spin-assume, zne-assume, loop-rotation, and bisimilarity, respectively.

SAVer is able to employ its transformations (even if only partially) on all the benchmarks andthis leads to a huge reduction in verification time over the unoptimized version. That is, even if in some cases, SAVer only applies spin-assume/zne-assume in some of the data-structure's methods, or even in some paths of a particular method, SAVer is still orders of magnitude faster than GenMC. Concretely, for all benchmarks, SAVer is able to transform at least one of the spinloops completely into an `assume` statement. For `seqlock`, SAVer reduces the read paths; for `mpmc-queue`, it reduces both the enqueue and dequeue methods; for `linuxrwlocks`, the read_lock and write_lock methods, for `chase-lev`, the steal method; for `treiber-stack`, the pop method; for `mutex`, `ttaslock`, and `twalock`, various spinloops in the lock and unlock paths; for `ms-queue`, the enqueue and dequeue methods; and for `scgather` the check method.

DYNAMIC PURITY/UNOBSERVABILITY CHECKS As it can be seen from table 10.11, in more than half of the benchmarks, SAVer checked the purity of a spinloop or the non-observability of its intermediate effects dynamically. Dynamic checking proves useful for three cases.

First, in cases like `ms-queue`, plain spin-assume is not enough to fully transform some spinloop iterations into `assume` statements because they contain possibly succeeding CAS operations. Recall from Fig. 7.8 that the second loopy path of the simplified dequeue implementation is not effect-free. By adding a dynamic check to the relevant backedge, SAVer only considers iterations where the CAS actually succeeds, thus greatly reducing the state space of the program.

Second, in other cases (e.g., `mutex` and `ttaslock`), dynamic-spin-assume is necessary as spinloops contain function calls possibly containing side-effects. As it is difficult to determine statically whether these side-effects will actually take place in the particular calling context, the check is deferred to runtime.

Third, the unobservability checks both for initialization writes in failed CAS loops (e.g., `treiber-stack`)[167]) and for ZNE loops (`linuxrwlocks` and `scgather`) cannot always be easily performed statically with sufficient precision. As such, performing them dynamically is the only viable option.

LOOP ROTATION AND BISIMILARITY REDUCTION Loop rotation and bisimilarity reduction are similarly important in some real-world test cases. Even though they do not yield any performance improvements on their own, they are instrumental in making the spin-assume and zne-assume transformations applicable to more complex cases. Specifically, in benchmarks like `ms-queue` and `linuxrwlocks`, spin-assume and zne-assume are not applicable without loop rotation and bisimilarity respectively. And, in fact, these are not the only cases that we have encountered; there are many ways to rewrite the same benchmarks so that they also require bisimilarity and/or loop rotation, thus rendering these transformations a necessity, as opposed to an enhancement.

As a further demonstration of their usefulness, we consider two synthetic test cases inspired by the LOOP-PEEL example. In these tests, some threads repeatedly write to a shared variable, which is read by readers that employ schemes similar to LOOP-PEEL's thread II. As explained in

[167] *If e.g., all nodes are statically allocated in a pool, and not at each push invocation.*

Table 10.12: Benefits of bisimilarity

		SAVER$_{\setminus B}$		SAVER	
	Executions	*Blocked*	*Time*	*Blocked*	*Time*
`ws+r-peeled(3)`	6720	10 080	0.82	2345	0.44
`ws+r-peeled(4)`	1 848 000	2 956 800	293.30	667 595	133.85
`w+rs-peeled(3)`	1	1848	0.36	3	0.06
`w+rs-peeled(4)`	1	79 506	13.00	7	0.05

§7.2.4, spin-assume is not directly applicable in such cases because the live variables of the header are redefined within the loop. Thus, we used an unroll value of 3, and manually unrolled any loops utilized by the writer threads. For these benchmarks, we used two SAVER versions: the default version that employs both bisimilarity and loop rotation (SAVER), and a version where bisimilarity is disabled (SAVER$_{\setminus B}$)[168]. The results can be seen in table 10.12.

With bisimilarity reduction, SAVER transforms the spinloops into `assume` statements and only explores very few executions, since only a handful of value combinations satisfy the `assumes`. . On the other hand, SAVER$_{\setminus B}$ explores a much larger number of executions, which affects the verification time. These results highlight the necessity of being resilient against small syntactic variations as, even if a single read is not taken into account when transforming a spinloop into an `assume`, the state space might grow exponentially.

10.2.5 *Blocking Prevention*

Next, we show that load annotations and the detection of futile explorations can be effective in practice by ruling out blocked executions, though they may not always offer exponential benefits.

Concretely, we run GENMC against a version that does not rule out blocked executions with annotations/futile executions (henceforth No Blocking Prevention), and measure the number of encountered blocked executions on a variety of benchmarks.

In table 10.13, we distinguish two types of benchmarks: (synthetic) benchmarks like `IRIW-iter` and `assume-exp` represent cases where the difference in the number of blocked executions between the two versions explored is exponential (e.g., three orders of magnitude in `IRIW-iter`), while synchronization algorithms (without RMWs) like `szymanski` and `lamport-sc` represent cases where the number of blocked executions does not reduce significantly (yet the difference is still noticeable).

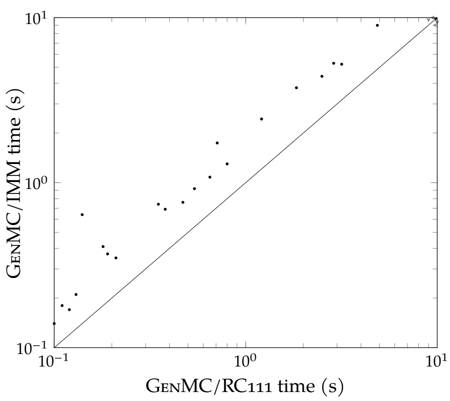
[168] *GENMC does not perform loop rotation by default.*

Table 10.13: Benefits of blocking prevention

		No Blocking Prevention		GenMC	
	Executions	*Blocked*	*Time*	*Blocked*	*Time*
`IRIW-iter`	1	7224	0.10	3	0.04
`assume-exp(2)`	6561	275 562	7.54	0	0.22
`assume-exp(3)`	65 536	2 752 512	75.65	0	1.73
`assume-exp(4)`	390 625	16 406 250	444.64	0	10.11
`lamport-sc(2)`	16	16	0.05	7	0.04
`lamport-sc(3)`	9216	11 525	1.78	4368	1.70
`szymanski(2)`	78	1913	0.28	1205	0.25
`szymanski(3)`	1068	26 883	4.99	16 935	4.51

The ratio between the total number of executions explored, however, does not always translate to an equal ratio in verification time, since the cost of exploring a blocked execution depends very much on the program structure and the exploration order. For `assume` statements encountered near the end of a graph (meaning that the model checker does not have to execute many instructions after the `assume`), exploring blocked executions incurs negligible overhead, especially since GenMC avoids rerunning any blocked or terminated threads. If, however, many accesses are encountered after a thread blocks due to an `assume` (or many effect-free accesses mediate between the annotated load and the `assume`), then the model checker will have to pay the penalty of exploring these accesses (and possible subexplorations they induce).

Following the above, for the benchmarks we used, the time difference between the two GenMC versions is proportional to the exploration cost per failed assume. As far as the synthetic benchmarks are concerned, for `IRIW-iter` the difference in running time is minor as the overhead per failed assume is low, while for `assume-exp` the time difference is substantial, as each failed assume leads to an exponential number of (also blocked) subexplorations. In fact, for `assume-exp`, the only `assume` of the program has only 42 possible reads-from options, from which only 1 is of interest. However, each time this `assume` fails, other threads can be scheduled, and these threads yield additional subexplorations, leading to a significant increase in the unoptimized version's running time. As far as the synchronization algorithms are concerned, the time difference between the two versions is not significant (approximately 7% difference for `lamport-sc` and 16% difference for `szymanski`), as all threads of these programs contain `assume` statements, and thus none of them can induce exponentially many subexplorations.



Dep.tracking
geo-mean: 1.78
max: 4.64

Algorithm
geo-mean: 1.05
max: 1.23

Total
geo-mean: 1.89
max: 4.57

Figure 10.1: Overhead of dependency tracking

10.2.6 *Tracking Dependencies*

In all the previous comparisons, we only saw how GenMC performs under RC11, a non-dependency-tracking model. To measure the overhead of dependency tracking, we ran GenMC both under RC11 and under IMM on the tool's default test suite (containing over 200 synthetic and non-synthetic tests), as well as on the benchmarks of §10.2.1.1. We used a threshold value of (0.1s) to exclude benchmarks that terminate almost instantly.

The results are shown in Fig. 10.1. As can be seen, GenMC/IMM has a standard overhead over GenMC/RC11, which is acceptable given the coverage of more behaviors.

Note, however, that this overhead is not due to the difference in complexity among the two models, but should rather be attributed to the dynamic calculation of dependencies. Indeed, to measure the overhead of calculating dependencies, we also replaced the interpreter of GenMC/RC11 with the dependency-tracking one. Of the average 89% overhead of GenMC/IMM over GenMC/RC11, 78% is due to the calculation of dependencies. The model itself has only 5% average (23% max) overhead over the instrumented version of GenMC that needlessly calculates dependencies.

10.2.7 *Parallelization*

In §6.3.3, we saw how GenMC equipped with TruSt is embarrassingly parallelizable. To demonstrate this point, let us now see how GenMC scales in practice as we increase the number of threads on the multicore architecture of 16 physical cores (32 logical cores with hyperthreading) of §10.2.

Figure 10.2 plots the speedup obtained by running GenMC over single-threaded performance (y-axis) against the number of worker threads

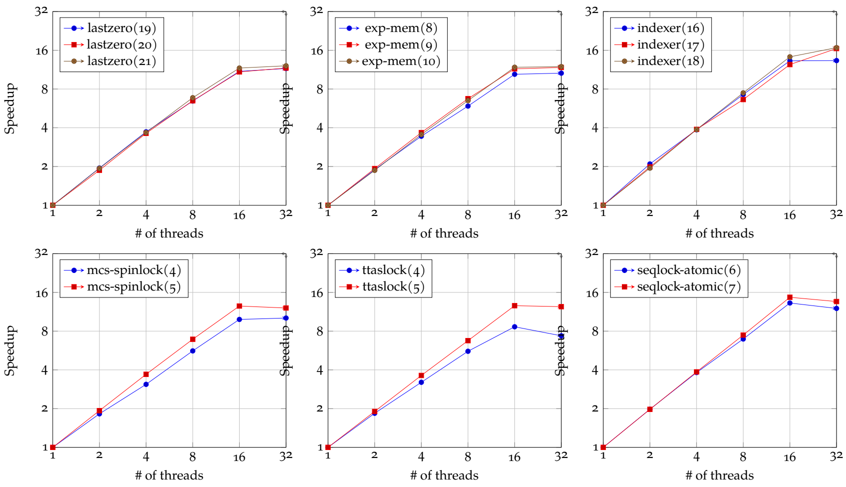


Figure 10.2: GenMC scalability on 16 physical (32 logical) cores

employed (x-axis). Both axes are in logarithmic scale, so that perfect scaling would correspond to the diagonal line.

As can be seen, GenMC achieves an almost linear speedup when scaling up to 16 threads, and then its performance flattens out and even deteriorates as we add more worker threads than physical cores on the machine. The speedup obtained by using up to 16 worker threads dramatically decreases the running time for some very intensive benchmarks. As an example, consider the `seqlock-atomic` benchmark: while the sequential version of GenMC needs more than 2.5 hours to terminate for this benchmark, with 16 cores it terminates in 10 minutes.

There are three additional takeaways from Fig. 10.2. First, GenMC's speedup is almost, but not exactly, linear, up to 16 cores. This is expected and in line with most results of parallel algorithms. Indeed, even though the design of GenMC allows for explorations to proceed completely in parallel, there is no guarantee that all different subexplorations will have the same "depth". Specifically, each thread necessarily has some small "idling" period where it tries to pick up work from other threads, thus limiting the scalability of GenMC.

Second, scalability flattens at 16 worker threads even though our machine does support hyperthreading and thus is deemed to have 32 logical cores. Again, this is expected because our computations are CPU/memory-bound (as opposed to I/O-bound) and thus hyperthreading does not succeed in running more tasks in parallel.

Third, GenMC scales better when either the state space of a benchmark or the cost per execution becomes larger. For instance, GenMC scales generally better for the last four benchmarks of Fig. 10.2 than for the first two of the same figure: this is because the per-execution cost of the weak-memory benchmarks is larger, which in turn means that the different threads have more work to do before trying to pick up their next tasks. In addition, observe that GenMC scales better for each benchmark as we increase the parameter controlling its state space. Again, this is expected as a larger state space entails more executions, which in turn implies that each of GenMC's worker threads will have more work to do.

10.3 THE INTERACTION BETWEEN KATER AND GENMC

Finally, let us evaluate the performance of Kater-generated consistency checks against the (previously) default ones.

In order to do so, we subsequently answer the following questions:

- How well do the Kater-generated consistency checks perform against the baseline GenMC implementation?
- How do the Kater-generated checks scale as the memory model becomes more complex?

As our evaluation demonstrates, Kater-generated consistency checks induce an average overhead of 30-40% over the (previous) default GenMC implementation (that in many cases skips consistency checks altogether), but greatly outperform the default implementation in cases where GenMC has to check full consistency.

As expected, checking consistency becomes more expensive as the memory model becomes more complex.

In all our tests below, we ran the default GenMC implementation under the RC11 memory model.

GENMC'S DEFAULT CONSISTENCY CHECKS It is worthwhile to discuss the default consistency checking routines previously employed by GenMC, as they encompass a few optimizations.

A first optimization example is that the default infrastructure does not check full consistency at each step. Indeed, precisely because checking consistency can be expensive, it only checks for full consistency when an error is detected, even though not checking for full consistency at every step means that in certain cases orders of magnitude more executions than necessary might be explored[169].

As another example, if there are only a handful of SC accesses, the default infrastructure only checks for consistency violations caused by these accesses, and does not take into account the full graph[170]. If there are no such accesses, full consistency does not need to be checked. This last optimization can have a large impact given that the complexity of the default consistency checking routines is $O(n^3)$.

10.3.1 *Default Checks vs Kater-generated*

To answer the first question we ran both versions on GenMC's test suite, excluding tests for which either version finished in less than 0.10

[169] *Usually this is not the case.*

[170] *Recall that RC11's consistency predicate is mostly concerned with SC accesses.*

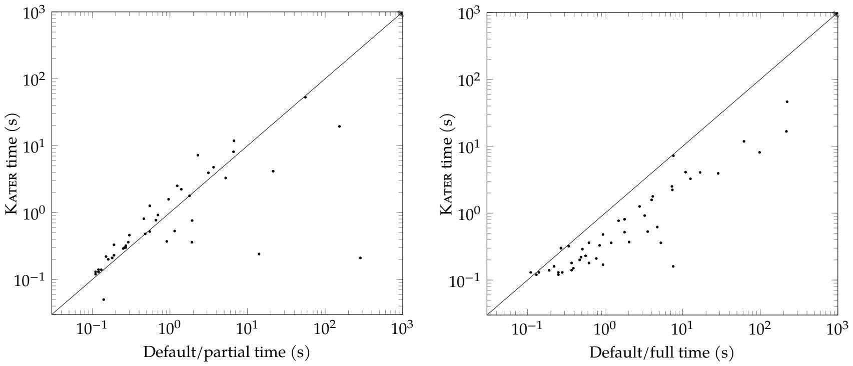


Figure 10.3: Default vs kater-generated consistency checks

seconds. The results can be seen in Fig. 10.3 (left). Kater-generated checks are on average 30-40% slower than the default ones, though in certain cases they outperform the default ones by a large factor.

While this may come off as a surprise at first, the reason (as explained above) is that GenMC does not check full consistency at each step, and is therefore generally faster. Of course, not checking for full consistency at every step means that in certain cases the default checks can explore orders of magnitude more executions than necessary, and therefore run slower than the Kater-generated ones.

Now, if we force the default implementation to check full consistency at each program step, the results change dramatically (cf. Fig. 10.3, right). Kater-generated checks are rarely slower than the default ones (on average it is two times faster), while in many cases it is an order of magnitude faster, thereby demonstrating the efficiency of the automatically generated code.

Still, one may wonder why Kater-generated checks are not always orders of magnitude faster than the default ones given that Kater's consistency checks are linear in the size of the product automaton of the memory model and the graph. The answer is twofold. First, most of the test-suite benchmarks solely utilize weakly ordered accesses, and thus do not even require checking full RC11 consistency. The default (handwritten) consistency checking mechanism is able to leverage the non-existence of SC accesses and optimize away the checks, while Kater-generated checks always performs certain calculations. Second, many of these tests have small graphs and so the worse complexity of the default implementation does not show.

To better evaluate how efficient the Kater-generated checks are, we conducted another case study in benchmarks containing many SC accesses. Such benchmarks do require extensive consistency checks under RC11, and give us a clearer picture of how Kater's checks compare against the built-in ones.

The results are summarized in table 10.14 (columns Default/partial, Default/full, and Kater-RC11). In the first four benchmarks, not check-

Table 10.14: SC benchmarks

	Default/partial			Default/full	Kater/SC	Kater/TSO	Kater/RC11
	Executions	*Time*	*Executions*	*Time*	*Time*	*Time*	*Time*
`szymanski(1)`	384	0.07	6	0.01	0.01	0.01	0.02
`szymanski(2)`	1 115 118	266.79	78	0.94	0.09	0.11	0.17
`szymanski(3)`	🕒	🕒	1068	37.86	1.36	1.89	3.34
`peterson(2)`	1848	0.11	48	0.04	0.02	0.02	0.03
`peterson(3)`	222 956	13.20	588	0.62	0.08	0.10	0.18
`peterson(4)`	🕒	🕒	7360	12.63	0.97	1.40	2.80
`parker(1)`	232	0.04	54	0.05	0.02	0.02	0.04
`parker(2)`	139 425	19.11	6701	10.93	2.30	2.76	4.13
`dekker_f(2)`	242	0.04	71	0.10	0.02	0.03	0.04
`dekker_f(3)`	13 789	1.77	1344	4.71	0.26	0.29	0.76
`dekker_f(4)`	906 142	147.89	26 797	216.61	5.59	6.61	19.41
`fib_bench(4)`	34 205	0.29	19 605	1.22	0.19	0.24	0.36
`fib_bench(5)`	525 630	3.67	218 243	16.92	1.97	2.56	4.08
`fib_bench(6)`	8 149 079	56.49	2 363 803	220.18	21.01	24.62	52.94
`lamport(2)`	28	0.03	16	0.02	0.01	0.02	0.02
`lamport(3)`	54 851	6.26	9216	12.57	1.06	1.27	3.26

ing for full consistency leads Default/partial to perform poorly compared to Default/full and Kater-RC11, as it explores orders of magnitude more executions than necessary. In the last two benchmarks, on the other hand, where Default/partial does not explore a lot of redundant executions, Default/full and Kater-RC11 are slower due to the complexity induced by the consistency checks. In all cases, however, Kater-RC11 outperforms Default/full, and is also competitive against Default/partial, even when the latter is faster than Default/full.

We end this part of our evaluation with two observations. First, in `dekker_f`, Default/full has comparable performance to Default/partial, even though it explores two orders of magnitude fewer executions. Kater-RC11, on the other hand, outperforms Default/partial by a much larger margin, thereby allowing us to observe first-hand the difference in the computational complexity between the checks of the two tools. Second, in `lamport`, something similar happens for Kater-RC11, which has comparable performance to Default/partial even though it explores fewer executions. In this case, however, Kater-RC11 does not explore exponentially fewer executions than Default/partial. In addition, when the cost per execution is small (which is the case for `lamport`), the difference in the running times becomes less pronounced.

10.3.2 *Consistency Checking under Different Models*

To evaluate how well Kater scales when the memory model becomes more complex, we added support for two models that GenMC does not have a specialized (handcrafted) consistency checker (SC and TSO), and compared Kater-RC11 against Kater-SC and Kater-TSO in the computationally expensive benchmarks of table 10.14 (columns Kater-SC, Kater-TSO and Kater-RC11).

As expected, as the memory model becomes more complex, Kater becomes slower. Both Kater-SC and Kater-TSO are much faster than Kater-RC11, since the generated automata for these models comprise just one state, in contrast to the one for RC11, which comprises twelve states. However, even though the automata for SC and TSO have the same number of states, checking for SC is faster than TSO since the transitions in the TSO automaton are composite (i.e., they contain both predicates and relations; see Fig. 9.8 and §9.3.2).

Part IV

CONCLUSION

11 RELATED WORK

In this chapter, we position Kater (§11.1) and GenMC (§11.2) relative to other verification techniques that support weak memory consistency. As both Kater and GenMC are automated tools, we only compare against other automated reasoning tools.

11.1 METATHEORY

As far as metatheoretic properties are concerned, most existing works proved such properties for specific (pairs of) memory models with manual proof efforts[171]. Many of these results were not even mechanized, which led to the publication of some incorrect results[172].

There do exist a handful of approaches for automatically checking metatheoretic properties of weak memory models. To the best of our knowledge, Mador-Haim, Alur, and Martin[173] first considered the problem of comparing memory models automatically, but used the rather naive technique of exhaustively generating all litmus tests up to a bounded size. Mador-Haim, Alur, and Martin[174] later showed that a fairly restricted class of memory models enjoyed a small model property and thus checking for whether a memory model is weaker than another is decidable if both models belong to that very restricted class, which is sufficient for expressing SC and TSO, but not Power, Arm or C11.

More recently, Wickerson et al.[175] developed MemAlloy, a tool that performs an incomplete bounded search through possible litmus test skeletons to distinguish between memory models and to validate correctness of compiler mappings and optimizations. MemSynth[176] is a synthesis-based tool that uses SMT-solvers in its backend to answer similar queries about memory models as MemAlloy does, and additionally can generate memory model definitions that match a given set of litmus test outcomes and a sketch of the model.

Note that these two techniquesare *not* sound. When, for example, checking for inclusion between weak memory model definitions, they search for counterexamples up to a given bounded size, and can thus provide no formal guarantees about whether the property holds.

11.2 VERIFICATION

We can broadly classify verification approaches into three categories: (a) explicit-state model checkers, (b) enumerative approaches (e.g., SMC, DPOR), and (c) SMT-based approaches, with the latter two cat-

[171] *To name a few:* [Bat+12; Flu+17; Pul+19; PLV19; Lah+17; LV16; LGV16; DSM18; Vaf+15].

[172] *"Synchronising C/C++ and POWER"* [Sar+12]

[173] *"Generating Litmus Tests for Contrasting Memory Consistency Models"* [MAM10]

[174] *"Litmus tests for comparing memory consistency models: how long do they need to be?"* [MAM11]

[175] *"Automatically Comparing Memory Consistency Models"* [Wic+17]

[176] *"Synthesizing memory models from framework sketches and Litmus tests"* [BT17]

egories falling into bounded verification. In what follows, we briefly review some representative techniques from each category.

11.2.0.1 *Explicit-State Model Checkers*

Given a program P, explicit-state model checking[177] explores the set of states of that are reachable from the initial state, and determines whether it includes any "bad" state (i.e., one violating a provided specification). To avoid exploring the same state over and over again, explicit-state techniques straightforwardly just record the set of already visited states. The biggest advantage of doing so is that such techniques do not require the executions of the input program to be of bounded length.

Despite this major advantage, there are two main downsides in explicit-state model checking. First, even though such techniques employ some (static) partial order reduction to reduce the number of states to be visited, saving visited states is often impractical due to the memory required to record this set. Second, they do not provide algorithmic support for weak memory. So while it is possible to encode a weak memory model in an operational semantics that records execution graphs, such an encoding blows up the state space.

Prominent explicit-state model checkers include Spin [178], Blast [179], Slam [180], Java Pathfinder [181], LTSMin [182], and DIVINE [183].

11.2.1 *Enumerative Approaches*

Enumerative techniques assume that the program under test has only a finite number of executions, and that all of its executions are of bounded length[184] Depending on their focus, we can categorize these into memory model simulators, which aim for memory-model parametricity (without much focus on the algorithmic component), and DPOR approaches, which aim to reduce the number of executions explored by partitioning them into equivalence classes, and exploring one execution per equivalence class.

11.2.1.1 *Memory Model Simulators*

As far as simulators are concerned, the only tools providing roughly similar functionality to GenMC (in terms of supporting multiple memory models) are Herd [185] and rmem [186].

Herd is a memory model simulator that takes the memory model definition as an argument and allows users to experiment with different consistency predicates on small "litmus test" programs. Unlike GenMC, Herd does not require models to satisfy conditions well-formedness, prefix-closedness and extensibility, and so accepts a wider range of models than GenMC.

[177] *"Automatic verification of finite-state concurrent systems Using temporal logics specification: A practical approach"* [CES83]; *"Specification and verification of concurrent systems in CESAR"* [QS82]

[178] *"The model checker SPIN"* [Hol97]

[179] *"Lazy Abstraction"* [Hen+02]

[180] *"SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft"* [Bal+04]

[181] *"Model Checking JAVA Programs using JAVA PathFinder"* [HP00]

[182] *"LTSmin: High-Performance Language-Independent Model Checking"* [Kan+15]

[183] *"Model Checking of C and C++ with DIVINE 4"* [Bar+17]

[184] *Depending on the technique, the bound may or may not need to be known in advance.*

[185] *"Herding cats: Modelling, simulation, testing, and data mining for weak memory"* [AMT14]

Nevertheless, it follows the simple approach of enumerating all possible executions and filtering them according to the user-supplied consistency predicate, and thus does not scale to larger programs.

rmem is a memory model simulator that, while not fully parametric, supports operational definitions of ARMv8 and RISC-V. Pulte et al. claim that rmem 's current operational definitions are suitable for model checking, as they are much faster than the previous ones [Pul+18], and tools like Herd. That said, rmem does not employ any DPOR techniques, and thus enjoys limited scalability (see §10.2.1).

11.2.1.2 *Dynamic Partial Order Reduction*

After seminal works like Verisoft [187] and Chess [188] paved the way for stateless model checking, there has been a large body of work on SMC and DPOR [189]. A major breakthrough in this line of work was made with Optimal-DPOR by Abdulla et al.[190], who developed the first optimal DPOR algorithm for the Shasha-Snir equivalence under SC (at the cost of exponential memory).

We can broadly classify the more recent works in this area into two main categories depending on their primary focus: (1) techniques that focus on extending DPOR to weak memory consistency (be it in an ad-hoc or a parametric fashion), and (2) techniques that aim to combat the state-space explosion problem by introducing coarser equivalence partitionings. In contrast to GenMC, however, no proposed technique manages to combine (a) being parametric w.r.t. the memory model, (b) operating under both Shasha-Snir and reads-from equivalence, (c) being optimal, and (d) maintaining polynomial memory consumption [191].

dpor & weak memory consistency The first attempt to extend DPOR to weak memory consistency was that of CDSChecker [192], a tool that targets the (original) C/C++11 memory model. Albeit non-optimal, CDSChecker was also the first tool to introduce a constraints-based coherence order similar to `wb`, effectively creating a reads-from equivalence partitioning.

Soon after Optimal-DPOR was presented, Abdulla et al. extended Source-DPOR (a non-optimal version of Optimal-DPOR with polynomial memory consumption) for TSO and PSO[193]. At about the same time, Zhang, Kusano, and Wang developed rInspect [194], a (non-optimal) DPOR algorithm for TSO and PSO.

Abdulla et al. also developed RSMC a stateless model checking algorithm for a part of the POWER/ARM model[195]. Their algorithm is non-optimal, but includes a scheme for systematically deriving operational execution models from declarative ones.

More recently, Abdulla et al. developed Tracer [196], an optimal DPOR

[186] *"Promising-ARM/RISC-V: A simpler and faster operational concurrency model"* [*Pul+19*]

[187] *"Software Model Checking: The VeriSoft Approach"* [*God05*]

[188] *"Finding and reproducing Heisenbugs in concurrent programs"* [*Mus+08*]

[189] *"Dynamic partial-order reduction for model checking software"* [*FG05*]

[190] *"Optimal dynamic partial order reduction"* [*Abd+14*]

[191] *To some degree, the combination between optimality and maintaining polynomial memory consumption has been explored by Quasi-optimal-DPOR; see §11.2.3*

[192] *"CDSChecker: Checking concurrent data structures written with C/C++ atomics"* [*ND13*]

[193] *"Stateless model checking for TSO and PSO"* [*Abd+15*]

[194] *"Dynamic partial order reduction for relaxed memory models"* [*ZKW15*]

[195] *"Stateless model checking for POWER"* [*Abd+16*]

algorithm for the RA model. Tracer is optimal (by sacrificing memory), and operates under the reads-from equivalence.

Using a different approach, Kokologiannakis et al.[197] developed RCMC, a DPOR algorithm for the RC11 model.RCMC is only optimal for a limited fragment of the model (excluding RMWs and SC accesses).

DPOR & EQUIVALENCE PARTITIONINGS Since Optimal-DPOR, a lot of work has been devoted into coarsening the equivalence partitioning under which DPOR operates. In what follows, we attempt to provide a brief (informal) overview of the different equivalence partitionings that have been proposed in the literature.

$$\begin{tikzcd}
 & & \texttt{rf} \arrow[dr, phantom, "\sqsubset" sloped] & & & & \\
\texttt{causal} \arrow[r, phantom, "\sqsubset"] & \texttt{rvf} \arrow[ur, phantom, "\sqsubset" sloped] \arrow[dr, phantom, "\sqsubset" sloped] & & \texttt{dc} \arrow[r, phantom, "\sqsubset"] & \texttt{obs} \arrow[r, phantom, "\sqsubset"] & \texttt{ss} \arrow[r, phantom, "\sqsubset"] & \texttt{naive} \\
 & & \texttt{vc} \arrow[ur, phantom, "\sqsubset" sloped] & & & &
\end{tikzcd}$$

Figure 11.1: A partial order of proposed equivalence partitionings

A graphical overview of these partitionings can be seen in Fig. 11.1, where ⊏ is read as "is-coarser-than". In Fig. 11.1, `naive` corresponds to plain SMC techniques (i.e., no partitioning is performed), `ss` to Shasha-Snir equivalence, and `rf` to reads-from equivalence. For SC, Abdulla et al. presented Optimal-RFSC [198], a version of Optimal-DPOR that provides an optimized consistency checking mechanism under `rf`. Bui et al. present a similarly optimized mechanism for TSO and PSO[199].

`causal`, `rvf`, `vc`, `dc` and `obs` have all been defined in the context of SC and are not straightforwardly sound under weak memory, as they fundamentally assume "*multi-copy atomicity*" (i.e., that writes propagate simultaneously to all other processors).

`causal` was proposed by Şerbănuţă, Chen, and Roşu[200], and effectively distinguishes executions based on the *values* read by the reads. Under `causal`, two (consistent) executions G_1 and G_2 are equivalent if

(1) $G_1.\mathsf{E} = G_2.\mathsf{E}$ (up to permutation), and

(2) for all $r \in G_1.\mathsf{E}$ it is $G_1.\mathtt{val}(r) = G_2.\mathtt{val}(r)$

`causal` is the coarsest equivalence proposed, but has not been used directly in DPOR, but rather only in hybrid approaches (see §11.2.3).

`rvf` is a read-value-from partitioning proposed by Agarwal et al.[201]. It can be a refinement of `causal` where, in addition to reading the same values, reads also have the same causal orderings. Under `rvf`, two executions G_1 and G_2 are deemed equivalent if

(1) they are considered equivalent under `causal`, and

(2) $G_1.\mathtt{porf}|_{\mathsf{R}} = G_2.\mathtt{porf}|_{\mathsf{R}}$

[196] *"Optimal stateless model checking under the release-acquire semantics"* [Abd+18]

[197] *"Effective stateless model checking for C/C++ concurrency"* [Kok+17]

[198] *"Optimal stateless model checking for reads-from equivalence under sequential consistency"* [Abd+19]

[199] *"The Reads-from Equivalence for the TSO and PSO Memory Models"* [Bui+21]

[200] *"Maximal Causal Models for Sequentially Consistent Systems"* [ŞCR13]

[201] *"Stateless Model Checking Under a Reads-Value-From Equivalence"* [Aga+21]

[202] *"Value-Centric Dynamic Partial Order Reduction"* [CPT19]

vc corresponds to a value-centric partitioning[202] which, similarly to rvf, distinguishes executions based on the values read. Two executions G_1 and G_2 are equivalent under vc if they are equivalent under rvf, and further, given a pre-selected thread t:

(1) for all reads r in $G_1.\mathsf{R}_t$, either $r \in rng(G_1.\texttt{rf}_i) \land r \in rng(G_2.\texttt{rf}_i)$, or $r \in rng(G_1.\texttt{rf}_e) \lor r \in rng(G_2.\texttt{rf}_e)$, and

(2) $G_1.\texttt{hb}_{\neq t} = G_2.\texttt{hb}_{\neq t}$

where $G.\texttt{hb}_{\neq t} \triangleq \{\langle e_1, e_2\rangle \mid \langle e_1, e_2\rangle \in G.\texttt{hb} \land \texttt{tid}(e_1) \neq t \land \texttt{tid}(e_2) \neq t\}$

dc corresponds to a data-centric partitioning[203], a coarser version of the reads-from partitioning. Two executions G_1 and G_2 are equivalent under dc if

(1) they are equivalent under rf, and

(2) given a pre-selected thread t, $G_1.\texttt{hb}_{\neq t} = G_2.\texttt{hb}_{\neq t}$

Finally, obs corresponds to a coarsening of ss[204], which distinguishes executions based on a notion of observability In contrast to rf, which defines two executions to be equivalent if each read reads from the same write in both executions, obs is based on observing interference of operations (i.e., whether some write is hb-before a given read r).

There are two things worth noting regarding the different partitionings. First, even though operating under a coarser equivalence partitioning can yield exponential benefits in theory, in practice the precise partitioning used is typically irrelevant[205]: observe that coarser partitionings yield better results if either (a) there are unordered, concurrent writes, or (b) there are multiple (unordered) writes writing the same value, conditions that typically do not manifest in properly synchronized programs[206]. Second, even when the difference does matter, there only exist optimal DPOR algorithms for ss and rf. In turn, a given DPOR algorithm that is non-optimal for a coarse partitioning may be less effective than an optimal algorithm for a finer one.

In a rather different line of work, algorithms like CDPOR [207] and CSDPOR [208] operate under the ss partitioning, but try to reduce the number of explored executions by leveraging *conditional independence*. Under conditional independence, certain revisits will not be considered, depending on the current execution state. To check for such independence, CDPOR uses checks based on pre-generated constraints, which may result in (potentially expensive) state-equivalence checks.

To see this, consider the following program:

$$r := x \;\Big\|\; x := v_1 \;\Big\|\; \dots \;\Big\|\; x := v_N$$

If $v_1 = v_2 = \dots = v_N$, then CDPOR detects that reading from any of the N writes leads to an equivalent state, and hence explores 2 executions (one reading the initial value, and one reading from one of the

[203] *"Data-centric dynamic partial order reduction"* [Cha+17]

[204] *"Optimal dynamic partial order reduction with observers"* [Aro+18]

[205] *At least in a shared-memory setting.*

[206] *Improperly synchronized programs are buggy by definition.*

[207] *"Constrained dynamic partial order reduction"* [Alb+18]

[208] *"Context-sensitive dynamic partial order reduction"* [Alb+17]

writes). On the other hand, GenMC explores $N + 1$ executions (one for each possible value the read can read, assuming `rf`), as it does not take conditional independence into account. However, if $v_1, v_2, \dots, v_N$ are pairwise distinct (and non-zero), CDPOR explores $(N+1)!$ executions (as the N writes all lead to different states), while GenMC still explores $N + 1$ executions.

11.2.2 *SMT-Based Approaches*

In contrast to enumerative approaches, SMT-based approaches encode all executions of a program (together with the memory model) in a SAT/SMT formula, and query a dedicated solver for its satisfiability. In order to generate the SAT/SMT encoding, SMT-based approaches unroll every loop of the program a fixed number of times.

One such approach is CBMC [209], which currently supports the SC, TSO and PSO models. For weak memory, CBMC makes use of the fact[210] that an execution of a program under a weak memory model can be viewed as a partially ordered set, which results in an algorithm aware of the underlying memory model when constructing the SMT/SAT formula.

CheckFence [211], focuses on verifying concurrent data structure implementations under SC and a (custom) Relaxed model. CheckFence encodes the memory model along with entire abstract program executions in SAT, and lazily unrolls loops when the solver indicates that a given loop may exceed the current unroll bound.

Deagle [212] is another SMT-based approach that supports the SC, TSO and PSO models. Unlike CBMC, however, Deagle equips the underlying solver with a dedicated consistency theory for multi-threaded program verification that results in a more efficient encoding, and generally better runtimes.

Other tools like MemSAT [213] and Dartagnan [214] take the memory model as an input along with the given program. Specifically, given a small bounded program and a memory model, MemSAT constructs a formula representing the possible executions of the program according to the model and queries a SAT/SMT solver to see whether a given program outcome is possible. In terms of scalability, MemSAT was only meant to be used for small litmus tests, and so does not scale to larger examples like the ones used in §10.2.1. Dartagnan, on the other hand, uses cleverer encodings into SAT and various optimizations and is thus able to scale reasonably well. We compare the performance of GenMC against that of Dartagnan in §10.2.

11.2.3 *Hybrid Approaches*

Apart from the techniques described above, a lot of work has been devoted into creating *hybrid* techniques, e.g., by combining DPOR and

[209] *"A tool for checking ANSI-C programs"* [CKL04]

[210] *"Partial orders for efficient bounded model checking of concurrent software"* [AKT13]

[211] *"CheckFence: Checking consistency of concurrent data types on relaxed memory models"* [BAM07]

[212] *"Consistency-Preserving Propagation for SMT Solving of Concurrent Program Verification"* [SFH22]

[213] *"MemSAT: checking axiomatic specifications of memory models"* [TVD10]

[214] *"BMC for weak memory models: Relation analysis for compact SMT encodings"* [Gav+19]

SMT-based approaches. Below we survey some representative examples.

Starting with a SAT-driven stateless model checking approach, Huang has proposed Maximal Causality Reduction (MCR)[215] to improve on DPOR, and also extended it to also handle TSO and PSO through a relaxed happens-before modeling. The key insight behind MCR is that a thread's behavior does not depend on the specific stores that the thread's loads take its values from, but rather on the values that these loads read. MCR uses an SMT solver to generate executions in which the loads of a thread read different combinations of values than previously explored executions (effectively operating under `causal`). MCR also assumes multi-copy atomicity.

In a similar spirit to MCR, Demsky and Lam proposed SATCheck [216], a branch-driven stateless model checking approach that aims to cover all branches and all the unknown behaviors of the uninterpreted functions by systematically exploring thread schedules under SC and TSO. In contrast to MCR, SATCheck does not try to generate executions where reads read a different value, but rather where a new (unexplored) direction on a branch is taken, a novel interleaving is visited, or a new input-output relation for an uninterpreted function is learned.

On a different level, unfolding techniques like the proposed by Kähkönen, Saarikivi, and Heljanko and Rodríguez et al. can be considered combinations of explicit-state model checking and DPOR. Such techniques use unfoldings[217] to cache certain visited states, can deal with programs that have infinite traces, and can obtain an even better reduction than optimal DPOR approaches. That said, unfolding-based techniques typically have significantly larger cost per test execution than vanilla DPOR techniques.

Along this line of work, Quasi-optimal-DPOR [218] explores the tradeoff between optimality and polynomial memory consumption. Quasi-optimal-DPOR is able to approximate an optimal DPOR with a user-provided constant k. As the value of k increases, memory consumption also increases in an exponential manner. Although Quasi-optimal DPOR theoretically achieves optimality only with $k = \infty$, it has been shown to practically be optimal, for small values of k.

11.2.4 *The Bounded Verification Landscape*

In table 11.1 we try to give a comprehensive summary of the bounded verification techniques described above. To that end, we compare the various techniques on several fronts:

- the memory models supported
- the equivalence partitioning used
- optimality (i.e., whether each behavior is explored exactly once, and no fruitless exploration is performed)

[215] *"Stateless model checking concurrent programs with maximal causality reduction"* [Hua15]; *"Maximal Causality Reduction for TSO and PSO"* [HH16]

[216] *"SATCheck: SAT-directed stateless model checking for SC and TSO"* [DL15]

[217] *"A Technique of a State Space Search Based on Unfolding"* [McM95]

[218] *"Quasi-optimal partial order reduction"* [Ngu+18]

- polynomial memory consumption
- whether the technique can be parallelized with no data sharing (not considering parallelization of a possible underlying solver)
- whether data-non-determinism is supported

We write "-" when some metric is inapplicable. Enumerative tools that employ declarative semantics in some form or another are highlighted. (Tools that use SAT/SMT solvers use declarative semantics by definition.)

In principle, whether a SAT-based or stateless model checking approach works best depends largely on the program to be verified. SAT-based tools tend to scale better for programs with a large state space and no local computation, while stateless model checkers work best for programs with a relatively small number of distinct program executions, but which may include a lot of arithmetic computations[219] (also see §10.2).

[219] *"Stateless model checking for POWER"* [*Abd+16*]; *"HMC: Model checking for hardware memory models"* [*KV20*]; *"Stateless model checking of the Linux kernel's read–copy update (RCU)"* [*KS19*]

		Models	*Equivalence*	*Optimal*	*Poly Mem*	*No Sharing*	*Nondet Data*
Simulators	RMEM	ARM,POWER	`naive`	✗	✗	✓[†]	✗
	HERD	parametric	`naive`	✗	✗	✓[†]	✗
DPOR	SOURCE-DPOR	SC/TSO/PSO	`ss`	✗	✓	✗[*]	✗
	DC-DPOR	SC	`rf`	✗	✓	✗[*]	✗
	VC-DPOR	SC	`vc`	✗	✓	✗[*]	✗
	RVF-DPOR	SC	`rvf`	✗	✓	✗[*]	✗
	OPTIMAL-DPOR	SC/TSO/PSO	`ss`	✓	✗	✗	✗
	OPTIMAL-OBSERVERS	SC/TSO/PSO	`ss,obs`	✓	✗	✗	✗
	OPTIMAL-RFSC	SC	`rf`	✓	✗	✗	✗
	TRACER	RA	`rf`	✓	✗	✗	✗
	RSMC	ARM[‡]/POWER[‡]	`ss`	✗	✓	✗[*]	✗
	CDSCHECKER	C11	`rf`	✗	✓	✗[*]	✗
	RCMC	RC11	`ss`	✗	✓	✗[*]	✗
SMT/SAT	CBMC	SC/TSO/PSO	-	-	✗	✗	✓
	DEAGLE	SC/TSO/PSO	-	-	✗	✗	✓
	CHECKFENCE	SC/Relaxed	-	-	✗	✗	✓
	DARTAGNAN	parametric	-	-	✗	✗	✓
Hybrid	MCR	SC/TSO/PSO	`causal`	✗	✗	✓	✓[†]
	SATCHECK	SC/TSO	-	✓	✗	✗	✓[†]
	GENMC	parametric	`ss,rf`	✓	✓	✓	✗

Table 11.1: An overview of the bounded verification landscape

[*] Can be parallelized with no sharing by exploring more duplicates.
[†] In principle, yes; not explicitly addressed in the paper.
[‡] Partial support.

12 SUMMARY

This thesis revolved around *automation* of weak memory consistency:

- KATER provides a decision procedure able to automate proofs that were previously only done manually. Leveraging the fact that most memory models are expressible in KAT, it reduces metatheoretic queries about memory models to a language-inclusion problem between regular languages.

- GENMC provides a new foundation for DPOR based on execution graphs. It is the first DPOR framework that is parametric in the choice of the memory model, while also maintaining polynomial memory consumption.

The thesis is also a testament to the elegance of declarative semantics. Being able to (a) specify a variety of models using a single framework, and (b) model each program behavior as an execution graph was crucial in enabling the contributions above.

Besides declarative semantics, key in achieving the contributions above was tool development. Apart from having a real-world impact, building and maintaining tools alongside algorithms was pivotal in locating the strengths and weaknesses of each approach, and instrumental in driving research from a practical perspective.

12.1 FUTURE WORK

There are various ways the results of this thesis could be built upon.

METATHEORY One drawback in the presentation of KATER so far is that users might be required to rewrite a given model so that it falls within the (decidable) KAT fragment. Even though alternative formulations of the same model can be checked for equivalence by the tool itself, rewriting models might sometimes be impossible[220]. As such, it would be interesting to investigate extensions of KAT that would increase the expressiveness of the tool, while still providing a decidable procedure.

As far as fragments for which language inclusion cannot be decided are concerned, it would be interesting to see whether KATER can be used in a semi-interactive mode. Specifically, whenever a given expression does not fall within the decidable fragment of the tool, the users should be responsible to provide all information required to complete the inclusion checks. Such an interactive mode would be more meaningful in case KATER is implemented as a library for a proof assistant.

[220] *As a few examples, consider arbitrary fixpoint calculations, restrictions of a relation to a given location, etc.*

In terms of tool support, it would be worth investigating whether it is possible to incorporate Kater's decision procedures into proof assistants like Coq (that have been traditionally used to prove metatheoretic properties of weak memory models), and whether Kater's infrastructure for generating consistency checks can be leveraged by SMT solver theories (as, e.g., in the work of He, Sun, and Fan[221]).

VERIFICATION In terms of verification, it would certainly be helpful to address one of the major weaknesses of DPOR, namely the lack of support for data-non-determinism. To that end, it should be straightforward to combine DPOR with concolic testing, treating symbolic variable definitions and branching decisions as extra backtracking points when constructing execution graphs in DPOR, and offloading path-feasibility constraints to an SMT solver. To a large extent, constraints on symbolic variables and path feasibility are thread-local, and should not induce a lot of changes in the underlying DPOR framework.

Another feature that would greatly improve the applicability of a framework like GenMC is support for arbitrary atomic sections, by lifting DPOR so that orderings of sections with no conflicting accesses are not explored. Such support would have to redefine what revisiting means and how it is performed, but would allow for much greater reductions as well. DPOR could then potentially be applied to handle transactions, mixed-size accesses (where multiple byte-level accesses can be performed in a single atomic step), or even interrupts in systems code (where an interrupt handler may run atomically, at any point of the program). The work of Bouajjani, Enea, and Román-Calvo[222] is a promising step in that direction, though the extent of DPOR's applicability is yet to be discovered.

On a more practical level, it would be interesting to investigate the integration of DPOR with techniques that check bounded correctness (e.g., preemption bounding). One could try to come up with algorithms that optimally combine the two techniques (a problem that is quite challenging[223]), or come up with novel bound notions that carry over to weak memory and enable effective verification.

Finally, as far as tool support is concerned, integrating interpreters or runtime environments for different languages into GenMC would enable the verification of a much broader class of programs, and therefore the potential discovery of more bugs. Another possibility would be to use GenMC in other contexts, e.g., fence insertion and program synthesis[224].

[221] *"Satisfiability modulo Ordering Consistency Theory for Multi-Threaded Program Verification"* [HSF21]

[222] *"Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels"* [BER23]

[223] *"Reconciling Preemption Bounding with DPOR"* [MKV23]; *"Optimal Bounded Partial Order Reduction"* [MV23]

[224] *"VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models"* [Obe+21b]

BIBLIOGRAPHY

[Abd+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. "Stateless model checking for TSO and PSO." In: *TACAS 2015*. DOI: 10.1007/978-3-662-46681-0_28. URL: http://dx.doi.org/10.1007/978-3-662-46681-0_28.

[Abd+14] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Optimal dynamic partial order reduction." In: *POPL 2014*. DOI: 10.1145/2535838.2535845. URL: http://doi.acm.org/10.1145/2535838.2535845.

[Abd+17] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Source sets: A foundation for optimal dynamic partial order reduction." In: *J. ACM* 64.4 (2017). DOI: 10.1145/3073408. URL: http://doi.acm.org/10.1145/3073408.

[Abd+19] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. "Optimal stateless model checking for reads-from equivalence under sequential consistency." In: *Proc. ACM Program. Lang.* 3 (OOPSLA 10, 2019). DOI: 10.1145/3360576. URL: https://doi.org/10.1145/3360576 (visited on 01/18/2021).

[Abd+16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. "Stateless model checking for POWER." In: *CAV 2016*. DOI: 10.1007/978-3-319-41540-6_8. URL: https://doi.org/10.1007/978-3-319-41540-6_8.

[Abd+18] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. "Optimal stateless model checking under the release-acquire semantics." In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276505. URL: http://doi.acm.org/10.1145/3276505.

[Aga+21] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. "Stateless Model Checking Under a Reads-Value-From Equivalence." In: *CAV 2021*. DOI: 10.1007/978-3-030-81685-8_16.

[Alb+17] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. "Context-sensitive dynamic partial order reduction." In: *CAV 2017*. DOI: 10.1007/978-3-319-63387-9_26.

[Alb+18] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. "Constrained dynamic partial order reduction." In: *CAV 2018*. DOI: 10.1007/978-3-319-96142-2_24.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. "Partial orders for efficient bounded model checking of concurrent software." In: *CAV 2013*. DOI: 10.1007/978-3-642-39799-8_9.

[Alg+18] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. "Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel." In: *ASPLOS 2018*. DOI: 10.1145/3173162.3177156. URL: http://doi.acm.org/10.1145/3173162.3177156.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." In: *ACM Trans. Program. Lang. Syst.* 36.2 (2014). DOI: 10.1145/2627752. URL: http://doi.acm.org/10.1145/2627752.

[Aro+18] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. "Optimal dynamic partial order reduction with observers." In: *TACAS 2018*. DOI: 10.1007/978-3-319-89963-3_14.

[Bal+04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft." In: *IFM 2004*. DOI: 10.1007/978-3-540-24756-2_1.

[Bar+17] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. "Model Checking of C and C++ with DIVINE 4." In: *ATVA 2017*. DOI: 10.1007/978-3-319-68167-2_14.

[Bat+12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. "Clarifying and compiling C/C++ concurrency: From C++11 to POWER." In: *POPL 2012*. DOI: 10.1145/2103656.2103717. URL: http://doi.acm.org/10.1145/2103656.2103717.

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ concurrency." In: *POPL 2011*. DOI: 10.1145/1926385.1926394. URL: http://doi.acm.org/10.1145/1926385.1926394.

[BE19] Ranadeep Biswas and Constantin Enea. "On the Complexity of Checking Transactional Consistency." In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360591. URL: https://doi.org/10.1145/3360591.

[BD14] Hans-Juergen Boehm and Brian Demsky. "Outlawing ghosts: Avoiding out-of-thin-air results." In: *MSPC 2014*. DOI: 10.1145/2618128.2618134. URL: http://doi.acm.org/10.1145/2618128.2618134.

[BP13] Filippo Bonchi and Damien Pous. "Checking NFA equivalence with bisimulations up to congruence." In: *POPL 2013*. DOI: 10.1145/2429069.2429124. URL: https://doi.org/10.1145/2429069.2429124.

[BT17] James Bornholt and Emina Torlak. "Synthesizing memory models from framework sketches and Litmus tests." In: *PLDI 2017*. DOI: 10.1145/3062341.3062353. URL: https://doi.org/10.1145/3062341.3062353.

[BER23] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. "Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels." In: *Proc. ACM Program. Lang.* 7.PLDI (2023). DOI: 10.1145/3591243. URL: https://doi.org/10.1145/3591243.

[Bui+21] Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. "The Reads-from Equivalence for the TSO and PSO Memory Models." In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021). DOI: 10.1145/3485541. URL: https://doi.org/10.1145/3485541.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. "CheckFence: Checking consistency of concurrent data types on relaxed memory models." In: *PLDI 2007*. DOI: 10.1145/1250734.1250737.

[CV19] Soham Chakraborty and Viktor Vafeiadis. "Grounding thin-air reads with event structures." In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290383.

[Cha+17] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. "Data-centric dynamic partial order reduction." In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158119. URL: http://doi.acm.org/10.1145/3158119.

[CPT19] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. "Value-Centric Dynamic Partial Order Reduction." In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360550. URL: https://doi.org/10.1145/3360550.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "Automatic verification of finite-state concurrent systems Using temporal logics specification: A practical approach." In: *POPL 1983*. DOI: 10.1145/567067.567080. URL: http://doi.acm.org/10.1145/567067.567080.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs." In: *TACAS 2004*. DOI: 10.1007/978-3-540-24730-2_15. URL: http://dx.doi.org/10.1007/978-3-540-24730-2_15.

[Con63] Melvin E. Conway. "Design of a separable transition-diagram compiler." In: *Commun. ACM* 6.7 (1963). DOI: 10.1145/366663.366704. URL: https://doi.org/10.1145/366663.366704.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[Sch16] Samuel Schetterer. *Crossbeam: Flat combining #63*. 2016. URL: https://github.com/crossbeam-rs/crossbeam/issues/63 (visited on 01/29/2021).

[DL15] Brian Demsky and Patrick Lam. "SATCheck: SAT-directed stateless model checking for SC and TSO." In: *OOPSLA 2015*. DOI: 10.1145/2814270.2814297. URL: http://doi.acm.org/10.1145/2814270.2814297.

[DSM18] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. "Bounding Data Races in Space and Time." In: *PLDI 2018*. DOI: 10.1145/3192366.3192421. URL: https://doi.org/10.1145/3192366.3192421.

[23] *DOT (graph description language)*. 2023. URL: https://en.wikipedia.org/wiki/DOT_(graph_description_language) (visited on 06/16/2023).

[daa] daanx. *fix memory order for weak CAS*. URL: https://github.com/microsoft/mimalloc/commit/444afa934ff8d53bf8c53602246bfc65828dc1d9 (visited on 10/16/2022).

[FG05] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software." In: *POPL 2005*. DOI: 10.1145/1040305.1040315. URL: http://doi.acm.org/10.1145/1040305.1040315.

[Flu+17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. "Mixed-size concurrency: ARM, POWER, C/C++11, and SC." In: *POPL 2017*. DOI: 10.1145/3009837.3009839. URL: https://doi.org/10.1145/3009837.3009839.

[Gav+19] Natalia Gavrilenko, Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. "BMC for weak memory models: Relation analysis for compact SMT encodings." In: *CAV 2019*. DOI: 10.1007/978-3-030-25540-4_19.

[Kok] Michalis Kokologiannakis. *GenMC: Generic model checking for C programs*. URL: https://github.com/MPI-SWS/genmc.

[GK97] Phillip B. Gibbons and Ephraim Korach. "Testing shared memories." In: *SIAM J. Comput.* 26.4 (1997). DOI: 10.1137/S0097539794279614. URL: http://dx.doi.org/10.1137/S0097539794279614.

[God05] Patrice Godefroid. "Software Model Checking: The VeriSoft Approach." In: *Form. Meth. Syst. Des.* 26.2 (2005). DOI: 10.1007/s10703-005-1489-x. URL: http://dx.doi.org/10.1007/s10703-005-1489-x.

[HP00] Klaus Havelund and Thomas Pressburger. "Model Checking JAVA Programs using JAVA PathFinder." In: *Int. J. Soft. Tool. Tech. Transf.* 2.4 (2000). DOI: 10.1007/S100090050043. URL: https://doi.org/10.1007/s100090050043.

[HSF21] Fei He, Zhihang Sun, and Hongyu Fan. "Satisfiability modulo Ordering Consistency Theory for Multi-Threaded Program Verification." In: *PLDI 2021*. DOI: 10.1145/3453483.3454108. URL: https://doi.org/10.1145/3453483.3454108.

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Lazy Abstraction." In: *POPL 2002*. DOI: 10.1145/503272.503279. URL: https://doi.org/10.1145/503272.503279.

[Hol97] G.J. Holzmann. "The model checker SPIN." In: *IEEE Trans. Software Eng.* 23.5 (1997). DOI: 10.1109/32.588521.

[Hua15] Jeff Huang. "Stateless model checking concurrent programs with maximal causality reduction." In: *PLDI 2015*. DOI: 10.1145/2737924.2737975. URL: http://doi.acm.org/10.1145/2737924.2737975.

[HH16] Shiyou Huang and Jeff Huang. "Maximal Causality Reduction for TSO and PSO." In: *OOPSLA 2016*. DOI: 10.1145/2983990.2984025. URL: http://doi.acm.org/10.1145/2983990.2984025.

[KSH15] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. "Unfolding Based Automated Testing of Multithreaded Programs." In: *Autom. Softw. Eng.* 22.4 (2015). DOI: 10.1007/s10515-014-0150-6. URL: http://dx.doi.org/10.1007/s10515-014-0150-6.

[Kan+17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. "A promising semantics for relaxed-memory concurrency." In: *POPL 2017*. DOI: 10.1145/3009837.3009850.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking." In: *TACAS 2015*. DOI: 10.1007/978-3-662-46681-0_61.

[KLV23a] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. *Kater: Automating Weak Memory Model Metatheory and Consistency Checking (Project page)*. 2023. URL: https://plv.mpi-sws.org/kater (visited on 01/22/2024).

[Kok23] Michalis Kokologiannakis. "Automated Reasoning under Weak Memory Consistency (replication package)." In: (2023). DOI: 10.5281/zenodo.10575926.

[Kok+21] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. "PerSeVerE: Persistency semantics for verification under ext4." In: *Proc. ACM Program. Lang.* 5.POPL (2021). DOI: 10.1145/3434324. URL: https://doi.org/10.1145/3434324.

[Kok+17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. "Effective stateless model checking for C/C++ concurrency." In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158105. URL: http://doi.acm.org/10.1145/3158105.

[KLV23b] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. "Kater: Automating Weak Memory Model Metatheory and Consistency Checking." In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571212. URL: https://doi.org/10.1145/3571212.

[Kok+22a] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. "Truly stateless, optimal dynamic partial order reduction." In: *Proc. ACM Program. Lang.* 6.POPL (2022). DOI: 10.1145/3498711. URL: https://doi.org/10.1145/3498711.

[Kok+22b] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. "Truly Stateless, Optimal Dynamic Partial Order Reduction (supplementary material)." In: (2022). URL: https://plv.mpi-sws.org/genmc.

[KMV23] Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. "Unblocking Dynamic Partial Order Reduction." In: *CAV 2023*. DOI: 10.1007/978-3-031-37706-8_12. URL: https://doi.org/10.1007/978-3-031-37706-8%5C_12.

[KRV19] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. "Model checking for weakly consistent libraries." In: *PLDI 2019*. DOI: 10.1145/3314221.3314609.

[KRV21] Michalis Kokologiannakis, Xiaowei Ren, and Viktor Vafeiadis. "Dynamic Partial Order Reductions for Spinloops." In: *FMCAD 2021*. DOI: 10.34727/2021/isbn.978-3-85448-046-4_25. URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4%5C_25.

[KS19] Michalis Kokologiannakis and Konstantinos Sagonas. "Stateless model checking of the Linux kernel's read–copy update (RCU)." In: *Int. J. Soft. Tool. Tech. Transf.* (2019). DOI: 10.1007/s10009-019-00514-6. URL: https://doi.org/10.1007/s10009-019-00514-6.

[KV20] Michalis Kokologiannakis and Viktor Vafeiadis. "HMC: Model checking for hardware memory models." In: *ASPLOS 2020*. DOI: 10.1145/3373376.3378480. URL: https://doi.org/10.1145/3373376.3378480.

[KV21a] Michalis Kokologiannakis and Viktor Vafeiadis. "BAM: Efficient Model Checking for Barriers." In: *NETYS 2021*. DOI: 10.1007/978-3-030-91014-3_16. URL: https://plv.mpi-sws.org/genmc.

[KV21b] Michalis Kokologiannakis and Viktor Vafeiadis. "GenMC: A model checker for weak memory models." In: *CAV 2021*. DOI: 10.1007/978-3-030-81685-8_20.

[Koz97] Dexter Kozen. "Kleene Algebra with Tests." In: *ACM Trans. Program. Lang. Syst.* 19.3 (1997). DOI: 10.1145/256167.256195. URL: https://doi.org/10.1145/256167.256195.

[KS96] Dexter Kozen and Frederick Smith. "Kleene Algebra with Tests: Completeness and Decidability." In: *CSL 1996*. DOI: 10.1007/3-540-63172-0_43. URL: https://doi.org/10.1007/3-540-63172-0%5C_43.

[LGV16] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. "Taming Release-acquire Consistency." In: *POPL 2016*. DOI: 10.1145/2837614.2837643. URL: http://doi.acm.org/10.1145/2837614.2837643.

[LV16] Ori Lahav and Viktor Vafeiadis. "Explaining Relaxed Memory Models with Program Transformations." In: *FM 2016*. DOI: 10.1007/978-3-319-48989-6_29. URL: http://dx.doi.org/10.1007/978-3-319-48989-6_29.

[Lah+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing sequential consistency in C/C++11." In: *PLDI 2017*. DOI: 10.1145/3062341.3062352. URL: http://doi.acm.org/10.1145/3062341.3062352.

[Lam79] Leslie Lamport. "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." In: *IEEE Trans. Computers* 28.9 (1979). DOI: 10.1109/TC.1979.1675439. URL: http://dx.doi.org/10.1109/TC.1979.1675439.

[LS20] Magnus Lång and Konstantinos Sagonas. "Parallel Graph-Based Stateless Model Checking." In: *ATVA 2020*. DOI: 10.1007/978-3-030-59152-6_21.

[03a] *lli - directly execute programs from LLVM bitcode*. 2003. URL: https://llvm.org/docs/CommandGuide/lli.html (visited on 01/29/2021).

[03b] *Writing an LLVM Pass*. 2003. URL: https://llvm.org/docs/WritingAnLLVMPass.html#introduction-what-is-a-pass (visited on 06/16/2023).

[MAM10] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. "Generating Litmus Tests for Contrasting Memory Consistency Models." In: *CAV 2010*. DOI: 10.1007/978-3-642-14295-6_26. URL: https://doi.org/10.1007/978-3-642-14295-6%5C_26.

[MAM11] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. "Litmus tests for comparing memory consistency models: how long do they need to be?" In: *DAC 2011*. DOI: 10.1145/2024724.2024842. URL: https://doi.org/10.1145/2024724.2024842.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java memory model." In: *POPL 2005*. DOI: 10.1145/1040305.1040336. URL: https://doi.org/10.1145/1040305.1040336.

[MKV23] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. "Reconciling Preemption Bounding with DPOR." In: *TACAS 2023*.

[MV23] Iason Marmanis and Viktor Vafeiadis. "Optimal Bounded Partial Order Reduction." In: *FMCAD 2023*. DOI: 10.34727/2023/ISBN.978-3-85448-060-0_16. URL: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0%5C_16.

[Maz87] Antoni Mazurkiewicz. "Trace Theory." In: *PNAROMC 1987*. DOI: 10.1007/3-540-17906-2_30. URL: http://dx.doi.org/10.1007/3-540-17906-2_30.

[McM95] Kenneth L. McMillan. "A Technique of a State Space Search Based on Unfolding." In: *Form. Meth. Syst. Des.* 6.1 (1995). DOI: 10.1007/BF01384314. URL: http://dx.doi.org/10.1007/BF01384314.

[MKV22] Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. “Model Checking on a Multi-execution Memory Model.” In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022). DOI: 10.1145/3563315. URL: https://doi.org/10.1145/3563315.

[Mus+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. “Finding and reproducing Heisenbugs in concurrent programs.” In: *OSDI 2008*. URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf (visited on 11/16/2020).

[Ngu+18] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. “Quasi-optimal partial order reduction.” In: *CAV 2018*. DOI: 10.1007/978-3-319-96142-2_22.

[ND13] Brian Norris and Brian Demsky. “CDSChecker: Checking concurrent data structures written with C/C++ atomics.” In: *OOPSLA 2013*. DOI: 10.1145/2509136.2509514. URL: https://doi.org/10.1145/2509136.2509514.

[Obe+21a] Jonas Oberhauser, Lilith Oberhauser, Antonio Paolillo, Diogo Behrens, Ming Fu, and Viktor Vafeiadis. “Verifying and Optimizing the HMCS Lock for Arm Servers.” In: *NETYS 2021*. DOI: 10.1007/978-3-030-91014-3_17. URL: https://doi.org/10.1007/978-3-030-91014-3_17.

[Obe+21b] Jonas Oberhauser et al. “VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models.” In: *ASPLOS 2021*. DOI: 10.1145/3445814.3446748. URL: https://doi.org/10.1145/3445814.3446748.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. “A better x86 memory model: x86-TSO.” In: *TPHOLs 2009*. DOI: 10.1007/978-3-642-03359-9_27. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_27.

[mar] mary3000. *Play nice with thread sanitizer #130*. URL: https://github.com/microsoft/mimalloc/issues/130#issuecomment-662666849 (visited on 10/16/2022).

[PLV19] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. “Bridging the gap between programming languages and hardware weak memory models.” In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290382. URL: http://doi.acm.org/10.1145/3290382.

[Fil] Yuval Filmus. *Is the power of a regular language regular? Is the root of a regular language regular?* Computer Science Stack Exchange. URL:https://cs.stackexchange.com/q/99371 (version: 2018-10-31). URL: https://cs.stackexchange.com/q/99371 (visited on 10/20/2022).

[17] *pthread.h man page*. 2017. URL: https://man7.org/linux/man-pages/man0/pthread.h.0p.html (visited on 03/19/2021).

[Pul+18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8." In: *Proc. ACM Program. Lang.* 2.POPL (2018). DOI: 10.1145/3158107. URL: https://doi.org/10.1145/3158107.

[Pul+19] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. "Promising-ARM/RISC-V: A simpler and faster operational concurrency model." In: *PLDI 2019*. DOI: 10.1145/3314221.3314624. URL: http://doi.acm.org/10.1145/3314221.3314624.

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR." In: *ISP 1982*. DOI: 10.1007/3-540-11494-7_22. URL: http://dx.doi.org/10.1007/3-540-11494-7_22.

[RV18] Azalea Raad and Viktor Vafeiadis. "Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model." In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276507. URL: https://doi.org/10.1145/3276507.

[Raa+19] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. "Persistency semantics of the Intel-x86 architecture." In: *Proc. ACM Program. Lang.* 4 (POPL 20, 2019). DOI: 10.1145/3371079. URL: https://doi.org/10.1145/3371079 (visited on 06/17/2020).

[RWV19] Azalea Raad, John Wickerson, and Viktor Vafeiadis. "Weak persistency semantics from the ground up." In: *Proc. ACM Program. Lang.* 3 (OOPSLA 10, 2019). DOI: 10.1145/3360561. URL: https://doi.org/10.1145/3360561 (visited on 02/07/2020).

[rme09] rmem. *rmem: Executable concurrency models for ARMv8, RISC-V, Power, and x86*. 2009. URL: https://github.com/rems-project/rmem (visited on 08/24/2019).

[Rod+15] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. "Unfolding-based Partial Order Reduction." In: *CONCUR 2015*. DOI: 10.4230/LIPIcs.CONCUR.2015.456. URL: http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.456.

[Sar+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. "Synchronising C/C++ and POWER." In: *PLDI 2012*. DOI: 10.1145/2254064.2254102.

[ŞCR13] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. "Maximal Causal Models for Sequentially Consistent Systems." In: *RV 2012*. DOI: 10.1007/978-3-642-35632-2_16.

[SS88] Dennis Shasha and Marc Snir. "Efficient and correct execution of parallel programs that share memory." In: *ACM Trans. Program. Lang. Syst.* 10.2 (1988). DOI: 10.1145/42190.42277. URL: http://doi.acm.org/10.1145/42190.42277.

[SPA94] SPARC International Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, 1994.

[SFH22] Zhihang Sun, Hongyu Fan, and Fei He. "Consistency-Preserving Propagation for SMT Solving of Concurrent Program Verification." In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022). DOI: 10.1145/3563321. URL: https://doi.org/10.1145/3563321.

[SV-19] SV-COMP. *Competition on Software Verification (SV-COMP)*. 2019. URL: https://sv-comp.sosy-lab.org/2019/ (visited on 03/27/2019).

[TVD10] Emina Torlak, Mandana Vaziri, and Julian Dolby. "MemSAT: checking axiomatic specifications of memory models." In: *PLDI 2010*. DOI: 10.1145/1806596.1806635. URL: https://doi.org/10.1145/1806596.1806635.

[Vaf+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. "Common compiler optimisations are invalid in the C11 memory model and what we can do about it." In: *POPL 2015*. DOI: 10.1145/2676726.2676995. URL: http://doi.acm.org/10.1145/2676726.2676995.

[Wic+17] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. "Automatically Comparing Memory Consistency Models." In: *POPL 2017*. DOI: 10.1145/3009837.3009838. URL: https://doi.org/10.1145/3009837.3009838.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. "Dynamic partial order reduction for relaxed memory models." In: *PLDI 2015*. DOI: 10.1145/2737924.2737956. URL: http://doi.acm.org/10.1145/2737924.2737956.

CURRICULUM VITAE

RESEARCH INTERESTS

Programming languages, compilers, weak memory models, and software verification. I am mainly interested in concurrent software verification with emphasis on the effects induced by the weak memory models employed by modern microprocessors.

EDUCATION

2018–2023 PhD student in Computer Science
Max Planck Institute for Software Systems — KL, Germany

Thesis: Automated Reasoning under Weak Memory Concurrency *MPI-SWS*
Advisor: Viktor Vafeiadis

2011–2016 MEng in Computer Engineering
National Technical University of Athens — Athens, Greece

Thesis: Systematic Concurrency Testing of Read-Copy-Update under Sequentially Consistent and Weak Memory Models *NTUA*
Advisor: Kostis Sagonas