PROGRAM ANALYSIS UNDER RELAXED MEMORY CONCURRENCY

SANJANA SINGH

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
AUGUST 2023

PROGRAM ANALYSIS UNDER RELAXED MEMORY CONCURRENCY

by

SANJANA SINGH

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of
Doctor of Philosophy

to the

Indian Institute of Technology Delhi
AUGUST 2023

Certificate

This is to certify that the thesis titled **PROGRAM ANALYSIS UNDER RELAXED MEMORY CONCURRENCY** being submitted by **Ms. SANJANA SINGH** for the award of **Doctor of Philosophy** in **Computer Science and Engineering** is a record of bona fide work carried out by her under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Subodh Sharma
Associate Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi- 110016

Acknowledgements

I am deeply grateful to everyone who has contributed to the completion of this work.

First and foremost, I express my heartfelt gratitude to my advisor, Subodh Sharma, for his unwavering guidance and support throughout the entire journey. His encouragement and belief in my abilities have not only shaped me as a researcher but also as a confident and level-headed individual. Subodh is a humble, highly motivated and ever curious person. Working with him taught me to enjoy the process and stay committed despite occasional disappointments. I am thankful for his valuable feedback and for keeping me focused on the right track. One of the qualities I truly admire in my advisor is his willingness to give his students the credit they deserve, which has helped us gain better exposure and recognition. Moreover, under Subodh's guidance, I have developed into a more confident and presentable professional. I am grateful to him for his belief in me and for his patience throughout my Ph.D. journey.

I extend my sincere thanks to my committee members, Sanjiva Prasad, Sorav Bansal, and S. Krishna, for for taking the time out to periodically review my ongoing work and provide valuable insights and feedback that significantly contributed to its refinement. I am indebted to S. Arun Kumar for his mentorship and continuous availability for technical and life advice. Guidance of S. Arun Kumar and Subodh Sharma has played a significant role in improving my presentation and communication skills.

My heartfelt thanks go to the staff at the CSE department for their unwavering support, especially Rekha Rathee, Hemant Prasad, S. S. Negi, and Monika Rajan.

I am also grateful to Sandeep K. Singh and Sanjay Goel for their belief in my potential

and encouraging me to pursue a Ph.D. Their support during my undergraduate and graduate years laid a strong foundation for my research journey. I would also like to thank Sapna Gupta who taught me Computer Science in school and developed in me a fascination for the field.

To my dear friends and colleagues, Divyanjali Sharma, Madhukar Yerraguntla, Shailja Pandey, and Vishal Sharma, thank you for making my Ph.D. tenure enjoyable and for being there for me both professionally and emotionally.

A special thanks to my dearest friend and partner Lav Singh, who has been a constant source of strength and belief in my abilities. I am grateful to him for pushing me to apply for Ph.D. at IIT Delhi. He has taught me to loosen up and approach life and work with an ease. His presence in my life has made me a better person.

I was also blessed with the joy of motherhood during my Ph.D. tenure. Balancing the responsibilities of being a young mother and a researcher has taught me valuable lessons in adapting to circumstances without losing sight of my ultimate objectives. My three year old is the source of my inspiration and strength because of whom I have developed enhanced levels of patience and honed my multitasking skills.

I owe my heartfelt appreciation to my parents, Sadhana Singh and Surender Singh, supporting me during my Ph.D. journey as a counselor, guide, and even a nanny for my kid. Their belief in me throughout my life gave me the confidence to take on challenges. I also thank my parents-in-law, Usha Singh and Ram Sharan Singh, for their blessings.

To my siblings, Samyukta Singh and Suraj Pratap Singh, and sibling-in-law, Aditi Singh, thank you for your constant well-wishes and emotional support.

Finally, I acknowledge that a Ph.D. journey is not a solo endeavor; it is made possible by the support and sacrifices of many individuals. I thank everyone in my life for enabling me to embark on this journey with ease.

Sanjana Singh

Abstract

Concurrency is ubiquitous. Concurrent processing improves the speed of execution, and offers better resource utilization. However, the analysis of concurrent programs is remarkably more challenging than sequential counterparts. Concurrency renders an inherently complex program structure and unintuitive control flow. The complexity is further escalated with synchronization primitives used to control access to shared resources. Moreover, the reachable state graph grows exponentially with an increase in the concurrent processing elements resulting in the state space explosion problem.

Relaxed memory models allow out-of-order execution of program instructions that further ravel the program structure and significantly expand the reachable state graph. As a result, the challenge in analyzing concurrent programs becomes more acute under relaxed memory concurrency. Consequently, it is hard to develop concurrent programs that efficiently utilize the permitted ordering relaxations, and equally hard to determine the feasible program outcomes under relaxed memory concurrency. Both the problems may become exacting even for expert programmers.

This work proposes techniques for the ease of development of efficient programs that produce expected outcomes on relaxed memory models. The techniques focus on

verification efficiency, targeting legacy and developed programs, and *developer productivity* and *runtime efficiency*, targeting programs under development.

Stateless model checkers mitigate the state space explosion problem by exploring a reduced state graph comprising representative program executions from each *equivalence class*; where, an equivalence class is a set of *equivalent* executions that is formed based on an *equivalence relation*. This work proposes a novel equivalence relation on the program executions called *view-equivalence* that is at least as coarse as any existing equivalence relation. Consequently, view-equivalence induces the smallest set of equivalence classes, positively reducing the analysis effort. This work also proposes a stateless model checker called `ViEqui`, which partitions program executions on the proposed view-equivalence relation. The fewer equivalence classes under view-equivalence help `ViEqui` achieve faster analysis and better scalability in comparison to the existing stateless model checkers on finer equivalence relations.

As the second contribution, this work explores stateless model checking under relaxed memory consistency specifications or *memory models*. Various program outcomes feasible under the memory model associated with languages (such as `C/C++`) may never manifest on underlying architectures, being disallowed by the architecture's stronger implicit ordering. Existing model checking techniques operate under the memory consistency specification of a language or of an architecture. This work proposes a stateless model checker called `MoCA` that analyzes programs with `C/C++` memory model under a class of architecture memory models called *multi-copy atomics*. `MoCA` performs a precise stateless model checking for program outcomes valid under the `C/C++` specification and feasible under multi-copy atomicity. The precise analysis of

MoCA reduces the cognitive load on developers of sorting the feasible from infeasible outcomes while also reducing the model checking effort by restricting the set of outcomes to analyze.

The C/C++ language is known to have intricate memory consistency semantics. The work with MoCA uncovers the complexity of the semantics that a C/C++ developer is exposed to. With a focus on reducing development effort, the final element of this study proposes the first `fence` synthesis technique for C/C++ memory model. The technique takes a buggy C/C++ program with a correctness specification and presents an automated fix of the program with additional synchronization on concurrent elements through `fences`. This work proposes a technique called FenSying that fixes the input program with minimal necessary synchronization overhead.

In summary, this work proposes analysis techniques for developing correct and efficient concurrent programs for relaxed memory models. The focus of this work is on verification efficiency (through ViEqui and MoCA), developer productivity (through MoCA and FenSying), and runtime efficiency (through FenSying). Each technique is accompanied by an effective tool support.

सार

समवर्तीता सर्वव्यापी है. समवर्ती प्रसंस्करण से निष्पादन की गति में सुधार होता है, और बेहतर संसाधन उपयोग प्रदान करता है। हालाँकि, समवर्ती कार्यक्रमों का विश्लेषण है अनुक्रमिक समकक्षों की तुलना में उल्लेखनीय रूप से अधिक चुनौतीपूर्ण। समवर्ती एक प्रस्तुत करता है स्वाभाविक रूप से जटिल कार्यक्रम संरचना और सहज ज्ञान युक्त नियंत्रण प्रवाह। जटिलता साझा तक पहुंच को नियंत्रित करने के लिए उपयोग किए जाने वाले सिंक्रोनाइज़ेशन प्रिमिटिव के साथ इसे और बढ़ाया गया है संसाधन। इसके अलावा, पहुंच योग्य स्थिति का ग्राफ वृद्धि के साथ तेजी से बढ़ता है समवर्ती प्रसंस्करण तत्वों के परिणामस्वरूप राज्य अंतरिक्ष विस्फोट की समस्या उत्पन्न होती है।

रिलेक्सड मेमोरी मॉडल प्रोग्राम निर्देशों के आउट-ऑफ़-ऑर्डर निष्पादन की अनुमति देते हैं जो कि प्रदान करते हैं- वे प्रोग्राम संरचना को स्पष्ट करते हैं और पहुंच योग्य स्थिति ग्राफ़ का महत्वपूर्ण रूप से विस्तार करते हैं। परिणामस्वरूप, समवर्ती कार्यक्रमों के विश्लेषण में चुनौती और अधिक गंभीर हो जाती है आरामदेह स्मृति संगामिति के अंतर्गत। परिणामस्वरूप, समवर्ती विकास करना कठिन है ऐसे प्रोग्राम जो अनुमत आदेश संबंधी छूटों का कुशलतापूर्वक उपयोग करते हैं, और समान रूप से कठिन भी रिलेक्सड स्मृति संगामिति के तहत व्यवहार्य कार्यक्रम परिणामों को निर्धारित करने के लिए। दोनों विशेषज्ञ प्रोग्रामर के लिए भी समस्याएँ विकट हो सकती हैं।

यह कार्य कुशल कार्यक्रमों के विकास में आसानी के लिए तकनीकों का प्रस्ताव करता है रिलेक्सड मेमोरी मॉडल पर अपेक्षित परिणाम उत्पन्न करें। तकनीकें सत्यापन दक्षता, विरासत और विकसित

कार्यक्रमों को लक्षित करने और डेवलपर समर्थक पर ध्यान केंद्रित करती हैं। लचीलापन और रनटाइम दक्षता, विकास के तहत कार्यक्रमों को लक्षित करना।

स्टेटलेस मॉडल चेकर्स खोज करके राज्य अंतरिक्ष विस्फोट की समस्या को कम करते हैं प्रत्येक समतुल्य से प्रतिनिधि कार्यक्रम निष्पादन को शामिल करते हुए कम किया गया राज्य ग्राफ़- लेंस वर्ग; जहां, एक समतुल्य वर्ग समतुल्य निष्पादन का एक सेट है जो बनता है तुल्यता संबंध पर आधारित. यह कार्य एक नवीन तुल्यता संबंध का प्रस्ताव करता है प्रोग्राम निष्पादन पर दृश्य-समतुल्यता कहा जाता है जो कम से कम उतना ही मोटा है मौजूदा तुल्यता संबंध. नतीजतन, दृश्य-समतुल्यता सबसे छोटे को प्रेरित करती है तुल्यता वर्गों का सेट, विश्लेषण प्रयास को सकारात्मक रूप से कम करता है। यह कार्य भी प्रो- ViEqui नामक एक स्टेटलेस मॉडल चेकर बनाता है, जो प्रोग्राम निष्पादन को विभाजित करता है प्रस्तावित दृश्य-समतुल्य संबंध पर। विचाराधीन कम तुल्यता वर्ग- समतुल्यता ViEqui को तुलना में तेज़ विश्लेषण और बेहतर स्केलेबिलिटी प्राप्त करने में मदद करती है बेहतर तुल्यता संबंधों पर मौजूदा स्टेटलेस मॉडल चेकर्स के लिए।

दूसरे योगदान के रूप में, यह कार्य आराम के तहत स्टेटलेस मॉडल चेकिंग की पड़ताल करता है मेमोरी संगति विनिर्देश या मेमोरी मॉडल। विभिन्न कार्यक्रम परिणाम- भाषाओं (जैसे C/C++) से जुड़े मेमोरी मॉडल के तहत सिबल कभी नहीं हो सकता अंतर्निहित आर्किटेक्चर पर प्रकट, आर्किटेक्चर के मजबूत द्वारा अस्वीकृत किया जा रहा है अंतर्निहित आदेश. मौजूदा मॉडल जाँच तकनीकें मेमोरी के अंतर्गत काम करती हैं किसी भाषा या वास्तुकला की स्थिरता विशिष्टता। यह कार्य प्रस्तावित है MoCA नामक एक स्टेटलेस मॉडल चेकर जो C/C++ मेमोरी वाले प्रोग्रामों का विश्लेषण करता है आर्किटेक्चर मेमोरी मॉडल के एक वर्ग के तहत मॉडल जिसे मल्टी-कॉपी एटॉमिक्स कहा जाता है। मोका के अंतर्गत मान्य कार्यक्रम परिणामों के लिए एक सटीक स्टेटलेस मॉडल जाँच करता है सी/सी++ विशिष्टता और बहु-प्रतिलिपि परमाणुता के तहत व्यवहार्य। MoCA का सटीक विश्लेषण व्यावहारिक से व्यवहार्य को छांटने में डेवलपर्स पर संज्ञानात्मक भार को कम करता है- सेट को सीमित करके मॉडल जाँच प्रयास को कम करते हुए परिणामों को ख़राब करें विश्लेषण करने के लिए परिणाम.

C/C++ भाषा को जटिल मेमोरी संगति शब्दार्थ के लिए जाना जाता है। MoCA के साथ काम करने से C/C++ डेवलपर के शब्दार्थ की जटिलता का पता चलता है से अवगत कराया। विकास प्रयास को कम करने पर ध्यान देने के साथ, इसका अंतिम तत्व अध्ययन C/C++ मेमोरी मॉडल के लिए पहली बाड़ संश्लेषण तकनीक का प्रस्ताव करता है। तकनीक एक त्रुटिहीन C/C++ प्रोग्राम को शुद्धता विनिर्देशन के साथ लेती है और प्रस्तुत करती है समवर्ती तत्वों पर अतिरिक्त सिंक्रनाइज़ेशन के साथ कार्यक्रम का एक स्वचालित समाधान- बाड़ के माध्यम से प्रवेश. यह कार्य फ़ेंसिंग नामक एक तकनीक का प्रस्ताव करता है जो ठीक करती है न्यूनतम आवश्यक सिंक्रोनाइज़ेशन ओवरहेड के साथ इनपुट प्रोग्राम।

संक्षेप में, यह कार्य सही और प्रभावोत्पादक विकास के लिए विश्लेषण तकनीकों का प्रस्ताव करता है। रिलेक्सड मेमोरी मॉडल के लिए वैज्ञानिक समवर्ती कार्यक्रम। इस काम पर फोकस है सत्यापन दक्षता (ViEqui और MoCA के माध्यम से), डेवलपर उत्पादकता (के माध्यम से)। MoCA और FenSying), और रनटाइम दक्षता (FenSying के माध्यम से)। प्रत्येक तकनीक है एक प्रभावी उपकरण समर्थन के साथ।

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrency in computer systems involves the simultaneous execution of multiple instructions by different processing elements. Concurrency can provide significant benefits, including improved response time, higher throughput, and better resource utilization. To facilitate concurrent programming, modern programming languages have included concurrent processing elements as a first-class programming construct. However, concurrency can also introduce complex interactions and data dependencies, which has led to increased interest in analyzing concurrent programs for correctness and efficiency within the research community.

Concurrent programs may be executed out-of-order, *i.e.*, their execution order may differ from that specified in the program. The out-of-order execution is done to reduce the stalls associated with store instructions and to support compiler optimizations. While this can increase efficiency, it also introduces additional scheduling overhead and programming complexity. Furthermore, if a program's correctness depends on the completion of memory accesses in a specific order, out-of-order execution can lead to errors.

In order to ensure correct execution of concurrent programs, it may be essential to impose limitations on the allowable orderings of memory accesses. This is where *memory consistency models* become significant, as they define the required constraints

and guarantee that concurrent programs run coherently.

Different architectures and programming languages define their own memory consistency models, which can vary in their strictness and performance characteristics. The most restrictive model is the *sequentially consistent* memory model, which only allows outcomes that are consistent with some sequential order of memory accesses. While intuitively appealing, this model severely limits the possible execution orders of instructions.

Most real-world architectures and programming languages support more relaxed memory consistency models that allow for greater performance but may permit out-of-order executions. However, these models exacerbate the *state space explosion* problem, where the number of reachable program states grows exponentially with the number of concurrent processing elements and instructions. This problem makes it difficult to reason about and analyze concurrent programs accurately.

Despite these challenges, significant progress has been made in the last decade in understanding the semantics of relaxed memory concurrency and developing effective analysis techniques. However, there are still significant challenges in scaling analysis techniques, accurately detecting feasible program outcomes, and understanding the complex prose-style memory model specifications with mathematical rigor.

To address these challenges, this work proposes techniques to facilitate the development of efficient concurrent programs that produce expected outcomes under various memory consistency models. The techniques focus on,

- *verification efficiency*, targeting legacy and developed programs, and
- *developer productivity* and *runtime efficiency*, targeting programs under development.

Note.

Verification is establishing the correctness of a concurrent program. The techniques analyze all possible program outcomes under a memory consistency model, to validate that a specified property is satisfied for every program run. Most verification

techniques also provide an error trace if the property may be violated. Verification efficiency focuses on improving the time of analysis, the memory requirement, and the scalability of the verification process.

One effective verification technique is *stateless model checking*, which analyzes a reduced state graph to tackle the *state space explosion*. The reduced state graph comprises representative program executions from each set of *equivalent* executions, where equivalence is established based on an *equivalence relation*. This work proposes a novel equivalence relation on program executions called *view-equivalence*, which is at least as coarse as any existing equivalence relation. As a result, view-equivalence induces the smallest number of partitions of equivalent executions, which reduces the number of representative executions to be analyzed and positively impacts the stateless model checking effort. This work also proposes a stateless model checker, called `ViEqui`, to analyze concurrent programs based on view-equivalence partitioning. `ViEqui` is sound, complete, and optimal in its analysis.

Further, this work addresses the challenge of *stateless model checking* for concurrent programs under relaxed memory consistency. Different computer architectures have varying memory consistency models, which dictate the feasible outcomes of a concurrent program. Likewise, programming languages define their memory models for proper compiler transformations and optimizations. However, outcomes that are permitted by a language's ordering specification may not manifest on an architecture's memory model with a stronger implicit ordering specification. It should be noted that existing verification techniques are either designed for the memory model of an architecture or the memory model of a language, which can lead to imprecise analysis for outcomes that are permitted by the language and producible on the architecture.

In this context, this work proposes a stateless model checker called `MoCA` that analyzes `C/C++` programs under a class of architecture memory models called *multi-copy atomics*. The model checker is designed to be sound and precise for coherent `C/C++` outcomes that can manifest on a *multi-copy atomic* architecture. The precise analysis reduces the reachable state graph, leading to better verification efficiency. Moreover,

the precise set of reported bugs reduces the burden of sorting feasible from infeasible bugs or the burden of fixing a larger set of bugs, thereby increasing developer productivity.

Note.
Developer productivity focuses on easing the development process by improving the programming constructs, unambiguous presentation of the memory model specifications, quick and precise detection of bugs, and automated program synthesis for fixing bugs or improving performance, among others.

The work with MoCA exposes the complex semantics of the C/C++ memory model that a developer of the language faces. Among existing memory models, the C/C++ memory model is one of the most relaxed, making it difficult to comprehend the feasible program outcomes. Even experts of the language struggle to find a balance between efficient use of the permitted relaxations and ensuring correctness.

To tackle this issue, this work proposes a fence synthesis technique called FenSying that can automatically correct concurrent buggy programs under the C/C++ memory model. The technique accomplishes this by using fences to preserve the ordering between program instructions that are relaxed under the memory model. Furthermore, the FenSying technique produces the most runtime-efficient version of the corrected C/C++ program.

Note.
Runtime efficiency deals with improving the performance of a concurrent program in terms of response time, execution time, and effective resource utilization. Runtime efficiency also focuses on efficiently exploiting the ordering relaxation permitted by the language and architecture.

FenSying is sound and optimal, meaning that it fixes a C/C++ buggy program while adding minimal fence overhead. However, the optimality of fence placement is an

NP-hard problem. To mitigate the optimality toll, this work proposes a related fence synthesis technique called fastFenSying. The fastFenSying technique is sound and significantly outperforms the optimal technique, albeit with a small possibility of incurring extra fences.

FenSying and fastFenSying are the first fence synthesis techniques developed for the C/C++ memory model. The techniques perform fence synthesis while maintaining the portability of the C/C++ program.

Each of the techniques, ViEqui, MoCA, FenSying, and fastFenSying, are accompanied with an implementation.

The ViEqui technique is implemented in C++, over a tool called Nidhugg, for concurrent C/C++ programs. The technique is validated on 16,154 litmus tests of multi-threaded C programs. ViEqui is compared on the time of analysis and scalability against state-of-the-art stateless model checkers. Due to the effective reduction in the state graph and optimal exploration, ViEqui performs verification over 10x faster in ∼23% of tests and times out in ∼29% lesser tests than the fastest comparative technique (∼30% and ∼46%, respectively on an average across techniques).

The MoCA technique is implemented in C++, over a tool called rInspect, for concurrent C/C++ programs using C/C++ memory model semantics. The technique is validated on diy7 family of litmus tests and small tests from SV-Comp suite. MoCA is compared on the time of analysis, scalability, and precision against state-of-the-art stateless model checkers for C/C++ memory model and non-multi-copy atomic memory models. MoCA alone precisely detects the feasible outcomes for the cross-section of C/C++ and multi-copy atomic memory models. Further, due to the precision of analysis, MoCA performs 3.3x faster and times out in ∼12.5% lesser tests than the comparative technique.

The FenSying and fastFenSying techniques are implemented in Python for concurrent C/C++ programs. The techniques are validated on 1,389 buggy C programs built on C/C++ memory model semantics. The performance of FenSying and fastFenSying are compared on the time of analysis and scalability. The *near*-optimal fastFenSying technique performs over ∼100x faster in nearly 41% of tests (67x in general) and times

out in ∼35% lesser tests than the optimal `FenSying` technique. The `fastFenSying` technique produces the optimal result in ∼99.5% of the tests.

Thesis Statement

This work proposes techniques for efficient development of concurrent programs under various memory consistency models, focusing on verification efficiency for both legacy and developed programs, as well as developer productivity and runtime efficiency for programs under development.

Document structure

The remaining document is structured as follows.

Chapter 2. Preliminaries

The chapter introduces terms and notations used through this document. The chapter also introduces the operational assumptions.

Chapter 3. Background

This chapter briefly presents background details on concepts and models relevant for this work. This chapter includes details on concurrency, relevant memory models, and brief introduction to model checking.

Chapter 4. ViEqui. Stateless Model Checking based on View-equivalence

This chapter discusses the various approaches for efficient model checking including reducing the number of equivalence partitions. It discusses the existing equivalence relations, and the difference and applicability of the novel view-equivalence relation in comparison to the existing relations. The chapter presents the ViEqui stateless model checker including the proposed analyses, representations and the ViEqui algorithm. It presents proofs of various claims on ViEqui and presents its time and space complexity. It further presents the implementation details and experimental study on ViEqui. The chapter also discusses the scope of the work and compares against similar notions. Finally, based on the scope, the chapter presents future directions.

Chapter 5. MoCA. Dynamic Verification of C11 Concurrency over Multi Copy Atomics

This chapter discusses the imprecision in the analysis of the existing techniques *wrt* feasibility of outcomes on multi-copy atomics (MCA). It presents the formal specification of the MCA model. It then introduces a novel restriction of the C/C++ memory model for MCA, and further restrictions for specific MCA models called TSO and ARM; including proposed analyses and representations. The chapter presents proofs

of various claims on MoCA and presents its time complexity. It further presents the implementation details and experimental study on MoCA. The chapter also discusses the scope of the work and presents future directions.

Chapter 6. (fast)FenSying. Fence Synthesis for the C11 Memory Model

The chapter discusses the complexity of C/C++ memory model and the need for a fence synthesis technique. It then discusses the first fence synthesis techniques for the C/C++ memory model, including proposed analyses, and the FenSying and fastFenSying algorithms. It presents proofs of various claims on FenSying and fastFenSying and presents their time complexity. It further presents the implementation details and experimental study on FenSying and fastFenSying. The chapter also discusses the scope of the work and compares against similar notions. Finally, based on the scope the chapter presents future directions.

Chapter 7. Conclusion

This chapter summarizes the work presented in this document.

Chapter 2

Preliminary Setup

Concurrent model

Consider an acyclic multi-thread program P on a finite set of program threads. Each thread of P has deterministic computations and terminating executions. Let $\mathbb{T}$ represent the finite set of threads of P. Each thread in $\mathbb{T}$ is represented by a unique whole number added as a subscript of $\mathbb{T}$, for instance the i^{th} thread of P is represented as $\mathbb{T}_i$. The threads in P access a fixed set of memory locations called objects. The set of objects is denoted by $\mathcal{O}$. The input program operates on fixed input values and the only source of non-determinism is the scheduling non-determinism.

Each thread of P constitutes a sequence of *actions* from the set $\mathcal{A} = \{$`write`, `read`, `rmw`, `fence`$\}$ representing, respectively, the `write` operation or store to an object in $\mathcal{O}$, the `read` operation or load of an object in $\mathcal{O}$, the read-modify-write operation that atomically reads an object in $\mathcal{O}$, modifies the object and writes it back, and a `fence` memory synchronization action (discussed in §3.2.3).

Memory access actions on objects that are shared between threads are called *shared* or *global* memory accesses and memory access actions on thread local objects are called *local* memory accesses. For the purpose of program analysis[1]conducted in this study, the local memory accesses are considered *invisible* actions, while the shared memory

accesses and `fences` are considered *visible* actions.

Program events

The runtime instance of a visible action is called an *event*. An event comprises the effect of a sequence of local actions followed by a visible action. A thread executes a sequence of events.

Note that, multiple executions of a program location yield different events. Thus, an event is uniquely identified in a program execution, however, multiple events may be associated with the same program location. An event is formally defined as,

Definition 1. (Event)

An event e is a tuple $\langle thr(e), idx(e)\ act(e), obj(e), ord(e), loc(e)\rangle$ where,

$thr(e)\ \in \mathbb{T}$, represents the thread of e;

$idx(e)$ represents the unique identifier of e;

$act(e)\ \in \mathcal{A}$, is the event action;

$obj(e)\ \subseteq \mathcal{O}$, is memory objects accessed by e;

$ord(e)$ is the memory order associated with e; and

$loc(e)$ is the program location corresponding to e.

Note that, the events may be referred to by their actions, for instance, 'a `read` event e' or simply 'a `read` e' refers to an event e such that $act(e) =$ `read`. The $obj(e)$ for a `fence` event e is not valid as a `fence` does not access an object. Under the C/C++ memory consistency model an event e is additionally associated with a *memory order*, represented as $ord(e)$. Memory orders are formally introduced in §3.2.3.

[1] We use the term program analysis as opposed to program verification for the proposed techniques to include a broader scope of analyses encompassing the verification techniques along with various static and dynamic analysis.

The set of events of P is represented by $\mathcal{E}$. The notations $\mathcal{E}^{\mathbb{W}}$, $\mathcal{E}^{\mathbb{R}}$ and $\mathcal{E}^{\mathbb{F}}$ are used to represent the set of `writes`, the set of `reads`, and the set of `fences` respectively. Note that, an `rmw` event belongs to both the sets $\mathcal{E}^{\mathbb{W}}$ and $\mathcal{E}^{\mathbb{R}}$.

Further, for each shared object o, consider a special *initial event* ($\mathbb{I}_o$). The initial event provides a well-defined initial value for its corresponding object.

Event sequences and program executions

A sequence of program events, $\tau = e_1.e_2, ...e_n$ where $e_i \in \mathcal{E}$ for $i \in [1, n]$, is maximal if after the sequence, there are no events *enabled* or available for execution. A maximal event sequence represents a *program execution* or an *execution sequence*. A *sequence* may refer to either non-maximal sequences or maximal sequences of events.

Given a sequence τ, notations $\mathcal{E}_\tau$, $\mathcal{E}_\tau^{\mathbb{W}}$ and $\mathcal{E}_\tau^{\mathbb{R}}$ represent respectively the set of events, the set of `writes` and the set of `reads` occurring in τ. In a sequence τ, $val_{[\tau]}(e)$ represents the value of an event e from a finite set of program values $\mathcal{V}$. This represents the value written for a `write` event and the value `read` for a read event.

Consider $e_1, e_2 \in \mathcal{E}_\tau$, e_1 occurs-before e_2 in τ is represented as $e_1 <_\tau e_2$. Given $e_r \in \mathcal{E}_\tau^{\mathbb{R}}$, the latest `write` of $obj(e_r)$ in the prefix of τ up to e_r is represented by $\mathsf{last}W_{[\tau]}(e_r)$. However, if there does not exist a `write` event in the prefix of τ up to e_r then $\mathsf{last}W_{[\tau]}(e_r) = \mathbb{I}_{obj(e_r)}$. Note that, under strong models of memory consistency (refer to §3.2) such as the *sequentially-consistent* memory model (refer to §3.2.1) e_r reads the value of $\mathsf{last}W_{[\tau]}(e_r)$ in τ.

A sequence τ is extended by an event e or a sequence τ' as $\tau.e$ (respectively $\tau.\tau'$). An empty sequence is represented by $\langle\rangle$. The notation $e \in A$ is used to represent that an event e is contained in A, where A can be a set or a sequence of events. Given sequences τ_1, τ_2, notation $\tau_1 \sqcap \tau_2$ represents the longest sequence τ that is a subsequence of τ_1 and τ_2, and $\tau_1 - \tau_2$ represents the remaining sequence of τ_1 after removing its subsequence τ_2.

Exploration states.

A state of exploration (or simply a state) is defined as the valuation of shared objects. The state reached after executing a sequence τ is denoted as $s_{[\tau]}$.

The set of enabled events at a state $s_{[\tau]}$ is represented as $\mathsf{En}(s_{[\tau]})$. An execution transitions from a state $s_{[\tau]}$ to the next state $s_{[\tau.e]}$ on execution of an enabled event e.

Given a state $s_{[\tau]}$, $\mathbf{shr}(s_{[\tau]})$ denotes the valuation of shared objects, that is, a set of pairs of $(\mathcal{O}, \mathcal{V})$ representing a shared object and its corresponding shared value. Similarly, $\mathbf{lcl}(s_{[\tau]})$ denotes the valuation of local objects, again a set of pairs of $(\mathcal{O}, \mathcal{V})$ representing a thread local object and its corresponding thread local value.

Relational Operators

R^{-1} represents the inverse and R^{+} represents the transitive closure of a relation R. Further, $R_1; R_2$ represents the composition of relations R_1 and R_2.

Given a relation R, $(a, b) \in R$ is also represented with a corresponding infix notion $a{\rightarrow}^{R}b$. Similarly, $(a, b) \notin R$ is represented with a corresponding infix notion $a{\not\rightarrow}^{R}b$.

Lastly, a relation R has a cycle (or is cyclic) if $\exists e \in \mathcal{E}$ s.t. $e{\rightarrow}^{R}e$.

Figure notations

The notation $W(o, v)$ represents a `write` to a shared object o of value v. Similarly, $R(o, v)$ represents the `read` of a shared object o and value v, and $R(o)$ represents the program event `read` of a shared object o (not associated with a sequence). The parallel bars ($\|$) represent the parallel composition of events of program threads. The events in figures may be referred to by their labels, for instance, consider the notation $a : W(x, 1)$, where a `write` event of object x with value 1 is labeled with a, the event may be referred to as the event a.

Chapter 3

Background

3.1 Introduction to Concurrency

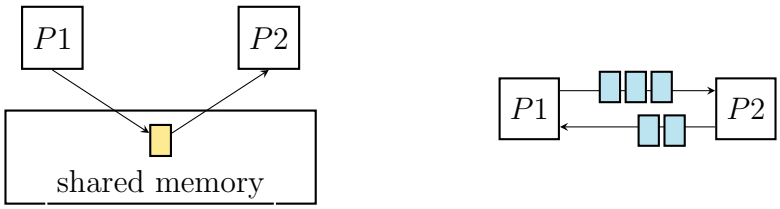
Simultaneous execution of multiple sequences of events is called *concurrency*. The sequences of events are executed by several concurrent processing elements such as program threads. Concurrent execution significantly improves the processing speed and offers better resource utilization.

Concurrency is ubiquitous. Various programming languages have included concurrent processing as a first-class programming construct. Multicore systems are globally used, and even on uniprocessor systems, the applications are typically concurrent.

3.1.1 Models of concurrency

There are two models of concurrency, the *shared memory* model, and the *message passing* model. The models are diagrammatically represented in Figure 3.1.

Under the shared memory model of concurrency, concurrent processing elements communicate by `read` accesses and `write` accesses to a shared memory. The model is



shared memory model message passing model

Figure 3.1: Models of concurrency

typically associated with a uniprocessor or a multiprocessor system, where the processing elements and the shared memory reside on the same system. The model allows a faster communication but explicit care is needed in ensuring a coherent access to the shared memory.

Under the message passing model, concurrent processing elements communicate by sending and receiving messages through a communication channel. The model is typically associated with a distributed system where the processing elements reside on separate machines. It is relatively slower than shared memory and is used for exchange of small amounts of data.

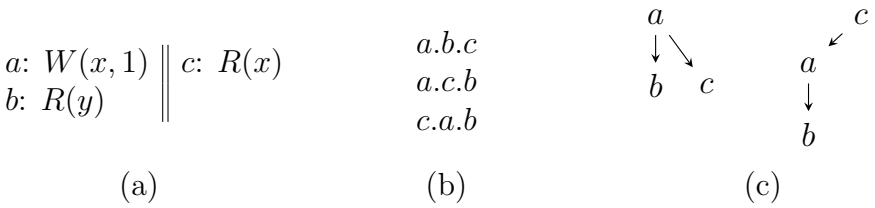
3.1.2 Models of reasoning about concurrency

There are two popular models of reasoning about concurrency, the *interleaving model* of reasoning and the *partial-order* based model of reasoning.

Under the interleaving model of reasoning events from different processing elements are non-deterministically *weaved* or *interleaved* to represent a program execution sequence. As a consequence, the set of all possible interleavings represents the set of all feasible program outcomes. The interleaving model of reasoning about concurrent programs abstracts from the multiple processing elements and reasons about the program as a single sequence of events.

The partial order model of reasoning represents the input program by a set of partial

orders such that each partial order represents a set of program executions that reach the same outcome.



(a) (b) (c)

Figure 3.2: WR-R. (a) input program, (b) interleavings, (c) partial orders

Consider the input program in Figure 3.2(a). Figure 3.2(b) represents the feasible interleavings of the input program, where the event c interleaves with the events of the other thread. Figure 3.2(c) represents the feasible partial orders of the input program. The order of execution of the `write` and `read` events of x result in different program outcomes. The partial orders represent those program outcomes.

3.1.3 State-space explosion problem

The combinations of interleaving of concurrent events grows exponentially with number of concurrent processing elements and concurrent events. The condition is called a *combinatorial explosion* in program interleavings.

As a result of the combinatorial explosion, the set of reachable program states, called the state space, also grows exponentially. The exponential increase of the state space with the increase in the number of concurrent processing elements and events is called the *state space explosion* problem.

The state space explosion escalates the challenge of reasoning about concurrent programs and negatively impacts the time of analysis and scalability of analysis techniques.

$$\begin{array}{l|l} a\colon W(x,1) & c\colon W(y,1) \\ b\colon R(y,0) & d\colon R(x,0) \end{array}$$

(a)

$$\begin{array}{l|l} a\colon W(x,1) & c\colon R(y,1) \\ b\colon W(y,1) & d\colon R(x,0) \end{array}$$

(b)

$$\begin{array}{l|l|l|l} a\colon W(x,1) & b\colon R(x,1) & d\colon W(y,1) & e\colon R(y,1) \\ & \downarrow_{addr} & & \downarrow_{addr} \\ & c\colon R(y,0) & & f\colon R(x,0) \end{array}$$

(c)

Figure 3.3: Concurrent programs. (a) Store buffer, (b) Message passing, (c) IRIW+addr ($\downarrow_{addr}$ represents address dependence)

3.2 Memory Models

Model of memory consistency, a.k.a memory model, is a specification of the permitted outcomes of multi-threaded programs executing with shared memory. A memory model describes the permitted reorderings on concurrent events, where reordering refers to execution of events in a different order than that specified in the program. A memory model also specifies the permitted accesses to the shared memory and, thus, the permitted inter-thread interactions.

The memory models can range from strong to weak. The stronger memory models allow a low degree of reordering and restrict the permitted inter-thread interactions. The weak memory models are liberal with permitted reorderings and inter-thread interactions. Weak memory models enable compiler and hardware optimizations that improves system performance. However, even for experts, comprehending the program outcomes under weak memory models is formidable.

3.2.1 Sequential consistency

Sequential consistency, or sequentially consistent memory model is defined by the following property [61],

"the result of any execution is the same as if the operations of all the threads were executed in some sequential order, and the operations of each individual thread appear in this sequence in the order specified by its program".

The definition implies that all threads agree on an order of occurrence of program events and the agreed upon order is consistent with the order of events in the input program (called the *program-order*). The model does not support reordering on program events.

Consider the three programs outcomes shown in Figure 3.3, where the `read` events read the values shown in the figure. Any of the three outcomes cannot be explained by a sequential occurrence order on the events for a single shared memory. As the result the three outcomes are infeasible under sequential consistency and cannot manifest on an architecture that follows sequential consistency.

Such sequential ordering levies a heavy performance penalty. The model also restricts a large set of compiler optimizations.

3.2.2 Multi-copy Atomics (MCA)

Multi-copy atomicity refers to a class of memory models where the `writes` of the memory models satisfy the conjunction of the following conditions [64].

- All `writes` to the same location are serialized, meaning they are observed in the same order by all threads, although some threads might not observe all of the `writes`.
- A `read` does not read the value of a `write` from a different thread until all threads observe that `write`.

Intuitively, under multi-copy atomicity, a `write` is advertised to all other threads simultaneously, however, the `write` is available to its own thread prior to being advertised to all other threads.

Multi-copy atomic (MCA) is a popular class of memory models that is supported by memory models such as Intel x-86 TSO, sparc PSO, newer versions of ARM (version 8 and later), and Alpha. However, each of the models that support multi-copy atomicity defines its own set of permitted reorderings. The reordering conditions are conjuncted with the above stated conditions of multi-copy atomicity to define the permitted outcomes under the models of various MCA architectures.

The outcomes shown in Figure 3.3(a) and (b) can be explained under the conditions stated above and, thus, represent multi-copy atomic outcomes. On the other hand, in Figure 3.3(c) the events of a thread are address dependent (represented by *addr*) disallowing any reordering. As a result, the outcome shown in Figure 3.3(c) is feasible only if the `writes` become visible to different threads at different times. Thus, the outcome is non-multi-copy atomic.

Total Store Order (TSO)

The TSO model relaxes the `write-read` ordering allowing `writes` from a thread to complete after a later `read` of different object from the same thread. Here, later implies a higher event index. x86 architecture supports the TSO memory model.

Consider the program outcome in Figure 3.3(a), the outcome can be explained by reordering the `write` events of either of the two threads with the `read` events of the same thread. Therefore, the outcome is feasible under TSO. A similar reordering cannot occur for Figure 3.3(b), and (c), thus, the outcomes are not valid under TSO.

Partial Store Order (PSO)

In addition to the `write-read` ordering, the PSO model also relaxes the `write-write` ordering allowing `writes` from a thread to complete after a later `write` of different objects from the same thread.

Consider the program outcomes in Figure 3.3(a), and (b), the outcomes can be ex-

plained by `write-read` and `write-write` reorderings respectively. Therefore, the outcomes are feasible under PSO. A similar reordering cannot occur for Figure 3.3(c) and, thus, the outcome is not valid under PSO.

ARM version 8 and higher (ARM)

The ARM memory model (associated with versions 8 and later) [64] allows the reordering of every pair of events from a thread. The model defines a set of coherence conditions that describe the valid reorderings. As a result a program outcome that is feasible with reorderings but does not satisfy the ARM coherence conditions is considered invalid. For instance, the model does not permit reordering of program (data, address, register, control) dependent events.

The necessary reorderings for the outcomes shown in Figure 3.3(a), and (b) are valid under ARM and, thus, the outcomes are feasible under the model. The outcome shown in Figure 3.3(c) is not feasible under ARM since the `reads` are address dependent and cannot reorder. In the absence of such an address dependence, the outcome would be feasible under ARM.

3.2.3 Non-multi-copy Atomics (non-MCA)

Non-multi-copy atomicity refers to a class of memory models where the `writes` may become visible to different threads at different times. The memory models that do not always satisfy the conditions of a MCA model fall in the class of non-multi-copy atomic (non-MCA) memory models.

The outcomes of a non-MCA model are popularly explained using the *Flowing model* of Flur *et al.* [36]. non-MCA is supported by memory models such as the memory model of earlier versions (version 7 and earlier) of ARM architecture, Power and language models such as the C/C++ memory model.

Each of the outcomes shown in Figure 3.3(a), (b) and (c) is a valid outcome under a

non-MCA model.

C/C++ ISO 2011 memory model (C11)

The C/C++ 2011 (and subsequent) ISO standard introduces a memory model to define the semantics of computer memory storage for the purpose of the C/C++ abstract machine. The model lays down semantics of concurrent memory accesses in C and C++. In this document the model is referred to as C11.

Non-atomic and atomic memory access. The model supports two categories of events - non-atomic and atomic.

The events of an atomic object that are called atomic events. Atomic events are specifically intended for shared objects. Data races on atomic events are permitted or defined under C11. However, the compilers and underlying architectures have to introduce memory barriers and impose additional restrictions to maintain the atomicity and impose ordering restrictions.

The events of a non-atomic object are called non-atomic events. They are the general events and can be translated to plain load and store instructions. However, data races on such events are not permitted under C11 and lead to *undefined* behavior.

Memory orders in C11. C11 provides a spectrum of ordering restrictions that can be imposed on an atomic event. The atomic events are tagged with *memory orders* that define the ordering restriction on atomic and non-atomic events around them. The memory orders provide options to pose no ordering restriction all the way to strong ordering restrictions that may restore sequential consistency.

Let $\mathcal{M} = \{\texttt{na}, \texttt{rlx}, \texttt{rel}, \texttt{acq}, \texttt{acq-rel}, \texttt{sc}\}$, represent the set of memory orders. A non-atomic event is recognized by the `na` memory order. The atomic events may be associated with the memory orders relaxed (`rlx`), release (`rel`), acquire/consume (`acq`), acquire-release (`acq-rel`) and sequentially consistent (`sc`) associating varying degrees of ordering guarantees. Let $\sqsubset \subseteq \mathcal{M} \times \mathcal{M}$ represent the relation *weaker* such that $m_1 \sqsubset m_2$ represents that the m_1 is weaker than m_2. As a consequence, tagging

an event with m_2 may order events that are unordered with m_1. The orders in $\mathcal{M}$ are related as,

$$\texttt{na} \sqsubset \texttt{rlx} \sqsubset \{\texttt{rel}, \texttt{acq}\} \sqsubset \texttt{acq-rel} \sqsubset \texttt{sc}.$$

Also, the relation $\sqsubseteq$ represents *weaker or equally weak*. Similarly, $\sqsupset$ represents *stronger* and $\sqsupseteq$ represents *stronger or equally strong*.

Let $\mathcal{E}^{(m)}$ represent the set of events ordered with the memory order m. Similarly, let $\mathcal{E}^{\mathbb{W}(m)}$ and $\mathcal{E}^{\mathbb{R}(m)}$ represent the set of `writes` and the set of `reads`, respectively, that are ordered with the memory order m. Further, let $\mathcal{E}_\tau^{(m)}$, $\mathcal{E}_\tau^{\mathbb{W}(m)}$ and $\mathcal{E}_\tau^{\mathbb{R}(m)}$ represent the set of events, `write` events and `read` events, respectively, of a sequence τ that are ordered with the memory order m.

The operators $\sqsubset$ and $\sqsubseteq$ are overloaded as unary operators that return the set of memory orders that are weaker, and weaker or equally weak respectively. For instance, $\sqsubseteq\texttt{rel} = \{\texttt{na}, \texttt{rlx}, \texttt{rel}\}$. Similarly, the operators $\sqsupset$ and $\sqsupseteq$ are overloaded to represent the set of memory orders that are stronger, and stronger or equally strong respectively. For instance, $\sqsupseteq\texttt{rel} = \{\texttt{rel}, \texttt{acq-rel}, \texttt{sc}\}$.

Given a relation R, the notation $R|_{\texttt{sc}}$ represents a subset of R on `sc` ordered events; *i.e.* $(e_1, e_2) \in R|_{\texttt{sc}} \Leftrightarrow (e_1, e_2) \in R \wedge e_1, e_2 \in \mathcal{E}^{(\texttt{sc})}$.

Definition 2. (C11 Trace)

$$
\begin{array}{lcl}
e_1 \rightarrow_\tau^{\mathbf{sb}} e_2 & \Leftarrow & \forall e_1, e_2 \in \mathcal{E}_\tau \text{ s.t. } thr(e_1) = thr(e_2) \text{ and } e_1 \text{ occurs before } e_2 \text{ in their thread.} \\
e_w \rightarrow_\tau^{\mathbf{sw}} e_r & \Leftarrow & e_w \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \texttt{rel})},\ e_r \in \mathcal{E}_\tau^{\mathbb{R}(\sqsupseteq \texttt{acq})},\ thr(e) \neq thr(e_1) \wedge e_w \rightarrow_\tau^{\mathbf{rf}} e_r. \\
e_w \rightarrow_\tau^{\mathbf{dob}} e_r & \Leftarrow & e_w \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \texttt{rel})},\ e_r \in \mathcal{E}_\tau^{\mathbb{R}(\sqsupseteq \texttt{acq})},\ thr(e) \neq thr(e_1) \wedge \exists e'_w \in \textit{release-sequence} \\
 & & \text{of } e_w \text{ s.t. } e'_w \rightarrow_\tau^{\mathbf{rf}} e_r. \\
e_1 \rightarrow_\tau^{\mathbf{ithb}} e_2 & \Leftarrow & \forall e_1, e_2, e_3 \in \mathcal{E}_\tau, \\
 & & \quad (e_1 \rightarrow_\tau^{\mathbf{sw}} e_2) \vee \\
 & & \quad (e_1 \rightarrow_\tau^{\mathbf{dob}} e_2) \vee \\
 & & \quad (e_1 \rightarrow_\tau^{\mathbf{sw}} e_3 \wedge e_3 \rightarrow_\tau^{\mathbf{sb}} e_2) \vee \\
 & & \quad (e_1 \rightarrow_\tau^{\mathbf{sb}} e_3 \wedge e_3 \rightarrow_\tau^{\mathbf{ithb}} e_2) \vee \\
 & & \quad (e_1 \rightarrow_\tau^{\mathbf{ithb}} e_3 \wedge e_3 \rightarrow_\tau^{\mathbf{ithb}} e_2). \\
e_1 \rightarrow_\tau^{\mathbf{hb}} e_2 & \Leftarrow & e_1 \rightarrow_\tau^{\mathbf{sb}} e_2 \vee e_1 \rightarrow_\tau^{\mathbf{ithb}} e_2.
\end{array}
$$

Figure 3.4: C11 hb_τ relation

A *trace* under C11, τ, is a tuple $\langle \mathcal{E}_\tau, \mathrm{hb}_\tau, \mathrm{mo}_\tau, \mathrm{rf}_\tau \rangle$, where

$\mathcal{E}_\tau$ $\subseteq \mathcal{E}$ represents the set of events in the trace τ;

hb_τ (*Happens-before*) $\subseteq \mathcal{E}_\tau \times \mathcal{E}_\tau$ is a partial order which captures the event interactions and inter-thread synchronizations, (discussed below);

mo_τ (*Modification-order*) $\subseteq \mathcal{E}_\tau^{\mathbb{W}} \times \mathcal{E}_\tau^{\mathbb{W}}$ is a total order on the `writes` of an object;

rf_τ (*Reads-from*) $\subseteq \mathcal{E}_\tau^{\mathbb{W}} \times \mathcal{E}_\tau^{\mathbb{R}}$ is a relation from a `write` event to a `read` event signifying that the `read` event takes its value from the `write` event in τ.

A valid outcome or behavior of an input program under C11 is called a *trace* of the program. A C11 trace is defined over a set of events of the input program and relations over the events, as shown in Definition 2.

C11 happens-before relation.

The most significant relation that defines a C11 trace τ is the irreflexive and acyclic happens-before relation, $\text{hb}_\tau \subseteq \mathcal{E}_\tau \times \mathcal{E}_\tau$. The hb_τ relation is composed of the following relations [46] (the relations are formally defined in Figure 3.4).

sb_τ (*Sequenced-before*): total occurrence order on the events of a thread[1].

sw_τ (*Synchronizes-with*) Inter-thread synchronization between a write e_w (ordered $\sqsupseteq$ `rel`) and a read e_r (ordered $\sqsupseteq$ `acq`) when $e_w \rightarrow_\tau^{\mathbf{rf}} e_r$.

dob_τ (*Dependency-ordered-before*): Inter-thread synchronization between a write e_w (ordered $\sqsupseteq$ `rel`) and a read e_r (ordered $\sqsupseteq$ `acq`) when $e'_w \rightarrow_\tau^{\mathbf{rf}} e_r$ for $e'_w \in$ *release-sequence*[2] of e_w in τ.

ithb_τ (*Inter-thread-hb*): Inter-thread relation computed by extending sw_τ and dob_τ with sb_τ.

hb_τ (*Happens-before*): defined as $\text{sb}_\tau \cup \text{ithb}_\tau$.

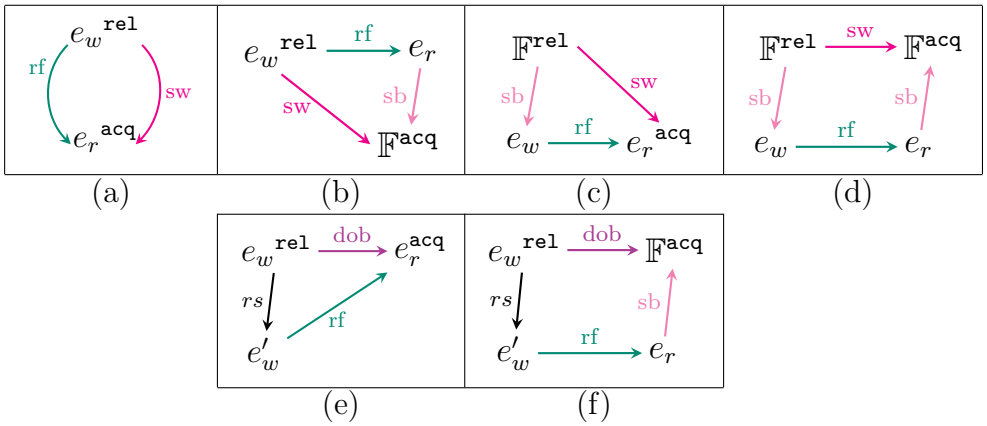
Happens-before relation with C11 `fences`.

C11 `fences` form ithb_τ relation with other events of a trace τ [20, 46]. A `fence` can be associated with the memory orders `rel`, `acq`, `acq-rel` and `sc`. An appropriately placed `fence` can form sw_τ and dob_τ from an rf_τ relation between events of different threads. The conditions for forming sw_τ and dob_τ relations with `fences` are formally presented in Figure 3.5.

Figure 3.5 also represents the conditions diagrammatically; where the black edges from e_w to e'_w labeled rs (in (e) and (f)) represent that e'_w is in the *release-sequence* of e_w. Further, the depictions (a) and (e) represent the conditions in the absence of `fences` (presented in Figure 3.4); the depictions (b)-(d) represent the conditions for forming sw_τ relation with `fences`; and, the depiction (f) represents the condition for forming dob_τ relation with `fences`.

[1] Some events of a thread may not be ordered (for example, the operands of ==). This work assumes sb_τ as a total order on the events of a thread, similar to previous works on C11 [51, 72].

[2] *release-sequence* of e_w in τ: maximal contiguous sub-sequence of mo_τ that starts at e_w and contains: (i) write events of $thr(e_w)$, (ii) rmw events of other threads [20, 46].



sw_τ using C11 `fences`

Given $e_w \rightarrow_\tau^{\mathbf{rf}} e_r$,

if $ord(e_w) \sqsupseteq \texttt{rel}$, $\exists \mathbb{F}^{\texttt{acq}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \texttt{acq})}$ s.t. $e_r \rightarrow_\tau^{\mathbf{sb}} \mathbb{F}^{\texttt{acq}}$ then $e_w \rightarrow_\tau^{\mathbf{sw}} \mathbb{F}^{\texttt{acq}}$;

if $ord(e_r) \sqsupseteq \texttt{acq}$, $\exists \mathbb{F}^{\texttt{rel}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \texttt{rel})}$ s.t. $\mathbb{F}^{\texttt{rel}} \rightarrow_\tau^{\mathbf{sb}} e_w$ then $\mathbb{F}^{\texttt{rel}} \rightarrow_\tau^{\mathbf{sw}} e_r$;

if $\exists \mathbb{F}^{\texttt{rel}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \texttt{rel})}$, $\exists \mathbb{F}^{\texttt{acq}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \texttt{acq})}$ s.t. $\mathbb{F}^{\texttt{rel}} \rightarrow_\tau^{\mathbf{sb}} e_w$, $e_r \rightarrow_\tau^{\mathbf{sb}} \mathbb{F}^{\texttt{acq}}$ then $\mathbb{F}^{\texttt{rel}} \rightarrow_\tau^{\mathbf{sw}} \mathbb{F}^{\texttt{acq}}$.

dob_τ using C11 `fences`

Given $e'_w \rightarrow_\tau^{\mathbf{rf}} e_r$, if $\exists e_w \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \texttt{rel})}$ s.t. $e'_w \in$ *release-sequence* of e_w and $\exists \mathbb{F}^{\texttt{acq}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \texttt{acq})}$ s.t. $e_r \rightarrow_\tau^{\mathbf{sb}} \mathbb{F}^{\texttt{acq}}$ then $e_w \rightarrow_\tau^{\mathbf{dob}} \mathbb{F}^{\texttt{acq}}$.

Figure 3.5: sw_τ and dob_τ using C11 `fences`

Intuitively, a `write` event with a `fence` placed before it in sb_τ and a `read` event with a `fence` placed after it in sb_τ enact the role of strongly ordered `write` and strongly ordered `read` events respectively.

Note that, C11 `fences` are not memory barriers and do not flush `write` values to the shared memory. They are synchronization tools that assist in forming ithb_τ relations [20, 46].

Trace coherence under C11. The hb_τ relation along with the mo_τ and rf_τ relations is used in specifying a set of six *coherence conditions* [46, 85].

(corf) $e_w \rightarrow^{\mathbf{rf}}_\tau e_r \Rightarrow e_r \not\rightarrow^{\mathbf{hb}}_\tau e_w$

(coWW) $\forall e_{w_1}, e_{w_2} \in \mathcal{E}^{\mathbb{W}}_\tau,\ e_{w_1} \rightarrow^{\mathbf{hb}}_\tau e_{w_2} \Rightarrow e_{w_1} \rightarrow^{\mathbf{mo}}_\tau e_{w_2}$

(coRR) $\forall e_{r_1}, e_{r_2} \in \mathcal{E}^{\mathbb{R}}_\tau,\ e_{w_1} \in \mathcal{E}^{\mathbb{W}}_\tau,\ e_{r_1} \rightarrow^{\mathbf{hb}}_\tau e_{r_2} \wedge e_{w_1} \rightarrow^{\mathbf{rf}}_\tau e_{r_1} \Rightarrow e_{w_1} \rightarrow^{\mathbf{rf}}_\tau e_{r_2} \vee (\exists e_{w_2} \in \mathcal{E}^{\mathbb{W}}_\tau$ s.t. $e_{w_1} \rightarrow^{\mathbf{mo}}_\tau e_{w_2} \wedge e_{w_2} \rightarrow^{\mathbf{rf}}_\tau e_{r_2})$

(coRW) $\forall e_{r_1} \in \mathcal{E}^{\mathbb{R}}_\tau,\ e_{w_1} \in \mathcal{E}^{\mathbb{W}}_\tau,\ e_{r_1} \rightarrow^{\mathbf{hb}}_\tau e_{w_1} \Rightarrow \exists e_{w_2} \in \mathcal{E}^{\mathbb{W}}_\tau,\ e_{w_2} \rightarrow^{\mathbf{mo}}_\tau e_{w_1}$ s.t. $e_{w_2} \rightarrow^{\mathbf{rf}}_\tau e_{r_1}$

(coWR) $\forall e_{w_1} \in \mathcal{E}^{\mathbb{W}}_\tau,\ e_{r_1} \in \mathcal{E}^{\mathbb{R}}_\tau,\ e_{w_1} \rightarrow^{\mathbf{hb}}_\tau e_{r_1} \Rightarrow e_{w_1} \rightarrow^{\mathbf{rf}}_\tau e_{r_1} \vee (\exists e_{w_2} \in \mathcal{E}^{\mathbb{W}}_\tau$ s.t. $e_{w_1} \rightarrow^{\mathbf{mo}}_\tau e_{w_2} \wedge e_{w_2} \rightarrow^{\mathbf{rf}}_\tau e_{r_1})$

(tosc) (i) $order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_\tau) \wedge total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_\tau) \wedge \mathrm{hb}_\tau|_{\mathtt{sc}} \cup \mathrm{mo}_\tau|_{\mathtt{sc}} \subseteq \mathrm{to}_\tau$ (coto)

(ii) $\forall e_w \rightarrow^{\mathbf{rf}}_\tau e_r$ s.t. $e_r \in \mathcal{E}^{(\mathtt{sc})}_\tau$

- $e_w \in \mathcal{E}^{(\mathtt{sc})}_\tau \wedge \mathtt{imm\text{-}scr}(\tau, e_w, e_r)$; or, (rfto1)
- $e_w \notin \mathcal{E}^{(\mathtt{sc})}_\tau \wedge \nexists e'_w \in \mathcal{E}^{\mathbb{W}(\mathtt{sc})}_\tau$ s.t. $e_w \rightarrow^{\mathbf{hb}}_\tau e'_w \wedge \mathtt{imm\text{-}scr}(\tau, e'_w, e_r)$. (rfto2)

where,

$order(S, R) \triangleq (\nexists a\ R(a,a)) \wedge (R^+ \subseteq R) \wedge (R \subseteq S \times S)$.

$total(S, R) \triangleq \forall a, b \in S \Rightarrow a = b \vee R(a,b) \vee R(b,a)$.

$\mathtt{imm\text{-}scr}(\tau, a, b) \triangleq a \in \mathcal{E}^{\mathbb{W}(\mathtt{sc})}_\tau,\ b \in \mathcal{E}^{\mathbb{R}(\mathtt{sc})}_\tau,\ a \rightarrow^{\mathbf{to}}_\tau b$ and $obj(a) = obj(b) \wedge \nexists c \in \mathcal{E}^{\mathbb{W}(\mathtt{sc})}_\tau$ s.t. $obj(c) = obj(a) \wedge a \rightarrow^{\mathbf{to}}_\tau c \rightarrow^{\mathbf{to}}_\tau b$

(tofen) $\forall e_w \rightarrow^{\mathbf{rf}}_\tau e_r$ s.t. $e_w \in \mathcal{E}^{(\mathtt{sc})}_\tau$, if $\exists \mathbb{F} \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_\tau$ s.t. $\mathbb{F} \rightarrow^{\mathbf{sb}}_\tau e_r$ then $e_w \rightarrow^{\mathbf{to}}_\tau \mathbb{F} \wedge \nexists e'_w \in \mathcal{E}^{\mathbb{W}(\mathtt{sc})}$ where $e_w \rightarrow^{\mathbf{to}}_\tau e'_w \rightarrow^{\mathbf{to}}_\tau \mathbb{F}$.

(cormw) $\forall\ e \in \mathcal{E}_\tau,\ act(e) = \mathtt{rmw}, \exists\ e_w \rightarrow^{\mathbf{rf}}_\tau e$ s.t. $e_w \rightarrow^{\mathbf{mo}}_\tau e \wedge \nexists\ e'_w$ s.t. $e_w \rightarrow^{\mathbf{mo}}_\tau e'_w \rightarrow^{\mathbf{mo}}_\tau e$

Figure 3.6: C11 coherence conditions

Formally, a trace is coherent under C11 if it satisfies the conjunction of the following `coherence conditions`. The conditions are formally presented in Figure 3.6 and discussed below.

(corf) A `read` event does not take its value from a `write` event hb_τ ordered after the `read`.

(coWW) `writes` ordered by hb_τ are also ordered by mo_τ.

(coRR) A `read` (e_{r_2}) hb_τ ordered after another `read` (e_{r_1}) does not read from a `write` mo_τ ordered before the `write` that e_{r_1} reads from.

(coRW) A `read` hb_τ ordered before a `write` reads from another `write` mo_τ ordered before that `write`.

(coWR) A `read` hb_τ ordered after a `write` (e_{w_1}) does not read from a `write` mo_τ ordered before e_{w_1}.

(tosc) The `sc` events[3]of a trace must form a total order (to_τ), such that, the hb_τ and mo_τ ordered `sc` events are also to_τ ordered. Further, a `sc read` reads from the immediately to_τ ordered before `sc write` or any non-`sc write` that is hb_τ ordered after the immediately to_τ ordered before `sc write`.

(tofen) If there exists an `sc fence` before a `read` and the `read` reads from an `sc write` then the `sc write` is immediately to_τ ordered before the `sc fence`.

(cormw) an `rmw` event reads from the immediately mo_τ ordered before event.

Coherence of a trace under the C11 model is also popularly interpreted as the conjunction of the following constraints [60]. The constraints are defined as compositions of event relations hb_τ, mo_τ, and rf_τ that must be irreflexive.

hb_τ is irreflexive.	(co-h)
$\mathrm{rf}_\tau; \mathrm{hb}_\tau$ is irreflexive.	(co-rh)
$\mathrm{mo}_\tau; \mathrm{hb}_\tau$ is irreflexive.	(co-mh)
$\mathrm{mo}_\tau; \mathrm{rf}_\tau; \mathrm{hb}_\tau$ is irreflexive.	(co-mrh)
$\mathrm{mo}_\tau; \mathrm{hb}_\tau; \mathrm{rf}_\tau^{-1}$ is irreflexive.	(co-mhi)
$\mathrm{mo}_\tau; \mathrm{rf}_\tau; \mathrm{hb}_\tau; \mathrm{rf}_\tau^{-1}$ is irreflexive.	(co-mrhi)

Figure 3.7 diagrammatically represents conditions on events of a trace that violate the above stated constraints.

3.3 Model checking concurrent software

Model checking is an automated verification technique that explores all possible program states in a systematic manner. Given a reachable state graph of an input

[3]Given a memory order m, 'm events' (similarly 'm `writes`' and 'm `reads`') is used to describe the set of events (similarly `writes` and `reads`) with the memory order m. For instance `sc` events refers to the set of events with memory order `sc`.

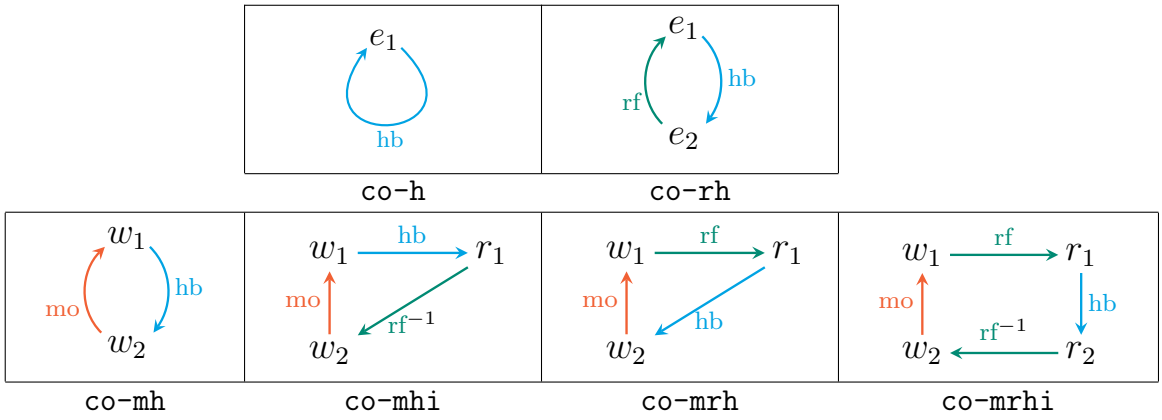


Figure 3.7: Conditions that violate C11 coherence

program and a formal property, a model checking technique examines all possible program outcomes. In this way, it can be shown that a given system model truly satisfies a certain property.

The model checking technique does not bias towards errors that are more likely to occur, thus, even the subtle errors that remain undiscovered using testing and simulation can be revealed using model checking.

Model checking suffers from the state space explosion problem. However, with the use of elegant algorithms and efficient data structures, model checkers can scale to larger state spaces.

Scheduling program executions. For the maximal coverage of the reachable state space, model checkers orchestrate the order of execution of program events to systematically reach various program states. A scheduler controls the execution steps to achieve a predetermined execution sequence.

At each state of exploration a model checker determines a representative set of the various schedules that must be explored from the state. The classic representations include *ample sets* [74] and *persistent sets* [39, 42, 86].

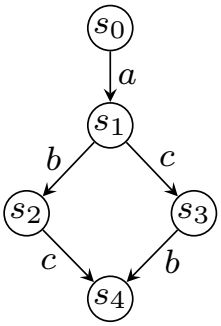


Figure 3.8: commutativity of concurrent events

3.3.1 Stateless and Stateful model checking

A stateful technique captures the local and shared program states at runtime. Thus the technique can accurately detect revisits to the same state during exploration of the state space and avoid repetitive explorations. Capturing the entire state space of an input program can be difficult to implement and expensive in memory requirement. However, the memory requirement may be tackled by efficiently tracking state changes.

A stateless technique explores the reachable state space without explicitly storing the program states. Stateless techniques reset the input program for each fresh execution and replay the execution steps to reach an intermediate state in exploration. Thus, a state can be represented by its execution prefix eliminating the need to store runtime states. Stateless model checkers are easier to design and implement but may suffer from repetitive explorations which effects the scalability of the technique.

3.3.2 Partial Order Reduction

Exploring all possible execution orders of events to determine correctness *wrt* a property may be wasteful. Execution order of two concurrent events does not necessarily change the outcome of the program. As an example consider again the input program in Figure 3.2(a), execution of events b and c in either order after the execution of a

reaches the same state, as show in Figure 3.8. The condition is called *commutativity* of events b and c.

Consequently, it suffices to explore any one of $b.c$ or $c.b$ from the state s_1 for a sound model checking. Thus, instead of exploring the set of execution sequences of a program, a model checker may focus on exploring representative executions for the set of partial orders, such as those shown in Figure 3.2(c) for the input program in Figure 3.2(a).

Chapter 4

ViEqui. Stateless Model Checking based on View-equivalence

4.1 Background

The interleaving model for reasoning about concurrent program outcomes suffers from the state explosion problem [30] due to the combinatorial explosion of thread interleavings (refer to §3.1.3). Early solutions to this problem were static in nature, with partial order reduction (POR) being a popular approach [29, 43, 39, 56]. POR techniques statically compute *persistent sets* [39, 42, 86] or *ample sets* [74], which represent a subset of the interleavings that must be explored. However, conservative over-approximation in static analyses often result in inaccurate sets.

Dynamic analyses, which construct persistent or ample sets *on-the-fly*, have proved to be more effective in addressing the combinatorial explosion. Dynamic analyses can be stateless or stateful (refer to §3.3.1). Seminal works like Verisoft [40, 41] and CHESS [69] popularized stateless model checking. Stateless model checking is a popular model checking technique under sequential consistency [1, 7, 12, 19, 24, 26, 27, 35, 52] and weak memory models [3, 9, 24, 52, 81] (refer to §3.2). Several stateless model checking techniques are coupled with POR [1, 3, 7, 19, 26, 27, 35, 81]

to combat the state space explosion.

Stateless Model Checkers (SMCs) investigate various schemes to further reduce the combinatorial explosion and improve their performance, such as bounding the state space and model checking using representatives, and further, minimization of the set of representatives and accurate computation of representatives to obtain smaller exploration graphs.

Bounding the exploration state space. Dynamic analyses compute persistent (or ample) sets *on-the-fly*, thus, bounding the exploration state by bounding the depth of exploration [84], loop unrolls in the input program [23, 28], or the number of context-switches [68, 69] limits the number of explored interleavings.

Model checking using representatives. Given a property ψ for an input program, ψ may be insensitive to the order of execution of certain concurrent events of the program. As a result any two execution sequences differing only in the order of such instructions would either both satisfy ψ or both satisfy $\neg\psi$. Model checking using representatives [74] addresses the state space explosion problem by exploring a reduced state graph generated of a subset of the interleavings, based on the observation made above.

SMCs that rely on model checking using representatives partition the execution sequences into *equivalence classes* based on an *equivalence relation* such that either all execution sequences of an equivalence class satisfy ψ or all satisfy $\neg\psi$. Such a representation generates a smaller state graph that contains (*ideally*) only one execution sequence from each equivalence class. Model checking using representatives is commonly referred to as *Dynamic* POR (DPOR) [35] and has been extensively studied over the past two decades [1, 3, 7, 13, 14, 19, 35, 81].

Minimizing the set of representatives. To ensure soundness, an SMC must explore at least one execution sequence from each equivalence class. Evidently, minimizing the set of equivalence classes positively impacts the model checking effort. The challenge remains in defining a suitable equivalence relation that partitions the execution sequences such that (i) all sequences in a partition behave similarly to

the property ψ (*i.e.* satisfy ψ or satisfy $\neg\psi$), and (ii) there exists an equivalence class whose sequences satisfy ψ (if feasible) and an equivalence class whose sequences satisfy $\neg\psi$ (if feasible).

Suitably defined equivalence relations may induce coarser partitioning on execution sequences towards a smaller set of equivalence classes. The coarse equivalence relations can be defined based on attributes of execution sequences relevant to ψ, such as data races [1, 3, 19, 35], visibility of events to other threads and events [7, 14, 19, 26], and other features than may render sequences indistinguishable such as read of the same data values [13, 27]. Section §4.1.1 discusses details of coarser notions of equivalence for partitioning execution sequences.

Accurate computation of representatives. The set of equivalence classes is not known to an SMC a priori. Moreover, computing the set of equivalence classes is as hard as exploring the entire state space. Thus, the SMCs compute equivalence classes while exploring execution sequences (*on-the-fly*). Inaccurate *on-the-fly* computation of the equivalence classes leads to exploration of multiple execution sequences from an equivalence class, called *redundant explorations*.

Accurate computation of representatives ensures exploration of *exactly* one execution sequence from each equivalence class, such an accurate exploration is called an *optimal exploration* and an SMC that performs optimal exploration is called an *optimal SMC* [1, 14, 19, 52].

4.1.1 Equivalence partitioning of program executions

As discussed previously, equivalence between program executions is established through an *equivalence relation*. The equivalence relation partitions the execution sequences into sets of equivalent execution sequences called *equivalence classes*. Various equivalence relations can be defined on the attributes of execution sequences that partition the sequences into finer or coarser equivalence classes. This section describes such equivalence relations.

Bisimulation equivalence. Bismimulation represents a fine equivalence relation that aims to identify transition systems which can simulate each other in a step-wise manner. Considering an execution sequence as a transition system with a deterministic order of occurrence on events in the sequence, bisimulation may be used to establish equivalence between execution sequences. However, such a fine equivalence relation may relate as equivalent only those execution sequence that vary on the order of occurrence of invisible events (refer to §2). As a result, execution sequences that have the same order of occurrence on visible events (refer to §2) are considered equivalent.

***Mazurkiewicz* traces and classical equivalence ($\sim^c$).** Mazurkiewicz [65] defines a finite, reflexive and symmetric relation on the events of an input program called *dependence*, D. Further, the relation D induces a relation *independence*, $I = (\mathcal{E} \times \mathcal{E}) \setminus D$. Equivalence classes formed using D are called *Mazurkiewicz traces*. A single trace arises by identifying all execution sequences that differ only in the order of the pairs of events in I that are adjacent in the execution sequence.

Classically, SMCs use Mazurkiewicz traces to define equivalence partitions, where, the dependence relation used to determine the Mazurkiewicz traces is defined over pairs of *racing events* (pairs of events, accessing the same shared object, where at least one event is a `write`) [1, 3, 35, 70, 89]. Program executions that have the same order of occurrence on racing events are considered equivalent, let this equivalence relation be referred to as the *classical* equivalence relation. Given two execution sequences τ_1, τ_2, $\tau_1 \sim^c \tau_2$ represents that the sequences are equivalent under the classical equivalence.

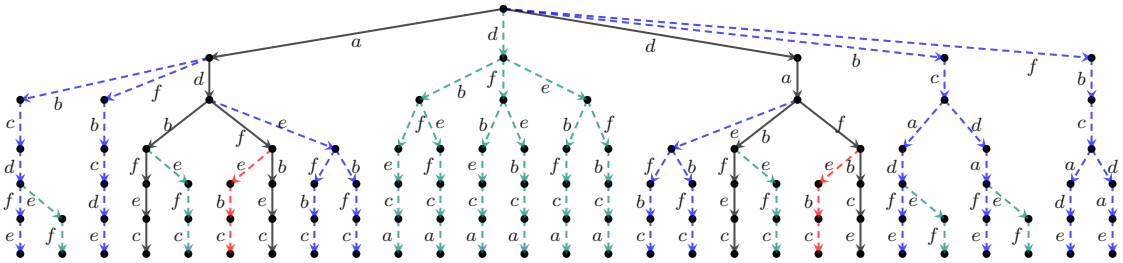
Consider the program in Figure 4.1. The program has three pairs of racing events on each shared object (x and y). Thus under the classical equivalence relation the program has 27 distinct equivalence classes; computed as $3! \times 3! - 9$ combinations that cannot be realized due to a cycle in the required execution resulting from $D \cup \mathrm{po}$, for example, any instance of D such that both $(c, d) \in D$ and $(e, b) \in D$ cannot be realized due to a cycle in $D \cup \mathrm{po}$ (where, po represents the total order on the events of a thread called *program order*).

The equivalence classes for the example program in Figure 4.1 are shown in Figure 4.2

Initially $x = 0, y = 0$

a: $W(x,1)$ $\|$ b: $R(y)$ c: $W(x,1)$ $\|$ d: $R(x)$ e: $W(y,1)$ $\|$ f: $W(y,1)$

Figure 4.1: motivational example



racing-pairs equivalence: blue, green, red dashed sequences and black solid sequences
observed racing-pairs equivalence: blue, red dashed sequences and black solid sequences
reads-from and reads-value-from equivalence: red dashed sequences and black solid sequences
view-equivalence: black solid sequences

Figure 4.2: Equivalence classes of the input program in Figure 4.1 under various equivalence relations.

(through representative executions). Every execution depicted in Figure 4.2 (including the dashed and the solid sequences) represents a distinct valid equivalence class under the classical equivalence relation.

The classical notion of equivalence is sound for detecting data-races and violations of safety properties such as violation of assert conditions. Recent works [7, 12, 13, 14, 19, 24, 26, 27] have explored notions of equivalence that are coarser than the classical equivalence relation. The coarseness reflects in smaller number of equivalence classes that are bulkier than Mazurkiewicz traces (*i.e.* the classes have a larger set of execution sequences).

The coarser notions of equivalence pivot on detection of safety property (assert) violations. Essentially, each coarse equivalence relation recognizes some attribute(s) of the execution sequences that do not contribute to the safety property. The relation is then designed to equate execution sequences that vary only in the order of occurrence of non-contributing attributes and, thus, collect them under the same equivalence class. The coarser equivalence relations are discussed below.

Equivalence on observed races ($\sim^o$). Given an execution sequence (τ), pairs of events in $\mathcal{E}_\tau$ that race in τ where (i) either one of the events in the pair is a `read` event, or (ii) there exists a `read` event that occurs after the racing pair in τ are considered *observed*. Program executions that have the same order of occurrence on *observed* racing event pairs are considered equivalent [14, 19]. Given two execution sequences τ_1, τ_2, $\tau_1 \sim^o \tau_2$ represents that the sequences are equivalent under equivalence on observed races.

The program in Figure 4.1 has 16 equivalence classes under this relation, *i.e.* eleven fewer equivalence classes from the classical relation. The classical equivalence classes that are combined with some other class under this equivalence relation are represented by green dashed sequences in the Figure 4.2. Consider, for instance, the left-most two sequences in Figure 4.2 ($a.b.c.d.f.e$ and $a.b.c.d.e.f$). These sequences vary in the order of execution of the `writes` e and f, which are not observed. As a result, the executions are clubbed under the same equivalence class.

Reads-from equivalence ($\sim^r$). Two program executions are considered reads-from equivalent if (i) both the executions contain the same set of `read` events and (ii) the `reads` obtain values from the same `write` events in both the executions [7, 24, 52]. Given two execution sequences τ_1, τ_2, $\tau_1 \sim^r \tau_2$ represents that the sequences are reads-from equivalent.

There are two `read` events in the program in Figure 4.1. Each `read` event can obtain values from three corresponding `write` events (including the initial `write` event). Thus, in total there are 6 equivalence classes under the reads-from equivalence. The red dashed along with the black solid sequences in Figure 4.2 represent those equivalence classes.

Reads-value-from equivalence ($\sim^v$). Program executions with the same set of `read` events that read the same value and maintain the same *causal-ordering* on the `read` events are considered equivalent [12, 24]. Given two execution sequences τ_1, τ_2, $\tau_1 \sim^v \tau_2$ represents that the sequences are reads-value-from equivalent.

The two `read` events of the program in Figure 4.1 can read two values each (*i.e.* 0

and 1). If the `read` d reads the value 1 from the `write` c, then the `read` b is causally ordered before d. However, there is no such causal ordering when d reads the value 1 from the `write` a. This leads to two separate equivalence classes where d reads 1 and an additional class where d reads 0. Similarly, there exist three equivalence classes under this relation corresponding to the `read` b. As a result, the program in Figure 4.1 has the same set of equivalence classes under this relation as under reads-from equivalence (represented by the red dashed and the black solid sequences).

4.2 The view-equivalence relation

View-equivalence is a coarser notion of equivalence that relies only on the values of `read` events. Program executions with the same set of `read` events that read the same values are considered equivalent. Given sequences τ_1 and τ_2, let $\tau_1 \sim \tau_2$ represent that τ_1 and τ_2 are *view-equivalent*. Formally,

Definition 3. (view-equivalence)

Given sequences τ_1 and τ_2,
$\tau_1 \sim \tau_2$ *iff* $\mathcal{E}^{\mathbb{R}}_{\tau_1} = \mathcal{E}^{\mathbb{R}}_{\tau_2} \wedge \forall e_r \in \mathcal{E}^{\mathbb{R}}_{\tau_1}, val_{[\tau_1]}(e_r) = val_{[\tau_2]}(e_r)$.

Theorem 1 shows that view-equivalence ($\sim$) represents an equivalence relation. Note that, for any given program view-equivalence is at least as coarse as any existing equivalence relation. The claim is formally stated and proven with Theorem 2.

Theorem 1. $\sim$ represents an equivalence relation.

Proof. *Reflexivity* holds trivially from Definition 3.

Given that the operator '=' is symmetric over sets and $\mathcal{V}$, $\sim$ is *Symmetric*.

Let $\tau_1 \sim \tau_2$ and $\tau_2 \sim \tau_3$
$\Rightarrow \mathcal{E}^{\mathbb{R}}_{\tau_1} = \mathcal{E}^{\mathbb{R}}_{\tau_2} = \mathcal{E}^{\mathbb{R}}_{\tau_3}$ and $\forall e_r \in \mathcal{E}^{\mathbb{R}}_{\tau_1}\ val_{[\tau_1]}(e_r) = val_{[\tau_2]}(e_r) = val_{[\tau_3]}(e_r)$.
$\Rightarrow \sim$ is *Transitive*. □

Given the program in Figure 4.1, each `read` event can *view* two values (*i.e.* value 0 and 1). Thus, under view-equivalence there are exactly four view-equivalence classes corresponding to the four combinations of values. The view-equivalence classes are represented by black solid sequences in Figure 4.2. Each dashed sequence can be partitioned with the view-equivalence class of one of the black solid sequences.

In moving from finer to coarser partitioning, an equivalence relation associates lesser ordering on the event of an execution sequence. For instance the classical notion orders all racing event pairs (including pairs of write events that may not be read in the program), while the reads-value-from equivalence only orders causal reads. The proposed view-equivalence relation completely dissociates from ordering on events of an execution sequence. As a result, view-equivalence is at least as coarse as any existing equivalence relation. Consequently, view-equivalence forms the smallest number of partitions for any input program.

The view-equivalence class of a program execution τ is represented by $[\![\tau]\!]$. Each program execution τ' such that $\tau' \sim \tau$, $\tau' \in [\![\tau]\!]$. Consider again the program in Figure 4.1 and its executions $\tau_1 = a.b.c.d.e.f$ and $\tau_2 = b.c.a.d.e.f$, $\tau_1 \sim \tau_2$, thus $[\![\tau_1]\!] = [\![\tau_2]\!]$ and $\tau_1, \tau_2 \in [\![\tau_1]\!]$ (or $[\![\tau_2]\!]$).

Further, given sequences τ_1 and τ_2, τ_1 is a prefix of τ_2 under view-equivalence, represented as $\tau_1 \lhd \tau_2$ if $\exists \tau_1'$ s.t. $\tau_1.\tau_1' \sim \tau_2$.

Applicability of view-equivalence. The applicability of view-equivalence is not reduced in context of safety property (assert) violations in comparison to the other coarse equivalence relations [7, 12, 13, 14, 19, 24, 26, 27]. In other words, given execution sequences τ_1 and τ_2 and a safety property (assert condition) ψ, $\tau_1 \sim \tau_2 \Leftrightarrow$ both τ_1 and τ_2 satisfy ψ or both τ_1 and τ_2 satisfy $\neg\psi$. In essence, an assert condition ψ is a boolean condition on the values of a set of `read` events. Thus, if τ_1 satisfies ψ and τ_2 satisfies $\neg\psi$ then there is a `read` $e \in \mathcal{E}_{\tau_1} \cap \mathcal{E}_{\tau_2}$ s.t. $val_{[\tau_1]}(e) \neq val_{[\tau_2]}(e)$. However, from Definition 3, then $\tau_1 \nsim \tau_2$.

Intuitively, every observable location and every branching condition in the input program such as the SSA phi nodes, output statements and assert conditions rely

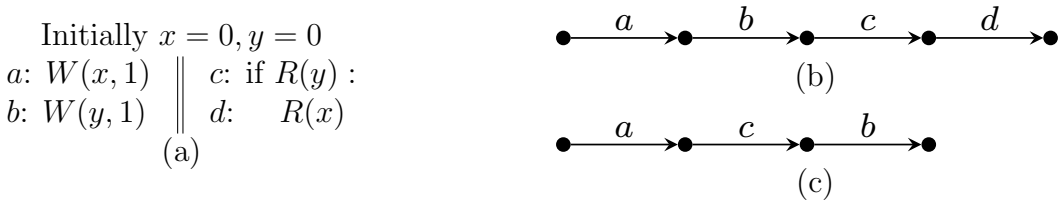


Figure 4.3: Message passing with condition

on the values of a set of corresponding memory locations. Since, computing the condition requires accessing the corresponding memory locations, thus, alternatively we can consider that the conditions rely on the values of a set of `read` events. Hence, by considering all combinations of `read` values in separate equivalence classes, view-equivalence ensures coverage of every branch of program's control flow graph, every program output and every feasible outcome of an assert condition in the program.

Trade-off of causality. The definition of view-equivalence does not incorporate a causal ordering on execution events, which sets it apart from other equivalence notions.

Including causal ordering in the definition of equivalence is beneficial as it provides the necessary information for constructing a coherent ordering, that leads to an execution of an equivalence class. Incoherent orderings may affect the completeness guarantee of SMCs. SMCs that detach from any causal ordering, such as ViEqui, must compute coherence operationally to ensure completeness, which is non-trivial.

Consider the example in Figure 4.3(a) and its execution in Figure 4.3(b). An SMC based on the classical notion of equivalence considers the order of execution on racing event pairs *i.e.*, $a \to d$ and $b \to c$, and flips either of these orders to explore a different equivalence class, such as the one shown in Figure 4.3(c) where $c \to b$.

On the other hand, ViEqui computes that the value 1 is read by `reads` c and d in the execution shown in Figure 4.3(b) and that value 0 may also be read by the two `reads` in some other execution. ViEqui then computes a state where the value can be read from, for instance, 0 can be read by d from the initial state and from no state after

Table 4.1: Performance comparison on program in Figure 4.1

	ODPOR		obs-ODPOR		RVF-SMC		ViEqui	
configuration N	#Seq	Time	#Seq	Time	#Seq	Time	#Seq	Time
Figure 4.1(10)	-	To	-	To	3703196	705.69	40	0.06
Figure 4.1(20)	-	To	-	To	-	To	80	0.28
Figure 4.1(100)	-	To	-	To	-	To	400	43.15
Figure 4.1(200)	-	To	-	To	-	To	800	737.00

the execution of a. Further, it computes a sequence that can enable and execute d from the initial state ensuring the read of the value 0. Since d can only be enabled by the sequence $a.b.c.d$. However, the enabling sequence leads to the read of value 1 by d and hence, ViEqui computes that value 0 cannot be read by d coherently.

However, SMCs such as ViEqui that can ensure completeness can reduce the number of program executions to explore and achieve higher performance. To demonstrate the savings with view-equivalence, we conducted experiments on various configurations of the program in Figure 4.1, where events from threads are repeated N times. Higher values of N leads to a higher degree of causal ordering, while the set of values read by the events remains the same. The results of the experiments with the program in Figure 4.1 are shown in Table 4.1 on the number of sequences explored (#Seq) and the time of analysis (Time). A timeout of analysis is set at 1800s (To). We observed exponential savings with view-equivalence compared to other techniques on finer equivalence relations (refer to §4.5 for details on the other techniques).

Theorem 2. $\forall \tau_1, \tau_2 \in E,\ \tau_1 \sim^c \tau_2 \vee \tau_1 \sim^o \tau_2 \vee \tau_1 \sim^r \tau_2 \vee \tau_1 \sim^v \tau_2 \Rightarrow \tau_1 \sim \tau_2$.
(view-equivalence is at least as coarse as any existing equivalence relation.)

Proof. Consider sequences τ_1, τ_2,

$\tau_1 \sim^v \tau_2 \Rightarrow \tau_1 \sim \tau_2$. (by definitions of $\sim^v$ and $\sim$)

Further, $\tau_1 \sim^r \tau_2 \Rightarrow \mathrm{po}_{\tau_1} = \mathrm{po}_{\tau_2}$. (by definition of $\sim^r$)
Let $\exists\ e_w \in \mathcal{E}^{\mathbb{W}}_{\tau_1} \cap \mathcal{E}^{\mathbb{W}}_{\tau_2},\ e_r \in \mathcal{E}^{\mathbb{R}}_{\tau_1} \cap \mathcal{E}^{\mathbb{R}}_{\tau_2}$ s.t. $e_w \rightarrow^{\mathbf{rf}}_{\tau_1} e_r$ and $e_w \rightarrow^{\mathbf{rf}}_{\tau_2} e_r$ but $val_{[\tau_1]}(e_r) \neq$

Event Relations

Consider the relations,

program-order (po_τ), *modification-order* (mo_τ), *observed-modification-order* (o-mo_τ), *reads-from* (rf_τ), *from-reads* (fr_τ), and *causal-order-on-reads* (co-r_τ), such that

$\mathrm{po}_\tau \cup \mathrm{mo}_\tau \cup \text{o-mo}_\tau \cup \mathrm{rf}_\tau \cup \mathrm{fr}_\tau \cup \text{co-r}_\tau \subseteq \mathcal{E}_\tau \times \mathcal{E}_\tau$ for a sequence τ.

Given $e_1 <_\tau e_2$, the relations are defined as:

po_τ	Consider the relation *spawn-order* (sno_τ) such that $e' \rightarrow_\tau^{\text{sno}} e$ if e' belongs to the thread that spawns the thread of e, and e' occurs before the spawn event in its thread. Consider the relation *join-order* (jno_τ) such that $e' \rightarrow_\tau^{\text{jno}} e$ if e belongs to the thread that joins the thread of e', and e occurs after the join event in its thread. Further, ${sno_\tau}^+ \subseteq sno_\tau$ and ${jno_\tau}^+ \subseteq jno_\tau$; then, $e_1 \rightarrow_\tau^{\mathbf{po}} e_2$ if e_1 occurs before e_2 in a thread $\vee$ $e_1 \rightarrow_\tau^{\text{sno}} e_2 \vee e_1 \rightarrow_\tau^{\text{jno}} e_2$.
mo_τ	a total order on the `write` events of an object.
o-mo_τ	$e_1 \rightarrow_\tau^{\mathbf{o\text{-}mo}} e_2$ if $e_1 \rightarrow_\tau^{\mathbf{mo}} e_2$ and $\exists e_r \in \mathcal{E}_\tau^{\mathbb{R}}$ s.t. $obj(e_r) = obj(e_1) \wedge e_2 <_\tau e_r$, *i.e.*, a total order on the observed `write` events of an object.
rf_τ	$e_1 \rightarrow_\tau^{\mathbf{rf}} e_2$ if $e_1 = \mathsf{last}W_{[\tau]}(e_2)$.
fr_τ	$e_1 \rightarrow_\tau^{\mathbf{fr}} e_2$ if $(e_1, e_2) \in \mathrm{rf}_\tau^{-1}; \mathrm{mo}_\tau$.
co-r_τ	$e_1 \rightarrow_\tau^{\mathbf{co\text{-}r}} e_2$ if $e_1, e_2 \in \mathcal{E}_\tau^{\mathbb{R}} \wedge e_1 \rightarrow_\tau^{\mathbf{co}} e_2$.

Formal Definitions of Equivalence Relations.

Classical equivalence ($\sim^c$). Given sequences τ_1, τ_2,

$\tau_1 \sim^c \tau_2$ if $\mathrm{po}_{\tau_1} \cup \mathrm{mo}_{\tau_1} \cup \mathrm{rf}_{\tau_1} \cup \mathrm{fr}_{\tau_1} = \mathrm{po}_{\tau_2} \cup \mathrm{mo}_{\tau_2} \cup \mathrm{rf}_{\tau_2} \cup \mathrm{fr}_{\tau_2}$.

Equivalence on observed races ($\sim^o$). Given sequences τ_1, τ_2,

$\tau_1 \sim^o \tau_2$ if $\mathrm{po}_{\tau_1} \cup \text{o-mo}_{\tau_1} \cup \mathrm{rf}_{\tau_1} \cup \mathrm{fr}_{\tau_1} = \mathrm{po}_{\tau_2} \cup \text{o-mo}_{\tau_2} \cup \mathrm{rf}_{\tau_2} \cup \mathrm{fr}_{\tau_2}$.

Reads-from equivalence ($\sim^r$). Given sequences τ_1, τ_2,

$\tau_1 \sim^r \tau_2$ if $\mathrm{po}_{\tau_1} \cup \mathrm{rf}_{\tau_1} = \mathrm{po}_{\tau_2} \cup \mathrm{rf}_{\tau_2}$.

Reads-value-from equivalence ($\sim^v$). Given sequences τ_1, τ_2,

$\tau_1 \sim^v \tau_2$ if $\mathrm{po}_{\tau_1} \cup \text{co-r}_{\tau_1} = \mathrm{po}_{\tau_2} \cup \text{co-r}_{\tau_2} \wedge V_{\tau_1} = V_{\tau_2}$.

View-equivalence ($\sim$). Given sequences τ_1, τ_2,

$\tau_1 \sim \tau_2$ if $\mathrm{po}_{\tau_1} = \mathrm{po}_{\tau_2} \wedge V_{\tau_1} = V_{\tau_2}$.

Where V_τ represent a map $\mathcal{E}_\tau^{\mathbb{R}} \mapsto \mathcal{V}$ such that $\forall e_r \in \mathcal{E}_\tau^{\mathbb{R}}$, $e_r \mapsto val_{[\tau]}(e_r)$ belongs to V_τ.

$val_{[\tau_2]}(e_r)$.
$\Rightarrow \exists e'_r \in \mathcal{E}^{\mathbb{R}}_{\tau_1} \cap \mathcal{E}^{\mathbb{R}}_{\tau_2}$ s.t. $e_r \rightarrow^{\mathbf{co}}_{\tau_1} e_w \wedge e_r \rightarrow^{\mathbf{co}}_{\tau_2} e_w \wedge \mathsf{last}W_{[\tau_1]}(e_r) \neq \mathsf{last}W_{[\tau_2]}(e_r)$
$\Rightarrow \mathrm{rf}_{\tau_1} \neq \mathrm{rf}_{\tau_2}$ that contradicts $\tau_1 \sim^r \tau_2$.
$\Rightarrow V_{\tau_1} = V_{\tau_2}$. (by contradiction)
Thus, $\tau_1 \sim^r \tau_2 \Rightarrow \tau_1 \sim \tau_2$.

Lastly, $\tau_1 \sim^c \tau_2 \vee \tau_1 \sim^o \tau_2 \Rightarrow \tau_1 \sim^r \tau_2$. (by definitions of $\sim^c$, $\sim^o$ and $\sim^r$)
$\Rightarrow \tau_1 \sim \tau_2$ (shown previously)
□

4.3 Stateless model checking based on view equivalence under sequential consistency

To ensure soundness, an SMC must explore at least one execution sequence for each equivalence class under the corresponding equivalence relation. Figure 4.2 shows that the number of equivalence classes under view-equivalence may be significantly smaller in comparison to other notions of equivalence. Evidently, model checking based on view-equivalence may exhibit significant savings in the model checking effort. The savings directly translates to a faster verification.

This work proposes ViEqui, an SMC based on view-equivalence under sequential consistency. The ViEqui technique is

1. *Sound.* ViEqui explores a representative execution sequence corresponding to each view-equivalence class of the input program.

2. *Complete.* Each execution sequence explored by ViEqui represents a view-equivalence class of the input program.

3. *Optimal.* ViEqui explores exactly one representative execution sequence corresponding to each view-equivalence class of the input program.

Operational Assumptions. The set of actions supported by ViEqui is the set $\{\texttt{write}, \texttt{read}\}$. As a consequence, $\mathcal{E}^{\mathbb{W}} \cup \mathcal{E}^{\mathbb{R}} = \mathcal{E}$ and $\mathcal{E}^{\mathbb{W}} \cap \mathcal{E}^{\mathbb{R}} = \emptyset$. Since, ViEqui operates under sequential consistency, a `read` $e_r \in \mathcal{E}_\tau^{\mathbb{R}}$ reads the value of the latest `write` of $obj(e_r)$ in the prefix of τ up to e_r, that is, e_r reads from $\mathsf{last}W_{[\tau]}(e_r)$. Further, since sequential consistency does not support out-of-order execution of events, the execution of an event from a thread $\mathbb{T}_i$ enables the next event in *program-order* from $\mathbb{T}_i$, thus, the enabled set contains the *next* event from each thread. This implies that when a `write` event e_w is enabled in a sequence τ, $val_{[\tau]}(e_w)$ is already computed for τ and available as a constant.

ViEqui explores (executes and analyzes) a representative execution sequence corresponding to each view-equivalence class of the input program. However, the set of view-equivalence classes is not known a priori. Moreover, computing the set of equivalence classes is as hard as exploring the entire state space. Thus, ViEqui computes view-equivalence classes while exploring execution sequences (*on-the-fly*).

ViEqui uses a scheduler to order the execution of program events in a way that facilitates optimal exploration of view-equivalence classes. This enables us to obtain a representative program execution that belongs to the desired view-equivalence class. During the exploration of an execution sequence τ, ViEqui gathers scheduling information such as the view-equivalence class of τ and other values for `reads` that are not read in τ. ViEqui then computes *scheduling directives* ($\Delta(s_{[\tau]})$) for each state of exploration $s_{[\tau]}$, reached after exploring an execution prefix τ. The directives are sets of pairs of (*next*: an event sequence, ϕ: a boolean formula) where,

- *next* is a sequence that is to be explored from $s_{[\tau]}$, and
- ϕ is a summary of the value combinations for the `read` events appearing in *next* that are read in some other execution sequence from $s_{[\tau]}$.

Given a scheduling directive $\delta \in \Delta(s_{[\tau]})$, its *next* sequence is represented as δ_n and its ϕ boolean formula is represented as δ_ϕ.

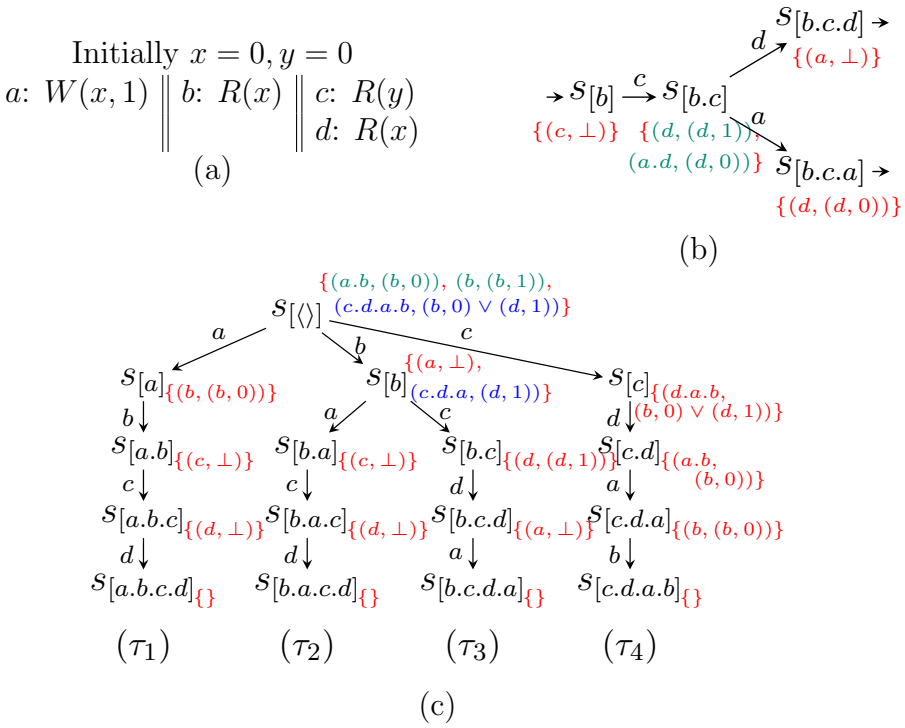


Figure 4.4: 1W2R. (a) input program, (b)-(c) explorations by ViEqui

Definition 4. (Properties of $\Delta(s_{[\tau]})$)

$\Delta(s_{[\tau]})$ at a state $s_{[\tau]}$ must satisfy the following properties.

1. For each view-equivalence class π that has a representative execution with τ as a prefix, there exists $\delta \in \Delta(s_{[\tau]})$ that extends τ to a representative execution of π. (*soundness*)

2. Each $\delta \in \Delta(s_{[\tau]})$ extends τ to a valid execution of the input program. (*completeness*)

3. No two discrete $\delta, \delta' \in \Delta(s_{[\tau]})$ can extend τ to equivalent executions. (*optimality*)

Consider the input program in Figure 4.4(a). The program has four view-equivalence

classes corresponding to the values 0 and 1 for the `read` events b and d. Figure 4.4(c) shows a complete exploration of the program where each of the four view-equivalence classes is explored optimally.

ViEqui technique follows a dual approach of eager and lazy analyses to compute the scheduling directives during exploration, called *forward-analysis* and *backward-analysis* respectively.

Forward-analysis. On reaching a state of exploration $s_{[\tau]}$ after exploring an execution prefix τ, ViEqui analyzes the set of events that are *enabled* (available for execution) and looks for a `read` event that can `read` multiple values from $s_{[\tau]}$. Consider the initial state $s_{[\langle\rangle]}$ in Figure 4.4(c) where the events a, b, c are enabled. ViEqui computes that the `read` b can read two values from $s_{[\langle\rangle]}$ (*i.e.* 1 and 0), and accordingly, computes $\Delta(s_{[\langle\rangle]}) = \{(a.b, (b, 0)), (b, (b, 1))\}$. In particular, on exploring the sequence $a.b$ of the directive $(a.b, (b, 0))$ from $s_{[\langle\rangle]}$, b reads the value 1, while its corresponding formula $(b, 0)$ summarizes that b reads the value 0 in the other directive, *i.e.*, $(b, (b, 1))$; similarly, on exploring sequence b of the directive $(b, (b, 1))$ from $s_{[\langle\rangle]}$, b reads the value 0, while is corresponding formula $(b, 1)$ summarizes that b reads the value 1 in the other directive, *i.e.*, $(a.b, (b, 0))$.

This computation of $\Delta(s_{[\tau]})$ is called *forward-analysis* since it is eagerly performed without executing the enabled events. The directives computed using forward-analysis are shown in green in Figure 4.4(c).

Forward-analysis is formally presented in §4.3.2.

Backward-analysis. Forward-analysis examines only the enabled events at a state $s_{[\tau]}$, hence, computing $\Delta(s_{[\tau]})$ precisely for all events is not always possible with forward-analysis. For example, the `read` event d is not available for analysis at $s_{[\langle\rangle]}$ in Figure 4.4(c). To address this problem, ViEqui computes scheduling directives at the end of each execution sequence for states in the execution prefix. This analysis is called *backward-analysis*, since it requires examining the prefix of an execution.

Consider the program execution $\tau_1 = a.b.c.d$ in Figure 4.4(c), where the `read` d is enabled after the execution of c at state $s_{[a.b.c]}$. The order of execution of events

in τ_1 results in the read of value 1 by d, however, a different order of execution of the events of τ_1 could result in the read of value 0 (initial value) by d. Therefore, with backward-analysis ViEqui would compute a directive that would lead to the exploration of value 0 for d from a state in the prefix of $s_{[\tau_1]}$.

However, there may be various such directives and they may not always result in the exploration of the same view-equivalence class. This is because a `read` e_r may read a certain value v in multiple view-equivalence classes; for example, the `read` d in the program in Figure 4.4(a) can read value 0 in two view-equivalence classes, *i.e.* those corresponding to $b{=}0, d{=}0$ and $b{=}1, d{=}0$.[1] Furthermore, it is possible that some of the view-equivalence classes where the `read` e_r reads v can be explored by existing directives; for example, in the exploration shown in Figure 4.4(c), the view-equivalence class corresponding to $b{=}0, d{=}0$ can be explored using the directive $(b, (b, 1))$ from $s_{[\langle\rangle]}$.

For optimality and termination it is crucial that ViEqui controls the view-equivalence classes that will be explored from a directive by carefully choosing its *next* sequence from the set of sequences where e_r reads v; for example, during the backward-analysis after τ_1 in Figure 4.4(c) for d to read 0, ViEqui must compute a directive that results in precisely the exploration of the view-equivalence class corresponding to $b{=}1, d{=}0$.

Thus, ViEqui computes a directive $(c.d.a.b, (b, 0) \vee (d, 1)) \in \Delta(s_{[\langle\rangle]})$ where the *next* sequence $c.d.a.b$ reads value 0 for d in combination of value 1 for b. The corresponding $\phi = (b, 0) \vee (d, 1)$ summarizes the values for d and b read in other sequences from $s_{[\langle\rangle]}$; specifically, the value 0 for b is covered in program executions τ_2 and τ_3 and the value 1 for d is covered in τ_1 and τ_2; note that, ϕ subsumes $(b, 0) \wedge (d, 1)$, which is explored in τ_2. The computed directive is added to a state in the prefix of τ_1 from where the value 0 can indeed be read by d, *i.e.* $\Delta(s_{[\langle\rangle]})$. Finally, during the exploration of τ_4, resulting from $(c.d.a.b, (b, 0) \vee (d, 1)) \in \Delta(s_{[\langle\rangle]})$, the formula $(b, 0) \vee (d, 1)$ is used by ViEqui to determine that no other scheduling directives need to be computed for b to read 0 or d to read 1 from τ_4. The directives computed using backward-analysis

[1] Note that, (i) we may refer to view-equivalence classes by the combinations of `reads` and their values; (ii) such a notation may not contain `reads` that have the same value in all view-equivalence classes, such as the `read` c in Figure 4.4(a).

are shown in blue in Figure 4.4(c).

Backward-analysis is formally presented in §4.3.2.

Note that, $\Delta(s_{[\tau]})$ at a state $s_{[\tau]}$ may not be unique since various event sequences may represent the same view-equivalence class. Figure 4.4(b) shows a fragment of an alternate exploration feasible with ViEqui where the technique selects c from the set of enabled events instead of a at $s_{[b]}$.

4.3.1 Computation of Scheduling Directives

To satisfy the three properties in Definition 4, scheduling directives are generated through a two-step process for each $\Delta(s_{[\tau]})$ using both forward- and backward-analysis. In the first step, individual scheduling directives are computed from available scheduling information and exploration summaries in a sequence. However, since directives of sequence are computed independently, they may not consider scheduling information from other sequences. Therefore, in the second step, the individual scheduling directives computed in step one are coherently combined with $\Delta(s_{[\tau]})$ to avoid redundancies and ensure soundness and completeness.

Computing the state to update for a directive. To perform the above mentioned steps, it is necessary to compute an exploration state $s_{[\tau]}$ whose $\Delta(s_{[\tau]})$ can be updated with the newly computed scheduling directive δ. Thus, prior to discussing these steps, it is important to consider the process of computing such a state. For example, during the backward-analysis after τ_1 in Figure 4.4(c), for d to read from $\mathbb{I}_x$, a newly computed directive is added to the state $s_{[\langle\rangle]}$ because at any other state in the prefix of τ_1 the `write` a overwrites $\mathbb{I}_x$.

A directive (δ) computed during forward-analysis at a state $s_{[\tau]}$ only updates $\Delta(s_{[\tau]})$, while, backward-analysis after a program execution τ, updates $\Delta(s_{[\tau']})$ for some τ' in the prefix of τ.

Consider the backward-analysis after the exploration of a program execution $\tau =$

$\tau_1.\tau_2.\tau_3$, where τ_2 represents the *next* sequence explored from $s_{[\tau_1]}$. The states $s_{[\tau_k]}$ between, but not including, $s_{[\tau_1]}$ and $s_{[\tau_2]}$ are marked *hidden* for the execution sequence τ, which signifies that backward-analysis does not update $\Delta(s_{[\tau_k]})$. Consider backward-analysis after τ, for $e_r \in \mathcal{E}_\tau^{\mathbb{R}}$ to read from $e_w \in \mathcal{E}_\tau^{\mathbb{W}}$, ViEqui updates $\Delta(s_{[\tau']})$ of a state $s_{[\tau']}$ if (i) $s_{[\tau']}$ is not *hidden*, and (ii) $\exists$ a sequence δ_n on the events of τ s.t. $e_w \rightarrow^{\mathbf{rf}}_{\tau'.\delta_n} e_r$. Intuitively, condition (ii) states that e_r can feasibly read from e_w after τ'. Note that, in the presence of multiple such states, ViEqui selects the state of the longest prefix τ' for operational efficiency. We use $\mathsf{pre}_{[\tau]}(\mathsf{e_w}, \mathsf{e_r})$ to represent such a state. For example, consider the backward-analysis after τ_2 in Figure 4.4(c), for determining the state where d reads from $\mathbb{I}_x$. While both $s_{[\langle\rangle]}$ and $s_{[b]}$ satisfy the two conditions outlined earlier, ViEqui selects $\mathsf{pre}_{[\tau_2]}(\mathbb{I}_\mathsf{x}, \mathsf{d}) = s_{[b]}$.

Step 1. Accurate computation of individual scheduling directives

In order to compute an accurate scheduling directive ($\delta = (\delta_n, \delta_\phi)$), we need to ensure accurate computation of both components (δ_n and δ_ϕ). This entails (i) computing a coherent and accurate δ_n that extends to program executions and precisely explores the desired view-equivalence classes, and (ii) representing the related exploration summary δ_ϕ succinctly, which can guide ViEqui to prevent redundant explorations without compromising soundness. The following text discuss these two requirements in more detail.

Computation of coherent and accurate *next* event sequence. The coherence of a sequence *next*, for a `read` e_r to read the value v after τ, means that it can result in a valid execution after τ. The accuracy of *next* is determined by its ability to extend to representative executions of the desired view-equivalence classes while excluding those of other view-equivalence classes. This notion of accuracy is intuitively introduced above, under 'Backward-analysis'. Recall the backward-analysis after τ_1 in Figure 4.4(c), for d to read 0 from $\mathbb{I}_x$. It was emphasized that the optimality of ViEqui relies on computing a *next* sequence that can exclusively extend to explore the view-equivalence class corresponding to b=1, d=0, rather than also explore the one corresponding to b=0, d=0. Therefore, the computed *next* sequence must include

$e_1 \rightarrow_\tau^{\text{po}} e_2 \triangleq$ e_1 occurs before e_2 in the same thread

$e_w \rightarrow_\tau^{\text{rf}} e_r \triangleq e_w, e_r \in \mathcal{E}_\tau$ and $e_w = \mathsf{last}W_{[\tau]}(e_r)$

$e_1 \rightarrow_\tau^{\text{co}} e_2 \triangleq (e_1, e_2) \in$ transitive closure of $(\text{po}_\tau \cup \text{rf}_\tau)$

$\tau_1 \oplus \tau_2 \triangleq$ if $\exists\ \tau$ (an event sequence) s.t. $\mathcal{E}_\tau = \mathcal{E}_{\tau_1} \cup \mathcal{E}_{\tau_2}$ and $\text{co}_\tau = \text{co}_{\tau_1} \cup \text{co}_{\tau_2}$, then $\tau_1 \oplus \tau_2 = \tau$, otherwise $\tau_1 \oplus \tau_2 = \langle\rangle$.

(a)

$\mathsf{nseq\text{-}fwd}_{[\tau]}(e_w, e_r) \triangleq \mathtt{w}.e_r$ (where, $\mathtt{w} = \langle\rangle$, if $e_w = \mathsf{last}W_{[\tau]}(e_r)$; $\mathtt{w} = e_w$, otherwise)

(b)

$\mathsf{eseq}_{[\tau]}(e) \triangleq$ smallest subsequence of τ s.t. $e \in \mathsf{eseq}_{[\tau]}(e) \wedge \forall e_{po} \rightarrow_\tau^{\text{po}} e\ (e_{po} \in \mathsf{eseq}_{[\tau]}(e) \wedge \forall e' \rightarrow_\tau^{\text{co}} e_{po},\ e' \in \mathsf{eseq}_{[\tau]}(e))$.
(note, for an initial event $\mathbb{I}_o$, $\mathsf{eseq}_{[\tau]}(\mathbb{I}_o) = \langle\rangle$)

$\mathsf{nseq\text{-}bkwd}_{[\tau]}(e_r, v) \triangleq \mathsf{eseq}_{[\tau-\tau']}(e_w) \oplus \mathsf{eseq}_{[\tau-\tau']}(e_r) \oplus \mathtt{w}.e_r \oplus \bar{\delta}_n(s_{[\tau']})$

(where, $\tau' = \mathsf{pre}_{[\tau]}(\mathsf{e_r}, \mathsf{v})$, and $\mathtt{w} = \langle\rangle$, if $e_w = \mathsf{last}W_{[\tau']}(e_r)$; $\mathtt{w} = e_w$, otherwise)

$\bar{\delta}_n(s_{[\tau]})$ represents the *next* sequence of $\delta \in \Delta(s_{[\tau]})$ that is currently being explored, for example, in Figure 4.4(c), $\bar{\delta}_n(s_{[b]}) = a$ and $c.d.a$ in τ_2 and τ_3 respectively.

(c)

Figure 4.5: Computation of (a) well-formed sequence, (b) *next* sequence using forward-analysis, (b) *next* sequence using backward-analysis

a subsequence $a.b$ that mandates the read of value 1 for b. As a result, for this backward-analysis, the accurate *next* sequence is $c.d.a.b$.

The computation of coherent and accurate δ_n using forward-analysis is formally presented as $\mathsf{nseq\text{-}fwd}_{[\tau]}(e_w, e_r)$ in Figure 4.5(b), where a `read` e_r reads from the last performed `write` or from an enabled `write` (e_w) by following it in execution order. On the other hand, the computation of δ_n using backward-analysis requires a more elaborate construct that is formally discussed below.

Consider the event relations *program-order* (po_τ), *reads-from* (rf_τ) and *coherence-order* (co_τ) on the events of a sequence τ, defined in Figure 4.5(a). Let a sequence τ be

well-formed if (i) co_τ is irreflexive, and (ii) rf_τ is feasible under sequential consistency. Let $\oplus$ represent the *well-formed join* to two well-formed sequences, formally defined in Figure 4.5(a).

A coherent and accurate δ_n, where e_r reads from e_w, is computed using backward-analysis, at a state $s_{[\tau']} = \mathsf{pre}_{[\tau]}(\mathsf{e_w}, \mathsf{e_r})$, by the well-formed join of four sequences: (i) a sequence that enables e_w after τ' (computed using *enabling sequence*, $\mathsf{eseq}_{[\tau-\tau']}(e_w)$, defined in Figure 4.5(c)); (ii) a sequence that enables e_r after τ' (computed similarly using $\mathsf{eseq}_{[\tau-\tau']}(e_r)$); (iii) a sequence where e_r reads from e_w; and, (iv) given that $\delta \in \Delta(s_{[\tau']})$ is currently being explored from $s_{[\tau']}$, its *next* sequence δ_n. The computation is formally presented as $\mathsf{nseq\text{-}bkwd}_{[\tau]}(e_w, e_r)$ in Figure 4.5(c).

To illustrate the computation of δ_n using backward-analysis, consider the analysis after τ_1 in Figure 4.4(c) for `read` d to read from $\mathbb{I}_x$; $\mathsf{pre}_{[\tau_1]}(\mathbb{I}_\mathsf{x}, \mathsf{d}) = s_{[\langle\rangle]}$, accordingly, $\mathsf{eseq}_{[\tau_1-\langle\rangle]}(\mathbb{I}_x) = \langle\rangle$, and $\mathsf{eseq}_{[\tau_1-\langle\rangle]}(d) = c.d$, thus, $\mathsf{nseq\text{-}bkwd}_{[\tau_1]}(\mathbb{I}_x, d) = \langle\rangle \oplus c.d \oplus \langle\rangle.d \oplus a.b = c.d.a.b$.

Note that, in including the the *next* sequence of δ that is being explored from $s_{[\tau']}$ as sequence (iv), ViEqui avoids redundant explorations with other sequences explored from $s_{[\tau']}$; for instance, in the above example by including $a.b$ in the computed sequence $c.d.a.b$, ViEqui mandates the read of value 1 for b, which subsequently avoids the exploration of the class corresponding to $b{=}0, d{=}0$. This class is explored from $s_{[\langle\rangle]}$ in the execution τ_3, from another directive at $s_{[\langle\rangle]}$, *i.e.* $(b, (b, 1))$.

Succinct representation of exploration summary. ViEqui summarizes the combinations of `read` events and corresponding values read in other explorations for a scheduling directive (δ) as a boolean formula (δ_ϕ). The formula can be formally presented by the following grammar.

$$L := \top \mid \bot \mid (e_r, v)$$

$$\phi := L \mid L \vee L \mid L \wedge L$$

The boolean constants *true* and *false* are represented by $\top$ and $\bot$, respectively. The

$\mathsf{simplify}(\delta_\phi, (e_r, v))$	$\triangleq$	the resulting formula after replacing (e_r, v) with $\top$ and (e_r, v') with $\bot$ in δ_ϕ, where $v' \neq v$
$(e_r, v)\Vdash\delta_\phi$	$\triangleq$	$\mathsf{simplify}(\delta_\phi, (e_r, v)) = \top$
$(e_r, v) \nVdash \delta_\phi$	$\triangleq$	$\mathsf{simplify}(\delta_\phi, (e_r, v)) \neq \top$
$\mathsf{summary}_{[\tau]}(\delta_n)$	$\triangleq$	$\bigvee_{l\in L} l$ where $L = \{(e_r, v) \mid \exists\tau'$ prefix of τ, $\exists e_r \in \mathcal{E}^{\mathbb{R}}_{\delta_n}$ s.t. $(e_r, v)\Vdash\bar{\delta}_\phi(s_{[\tau']})\}$

$\bar{\delta}_\phi(s_{[\tau]})$ represents the δ_ϕ boolean formula of $\delta \in \Delta(s_{[\tau]})$ that is currently being explored, for example, in Figure 4.4(c), $\bar{\delta}_\phi(s_{[b]}) = \bot$ and $(d, 1)$ in τ_2 and τ_3 respectively.

Figure 4.6: Notations and operations on boolean formula δ_ϕ

literal (e_r, v) is a shorthand notation for a predicate that evaluates to *true* when value of e_r is v and evaluates to *false* when value of e_r is not v. Figure 4.6 defines notations and operations on δ_ϕ. The operation $\mathsf{simplify}(\delta_\phi, (e_r, v))$ simplifies the formula by assuming e_r reads v and thus, replaces literal (e_r, v) with $\top$ and literal (e_r, v') with $\bot$, where $v \neq v'$. The notation $(e_r, v)\Vdash\delta_\phi$ represents that the literal (e_r, v) is a model of δ_ϕ, and the notation $(e_r, v)\nVdash\delta_\phi$ represents its negation.

To compute an accurate δ_ϕ using forward-analysis, ViEqui relies on the fact that a scheduling directive is created for each value of enabled `writes` (and the current value at a state) that can be read by a `read` e_r. Thus, δ_ϕ corresponding to a directive for a `read` e_r to read v, is computed such that for each value $v' \neq v$ that can be read by e_r using forward-analysis, $(e_r, v')\Vdash\delta_\phi$.

As an example recall the forward-analysis at $s_{[\langle\rangle]}$ in Figure 4.4(c). For each of the two computed scheduling directives, $\delta \in \{(a.b, (b, 0)), (b, (b, 1))\}$, $(b, v)\Vdash\delta_\phi$, where v represents the value of b read from the other directive.

Similarly, in case of backward-analysis after the execution sequence τ, for a `read` e_r to read value v from a `write` e_w, δ_ϕ is computed such that, for each value $v' \neq v$, of other `writes` of the same object in τ, $(e_r, v')\Vdash\delta_\phi$. Since a directive is created by backward-analysis for e_r to read each such value v', ensuring $(e_r, v')\Vdash\delta_\phi$ summarizes the same. Additionally, literals of the form (e'_r, v'') on `read` events e'_r of δ_n from τ

also satisfy δ_ϕ. In including such literals in δ_ϕ, ViEqui carries forward the exploration summary of τ to the newly computed directive. The computation of such summary of τ is formally presented as $\mathsf{summary}_{[\tau]}(\delta_n)$ in Figure 4.6.

As an example recall the backward-analysis after τ_1 in Figure 4.4(c) for d to read 0 from $\mathbb{I}_x$, where the computed directive is $(c.d.a.b, (b, 0) \vee (d, 1))$. The literal $(d, 1) \Vdash (b, 0) \vee (d, 1)$, where value 1 is the value of another `write` a in τ_1, and is explored by other directives at $s_{[\langle\rangle]}$, *i.e.* $(a.b, (b, 0))$ and $(b, (b, 1))$. Also, the literal $(b, 0) \Vdash (b, 0) \vee (d, 1)$, that carries forward the summary that value 0 for b is explored from another directive at $s_{[\langle\rangle]}$, *i.e.* $(b, (b, 1))$

The computation of δ_ϕ, of a scheduling directive δ, is formally presented as a part of *forward-analysis* and *backward-analysis*, in §4.3.2.

Step 2. Addition of the computed scheduling directives to $\Delta(s_{[\tau]})$

Scheduling directives of a sequence are computed independently of scheduling directives of other sequences and may redundantly explore the same view-equivalence classes. To prevent such redundant exploration and ensure soundness and completeness, the set of scheduling directives computed across different execution sequences from $s_{[\tau]}$, we introduce a *coherent-union* operator (represented as $\uplus$), that combines a newly computed $\delta = (\delta_n, \delta_\phi)$ with $\Delta(s_{[\tau]})$. The operation $\Delta'(s_{[\tau]}) = \Delta(s_{[\tau]}) \uplus \delta$ is defined with the following properties.

1. $\Delta'(s_{[\tau]}) \neq \{\}$.
2. *properties concerning* δ_n.
   (a) Each δ'_n, δ''_n corresponding to $\delta', \delta'' \in \Delta'(s_{[\tau]})$ explore distinct view-equivalence classes.
   (b) The same set of view-equivalence classes are explored by extending the *next* sequences of $\Delta(s_{[\tau]}) \cup \{\delta\}$ and $\Delta'(s_{[\tau]})$.
3. *properties concerning* δ_ϕ

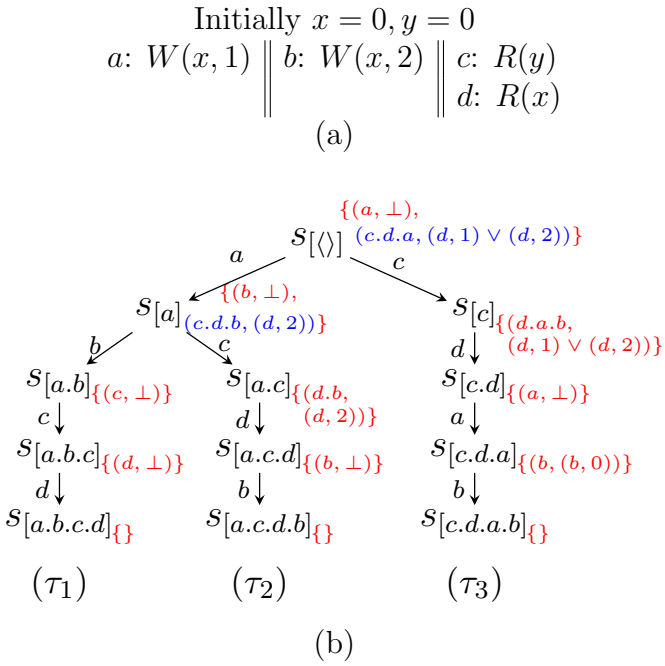


Figure 4.7: 2W1R. (a) input program, (b) exploration by ViEqui

(a) For each $\delta'' \in \Delta'(s_{[\tau]})$, $\delta' \in \Delta(s_{[\tau]}) \cup \{\delta\}$ s.t. δ''_n can also extend to explore a view-equivalence class of δ'_n, for each $e_r \in \mathcal{E}^{\mathbb{R}}_{\delta'_n} \cap \mathcal{E}^{\mathbb{R}}_{\delta''_n}$, if $(e_r, v)\Vdash\delta'_\phi$ then $(e_r, v)\Vdash\delta''_\phi$.

(b) For each $\delta'' \in \Delta'(s_{[\tau]})$, if $(e_r, v)\Vdash\delta''_\phi$ then $\exists\delta' \in \Delta(s_{[\tau]}) \cup \{\delta\}$ s.t. δ''_n can explore a view-equivalence class of δ'_n.

Consider the program given in Figure 4.7(a) and its exploration by ViEqui in Figure 4.7(b). $\Delta(s_{[\langle\rangle]})$ is computed using $\uplus$ after various executions as:

after τ_1: $\{(a,\perp)\} \uplus (c.d.a,(d,2)) = \{(a,\perp),(c.d.a,(d,2))\}$

after τ_2: $\{(a,\perp),(c.d.a,(d,2))\} \uplus (c.d.a,(d,1)) = \{(a,\perp),(c.d.a,(d,1)\vee(d,2))\}$

Backward-analysis after τ_1 for d from $\mathbb{I}_x$ computes a scheduling directive $(c.d.a,(d,2))$

Initially x=0
a: $W(x,0) \parallel b$: $W(x,1) \parallel c$: $W(x,1) \parallel d$: $W(x,2) \parallel e$: $R(x)$

Figure 4.8: example of forward-analysis

```
1 Function fwd(explored sequence τ, read event e_r):        /* let o = obj(e_r) */
2 |  W^o := {e_w ∈ E^W ∩ En(s_[τ]) | obj(e_w) = o}
3 |  W := unique_[τ](W^o \ done_[τ](e_r))      /* co-enabled and unique in value */
4 |  if lastW_[τ](e_r) ⊬ δ̄_φ(s_[τ]) then          /* include current value in W  */
5 |  |  W := W ∪ {lastW_[τ](e_r)}
6 |  forall e_w ∈ W do
7 |  |  δ_n := nseq-fwd_[τ](e_w, e_r)              /* event sequence for e_w →^rf_τ e_r */
8 |  |  δ_φ := ⋁_{e'_w ∈ W\{e_w}} (e_r, val_[τ](e'_w))          /* execution summary */
9 |  |  Δ(s_[τ]) ⊎= (δ_n, δ_φ)                    /* add to scheduling directives */
```

that is added to $s_{[\langle\rangle]}$. A similar backward-analysis after τ_2 computes a scheduling directive $(c.d.a, (d,1))$. To avoid redundant exploration from the two scheduling directives, ViEqui combines the them into $(c.d.a, (d,1) \vee (d,2))$ using $\uplus$, as shown above.

The resulting $\Delta(s_{[\langle\rangle]})$, shown at $s_{[\langle\rangle]}$ in Figure 4.7(b) satisfies the aforementioned properties (1), (2) and (3).

4.3.2 ViEqui algorithm

Forward-analysis and backward-analysis intuitively introduced earlier in this section are formally presented as functions fwd and bkwd respectively.

The functions fwd and bkwd use $\bar{\delta}_\phi(s_{[\tau]})$ to represent δ_ϕ of $\delta \in \Delta(s_{[\tau]})$ that is currently being explored. Further, the functions use $\mathsf{done}_{[\tau]}(e_r)$ to represent a set on `write` events that includes `writes` whose values are read in some other sequence or can be read at τ; formally, $\mathsf{done}_{[\tau]}(e_r) = \{e_w \in (\mathcal{E}^{\mathbb{W}}_\tau \cup \mathsf{En}(s_{[\tau]})) \cap \mathsf{W}^o \mid val_{[\tau]}(e_w) = val_{[\tau]}(\mathsf{last}W_{[\tau]}(e_r)) \vee (e_r, v) \Vdash \bar{\delta}_\phi(s_{[\tau]})\}$, where W^o represents writes of object $obj(e_r)$.

1 **Function** `bkwd`(*program execution* τ, ***read*** *event* e_r`):` `/* let` $o = obj(e_r)$ `*/`
2 $\quad \mathsf{W}^o_\tau := \{e_w \in \mathcal{E}^{\mathbb{W}}_\tau \mid obj(e_w) = o\}$
3 $\quad \mathsf{W} := \mathsf{unique}_{[\tau]}(\{e_w \in \mathsf{W}^o_\tau \cup \mathbb{I}_o \mid \mathsf{nseq\text{-}bkwd}_{[\tau]}(e_w, e_r) \neq \langle\rangle\} \backslash \mathsf{done}_{[\tau]}(e_r))$
4 $\quad$ **forall** $e_w \in \mathsf{W}$ **do** `/* writes of` τ`, unique in value */`
5 $\quad\quad \delta_n := \mathsf{nseq\text{-}bkwd}_{[\tau]}(e_w, e_r)$ `/* event sequence for` $e_w \rightarrow^{\mathtt{rf}}_\tau e_r$ `*/`
6 $\quad\quad \delta_\phi := \bigvee_{e'_w \in \mathsf{W} \backslash \{e_w\}} (e_r, val_{[\tau]}(e'_w)) \vee \mathsf{summary}_{[\tau]}(\delta_n)$ `/* exn. summary */`
7 $\quad\quad \Delta(pre_{[\tau]}(e_w, e_r)) \uplus= (\delta_n, \delta_\phi)$ `/* add to scheduling directives */`

Forward-analysis

The function fwd computes a set of enabled `writes` W^o that are of object o, same as that of the `read` e_r (line 2). Further from the set W^o, the function computes a subset W unique in value. The subset contains `writes` of W^o that are not in $\mathsf{done}_{[\tau]}(e_r)$ (line 3). If the value of $\mathsf{last}W_{[\tau]}(e_r)$ is not explored in some other sequence then it is also included in W (lines 4-5). After computing the set W, function fwd computes the *next* event sequence (line 7) and corresponding exploration summary (line 8) for each `write` in W as explained in §4.3.1. Finally, the computed directive for each `write` in W is added to $\Delta(s_{[\tau]})$ (line 9).

Consider the program in Figure 4.8, ViEqui performs forward-analysis at the initial state for `read` e. The set $\mathsf{done}_{[\langle\rangle]}(e) = \{a\}$, since the value of a is same as the current value of x; accordingly, $\mathsf{W} = \mathsf{unique}_{[\langle\rangle]}(\{b, c, d\}) \cup \{\mathbb{I}_x\} = \{\mathbb{I}_x, b, d\}$ or $\{\mathbb{I}_x, c, d\}$. Assuming $\mathsf{W} = \{\mathbb{I}_x, b, d\}$, lines 6-9 compute and add $(e, (e, 1) \vee (e, 2))$, $(b.e, (e, 0) \vee (e, 2))$ and $(d.e, (e, 0) \vee (e, 1))$ to $s_{[\langle\rangle]}$.

Backward-analysis

The function bkwd computes a set W^o_τ of `writes` of τ that are of object o (line 2). Further from the set W^o_τ, the function computes a subset W unique in value. The subset contains `writes` of W^o_τ that are not in $\mathsf{done}_{[\tau]}(e_r)$ such that a well-formed *next* sequence can be formed for e_r to read from $e_w \in \mathsf{W}$ (line 3). Function bkwd then computes the *next* event sequence (line 5) and the corresponding exploration

Algorithm 1: ViEqui algorithm (**Initially** `Explore`$(\langle\rangle, (\langle\rangle, \bot))$)

```
1  Function Explore(explored sequence τ, to explore (δ_n, δ_φ)):
2      if En(s_[τ]) = ∅ then                          /* maximal sequence explored */
3          forall e_r ∈ E^R_τ do bkwd(τ, e_r)   /* do backward-analysis and return */
4          return
5      if δ_n ≠ ⟨⟩ then                              /* has sequence to explore (δ_n) */
6          Δ(s_[τ]) := (δ_n, δ_φ)
7          Explore(τ.δ_n:hd, remaining(δ)); return   /* explore next in δ_n */
8      else                                          /* find next event to explore */
9          if ∃e_r ∈ E^R s.t. |co-en_[τ](e_r)| > 0 then
10             fwd(τ,e_r)                          /* forward-analysis possible on e_r */
11         else                                    /* forward-analysis not possible */
12             e := pickAny(En(s_[τ]))             /* pick any enabled event to explore */
13             Δ(s_[τ]) := (⟨⟩.e, ⊥)               /* scheduling directive to explore e */
14         forall (δ'_n, δ'_φ) ∈ Δ(s_[τ]) do       /* explore all scheduling directives */
15             Explore(τ.δ'_n:hd, remaining(δ'_n))
```

where, (i) for a sequence $\tau = e_1.e_2...e_n$, $\tau{:}hd = e_1$ and $\tau{:}tl = e_2...e_n$.
(ii) $remaining(\delta) = (\delta_n{:}tl,\ \mathsf{simplify}(\delta_\phi, (\delta_n{:}hd, val_{[\tau]}(\delta_n{:}hd))))$.
(iii) $\mathsf{co\text{-}en}_{[\tau]}(e_r) = \mathsf{En}(s_{[\tau]}) \cap \mathcal{E}^{\mathbb{W}}$ of $obj(e_r)$, if $e_r \in \mathsf{En}(s_{[\tau]})$; and $\{\}$, otherwise.

summary (line 6) for each `write` in W as explained in §4.3.1. Finally, the computed directive for each `write` in W is added to $\Delta(s_{[\tau']})$, where $s_{[\tau']} = \mathsf{pre}_{[\tau]}(\mathsf{e_w}, \mathsf{e_r})$ (line 7).

Consider again the backward-analysis after τ_1 in Figure 4.4(c) for d to read from $\mathbb{I}_x$. The set $\mathsf{done}_{[\langle\rangle]}(e) = \{a\}$, since the value of a is read by d in τ_1; accordingly, $\mathsf{W} = \mathsf{unique}_{[\tau_1]}(\{\mathbb{I}_x\}) = \{\mathbb{I}_x\}$; further lines 4-7 compute and add $(c.d.b.a, (b, 0) \vee (d, 1))$ to $\mathsf{pre}_{[\tau_1]}(\mathbb{I}_\mathsf{x}, \mathsf{d}) = s_{[\langle\rangle]}$ using $\uplus$.

The ViEqui algorithm

The algorithm takes an explored sequence (τ) and a previously computed scheduling directive to be explored (δ_n, δ_ϕ). If there are no enabled events at $s_{[\tau]}$, then the algorithm has explored a maximal sequence (line 2). As a final step, the algorithm performs backward-analysis for each `read` event of τ (line 3). If there are enabled events then the algorithm continues to explore and add scheduling directives (lines 5-

15). If the algorithm is in the middle of exploring a scheduling directive (*i.e.* $\delta_n \neq \langle\rangle$), then it executes the next event in δ_n (line 7). Additionally, a directive is computed for the next state to continue the exploration of the current scheduling directive. The directive for the next state is computed on the remaining *next* sequence and simplifying the corresponding formula using the values read by next event of δ_n (line 7). If the algorithm is not exploring a scheduling directive (*i.e.* $\delta_n = \langle\rangle$), then it chooses an enabled event to proceed (lines 9,12). The algorithm first looks for an enabled `read` with co-enabled `writes`, for feasible forward-analysis (line 9-10). If there does not exist such a `read` then any enabled event is selected to proceed (lines 12-13). Finally, all scheduling directives computed by this algorithm and also by forward- and backward-analyses are explored from $s_{[\tau]}$ (lines 14-15).

Soundness, completeness and optimality of ViEqui

Let Π represent the set of view-equivalence classes of the input program, and let E be the set of executions explored by ViEqui. We use $[\![\tau]\!]$ to denote the view-equivalence class represented by the sequence τ.

Theorem 3. ViEqui is complete. $\forall \tau \in E, \exists \pi \in \Pi$ s.t. $[\![\tau]\!] = \pi$.
(each maximal sequence explored by ViEqui represents a view-equivalence class of the input program.)

Proof. Given that ViEqui is an SMC, the technique can only execute an enabled event at each state of exploration $s_{[\tau']}$. Thus, the theorem can be restated as:
for each states of exploration $s_{[\tau']}$, $\forall \delta \in \Delta(s_{[\tau']})$, δ_n can be explored from $s_{[\tau']}$, *i.e.* $\forall e \in \delta_n$ s.t. $\delta_n = \tau_1.e.\tau_2$, $e \in \mathsf{En}(s_{[\tau'.\tau_1]})$.

Case 1. $\Delta(s_{[\tau']})$ is formed using forward-analysis.
$\Rightarrow \forall e \in \delta_n$, $e \in \mathsf{En}(s_{[\tau']})$. (by construction of sequences using forward-analysis)

Case 2. $\Delta(s_{[\tau']})$ is formed by backward-analysis. δ_n is formed by the well-formed join ($\oplus$) of enabling-sequences and $\bar{\delta}_n(s_{[\tau']})$, defined in Figure 4.5(c).
$\forall e \in \tau'$, where, τ' is an enabling sequence, e is enabled after its prefix in τ' (by

definition of enabling sequences), similarly, $\forall e \in \bar{\delta}_n(s_{[\tau']})$, e is enabled after its prefix in $\bar{\delta}_n(s_{[\tau']})$ (since, $\bar{\delta}_n(s_{[\tau']})$ is previously explored from $s_{[\tau']}$).

As a consequence, an event e in δ_n is not enabled after its prefix in δ_n
$\Rightarrow$ well-formed join of enabling sequences and $\bar{\delta}_n(s_{[\tau']})$ does not enable e after it prefix in δ_n.
$\Rightarrow$ a program branch that does not contain e will be explored by δ_n.
$\Rightarrow \exists$ a `read` e_r in the prefix of e in δ_n s.t. $val_{[\delta_n]}(e_r) \neq val_{[\bar{\delta}_n(s_{[\tau']})]}(e_r)$, or $val_{[\delta_n]}(e_r) \neq$ the value read in the corresponding enabling sequence.
$\Rightarrow \text{co}_{\delta_n} \neq \text{co}_{\bar{\delta}_n(s_{[\tau']})} \cup \text{co}_{\tau_e}$ (where, τ_e represents the enabling sequences).
$\Rightarrow$ the result of well-formed join $= \langle\rangle$. (by definition of $\oplus$) $\Rightarrow \delta_n$ is not formed by backward-analysis.

Hence, by contradiction, ViEqui is complete. □

Theorem 4. ViEqui is sound. $\forall \pi \in \Pi, \exists \tau \in E$ s.t. $[\![\tau]\!] = \pi$.
(for each view-equivalence class of the input program there exists a maximal sequence explored by ViEqui.)

Proof. Consider a state $s_{[\tau']}$ s.t. a `read` e_r can read the value v after $s_{[\tau']}$. (*cond*1)

Let $o = obj(e_r)$. Let $\exists \tau \in E$ s.t. τ' is prefix of τ.

Assume,

(A1) e_r is enabled in $\tau \Rightarrow \exists \tau''$ s.t. $\tau'.\tau''.e_r$ is a prefix of τ.

(A2) $\exists e_w \in \mathcal{E}_\tau^{\mathbb{W}} \cup \mathbb{I}_o$ s.t. $val_{[\tau]}(e_w) = v$.

(A3) $\forall$ states $s_{[\tau_i]}$, $\forall \delta_i \in \Delta(s_{[\tau_i]})$, $\delta_{i\phi} = \bot$, representing an absence of exploration summary at each state.

There exist three cases for e_w in τ, (i) $e_w \in \mathcal{E}_{\tau'.\tau''} \cup \mathbb{I}_o$ (explored before e_r), (ii) $e_w \in \mathsf{En}(s_{[\tau'.\tau'']})$ (enabled with e_r), and (iii) not cases (i) and (ii) (explored after e_r).

- In case (i), if $e_w = \mathsf{last}W_{[\tau]}(e_r)$, and in case (ii), forward-analysis computes a scheduling directive $\delta = (\delta_n, \delta_\phi)$ s.t. $val_{[\delta_n]}(e_r) = v$.

- In case (i), where $e_w \neq \mathsf{last}W_{[\tau]}(e_r)$, and in case (iii), $(cond1) \Rightarrow \mathsf{nseq\text{-}bkwd}_{[\tau]}(e_w, e_r) \neq \langle\rangle$, thus, backward-analysis computes a scheduling directive $\delta = (\delta_n, \delta_\phi)$ s.t. $val_{[\delta_n]}(e_r) = v$.

Hence, in the absence of exploration summary (assumption A3), forward- and backward-analysis compute a scheduling directive δ for e_r to read v. *inf(1)*

Further, let us relax the assumption A3.
Let $\nexists\ \tau \in E$ where τ' is a prefix of τ and $val_{[\tau]}(e_r) = v$.

inf(1) $\Rightarrow \forall \delta \in \Delta(s_{[\tau']})$ either $(e_r, v)\Vdash\delta_\phi$, or $(e_r, v)\nVdash\delta_\phi$ but $val_{[\delta_n]}(e_r) \neq v$.

Consider a scheduling directive $\delta \in \Delta(s_{[\tau']})$.

- $(e_r, v)\Vdash\delta_\phi$ and δ is computed using forward-analysis $\Rightarrow \exists \delta' \in \Delta(s_{[\tau']})$ s.t. $val_{[\delta'_n]}(e_r) = v$. (by construction of δ_ϕ using forward-analysis).

- $(e_r, v)\Vdash\delta_\phi$ and δ is computed using backward-analysis $\Rightarrow e_r \in \delta_n$ (by construction of δ_ϕ using backward-analysis) $\Rightarrow \delta$ is computed to explore value v' for e_r from an execution where e_r reads v, where $v' = val_{[\delta_n]}(e_r)$ (by definition of backward-analysis).

- If $(e_r, v)\nVdash\delta_\phi$ but $val_{[\delta_n]}(e_r) \neq v \Rightarrow$ backward-analysis computes a scheduling directive $\delta' = (\delta'_n, \delta'_\phi)$ s.t. $val_{[\delta_n]}(e_r) = v$ (using *inf(1)*).

Using properties (1), (2)(b), and (3)(b) of $\uplus$, we know that the view-equivalence classes of a δ computed using forward- or backward-analysis will not be omitted because of $\uplus$.

Thus, by contradiction ViEqui is sound under assumptions A1 and A2.

Corollary 1. Since every value that can be read by a `read` event at a state is explored by ViEqui, it implies that ViEqui explores every branch in the control flow graph of the input program.

Using the corollary stated above, the assumptions A1 and A2 always hold. □

Theorem 5. ViEqui is optimal. $\nexists \tau_1, \tau_2 \in E$, where $\tau_1 \neq \tau_2$, s.t. $[\![\tau_1]\!] = [\![\tau_2]\!]$.
(no two execution sequences explored by ViEqui belong to the same view-equivalence class.)

Proof. Using properties (1) and (2)(a) of $\uplus$ we can state that scheduling directives at a state $s_{[\tau]}$, cannot result in redundant explorations. *inf(1)*

Further, let $\exists \delta \in \Delta(s_{[\tau]})$ and $\exists \delta' \in \Delta(s_{[\tau']})$, where $\tau \neq \tau'$, such that δ and δ'can extend to representative executions of the same view-equivalence class.

The property (3)(a) of $\uplus$ ensures that the exploration summary δ_ϕ computed for a scheduling directive (δ) from a sequence τ is carried forward to each δ' computed from exploration of δ. As a result, property (3)(a) of $\uplus$ prevents redundant execution of τ.

However, consider the computation of $\delta \in \Delta(\tau)$ and $\delta' \in \Delta(\tau')$ from two executions where neither one is computed during the exploration of the other execution. Assume that δ and δ' can both explore a view-equivalence class.

Since, all explorations start at a unique initial state $s_{[\langle\rangle]}$
$\Rightarrow \exists \tau_1$ in prefix of both τ and τ'.

$\exists \delta_1, \delta_1' \in \Delta(s_{[\tau_1]})$ that extend τ_1 to τ and τ' respectively. The directives δ_1 and δ_1' explore distinct view-equivalence classes. (using *inf(1)* at $s_{[\tau_1]}$)

Thus, δ and δ' can both explore a view-equivalence class
$\Rightarrow$ that τ is a subsequence of δ_n or τ' is a subsequence of δ_n'.
$\Rightarrow s_{[\tau]}$ or $s_{[\tau']}$ is a *hidden* state (refer to §4.3.1).
$\Rightarrow \delta_{1n} = \delta_n - \tau$ can only explore the view-equivalence classes of δ, and ${\delta_1}'_n = \delta_n' - \tau'$ can only explore the view-equivalence classes of δ'.

Hence, by contradiction, ViEqui is optimal. □

4.3.3 Time and Space complexity

Consider the following observations.

(o1) Each complete exploration of the input program (exploration of all equivalence classes) may have at most $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ maximal sequences. This computation includes a maximal sequence for each combination of $|\mathcal{V}|$ values for each `read` event, where in the worst case all such combinations are feasible.

(o2) Given the optimal nature of the algorithm, the maximum number of scheduling directives computed by forward- and backward-analysis is $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$.

Worst-case time complexity. The worst-case time complexity of forward-analysis is $\mathscr{F} = \mathcal{O}(|\mathbb{T}|^2 + |\mathbb{T}|.\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}))$ where the component $|\mathbb{T}|^2$ corresponds to the computation of the scheduling directives for each `write` in W (line 3, forward-analysis, module 1) and the component $|\mathbb{T}|.\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|})$ corresponds to the computation of W.

The worst-case time complexity of backward-analyses, function `bkwd` (module 2) is $\mathscr{B} = \mathcal{O}(|\mathcal{E}^{\mathbb{W}}|.(|\tau|^3 + |\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|))$. The component $|\tau|^3$ corresponds to the computation of nseq-bkwd (line 5), and component $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|$ corresponds to the computation of $\uplus$ (line 7). The computations are done for each relevant `write` event.

The complexity analysis of forward- and backward-analyses is detailed in Appendix 1.

The ViEqui algorithm (Algorithm 1) computes forward- and backward-analyses for each `read` event in a sequence.

Thus, the complexity of exploring one maximal sequence is $\mathcal{O}(|\mathcal{E}^{\mathbb{R}}|.(\mathscr{F} + \mathscr{B}))$ and the complexity of the ViEqui algorithm is $\mathcal{O}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\mathcal{E}^{\mathbb{R}}|.(\mathscr{F} + \mathscr{B}))$.

Best-case time complexity. The best-case time can be achieved by the ViEqui algorithm in case of exclusive forward-analysis. Under such a case the computation of done would be reduced to $\mathcal{O}(\texttt{log}(|\mathcal{E}^{\mathbb{R}}|.|\mathcal{V}|))$ due to the construction of boolean formula by forward-analysis.

As a result, the best-case time complexity of the algorithm is $\theta(|\mathbb{T}|^2 + |\mathbb{T}|.\texttt{log}(|\mathcal{E}^{\mathbb{R}}|.|\mathcal{V}|))$.

Worst-case space complexity. Given observations (o1) and (o2), in the worst-case, the number of scheduling directives at a state may be $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$, if each of those scheduling directives is formed at the same state.

Further, if the state where $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ scheduling directives are formed is the initial state, then, in the worst-case the length of each scheduling directive may be $|\mathcal{E}|$ *i.e.* the length of a maximal sequence (feasible when scheduling directives are formed with backward-analysis alone). However, in such a case the scheduling directives at the remaining states would be computed by line 12 of Algorithm 1 and thus, would be sub-sequences of a scheduling directive of the initial state.

As a result, for each maximal sequence the size of scheduling directives can be computed as $|\mathcal{E}| + (|\mathcal{E}| - 1) + (|\mathcal{E}| - 2) + ... + 1 = |\mathcal{E}|.(|\mathcal{E}| + 1)/2$. Given $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ maximal sequences, each with scheduling directives of size $|\mathcal{E}|.(|\mathcal{E}|+1)/2$, the space complexity of the algorithm, in the worst-case, is $\mathcal{O}(|\mathcal{E}|^2.|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|})$.

Best-case space complexity. Assuming all $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ combinations of `read` events and their corresponding values are feasible then, in the best-case, each of the $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ scheduling directives can be computed using forward-analysis. As the result, the lengths of the scheduling directives are constant in size ($\leq$2), by design. Thus, given $|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}$ maximal sequences, each with scheduling directives of constant size, the space complexity of the algorithm in the best-case is $\theta(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|})$.

4.4 Implementation details of ViEqui SMC

The implementation of the ViEqui technique consists of two major components,

1. an *execution component* responsible for executing the input program for each maximal sequence explored by the technique, and
2. an *analysis component* that applies Algorithm 1 on the executing program and controls the interleaving order to ensure execution of the desired sequence.

For each maximal sequence the execution component restarts the input program and awaits directives on the next program instruction to execute. After executing a program instruction the component returns back with the set of next available program instructions (the instructions made available on execution of the last instruction).

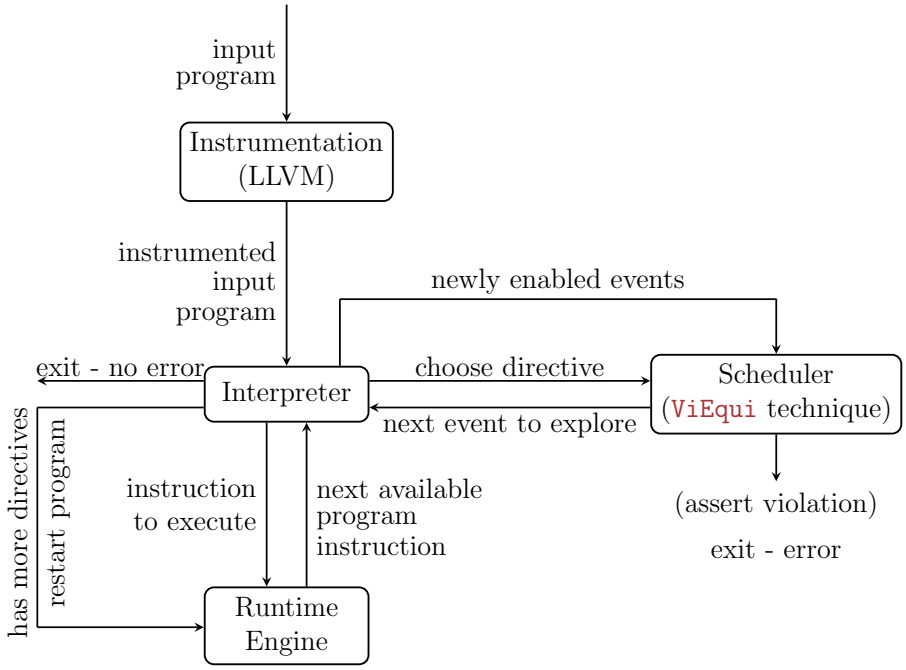


Figure 4.9: Structural overview of ViEqui tool

The set of available program instructions are interpreted as program events and provided to the analysis component which in turn chooses the next event to execute as per Algorithm 1. The next event is suitably interpreted as the program instruction to execute for the execution component.

The process continues till a maximal execution is executed or a violation of a safety property is detected. Further, the execution component restarts the input program for the next sequence to be explored when directed by the analysis component (*i.e.* when there are unexplored scheduling directives).

The key modules of the implementation design are presented in Figure 4.9 and discussed below.

- **Instrumentation.** As the first step, the implementation performs a source-to-source transformation of the input program called *instrumentation*. The

instrumentation inserts code in the input program at relevant control points such as memory access, thread creation, thread join, assert condition etc.

The instrumentation calls respective modules on reaching the relevant control points to supply data for the analyses and receive execution instructions. The instrumentation, thus, allows ViEqui tool to take control of the interleaving order to realize the desired execution sequence. The instrumentation preserves program behaviors.

- **Runtime Engine.** The runtime engine executes the instrumented program as per the directed execution order.

- **Scheduler.** The scheduler implements the ViEqui technique. At each exploration state the technique receives a fresh set of enabled events and chooses the next event to transition on.

- **Interpreter.** The interpreter forms the communication channel between the runtime engine and the scheduler. The modules called by the instrumentation, on reaching the relevant control points, belong to the interpreter.

  The interpreter translates the information flowing between the runtime engine and the scheduler (for instance mapping of program instructions and corresponding events).

4.4.1 Tool description

The implementation of the ViEqui technique is done in `C++` language over `Nidhugg` [71] tool. The tool takes a `C` or `C++` program as input and uses the `pthread` library for multi-threading. The *intermediate representation* of the source program generated after compilation is instrumented using `LLVM`. The instrumentation performs a source-to-source transformation of the intermediate representation by inserting suitable calls to interpreter modules. The tool detects the violations of safety properties provided as assert statements in the input program.

Initiating the verification of an input program on ViEqui tool invokes a central control unit called the *trace builder*. The trace builder invokes the runtime engine and starts a fresh execution. The trace builder further launches the interpreter and the scheduler. The interpreter receives the initial set of enabled events and communicates the set to the scheduler.

The interpreter then starts the exploration and invokes the scheduler requesting for the next event to execute. The scheduler then applies the ViEqui technique on the set of enabled events to form suitable scheduling directives and returns the next event to be executed as per the technique's analysis. The interpreter in turn instructs the Runtime Engine to execute the corresponding thread and gather a fresh set of enabled events that are further communicated to the scheduler.

The trace builder monitors the execution and detects violations of assert conditions. The tool halts on detecting an assert violation and returns a *bug trace*, *i.e.* an event sequence leading to the assert violation.

If the tool explores a maximal sequence (no assert condition in the input program was violated) then the scheduler identifies a state $s_{[\tau]}$, where τ is the longest prefix of the maximal sequence such that $s_{[\tau]}$ has an unexplored scheduling directive. If such a state exists the the trace builder restarts the input program and the interpreter and scheduler are given the initial set of enabled events. However, for a restarted execution the scheduler returns the events of the prefix τ, in the order of occurrence in τ, on iterative invocations from the interpreter till it reaches the state $s_{[\tau]}$. At $s_{[\tau]}$ the scheduler returns the next event of the *next* sequence of an unexplored scheduling directive.

Supported data-types and operations

The tool supports atomic and non-atomic data-types and all data structures. The tool further supports the following operations on global variables (program events): read, write and the following `rmw` operations, *fetch-and-add*, *fetch-and-subtract*, *fetch-and-and*, *fetch-and-nand*, *fetch-and-or*, *fetch-and-xor*, *fetch-and-max*, *fetch-and-min*,

exchange and *compare-and-exchange*. The tool supports all operations on local variables. The support for coarse-grained locking is supported through the *pthread-mutex* synchronization primitive.

Support for read-modify-write events

The tool supports `rmw` events (refer to 2) by considering an `rmw` event as a combination of a `read` event followed by a `write` event. As a consequence, the implementation reuses the infrastructure for `read` and `write` events. However, the implementation performs additional analysis to ensure atomicity of the `read` and the `write` event corresponding to an `rmw` event, such as (i) ensuring both the `read` and the `write` belong to a scheduling directive on an `rmw` event, and (ii) the `write` event of an `rmw` event is explored immediately after the corresponding `read` event, further all relevant analysis on the memory location occur only after the occurrence of the `write` event.

Write value of `rmw` events. A fundamental difference between a general `write` event and the `write` event corresponding to an `rmw` event is that the value of a general write event is known when the `write` event is enabled (refer to 2). The same does not hold for the `write` event of an `rmw` event whose value is computed after its corresponding `read` event gets its value. As a consequence, the implementation performs dry computation after the `read` of an `rmw` event to compute the `write` value of the already enabled corresponding `write` event.

Modification to computation of *next* sequences. Consider the input program in Figure 4.10(a), the program contains two *fetch-and-add*(`FAA`)[2]instructions from different threads. The instructions are handled as separate `read` and `write` events by the ViEqui tool as shown in Figure 4.10(b). Since the write values of the events are not known until the execution of the corresponding `read` components, the write values are represented as v_1 and v_2 respectively.

Figure 4.10(c) shows an execution sequence of the program where the red values in

[2]A *fetch-and-add* instruction `FAA`(x, v) reads the value of the memory location x, add the value v to the value read for x and writes the updated value back to the memory location x.

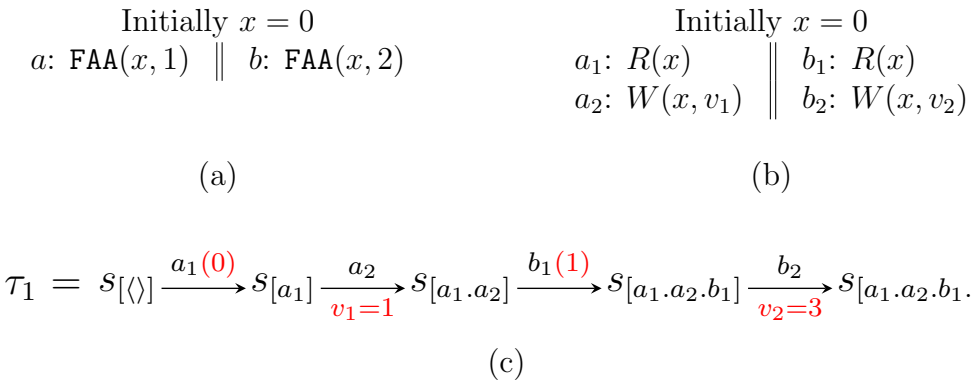


Figure 4.10: 2FAA

bracket represent the values read by the `read` components of the `rmw` events and the concrete values for v_1 and v_2 are shown in red, below the corresponding `write` event labels.

Two backward analyses will be performed in the sequence corresponding to (i) the initial value 0 for the `read` event b_1, and (ii) the value 3 (from b_2) for the `read` event a_1. No scheduling directive would be formed for analysis (ii) due to the coherence dependence of b_2 on a_1. Also, no scheduling directive would be formed for analysis (i) as a well-formed join for b_1 to `read` 0 and a_1 to read 0 would result in $\langle\rangle$. The result of analysis (i) is incorrect and would lead to a fail of completeness.

The above stated condition occurs as a consequence of considering an `rmw` event as two separate events. To fix the error of analysis ViEqui tool considers the scheduling directive being explored at state s as $(\langle\rangle, \bot)$ if the conjunction of the following conditions is met;

1. forward-analysis was performed at s,
2. the forward-analysis was performed for the `read` of an `rmw` event, and
3. the event for which the backward-analysis is being performed is an `rmw` event.

4.5 Experiments and Results on ViEqui SMC

4.5.1 Experimental setup

The experiments are conducted on an Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz with 32GB RAM and 32 cores running Ubuntu 20.04.1 LTS. LLVM 6.0.0 is used for to perform the instrumentation.

The performance of ViEqui is compared against two techniques that claim optimality of exploration under their respective equivalence relation, namely, ODPOR [1] and ODPOR-with-observers (obs-ODPOR) [19]. The technique ODPOR claims optimality under the classical notion of equivalence and the technique obs-ODPOR claims optimality under equivalence based on observed races.

Further, ViEqui is also compared against a non-optimal technique that uses `read` values to define equivalence called RVF-SMC [12]. The technique is based on reads-value-from equivalence.

All the four techniques, ODPOR, obs-ODPOR, RVF-SMC, and ViEqui are built on the `Nidhugg` [71] tool. The implementation of the techniques are structurally similar (refer to Figure 4.9). The *scheduler* modules, of the respective implementations, carry out the steps of the corresponding techniques and the *interpreter* modules may be suitably adapted.

4.5.2 Litmus testing

ViEqui is tested on 16,154 litmus tests of multi-threaded `C` programs, focusing on

(i) **reporting assert violations**: if there exists a program execution that violates an assert condition then the ViEqui tool generates a bug trace;

(ii) **completeness**: if the ViEqui tool explores a maximal sequence then the sequence represents a valid execution of the input program;

(iii) **optimality**: the ViEqui tool does not explore a redundant maximal sequence.

Identifying failure in detecting assert violations. For the set of litmus tests, if an assert condition for a test may violate in some program execution, then ViEqui must report an assert violation. The set of tests where an assert condition may violate is known for the set of litmus tests. Note that, as the set of equivalence classes or the set of all feasible executions is not known a priori, ViEqui uses assert violations to test the soundness of the implementation.

Identifying failure of completeness. Given that the input program is indeed executed (and not symbolically inspected) for every sequence explored by the algorithm, a successful completion of an execution implies that the sequence explored truly represents a program execution. Thus, a completeness failure occurs when a *next* sequence scheduled to be explored from a state of exploration cannot be translated to an execution, for instance, when the next event in the sequence is not enabled or does not represent a program event.

Identifying failure of optimality. After exploration of a maximal sequence, ViEqui saves the trace signature as a set of (event, value) pairs representing the `read` events of the trace along with the corresponding values read in the trace. ViEqui then compares the trace signatures using Definition 3. An optimality fail occurs when the trace signatures of two traces are equal, implying that the corresponding sequences are equivalent according to Definition 3 and represent the same equivalence class.

Litmus tests

The 16,154 litmus tests contain tests borrowed from previous work, modifications of borrowed tests and synthesized tests.

8,058 tests are borrowed from [9], a SMC called obs-ODPOR, based on equivalence of observed races. The entire set of 8,058 tests borrowed from [9] has an assert condition in the input program that is not violated in any program execution. The assert condition in the 8,058 tests is negated to generate another 8,058 tests where

Table 4.2: Litmus Tests

Category	#tests	Avg. #Seq	Avg. Time	Total #Seq	Total Time	#complete+optimal
No assert violation	8091	10.15	0.02s	82124	171.71	8091
Has assert violation	8063	1.00	0.017	8066	137.87	8063

the assert condition is violated in some program execution. The remaining 38 tests are synthesized to test various features of the ViEqui algorithm. The average length of litmus tests is 68.46 lines of code.

The results of litmus testing is summarized in Table 4.2. The results are grouped under two categories.

1. *No assert violation.* This category represents the tests where the assert condition in the input program is not violated in any program execution. For such cases, ViEqui explores the entire set of equivalence classes (through representative executions) and provides a proof of correctness of the input program by showing the absence of assert violation for any equivalence class.

   This category includes the original set of 8,058 tests borrowed from [9] and 33 synthesized tests, that is, a total of 8,091 tests.

   ViEqui performs complete and optimal exploration for each of the 8,091 tests of this category.

2. *Has assert violation.* This category represents the tests where the assert condition in the input program is violated in some execution of the program. For such cases, ViEqui detects the assert violation and reports a bug trace. ViEqui halts its exploration at the detection of the first assert violation.

   This category includes the 8,058 tests generated by modifying the tests from [9] (by negating the assert condition) and 5 synthesized tests.

   ViEqui detected an assert violation for each of the 8,063 tests of this category.

Column 'Category' in Table 4.2 represents the category of the tests, column '#tests' shows the number of tests in the category and column '#complete+optimal' rep-

resents the number of tests in a category that performed a complete and optimal exploration.

Columns 'Total #Seq' and 'Avg. #Seq' show the total and average number of sequences explored across the tests of a category respectively, that is, the number of sequences explored for sound exploration, under the 'No assert violation' category, and to reach the first execution sequence that results in an assert violation, under the 'Has assert violation' category.

Similarly, columns 'Total Time' and 'Avg. Time' show the total and average time of analysis across the tests of a category respectively. The time of analysis is recorded over 5 runs for each test.

4.5.3 Performance analysis

The performance analysis of ViEqui is done over challenging benchmarks borrowed from SV-comp benchmark suite [22], SCTBench benchmark suite [66] and previous works [9, 19]. The performance is measured on various configurations of each benchmark, where the configurations vary on the problem size determined by program features such as the number of loop unrolls and the number of concurrent processing elements (or threads). In essence, higher configurations of the benchmarks result in a higher number of program events.

The performance on the benchmarks is measured on three aspects.

1. *Time of analysis.* The time taken to verify a configuration of a benchmark input program.
2. *Scalability.* The highest configuration of a benchmark that can be verified within a reasonable time of analysis, also known as *Timeout of analysis* or simply *Timeout* (To), set at 1800 seconds.
3. *Number of maximal sequences explored.* The number of maximal sequences analyzed for either a complete exploration (in case the input program does not

violate an assert condition), or reaching the first assert violation (in case the input program violates an assert condition).

The performance analysis of ViEqui against that of techniques ODPOR, obs-ODPOR, and RVF-SMC is presented in Table 4.3 and Table 4.4. Table 4.3 shows the results on benchmarks where the assert condition is not violated in any program execution. Similar to the respective category of litmus tests, the techniques explore the entire set of equivalence classes for such benchmarks and provide a proof of correctness for the input program. On the other hand, Table 4.4 shows results on benchmarks where the assert condition in the input program is violated in some execution of the input program. For such cases, the techniques report the assert violation and halt the exploration after detecting the first assert violation.

The column 'benchmark' of Table 4.3 and Table 4.4 represents the name of the benchmark along with its configuration and the column 'Test ID' represents a unique ID for each unique configuration of the benchmarks.

The columns '#Seq' represent the number of sequences explored by the respective techniques. For Table 4.3 this value represents the number of sequences analyzed by each technique to explore the entire set of equivalence classes, while for Table 4.4 this value represents the number of sequences analyzed for reaching the first sequence that leads to an assert violation. Note that, since the techniques ODPOR, obs-ODPOR and ViEqui claim optimality of exploration, the corresponding columns '#Seq' in Table 4.3 represent the number of equivalence classes under the respective equivalence relations. However, the technique RVF-SMC is non-optimal and thus the column '#Seq' of Table 4.3 corresponding to RVF-SMC also includes redundant explorations (multiple explorations corresponding to the same equivalence class) and partial explorations (non-maximal explorations due to inaccuracies in scheduling).

The columns 'Time' represent the time of analysis recorded in seconds for each technique. The value in the columns represents the time of analysis, averaged over 5 runs. The value 'To' represents the timeout of analysis set at 1800s.

Configurations of benchmarks. The configurations of a benchmark vary the prob-

Table 4.3: ViEqui performance analysis (benchmarks with no assert violation)

test ID	benchmark	ODPOR #Seq	ODPOR Time	obs-ODPOR #Seq	obs-ODPOR Time	RVF-SMC #Seq	RVF-SMC Time	ViEqui #Seq	ViEqui Time
1	pgsql(5,5)	781	0.72	781	0.70	19900	3.06	781	0.73
2	pgsql(6,7)	55987	68.57	55987	77.51	2292077	654.66	55987	183.77
3	pgsql(7,7)	137257	171.45	137257	199.18	5356580	1620.66	137257	933.77
4	monabsex(100)	To	-	To	-	101	0.99	1	0.08
5	monabsex(500)	To	-	To	-	501	162.84	1	2.80
6	unverif(5,5)	14400	2.74	14400	3.13	68890	11.70	14400	198.61
7	unverif(5,10)	14400	2.98	14400	3.31	70890	12.76	14400	201.80
8	unverif(6,5)	518400	110.60	518400	129.32	2625944	699.47	To	-
9	redundant-co(10)	To	-	12431	3.10	11	0.01	7	0.02
10	redundant-co(50)	To	-	To	-	11	0.02	7	0.03
11	redundant-co(1000)	To	-	To	-	11	0.11	7	3.59
12	swsc-co1(20)	To	-	To	-	8060	14.14	7240	4.93
13	swsc-co1(50)	To	-	To	-	125150	1705.93	120100	322.98
14	swsc-co1(60)	To	-	To	-	To	-	208920	764.64
15	swsc-co10(10)	To	-	10	0.06	11	0.02	10	0.02
16	swsc-co10(100)	To	-	100	42.17	101	7.46	100	0.60
17	swsc-co10(250)	To	-	250	1732.37	251	278.73	250	6.83
18	alpha2(100)	To	-	To	-	10203	741.98	10101	183.67
19	alpha2(150)	To	-	To	-	To	-	22651	1054.27
20	burns(5)	2353602	1046.92	2353602	1155.09	17382	5.14	36	0.05
21	burns(10)	To	-	To	-	To	-	121	0.31
22	burns(40)	To	-	To	-	To	-	1681	185.75
23	burns(60)	To	-	To	-	To	-	3721	1532.97
24	dekker-simple(10)	739021	420.96	739021	468.19	2713870	704.97	21	0.03
25	dekker-simple(100)	To	-	To	-	To	-	201	32.05
26	dekker-simple(150)	To	-	To	-	To	-	301	288.25
27	dekker-simple(200)	To	-	To	-	To	-	401	1269.84
28	peterson(5)	2782162	1432.44	2782162	1584.59	To	-	31	0.04
29	peterson(50)	To	-	To	-	To	-	301	19.63
30	peterson(100)	To	-	To	-	To	-	601	474.40
31	peterson(120)	To	-	To	-	To	-	721	1186.56
32	szymanski(4)	396583	198.87	396583	221.96	1444246	319.78	5335	4.87
33	szymanski(5)	To	-	To	-	To	-	19349	25.81
34	szymanski(7)	To	-	To	-	To	-	264209	659.04
35	nondet-array-2(4,4)	2616	0.89	688	0.32	534	0.08	51	0.04
36	nondet-array-2(6,6)	To	-	711276	519.29	63491	6.50	2163	2.65
37	nondet-array-2(14,7)	To	-	To	-	908984	128.72	18731	90.68

Table 4.4: ViEqui performance analysis (benchmarks with assert violation)

test ID	benchmark	ODPOR #Seq	ODPOR Time	obs-ODPOR #Seq	obs-ODPOR Time	RVF-SMC #Seq	RVF-SMC Time	ViEqui #Seq	ViEqui Time
38	nondet-array-1(100,100)	1	0.22	1	0.05	1	0.04	1	0.22
39	nondet-array-1(1000,500)	1	0.03	1	0.03	1	0.02	1	0.09
40	tas(20,50)	To	-	To	-	23	0.08	3	46.05
41	tas(30,50)	To	-	To	-	33	0.15	3	100.51
42	tas(40,50)	To	-	To	-	43	0.26	3	178.78
43	IBM-incdec(50)	To	-	To	-	To	-	3	9.36
44	IBM-incdec(100)	To	-	To	-	To	-	3	45.57
45	triangular-2(5)	20172	2.69	20172	3.12	26272	2.41	1576	0.85
46	triangular-2(7)	1695856	266.81	1695856	311.04	644193	70.10	32517	470.08
47	triangular-2(8)	To	-	To	-	3045756	360.65.10	To	-
48	FreeBSD-abd-kbd	1	0.03	1	0.02	1	0.02	1	0.04
49	FreeBSD-rdma-addr	1	0.02	1	0.03	1	0.01	1	0.03
50	NetBSD-sysmon-power	4	0.03	4	0.02	6	0.02	5	0.05
51	Solaris-space-map	2	0.03	2	0.03	1	0.02	1	0.03

lem size such that higher configurations require a higher effort of analysis. The configurations typically vary on the number of concurrent elements (threads), the number of loop iterations, and the number of `read` or `write` events. The configurations of the benchmarks in Tables 4.3, 4.4 vary on the following aspects.

The configurations of benchmark `'pgsql'` (test IDs 1-3) vary on the number of loop iterations and the number of threads, where loop body contains `read` events. The configurations of benchmark `'monabsex'` (test IDs 4-5) vary on the number of threads. The configurations of benchmark `'unverif'` (test IDs 6-8) vary on the number of threads and the number of loop iterations, where the loop body contains `write` and `read` events. The configurations of benchmark `'redundant-co'` (test IDs 9-11) vary on the loop iterations, where the loop body contains `write` events.

The configurations of benchmarks `'SWSC-co1'`, `'SWSC-co10'` and `'alpha2'`, test IDs 12-14, 15-17 and 18-19 respectively, vary on the number of writer threads, that is, threads that perform `write` operations. The configurations of benchmarks on mutual

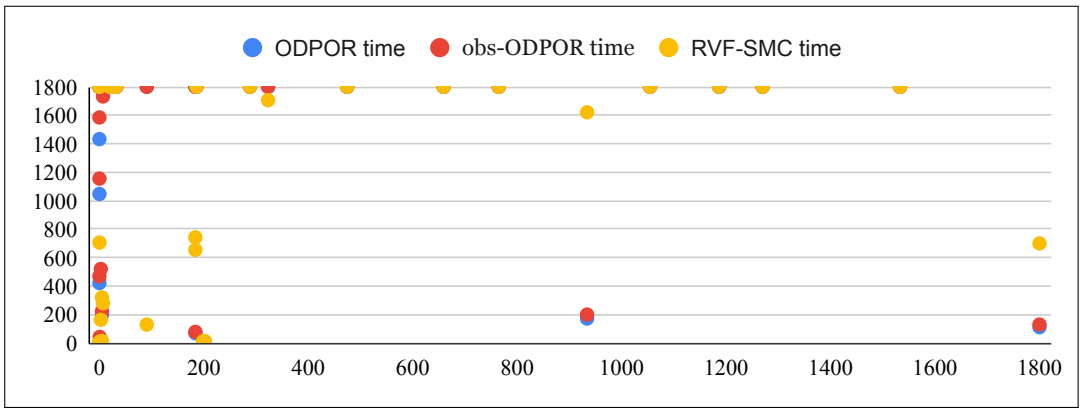


Figure 4.11: Time of analysis of existing SMCs vs Time of analysis of ViEqui (seconds)

exclusion algorithms `'burns'`, `'dekker-simple'`, `'perterson'` and `'szymanski'`, test IDs 20-23, 24-27, 28-31 and 32-34 respectively, vary on the number of attempts to enter the critical section, where the critical sections contain `read` and `write` events.

The configurations of benchmarks `'nondet-array-1'` and `'nondet-array-2'`, test IDs 38-39 and 35-37 respectively, vary on the size of the array and the number of threads.

Observations. A scatter plot contrasting the performance of existing techniques against ViEqui is shown in Figure 4.11. Each point in the graph represents the time of analysis of the corresponding technique (on y-axis) against the time of analysis of ViEqui (on x-axis) on the tests in Table 4.3. It can be observed that the points in the graph are concentrated near the origin of the x-axis, however, the points are *not* concentrated near the origin of the y-axis. This represents that the time of analysis of the other techniques is typically higher in comparison to that of ViEqui.

A similar trend can also be observed from the Table 4.3 where the time of analysis of other techniques (for providing a proof of correctness of the tests) is seen to be higher than that of ViEqui and the other techniques are seen to timeout (To) for a significantly larger number of tests. Similarly, it can be seen from Table 4.4 that ViEqui outperforms the techniques ODPOR and obs-ODPOR and performs comparable to RVF-SMC in detecting violations of assert conditions.

The time of analysis per execution can be higher for ViEqui in comparison to the other optimal techniques, ODPOR and obs-ODPOR. The speedup observable from the Tables 4.3, 4.4 is a result of having to consider fewer equivalence classes during examination. The said speed-up is witnessed specifically with the test IDs 4-5 (`'monabsex'`), 9-11 (`'redundant-co'`), and 43-44 (`'IBM-incdec'`). The set of values for the tests, $|\mathcal{V}| < 4$, and the set of `reads` ($\mathcal{E}^{\mathbb{R}}$) remains the same across configurations for these tests. Hence, increasing the set of `writes` does not increase the view-equivalence classes and we witness an exponential saving in the time of analysis.

In contrast, for test IDs 1-3 (`'pgsql'`) and 6-8 (`'unverif'`) the set of equivalence classes under the classical equivalence and view-equivalence is the same. A slowdown is witnessed on the benchmarks with ViEqui, and other SMCs based on coarser equivalence relations, empirically establishing that SMCs based on coarser equivalence relations may take a higher time of analysis per execution.

obs-ODPOR shows an exponential saving over ODPOR with `'SWSC-co10'` (test IDs 15-17). The benchmark has N `writes` of N different values from separate threads and a single `read` event at the end (after the join of the writer threads). As a result, obs-ODPOR computes an exponentially smaller number of equivalence classes in comparison to the equivalence classes of ODPOR. ViEqui also computes the same set of equivalence classes as obs-ODPOR while performing exponentially faster than obs-ODPOR. `'SWSC-co1'` and `'alpha2'` (test IDs 12-14,18-19) are a similar benchmarks but with more `reads`, hence they show similar results but take comparatively longer to analyze.

Benchmarks `'nondet-array-1'` and `'nondet-array-2'` (test IDs 35-39) concurrently update an array. The `reads` are performed for determining array indices, hence, the sets $\mathcal{E}^{\mathbb{R}}$ and $\mathcal{V}$ are small (for an array of length L, $|\mathcal{E}^{\mathbb{R}}| = |\mathcal{V}| = L$). Thus, ViEqui performs well on these benchmarks.

Test IDs 43-47 ('IBM-incdec', 'triangular-2') have long causal chains. The causal chains add to the time of backward-analysis, slowing down the time of analysis of ViEqui. However, test IDs 43-44 have, relatively, fewer `reads` and values allowing ViEqui to scale better.

Benchmark `'tas'` (test IDs 40-42) showcases a scenario where forward-analysis slows-down the analysis. Various states along the execution present opportunity of forward-analysis and as a result the assert condition is reached slower. Thus, although the number of execution sequences explored to reach the assert violation is very small, the time of analysis is significantly high.

The benchmarks of test IDs 20-34 correspond to benchmarks on mutual-exclusion algorithms. Such algorithms typically have a large set of `writes` of a small set of values. ViEqui thus performs well on the tests.

Test IDs 48-51 represent slices of bugs in FreeBSD, NetBSD and Solaris [22]. The results show that the bugs can be detected with ViEqui, thereby, empirically showing the applicability of stateless model checking under view-equivalence.

4.6 Scope, Limitations, and Future directions

Scope of view-equivalence relation. The coarser notion of view-equivalence pivots on detection of safety property (assert) violations, similar to other notions of equivalence (except the classical equivalence relation) (discussed in §4.1.1). The applicability of view-equivalence is not reduced in context of assert violations in comparison to the other coarse equivalence relations (refer to §4.1.1).

All such coarse notions of equivalence cannot detect data-races in the input program. However, various modern languages, such as `C` and `C++`, have introduced atomic data types[3]thereby eliminating the harmful effects of data-races.

Furthermore, the definition of view-equivalence is applicable for all memory models.

Difference with view-equivalence of DBMS schedules. View-equivalence as a term has been previously used to define equivalence of schedules in database management systems (DBMS). The two concepts are similar only in name.

[3]Atomic data types ensure that all bytes of an atomic object are accessed atomically. As a result, the behavior of data-races on atomic objects is well-defined.

Two schedules S_1 and S_2 of transactions are said to be view-equivalent in DBMS, if they satisfy the following conditions:

1. Each reading transaction reads the same value.
2. Each reading transaction reads from the same transaction.
3. Final write is performed by the same transaction.

Here, a reading transaction refers to a transaction performing a read.

If we collapse a transaction to a single memory access operation to enable comparison against the view-equivalence relation ($\sim$), then by definition the view-equivalence notion under DBMS is a finer notion of equivalence than the relation $\sim$.

Scope of ViEqui technique. ViEqui SMC is defined for the *sequential-consistency*. Intuitively, the model considers a single shared memory between concurrent processing elements (threads) and the program outcomes are generated by interleavings of concurrent events. The model does not support out-of-order execution of events from a program thread (refer to §3.2.1).

ϕ boolean formula and existing representation for equivalence classes. Various orders of occurrence on events may correspond to the same view-equivalence class, even those where a `read` event reads from different `write` events, if they write the same value. It is non-trivial to associate ordering on events with a view-equivalence class, where a view-equivalence class is defined on an unordered set of `reads` and their values.

As a consequence, representations of equivalence classes that exploit the commutativity of concurrent events, such as *ample sets* [74], *persistent sets* [39], and *source sets* [4], become unusable under view-equivalence. Similar representations for previously explored equivalence classes, such as *done sets* [35] and *sleep sets* (over events [39] or event sequences [13]) cannot effectively represent view-equivalence classes.

Scheduling directives and wakeup trees. *Wakeup trees* are used by the ODPOR technique [1] to store sequences that must be explored from a state. Wakeup trees are a tree data structures that induce an ordering on the sequences to be explored

Table 4.5: Performance of SMCs on known bugs

benchmark	ODPOR		obs-ODPOR		RVF-SMC		ViEqui		violation	
	#Seq	Time	#Seq	Time	#Seq	Time	#Seq	Time	detected?	
FreeBSD-abd-kbd	1	0.03	1	0.02	1	0.02	1	0.04	Yes	(a)
FreeBSD-rdma-addr	1	0.02	1	0.03	1	0.01	1	0.03	Yes	
NetBSD-sysmon-power	4	0.03	4	0.02	6	0.02	5	0.05	Yes	
Solaris-space-map	2	0.03	2	0.03	1	0.02	1	0.03	Yes	
Safestack	oom	-	oom	-	To	-	To	-	-	(b)

from a state. The ordering helps the technique achieve optimality.

On the other hand, scheduling directives are unordered sets of sequences and boolean formula. As a consequence the operational semantics of the two representations is unrelated. The representations are same only in purpose.

Threats to Validity

Scalability. SMCs are limited in scalability. While ViEqui outperforms SMCs based on finer notions of equivalence there is still a significant scope for improvement in performance of SMCs in terms of the time of analysis and scalability.

Consider the performance of various SMCs on known bugs shown in Table 4.5, where oom represents that the analysis ran *out of memory* and To represents a timeout of analysis which is set at 1800s. The tests in category (a) have a high number of program branches but the executions are small in length. We can observe that the SMCs perform well on such tests. However, the test in category (b) has long execution sequences and none of the SMCs used in this study scale on the test.

Establishing ground truth. The view-equivalence classes are not known to ViEqui a priori. In this work the ground truth is established based on other techniques as follows: for each program execution explored by the other techniques (ODPOR, obs-ODPOR, and RVF-SMC) there exists a view-equivalent execution sequence explored by ViEqui.

4.6.1 Future directions

SMC under view-equivalence for RMMs. In future, stateless model checking under view-equivalence can be investigated for relaxed/weak memory models (RMMs). View-equivalence as a notion is oblivious to a memory consistency model. ViEqui proposes a SMC under view-equivalence for a strong sequential-consistency memory model. A ViEqui like approach can be naturally extended to weaker memory models such as TSO and PSO (refer to 3.2.2) that are still significantly strong. The outcomes under TSO and PSO memory models can be justified as total occurrence order on events or execution sequences on a single shared memory. Thus, a ViEqui like approach with support for out of order execution of events can extend support for stateless model checking under view-equivalence for TSO and PSO.

Weaker memory models, such as the C11 memory model or the memory model of ARM architecture (ARM), use a set of events and event relations to define consistent outcomes (refer to 3.2.3). View-equivalence as a notion is directly applicable to such memory models, however, a ViEqui like approach cannot encompass all outcomes of such a weak memory model.

View-equivalence of transactions. As discussed previously, the existing notion of view-equivalence under DBMS is conceptually different and finer. A notion similar to view-equivalence that defines equivalence on schedules of transactions that read the same values can be defined. A model checker for verifying schedules of transactions based on view-equivalence can also be investigated.

Parallelization of ViEqui. The ViEqui technique presents viable opportunities for parallelization. For instance the scheduling directives generated by forward-analysis can be explored in parallel; or the backward-analysis on line 4 of Algorithm 1 may be performed for different values in parallel. Redundancies of scheduling directives may likely arise as a result of the parallelization, however, the same may be mitigated by repeated redundancy check, using a $\uplus$ like approach.

Support for richer constructs. The ViEqui technique introduced in §4.3 performs analysis at instruction level on `read` and `write` memory accesses. The technique does

not comprehend coarse-grained synchronization mechanisms such as locks. The main challenge of introducing lock-awareness in the ViEqui technique is in generating *next* sequences (refer to §4.3.1) that enable the respective events and ensure the required reads-from while maintaining consistency of `lock` acquisitions and `unlocks`. the introduction of *lock-awareness* introduces the notion of *disabling events*. An event once enabled in ViEqui is never disabled throughout an execution sequence. However, with coarse-grained synchronization, an enabled event may be disabled later in an execution. The ViEqui technique can be made lock-aware primarily with a lock-aware construction of *next* sequences.

4.7 Concluding remarks

This work presents a novel equivalence relation for trace partitioning called *view-equivalence*. The relation is shown to be at least as coarse as any existing equivalence relation, that is, for any input program, the number of equivalence classes formed under view-equivalence is not larger than the number of equivalence classes formed under any other equivalence relation.

View-equivalence pivots on detection of safety property (assert) violations same as other coarse notions of equivalence. However, the applicability of view-equivalence is not reduced in context of safety property (assert) violations in comparison to the other coarse equivalence relations.

This work also presents an SMC that performs stateless model checking based on view-equivalence under sequential consistency memory model, called ViEqui. ViEqui is shown to be *sound*, *complete*, and *optimal* in its exploration. This work also presents the worst and best case, time and space complexity analysis for the ViEqui algorithm.

The ViEqui technique is accompanied with an implementation for `C` and `C++` input programs. This work presents the corresponding implementation details including the (i) structural overview and key components, (ii) details of libraries and platforms used, and (iii) details of data-types and operations supported in the input program.

The implementation of the ViEqui technique is tested over 16000+ litmus tests from previous works with the focus on reporting assert violations, completeness of exploration and optimality of exploration. The tool passes the 16000+ litmus tests on the three focal considerations.

To determine the effectiveness of the ViEqui technique and implementation, the tool is tested on challenging benchmarks. The test results are compared against existing optimal stateless model checkers and model checkers that use `read` values to determine equivalence. The tests compare the techniques on the time of analysis, the number of execution sequences explored and scalability. The comparative results on benchmarks highlight that ViEqui significantly outperforms the other techniques in terms of the time of analysis and scalability, thus, establishing the efficacy of view-equivalence and the effectiveness of the ViEqui SMC.

In addition to the core ViEqui technique designed for `read` and `write` memory access events, this work presents the extended versions of ViEqui technique that support read-modify-write events and coarse-grained synchronization primitives. The extensions are presented with formal definitions and algorithmic modifications.

Finally, this work discusses the scope of view-equivalence and ViEqui SMC, and highlights differences with some popular and similar notions in related fields. Based on the scope of view-equivalence and ViEqui SMC this work also proposes worthy future directions.

Appendix A. ViEqui time complexity details

Complexity of operators.

operator $\mathsf{eseq}_{[\tau]}(e)$. Computing an enabling sequence requires one pass over the sequence hence, its worst-case time complexity is $\mathcal{O}(\tau)$.

operator $\tau_1 \oplus \tau_2$. The complexity of operator is $|\tau_1|^2.|\tau_2|$ *i.e.* $\mathcal{O}(|\tau|^3)$.

operator $\uplus$. Given S number of scheduling directives at the state $s_{[\tau]}$ and N calls to $\uplus$, the complexity of computing $S \uplus{=}N$ is $\mathcal{O}(N.|\tau| + N.S.|\tau|) = \mathcal{O}(N.S.|\tau|)$.

operator $\mathsf{nseq\text{-}bkwd}_{[\tau]}(e', e)$. The worst-case complexity of this operator is dominated by the complexity of computation of causal-join ($\oplus$), *i.e.* $\mathcal{O}(|\tau|^3)$.

Complexity of module $\texttt{fwd}(\tau, e_r)$ (module 1)

Complexity of $\mathsf{done}_{[\tau]}(e_r)$ is $\mathcal{O}(\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}))$ corresponding to the largest size of the boolean formula. Complexity of computing $\mathsf{unique}_{[\tau]}(W^o\mathsf{done}_{[\tau]}(e_r))$ is $\mathcal{O}(|\mathbb{T}|)$ as only one event can be enabled from each thread. Similarly, the complexity of computing W is $\mathcal{O}(|\mathbb{T}|.\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}))$.

Since, the enabling sequence for forward analysis are of constant lengths, the computation of $\mathsf{nseq\text{-}fwd}$ takes constant time. Thus, the complexity of computing scheduling directives is dominated by the computation of boolean formula *i.e.* $|\mathbb{T}|$. Given $|\mathbb{T}|$ number of computations of boolean formula where each computation takes $|\mathbb{T}|$ time, the worst-case complexity of forward-analysis is $\mathcal{O}(|\mathbb{T}|^2 + |\mathbb{T}|.\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}))$.

Complexity of module $\texttt{bkwd}(\tau, e_r)$ (module 2)

Computation of W requires computing $\mathsf{nseq\text{-}bkwd}$ whose complexity is $|\tau|^3$. Complexity of computing done is $\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|})$. Thus, computation of W is $\mathcal{O}(|\mathcal{E}_\tau^{\mathbb{W}}|.|\tau|^3 +$

$|\mathcal{E}_\tau^{\mathbb{W}}|.\texttt{log}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}))$ (lines 2-3).

In the worst-case, the complexity of computing $\uplus$ is $\mathcal{O}(|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|)$ (line 7). The complexity of computing pre is $\mathcal{O}(|\tau|)$ (line 7).

Thus, the complexity of the steps inside the loop (lines 5-7) is $\mathcal{O}(|\tau|+|\tau|^3+|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|+|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}) = \mathcal{O}(|\tau|^3+|\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|)$. Since, the loop iterates $|\mathcal{E}_\tau^{\mathbb{W}}|$ times in the worst-case (for the maximum number of elements in W) the complexity of the loop is $\mathcal{O}(|\mathcal{E}_\tau^{\mathbb{W}}|.(|\tau|^3 + |\mathcal{V}|^{|\mathcal{E}^{\mathbb{R}}|}.|\tau|))$, which is also the complexity of `bkwd`

Chapter 5

MoCA. Dynamic Verification of C11 Concurrency over Multi Copy Atomics

5.1 Background

Models of memory consistency, popularly known as Memory Models, describe the consistent interactions between threads through the shared memory (refer to §3.2). The memory model specification determine the permitted `read` and `write` memory accesses on shared memory locations by specifying the permitted out-of-order execution of events from a thread and visibility of an event to other threads and events. Consequently, the allowed outcomes of an input program under a memory model can be interpreted by the memory model specification.

Most modern architectures define their memory models, for instance the memory model TSO of x-86 architecture, PSO of sparc and the ARM and Power memory models associated with the ARM and Power architectures. The set of feasible orderings on memory accesses, or execution on an architecture can be interpreted from the architecture's memory model. Therefore, the set of possible outcomes for the same

input program may differ across various architectures.

In addition to architectures, modern programming languages, such as `C`, `C++`, `Java` and `OCaml`, also specify a memory model. The memory models of languages define the semantics of memory storage and memory management for the language abstract machines, such as JVM (`Java` virtual machine) associated with the `Java` language. They are also used by compilers to optimize programs. The consistency specification of the language memory models helps compilers determine the permitted optimizations.

The 2011 (and subsequent) ISO standard for C/C++ includes the `C11` memory model, which introduces *memory orders* (refer to §3.2.3). The memory orders can be syntactically associated with atomic memory access events (refer to §3.2.3) and dictate how other memory accesses (non-atomic or atomic) can be ordered with respect to the current memory access. By syntactically tagging memory access events, developers can specify a desired set of permitted event orderings, with the assumption that the underlying architecture will execute the input program within the developer's specification of permitted orderings.

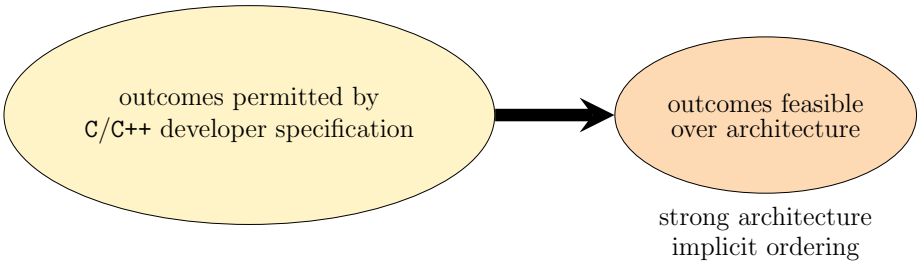


Figure 5.1: Set of feasible outcomes on an architecture with strong implicit ordering may be smaller than the set of outcomes permitted by `C/C++` developer specification.

In some cases, the implicit ordering specified by the underlying architecture's memory model may be stronger than the ordering specified in a C/C++ input program using memory orders. In this scenario, the set of feasible outcomes for the program may be restricted to a subset of the outcomes permitted by the developer's specification. This is illustrated in Figure 5.1.

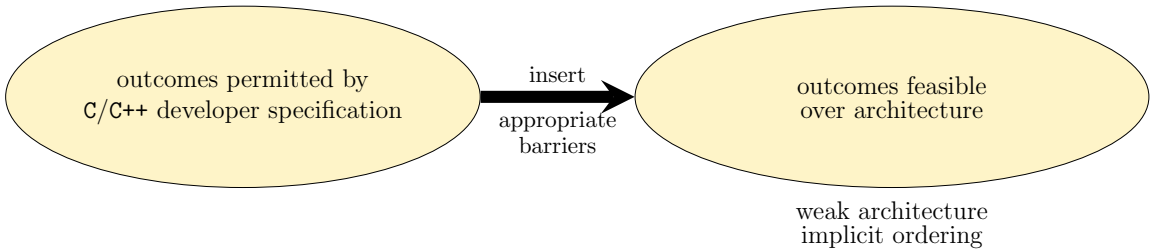


Figure 5.2: Set of feasible outcomes on an architecture with weak implicit ordering is same as the set of outcomes permitted by C/C++ developer specification.

On the other hand, when a C/C++ input program specifies a strong ordering, it is possible that the underlying architecture does not implicitly form the specified ordering. In such cases, the architecture still executes the input program within the developer's permitted orderings. To achieve this, appropriate barriers are introduced in the intermediate/machine code to restrict the feasible outcomes that are not permitted under the developer specification. This ensures that the set of feasible program outcomes is identical to the set of program outcomes permitted by the developer specification, as illustrated in Figure 5.2.

5.1.1 Verification under various memory models

Verification under sequential consistency

Several prior investigations have focused on stateless model checking to analyze programs under sequential consistency. Most of these techniques [1, 7, 13, 15, 26, 27, 80, 88] use DPOR [35] as the basis to achieve better scalability. Central to such techniques is the notion of equivalence of execution sequences such that the techniques must explore any one of the equivalent execution sequences for sound verification. The techniques explore efficient representation of *ample sets* [1, 13] to avoid *redundant* explorations (refer to §4.1). Various techniques use coarser notions of equivalence [7, 12, 26, 27] to reduce the number of execution sequences to explore. To mitigate the scalability challenge of SMCs based on DPOR, the technique in [80] pro-

poses a distributed DPOR SMC and the technique in [88] proposes a stateful DPOR SMC. DPOR has also been combined with parameterized unfolding semantics and proposed under sequential consistency [70, 78].

Symbolic and predictive trace analysis by encoding the thread interleaving of a program is also a popular approach investigated in [37, 44, 87].

Static analysis using either thread modular analysis abstract interpretation has also been explored in the context of concurrent programs under sequential consistency. Techniques have been proposed for flow-insensitive [34] and flow-sensitive [58] thread modular analysis with abstract interpretation.

Verification under architecture specific memory models

SMCs have also been proposed for weak architectural memory models such as TSO and PSO [2, 24, 89], and Power [8]. An SMC technique for a superset of architectural memory models (including the TSO, PSO, ARM and RISC-V memory models) is proposed by [54]. The focus of such techniques is on recognizing ordering on memory access events that are feasible under the respective memory model and verifying each such permitted ordering.

Thread modular analysis with abstract interpretation has also been used to verify programs under weak memory models such as TSO [57, 82], and PSO and sparc-RMO [82].

Verification under C11 memory model or its variants

The C11 memory model has been the object of intense study in the past decade [5, 20, 51, 53, 72]. The work [20] establish mathematical (axiomatic) semantics for C11 concurrency. The works [5, 51, 53, 72] present verification techniques for C/C++ input programs under the C11 memory model or its variants. The verification techniques explore the orderings on memory access events that are feasible under the C11 seman-

tics and permitted by the developer specification through memory orders specified by tagging the memory accesses.

The semantics of the C11 model are known to be complex, and for a fragment of the standard, called *release-acquire*[1](RA) [50, 75], the state-reachability problem is shown to be undecidable [5].

The techniques in [5, 72] use DPOR to explore the feasible orderings. The technique in [72] performs verification for the C11 memory model, while the technique in [5] performs verification under RA fragment of C11.

The techniques in [51, 53] verify C/C++ input programs under a variant of C11 model called the RC11 model [60]. The techniques form execution graphs representing the unique program outcomes and perform optimal stateless model checking.

Recently, a technique using thread modular analysis with abstract interpretation has been proposed for the RA fragment of C11 [79].

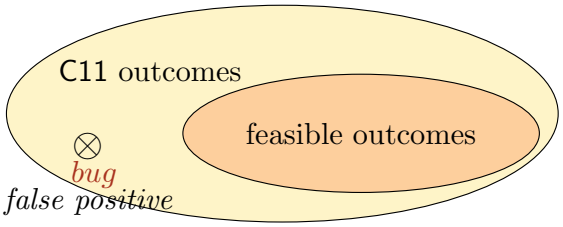
5.2 Imprecise analysis of C/C++ programs

Imprecise analysis by C11 verification techniques

The state-of-the-art verification techniques designed for the C11 memory model (or its variants and fragments), such as CDSChecker [72] and GenMC [51], perform analysis on the outcomes permitted by the developer specification in the input program. However, each underlying architecture has its own associated memory model.

As discussed in §5.1, the architecture memory model may have a strong implicit ordering thereby restricting some of the program outcomes permitted under C11. As a consequence, notwithstanding the sophistication of the C11 verification techniques,

[1]Under the RA model all `writes` are release accesses, while all `reads` are acquire accesses. RA is accepted as a useful and well-behaved subset of C11 that provides a good balance between performance and reasoning.



(a)

Initially $x = 0, y = 0$

a: $W^{\texttt{rlx}}(x, 1)$	c: $R^{\texttt{rlx}}(y, 1)$
b: $W^{\texttt{rlx}}(y, 1)$	d: $R^{\texttt{rlx}}(x, 0)$

outcome under C11: YES
outcome under TSO: NO

(b)

Figure 5.3: Imprecise analysis under weak C11 and strong architecture specifications.

some of the C11 outcomes flagged buggy by these these techniques would not occur when the input program executes on an architecture with a strong implicit ordering. The condition is illustrated in Figure 5.3(a) where a flagged bug exists in the set of C11 outcomes (yellow blob) but not in the subset of outcomes feasible on the architecture (orange blob). Thus, the bug is tagged as a *false positive*.

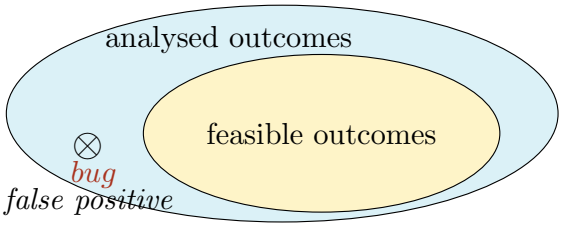
Consider the outcome shown in Figure 5.3(b), where all memory accesses are tagged with memory order `rlx`. The program outcome is permitted under the C11 memory model with the given `rlx` specification for memory accesses. However, the outcome is not feasible under memory models with strong implicit ordering such as TSO. As a result, if the given outcome is deemed undesirable then the outcome would be flagged as a bug by any verification technique designed for the C11 model.

Thus, a verification technique designed for the C11 model may flag bugs that may never manifest as an execution.

Imprecise analysis by architecture specific verification techniques

Further, the state-of-the-art verification techniques designed for memory models such as TSO, PSO, ARM and Power do not consider the memory orders specified in the input program. As a result, the techniques disregard the developer specification.

As discussed in §5.1, the underlying architecture may not implicitly form the strong ordering in the developer specification. In such a case the architecture restricts the



(a)

Initially $x = 0, y = 0$

a: $W^{\mathtt{sc}}(x, 1)$	c: $R^{\mathtt{sc}}(y, 1)$
b: $W^{\mathtt{sc}}(y, 1)$	d: $R^{\mathtt{sc}}(x, 0)$

outcome under C11: NO
outcome under PSO + barriers: NO
outcome under PSO − barriers: YES

(b)

Figure 5.4: Imprecise analysis under strong C11 and weak architecture specifications.

additional outcomes (not permitted under C11 specification) by inserting appropriate barriers. Since the existing verification techniques operate at the source level, no such barriers are inserted before performing the verification.

As a consequence, some of the outcomes flagged buggy by these these techniques would not occur when the input program executes on the underlying architecture in the presence of appropriate barriers. The condition is illustrated in Figure 5.4(a) where a flagged bug is in the set of outcomes feasible in the absence of barriers (blue blob) but not with barriers (yellow blob). Thus, the bug is tagged as a *false positive*.

Consider the outcome shown in Figure 5.4(b), where all memory accesses are tagged with memory order `sc`. The program outcome is *not* permitted under the C11 memory model with the given `sc` specification for memory accesses. When the program is translated for a sparc (PSO) architecture, appropriate full memory barriers are inserted between the events a and b, and c and d. Thus, the outcome is also not feasible on PSO. However, any existing verification technique for PSO does not consider the `sc` memory order and permits the outcome. As a result, if the given outcome is deemed undesirable then the outcome would be flagged as a bug by a verification technique designed for a model with weak implicit ordering such as PSO.

Thus, a verification technique designed for architecture specific models may also flag bugs that may never manifest as an execution.

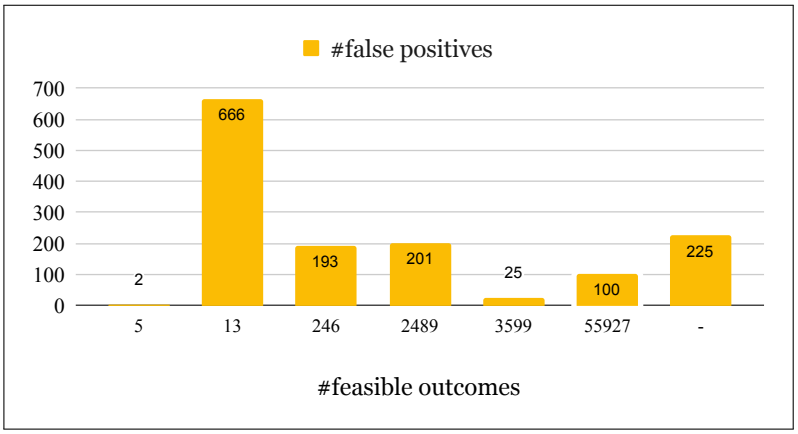


Figure 5.5: #feasible program outcomes vs #false positive bugs

5.3 Precise analysis for C11 programs under MCA

As a consequence of the imprecise analysis of C11 programs, if a verification technique detects a property violation, it must be scrutinized further for viability under C11 and a given architecture memory model. The laborious task of sorting the feasible from infeasible violations is a burden on the developer.

Our experiments show that even for small programs (less than 100 lines of code) the verification techniques designed for C11 model and architectural models can flag hundreds of bugs that can never manifest as an execution. As a consequence the bugs may be considered false positives. The graph in Figure 5.5 summarizes the results of the said experiments. The x-axis of the graph represents the number of program outcomes that are feasible under the C11 model and the underlying architecture[2], while the y-axis represents the number of bugs flagged by a C11 verification technique called CDSChecker, such that each of the outcomes flagged buggy are not feasible on the underlying architecture.

As can be seen from the graph in Figure 5.5, the number of false positive bugs reported by the existing techniques may be hundreds in number and the number of execution

[2]Note that, '-' for '#feasible outcomes' represents that the number of feasible outcomes could not be determined because the corresponding analysis could not scale.

sequences explored by the techniques to detect those bugs may be much higher, when the actual set of feasible outcomes (shown on the x-axis) may be very small.

As a consequence, the developer may take on the task of fixing the bugs when a large number of them may never manifest, or the developer may sort the feasible from infeasible bugs. Both the tasks are highly non-trivial in nature and may be exacting event for expert programmers.

Therefore, an important desideratum is to engineer analysis tools/techniques to analyze C11 programs under restricted architecture models with *precision*.

This work investigates the problem of precise dynamic analysis of C11 programs under the *Multi-Copy-Atomic* model (MCA). MCA is a popular architecture memory model that is supported by memory models such as Intel's x-86 TSO, newer versions of ARM[3](version 8 and later), and Alpha, with varying degrees of permitted reordering. A noteworthy aspect of the MCA model is the assumption of a single abstract view of shared memory between the processing elements (or threads), leading to the observation that permitted program behaviors under this model can be explained solely through interleaving and reordering of thread events (refer to §3.2.2 for details on MCA). Note that, the number of feasible outcomes depicted on the x-axis of the graph in Figure 5.5 correspond to the C11 outcomes feasible under MCA.

Precise analysis of C11 concurrent programs over MCA has the following merits

(M1) The analysis takes into consideration the developer C11 specification along with the architecture implicit ordering guarantees.

(M2) MCA is supported by some of the most widely used architectures. As a result, correctness results over MCA hold for most systems in use.

(M3) Precise analysis only flags the outcomes as buggy that can indeed manifest as executions reducing the developer overhead.

[3]ARM-version-8 calls its model *other*-MCA [64], however, the difference of *other*-MCA with MCA is that of terminology, not semantics (as clarified by [76]).

(M4) Precise analysis also implies a restricted (to MCA) number of equivalence classes, reducing the analysis overhead.

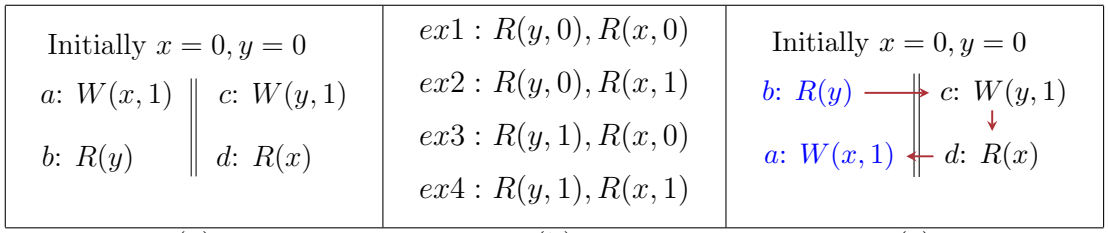
5.4 Operational semantics of MCA model

Under the MCA model if a `write` to an object o from a thread, $\mathbb{T}_i$, is *observed* by a different thread, $\mathbb{T}_j$, then the `write` is *coherently* observable by all other threads that access the object o. It is, however, permitted for a thread to observe its own `writes` prior to making them visible to other threads in the system [64]. Here, the term *observed* refers to the thread $\mathbb{T}_j$ becoming aware of a `write` from the thread $\mathbb{T}_i$, either directly when a `read` of $\mathbb{T}_j$ reads-from the `write` or indirectly through a chain of intra- and inter-thread dependencies [64]. The terms *observed* and *coherent* access are formally revisited in §5.5 after formally defining event interactions.

The MCA model formalized in [31] is used for the formal semantics of MCA, for this work. The relevant details of the model are briefly presented below.

The model allows for a sequence of events $\mathbb{T}_i{:}\tau$ of thread $\mathbb{T}_i$ to be reordered to a sequence $\mathbb{T}_i{:}\tau'$. Consider two events, $e', e \in \mathcal{E}_{\mathbb{T}_i:\tau}$, such that e originally occurs *after* e' in $\mathbb{T}_i$. The reordering of the two events such that now e occurs *before* e' is represented by $e' \overset{R}{\Leftarrow} e$. In the absence of such a reordering the program executions are outcomes accepted under the *interleaving semantics* (refer to §3.1.2). The reordering of program events of a thread may introduce outcomes that cannot be justified by any interleaved order and require modification to the order of events from a thread.

Consider the example shown in Figure 5.6(a). The possible outcomes of the input program are shown in Figure 5.6(b) through the respective `read` events. The outcomes ex2-4 can be justified by interleaving of events from the two threads. However, the outcome $ex1$ can only be justified by reordering the events of at least one of the two threads, such as the one shown in Figure 5.6(c), where the edges represent the order of execution of events after reordering shown in blue.



(a) (b) (c)

Figure 5.6: Store buffer. (a) input program, (b) outcomes, (c) reordered program

Reordering of events

For any two events e', e the reordering $e' \overset{R}{\Leftarrow} e$ is defined by a set of rules under an input memory model. Let the set of event pairs allowed to reorder be referred as the *degree of permitted reordering* under a memory model. Note that, larger the set of event pairs that can reorder from a thread under a memory model, higher is the memory model's degree of permitted reordering.

While memory models provide their own set of rules for reordering, all memory models under MCA follow an invariant restriction on reordering of two events that share a program dependence (which includes data, address and register). Let $\texttt{dep}(e', e)$ represent the dependence of e on e'.

Forwarding of events

In the course of executing the input program a condition may arise where two program statements accessing the same shared object translate to independent register operations. Consider the events $W(x,1)$ and $(r := R(x))$ in the event sequence $\mathbb{T}_i{:}\tau$ where x is a shared object and r is a local object and $(r := R(x))$ represents that the read value of x is stored in r. Further, let $W(x,1){<}_{\mathbb{T}_i:\tau}(r := R(x))$. The effect of $W(x,1)$ when captured in $(r := R(x))$ will translate the event to $W(r,1)$. As a consequence, the pair of events which were earlier data dependent have now become data *independent*.

The possible reordering under MCA of events e', e related as $\texttt{dep}(e', e)$ where e can absorb the effect of e' (represented as $e_{\langle e' \rangle}$) is called *forwarding* [31]. Using reordering with forwarding the event $W(x, 1)$ and $(r := R(x))$ can be reordered by capturing the effect of $W(x, 1)$ in $(r := R(x))$ as $W(x, 1) \overset{R}{\Leftarrow} (r := R(x))_{\langle W(x,1) \rangle}$, where the event $(r := R(x))_{\langle W(x,1) \rangle} = W(r, 1)$.

Intuitively, reordering with forwarding forces the visibility of `writes` of a thread to all the `reads` originally occurring after the `write` in the same thread, even after any possible reordering.

Semantic preserving reordering with forwarding

To ensure a semantic preserving reordering with forwarding, the following conditions must hold [31]:

(spr1) $\mathcal{E}_{\mathbb{T}_i:\tau'} = \mathcal{E}_{\mathbb{T}_i:\tau}$ (the event sets are the same)

(spr2) each `read` event of $\mathbb{T}_i{:}\tau$ must have the same set of `writes` to read from in $\mathbb{T}_i{:}\tau$ as well as in $\mathbb{T}_i{:}\tau'$ (*thread semantics*)

(spr3) $\mathbb{T}_i{:}\tau'$ preserves the order of updates and accesses of each shared object with respect to $\mathbb{T}_i{:}\tau$ (*coherence-per-location*).

Language and rules of MCA operational semantics

The language that represents an MCA model as specified by [31] is,

processing element,	$p := (\mathbb{T}_i \ \mathbf{lcl}\ \sigma \cdot \mathbb{T}_i{:}\tau) \mid p1 \parallel p2$
system,	$s := (\mathbf{shr}\ \sigma \cdot p)$

The key element of the language is a *processing element*. A processing element is identified by a unique identifier ($\mathbb{T}_i$), the local state ($\mathbf{lcl}\ \sigma$), and a sequence of events

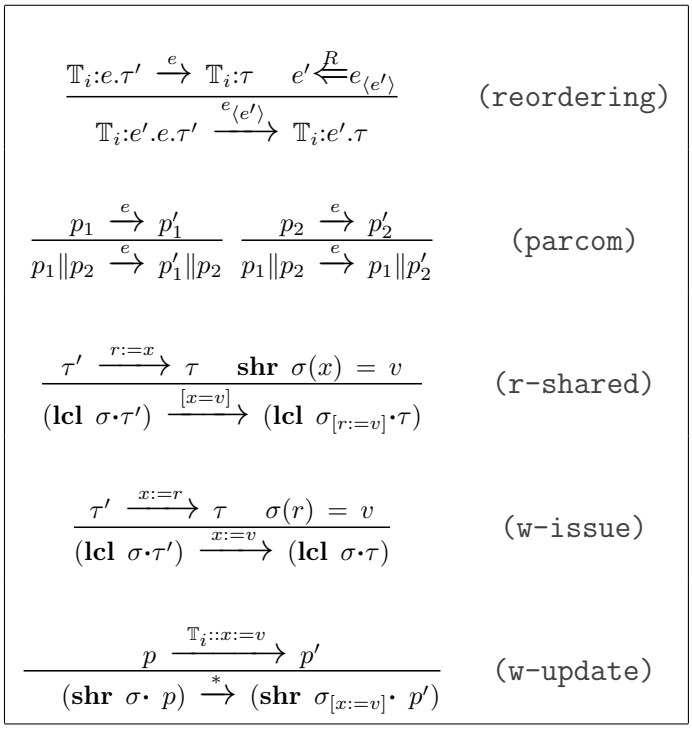


Figure 5.7: Operational semantics of MCA model [31]

to be executed ($\mathbb{T}_i{:}\tau$). The entity *system* is identified with a shared state (**shr** σ) and a parallel composition of processing elements. Thus, a system term is of the form (**shr** $\sigma \cdot (\mathbb{T}_0$ **lcl** $\sigma \cdot \mathbb{T}_0{:}\tau_0) \parallel (\mathbb{T}_1$ **lcl** $\sigma \cdot \mathbb{T}_1{:}\tau_1) \parallel \ldots$).

The operational semantics for a program P under MCA is listed in Figure 5.7. The notation $x := r$ represents the `write` action to an object x with an expression r, where r can be a local object or a constant value. The notation $[x = v]$ represents the `read` action on an object x where v is the value of x in memory. Notation **shr** $\sigma_{[x:=v]}$ represents that the value of the shared object x in the shared memory is v.

The `(reordering)` rule states that if an event $e_{\langle e'\rangle}$ can reorder before another event e' (where $e' <_{\mathbb{T}_i:\tau} e$) then the processing element $\mathbb{T}_i$ can execute $e_{\langle e'\rangle}$ before e' and suitably update the remaining sequence to be executed further.

The rule (parcom) shows the parallel composition of the processing elements. It states that one step of the system is taken by one processing element at a time.

The (r-shared) rule captures the read of a shared object from the shared storage into a local object.

The (w-issue) rule shows that a processing element initiates a write operation of value v to a shared object x, and moves to the next event.

A write initiated by $\mathbb{T}_i$ is updated to the shared storage by the system as shown in rule (w-update).

5.5 Restricted C11 for MCA

This work proposes a dynamic verification technique called MoCA for the precise analysis under C11 (discussed in §3.2.3) restricted for MCA (discussed in §5.4). As a result the MoCA technique analysis only those program outcomes and flags only those bugs that are permitted by the developer specification in a C11 input program and are feasible on an MCA architecture.

The key contribution of MoCA is in restricting the C11 outcomes of an input program to those admissible under an MCA architecture. As discussed in §3.2.3 a C11 outcome is represented as a C11 *trace*, *i.e.* a tuple of $\langle \mathcal{E}_\tau, \text{hb}_\tau, \text{mo}_\tau, \text{rf}_\tau \rangle$, such that the event relations in the trace satisfy a set of *coherence conditions*. MoCA redefines the hb_τ relation to $[m]::\text{hb}_\tau$, the *C11-MCA-happens-before* relation, that restricts the C11 outcomes to those feasible on a single shared memory. Further, MoCA introduces a set of *C11-MCA coherence conditions* such that a trace is coherent under C11 restricted to MCA if it satisfies these conditions.

A central feature of MoCA that helps realize the precise C11-MCA-happens-before relation and C11-MCA coherence conditions is the introduction of a new event type called `shadow-writes` that simulate *reordering through interleaving*.

The above mentioned key features for restriction of C11 to MCA are discussed below.

5.5.1 Shadow-write events

A `write` event when executed may not be immediately available to all threads under multi-copy atomics. To accommodate the visibility of a `write` event to other threads at a later timestamp, MoCA associates each `write` or `rmw` event with a corresponding `shadow-write` event. The `shadow-write` event updates the shared memory with the value of its corresponding `write` or `rmw` event, thus marking its visibility to all threads.

`Shadow-writes` break a `write` operation into two (not necessarily consecutive) events: (i) the `write` event from the program that is visible only to events of the same thread, and (ii) the `shadow-write` event that updates the shared memory with the `write` event's value at a later timestamp; thus, completing the `write` operation and making it visible to all threads. `Shadow-write` events are issued by *shadow-threads* that are spawned by MoCA (separate from the program threads). MoCA maintains a separate shadow-thread per program thread per object.

For an event $e \in \mathcal{E}^{\mathbb{W}}$, $shw(e)$ represents the `shadow-write` event associated with e. Similarly, $prw(e')$ denotes the `write` event corresponding to the `shadow-write` e'. The set of shadow-threads associated with $thr(e)$ is denoted by $sth(e)$. Note that, the `shadow-write` events of a shadow-thread in $sth(e)$ can interleave with the events of the corresponding program thread ($thr(e)$), in other words, a `shadow-write` event can interleave with the events of the thread with the `write` event. Such interleaving allows a `write` event e_w that occurs before an event e in a thread to take effect in memory after e (if its corresponding `shadow-write` ($shw(e_w)$) executes after e). Thus interleaving with `shadow-writes` gives rise to reordering in effect thereby enabling *reordering through interleaving*.

Consider the example in Figure 5.8(a). The `shadow-write` events corresponding to the `write` events labeled a and c are represented by events labeled a' and c' respectively. The shadow-threads $sth_x(\mathbb{T}_1)$ and $sth_x(\mathbb{T}_2)$ execute the `shadow-write`

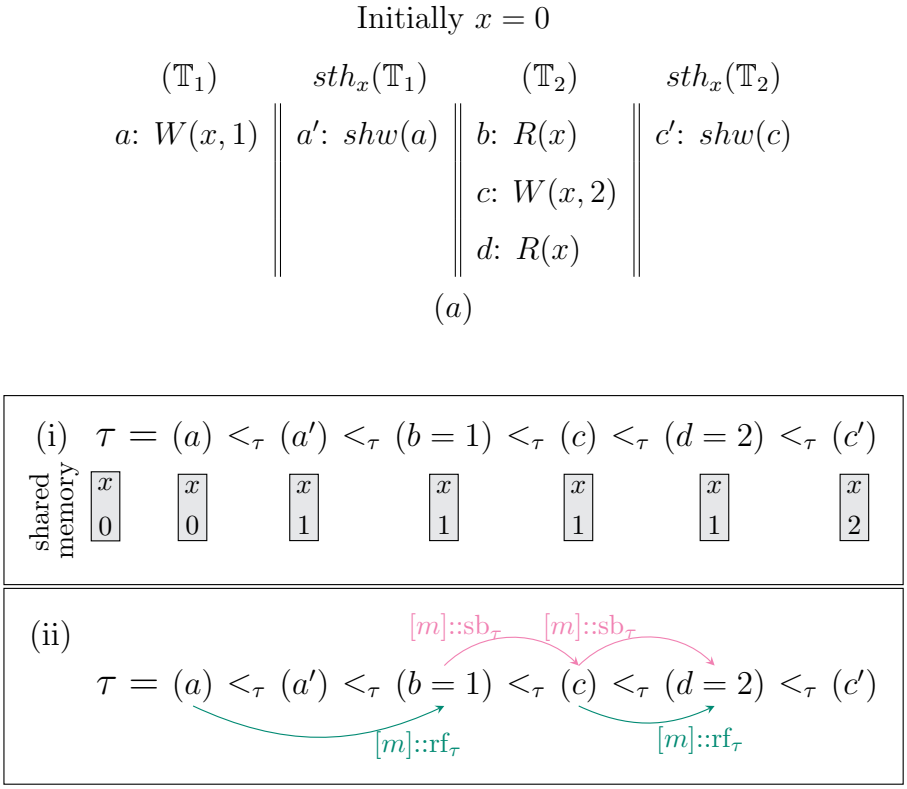


Figure 5.8: Execution with `shadow-writes`

events a' and c' respectively. Figure 5.8(b)-i shows an execution sequence τ where updates to the memory by `shadow-writes` are illustrated. Note that, the values below x in the gray boxes show the value of the shared object x in the shared memory at various stages of execution. The values after '=' against labels of `read` events show the value read by the `read` event (*i.e.* b=1 and d=2). As shown in the Figure 5.8(b)-i the value of x in the shared memory remains the same as the previous value after the execution of `write` events a and c and is updated only when their corresponding `shadow-write` events a' and c' are executed. The values read by the `read` events may be ignored for now and will be discussed after introduction of the *C11-MCA-reads-from* relation, in the following section.

5.5.2 Program executions and traces

A trace under C11 is defined by a tuple $\langle \mathcal{E}_\tau, \text{hb}_\tau, \text{mo}_\tau, \text{rf}_\tau \rangle$ (refer to Definition 2). The event relations hb_τ, mo_τ, and rf_τ on the set of events of the trace ($\mathcal{E}_\tau$) define the outcome or behavior represented by the trace.

Therefore, MoCA redefines the relations formed over the events of a trace, to eventually redefine the set of valid outcomes.

MoCA defines a C11 trace restricted to MCA by a tuple $\langle \mathcal{E}_\tau, [m]::\text{hb}_\tau, [m]::\text{mo}_\tau, [m]::\text{rf}_\tau \rangle$. The components of C11-MCA trace are discussed below.

Program executions and equivalence of executions

A key feature of the MCA model is that a program execution can be represented as a total-order on the events of the input program, where every update to the memory when reflected to the single shared memory is accessible to all threads. The total-order is represented by the occurs-before relation ($<_\tau$).

The outcome of an execution is captured as a trace, *i.e.* a tuple $\langle \mathcal{E}_\tau, [m]::\text{hb}_\tau, [m]::\text{mo}_\tau, [m]::\text{rf}_\tau \rangle$ representing the set of events of the the execution sequence and the set of relations formed on the events by MoCA.

The set of executions or non-maximal sequences that represent the same outcome are called *equivalent*. Formally two event sequences τ_1 and τ_2 are equivalent, represented as $\tau_1 \approx \tau_2$ if they represent the same trace, that is,

- $\mathcal{E}_{\tau_1} = \mathcal{E}_{\tau_2}$,
- $[m]::\text{hb}_{\tau_1} = [m]::\text{hb}_{\tau_2}$,
- $[m]::\text{mo}_{\tau_1} = [m]::\text{mo}_{\tau_2}$, and
- $[m]::\text{rf}_{\tau_1} = [m]::\text{rf}_{\tau_2}$.

MoCA event categories

For ease of presentation, the memory access events of $\mathcal{E}$ are categorized as: (i) the set of `writes` ($\mathcal{E}^{\mathbb{W}}$) that issue a write (*i.e.*, `write` events and `rmws`), (ii) the set of `modifiers` ($\mathcal{E}^{\mathbb{M}}$) that update the shared memory for an issued `write` (*i.e.* `shadow-write` events), and (iii) the set of `reads` ($\mathcal{E}^{\mathbb{R}}$) (*i.e.* `read` events and `rmws`). Accordingly, $\mathcal{E}_\tau^{\mathbb{W}}$, $\mathcal{E}_\tau^{\mathbb{M}}$, and $\mathcal{E}_\tau^{\mathbb{R}}$ represent the events of a sequence τ from the respective event categories; and, $\mathcal{E}_\tau^{\mathbb{W}(m)}$, $\mathcal{E}_\tau^{\mathbb{M}(m)}$, and $\mathcal{E}_\tau^{\mathbb{R}(m)}$ represent the events with memory order m of a sequence τ from the respective event categories.

Reads-from relation for C11 restricted to MCA ($[m]::\mathbf{rf}_\tau$)

Apart from `shadow-writes`, a second central contribution to realize the restriction of C11 for MCA, is a *C11-MCA-reads-from* relation ($[m]::\mathrm{rf}_\tau$) based on the `shadow-write` events. As a consequence of representing an execution sequence as a total-order, consider the notation $[m]::\mathsf{last}W_{[\tau]}(e_r)$ that represents the `write` corresponding to the latest `shadow-write` of $obj(e_r)$ in the prefix of τ upto e_r. The reads-from relation for MoCA is defined as,

Definition 5. **($[m]::\mathbf{rf}_\tau$ relation)**

Given $e_r \in \mathcal{E}_\tau^{\mathbb{R}}$, the $[m]::\mathrm{rf}_\tau$ relation for e_r is defined as:

(*general case*) $[m]::\mathsf{last}W_{[\tau]}(e_r) \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$; unless

(*special case*) $\exists e_w \in \mathcal{E}_\tau^{\mathbb{W}}$, which is the latest `write` of $obj(e_r)$ from $thr(e_r)$ s.t. $shw([m]::\mathsf{last}W_{[\tau]}(e_r)) <_\tau e_w <_\tau e_r$ then $e_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$.

Intuitively, a `read` event takes its value from the last `write` whose `shadow-write` updated the shared memory, unless there is a later `write` from the same thread. Consider the execution sequence τ in Figure 5.8(b)-ii corresponding to the input program in Figure 5.8(a). The event b reads from a (since $\mathsf{last}W_{[a.a']}(b) = a$); however,

for event d, $\mathsf{last}W_{[a.a'.b.c]}(d) = a$ but `write` c from same thread occurs after a', thus, d reads from c.

Note that, by ensuring read from the `write` of the same thread whose `shadow-write` has not executed, as defined under (*special case*), MoCA replicates the effects of forwarding (introduced in §5.4) through the $[m]$::rf_τ relation. In the input program shown in Figure 5.8(a), $c <_{\mathbb{T}_2} d$, however in the execution τ in Figure 5.8(b), $c <_\tau d <_\tau c'$, where $c' = shw(c)$. By the special case of $[m]$::rf_τ relation, `read` d reads the value 2 from the `write` c while the value of the `write` takes effect in memory later. The occurrence can effectively be interpreted as the `write` c reordering after the read d while the effect of the `write` is forwarded to d for reordering.

Modification-order relation for C11 restricted to MCA ($[m]$::mo_τ)

MoCA defines a total-order on the `write` events of an object in an execution sequence τ, called the *C11-MCA-modification-order* ($[m]$::mo_τ). However, unlike the mo_τ relation of C11, the $[m]$::mo_τ relation of MoCA is based on the order of occurrence of the corresponding shadow-writes.

Two `writes` e_{w1}, e_{w2} of the same shared object are modification-ordered if $shw(e_{w1})$ occurs before $shw(e_{w2})$ in an execution sequence, formally,

Definition 6. **($[m]$::mo_τ relation)**

$$\forall\ e_1, e_2 \in \mathcal{E}_\tau^{\mathbb{M}},\ obj(e_1) = obj(e_2),\ e_1 <_\tau e_2 \Rightarrow prw(e_1) \rightarrow_\tau^{[m]::\mathrm{mo}} prw(e_2)$$

Happens-before relation for C11 restricted to MCA

Based on `shadow-writes` and the hence defined $[m]$::rf_τ relation, MoCA defines a *C11-MCA-happens-before* relation for the restriction of C11 to MCA, represented as $[m]$::hb_τ. The $[m]$::hb_τ relation is composed of the following relations. The relations are formally defined in Figure 5.9.

$[m]$::sb_τ (*Sequenced-before*): total occurrence order on the events of a thread[4].

$[m]$::sw_τ (*Synchronizes-with*) Inter-thread synchronization between a write e_w (ordered $\sqsupseteq$ `rel`) and a read e_r (ordered $\sqsupseteq$ `acq`) when $e_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$.

$[m]$::dob_τ (*Dependency-ordered-before*): Inter-thread synchronization between a write e_w (ordered $\sqsupseteq$ `rel`) and a read e_r (ordered $\sqsupseteq$ `acq`) when $e'_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$ for $e'_w \in$ *release-sequence*[5] of e_w in τ.

$[m]$::ithb_τ (*Inter-thread-hb*): Inter-thread relation computed by extending $[m]$::sw_τ and $[m]$::dob_τ with $[m]$::sb_τ.

$[m]$::hb_τ (*Happens-before*): Inter-thread relation defined as $[m]$::$\mathrm{sb}_\tau \cup [m]$::$\mathrm{ithb}_\tau$.

$[m]$::hb_τ *relation on C11* `fences`

C11 `fences` form $[m]$::ithb_τ relation with other events of the trace τ. Appropriately placed `fences` can form $[m]$::sw_τ and $[m]$::dob_τ relations from an $[m]$::rf_τ relation between events of different threads. The conditions for forming $[m]$::sw_τ and $[m]$::dob_τ relations with `fences` are formally presented in Figure 5.10.

Figure 5.10 also represents the conditions diagrammatically; where the depictions (a)-(b) represent the conditions in the absence of `fences` (presented in Figure 5.9); the depiction (c) represents the condition for forming $[m]$::dob_τ relation with `fences`; and, the depictions (d)-(f) represent the conditions for forming $[m]$::sw_τ relation with `fences`. The relation $e_w \xrightarrow{rs} e'_w$ implies that e'_w belong to the *release sequence* of e_w.

Intuitively, a `write` event with a `fence` placed before it in $[m]$::sb_τ and a `read` event with a `fence` placed after it in $[m]$::sb_τ enact the role of strongly ordered `write` and strongly ordered `read` events respectively.

Theorem 6. $[m]$::hb_τ for a trace τ represents a valid happens-before relation.

[4] Some events of a thread may not be ordered (for example, the operands of ==). Relation $[m]$::sb_τ assumes a total order on the events of a thread, similar to previous works on C11 [72, 51].

[5] *release-sequence* of e_w in τ: maximal contiguous sub-sequence of $[m]$::mo_τ that starts at e_w and contains: (i) write events of $thr(e_w)$, (ii) rmw events of other threads [20, 46].

$$e_1 \rightarrow_\tau^{[m]::\mathbf{sb}} e_2 \quad \Leftarrow \quad \forall e_1, e_2 \in \mathcal{E}_\tau \text{ s.t. } thr(e_1) = thr(e_2) \text{ and } e_1 \text{ occurs-before } e_2 \text{ in their thread.}$$

$$e_w \rightarrow_\tau^{[m]::\mathbf{sw}} e_r \quad \Leftarrow \quad e_w \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \mathtt{rel})},\ e_r \in \mathcal{E}_\tau^{\mathbb{R}(\sqsupseteq \mathtt{acq})},\ thr(e) \neq thr(e_1) \wedge e_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r.$$

$$e_w \rightarrow_\tau^{[m]::\mathbf{dob}} e_r \quad \Leftarrow \quad e_w \in \mathcal{E}_\tau^{\mathbb{W}(\sqsupseteq \mathtt{rel})},\ e_r \in \mathcal{E}_\tau^{\mathbb{R}(\sqsupseteq \mathtt{acq})},\ thr(e) \neq thr(e_1) \wedge \exists e'_w \in \textit{release-sequence} \text{ of } e_w \text{ s.t. } e'_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r.$$

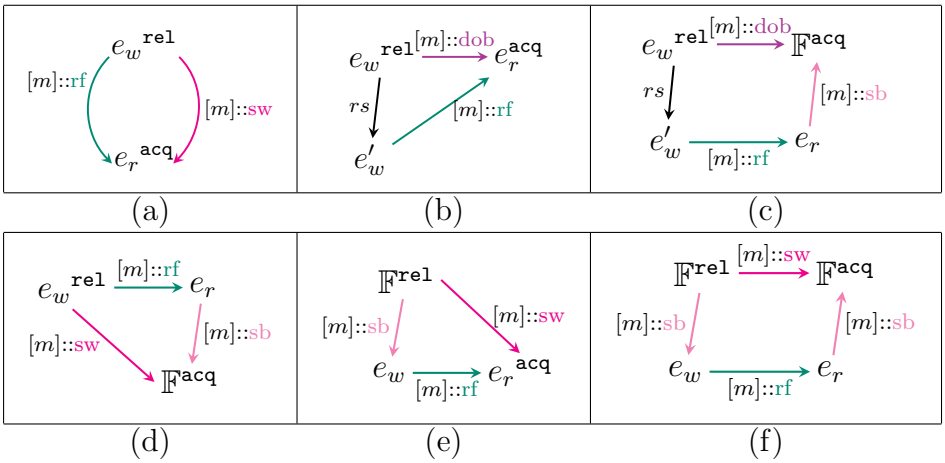
$$\begin{aligned} e_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} e_2 \quad \Leftarrow \quad & \forall e_1, e_2, e_3 \in \mathcal{E}_\tau, \\ & (e_1 \rightarrow_\tau^{[m]::\mathbf{sw}} e_2) \vee \\ & (e_1 \rightarrow_\tau^{[m]::\mathbf{dob}} e_2) \vee \\ & (e_1 \rightarrow_\tau^{[m]::\mathbf{sw}} e_3 \wedge e_3 \rightarrow_\tau^{[m]::\mathbf{sb}} e_2) \vee \\ & (e_1 \rightarrow_\tau^{[m]::\mathbf{sb}} e_3 \wedge e_3 \rightarrow_\tau^{[m]::\mathbf{ithb}} e_2) \vee \\ & (e_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} e_3 \wedge e_3 \rightarrow_\tau^{[m]::\mathbf{ithb}} e_2). \end{aligned}$$

$$e_1 \rightarrow_\tau^{[m]::\mathbf{hb}} e_2 \quad \Leftarrow \quad e_1 \rightarrow_\tau^{[m]::\mathbf{sb}} e_2 \vee e_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} e_2.$$

Figure 5.9: Happen-before relation for C11-MCA model ($[m]::\mathrm{hb}_\tau$)

A happens-before assignment is valid if $\forall$ execution sequences E, hb_E satisfies the following properties [1].

1. hb_E is a partial order on $\mathcal{E}_E$, which is included in $<_E$.
2. Events of each processing element are totally ordered, *i.e.* $\langle \mathbb{T}_t, i, a', o', m', l' \rangle$ hb_E $\langle \mathbb{T}_t, i+1, a, o, m, l \rangle$, whenever $\langle \mathbb{T}_t, i+1, a, o, m, l \rangle \in \mathcal{E}_E$.
3. If E' is a prefix of E, then $\mathrm{hb}_{E'}$ and hb_E are the same on $\mathcal{E}_{E'}$
4. Any linearization E' of $\mathrm{hb}_E \cup \mathrm{rf}_E$ is an execution sequence s.t. $\mathrm{hb}_{E'} \cup \mathrm{rf}_{E'} = \mathrm{hb}_E \cup \mathrm{rf}_E$ and E' and E are equivalent
5. If E' is equivalent to E then $s_{[E']} = s_{[E]}$.
6. If $E.E_1$ is an execution sequence, then E' is equivalent to $E \Leftrightarrow E'.E_1$ is equivalent to $E.E_1$



$[m]::\mathrm{sw}_\tau$ using C11 `fences`

Given $e_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$,

if $ord(e_w) \sqsupseteq \mathtt{rel}$, $\exists \mathbb{F} \in^{\mathtt{acq}} \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \mathtt{acq})}$ s.t. $e_r \rightarrow_\tau^{[m]::\mathrm{sb}} \mathbb{F}^{\mathtt{acq}}$ then $e_w \rightarrow_\tau^{[m]::\mathbf{sw}} \mathbb{F}^{\mathtt{acq}}$;

if $ord(e_r) \sqsupseteq \mathtt{acq}$, $\exists \mathbb{F}^{\mathtt{rel}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \mathtt{rel})}$ s.t. $\mathbb{F}^{\mathtt{rel}} \rightarrow_\tau^{[m]::\mathrm{sb}} e_w$ then $\mathbb{F}^{\mathtt{rel}} \rightarrow_\tau^{[m]::\mathbf{sw}} e_r$;

if $\exists \mathbb{F}^{\mathtt{rel}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \mathtt{rel})}$, $\exists \mathbb{F}^{\mathtt{acq}} \in \mathcal{E}_\tau^{\mathbb{F}(\sqsupseteq \mathtt{acq})}$ s.t. $\mathbb{F}^{\mathtt{rel}} \rightarrow_\tau^{[m]::\mathrm{sb}} e_w$, $e_r \rightarrow_\tau^{[m]::\mathrm{sb}} \mathbb{F}^{\mathtt{acq}}$ then $\mathbb{F}^{\mathtt{rel}} \rightarrow_\tau^{[m]::\mathbf{sw}} \mathbb{F}^{\mathtt{acq}}$.

$[m]::\mathrm{dob}_\tau$ using C11 `fences`

Given $e'_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$, if $\exists e_w \in \mathcal{E}_\tau^{\mathbb{W}(\mathtt{rel})}$ s.t. $e'_w \in$ *release-sequence* of e_w and $\exists \mathbb{F}^{\mathtt{acq}} \in \mathcal{E}_\tau^{\mathbb{F}(\mathtt{acq})}$ s.t. $e_r \rightarrow_\tau^{[m]::\mathrm{sb}} \mathbb{F}^{\mathtt{acq}}$ then $e_w \rightarrow_\tau^{[m]::\mathbf{dob}} \mathbb{F}^{\mathtt{acq}}$.

Figure 5.10: $[m]::\mathrm{sw}_\tau$ and $[m]::\mathrm{dob}_\tau$ using C11 `fences`

7. If $e_1 \rightarrow^{\mathbf{hb}}_{E.e_1.e_2} e_2$ and $e_1 \not\rightarrow^{\mathbf{hb}}_{E.e_1.e_3} e_3$ then $e_1 \rightarrow^{\mathbf{hb}}_{E.e_1.e_3.e_2} e_2$.

Proof. Consider a trace τ.

Property 1,3 follow directly from construction of $[m]::\mathrm{hb}_\tau$.

Property 2 follows from the definition of $[m]::\mathrm{sb}_\tau$.

For property 4, assume $\exists \tau_1$ s.t. $\tau_1 \neq \tau$ then $\exists e', e$ s.t. $e \rightarrow_{\tau_1}^{[m]::\mathbf{hb}} e'$ but $e \not\rightarrow_\tau^{[m]::\mathbf{hb}} e' \Rightarrow [m]::\mathrm{hb}_{\tau_1} \neq [m]::\mathrm{hb}_\tau$. Thus, Property 4 follows from construction of $[m]::\mathrm{hb}_\tau$.

Property 5 is satisfied as follows: $\tau' \approx \tau \Rightarrow [m]::\text{hb}_{\tau'} \cup [m]::\text{rf}_{\tau'} = [m]::\text{hb}_{\tau} \cup [m]::\text{rf}_{\tau}$, $\mathcal{E}'_{\tau} = \mathcal{E}_{\tau} \Rightarrow \forall e \in \mathcal{E}_{\tau}, val_{[\tau']}(e) = val_{[\tau]}(e). \Rightarrow s_{[\tau']} = s_{[\tau]}$.

Property 6 is satisfied as follows: $\tau' \approx \tau \Rightarrow [m]::\text{hb}_{\tau'} \cup [m]::\text{rf}_{\tau'} = [m]::\text{hb}_{\tau} \cup [m]::\text{rf}_{\tau}$ $\Rightarrow [m]::\text{hb}_{\tau'} \cup [m]::\text{rf}_{\tau'} \cup [m]::\text{hb}_{\tau_1} \cup [m]::\text{rf}_{\tau_1} = [m]::\text{hb}_{\tau} \cup [m]::\text{rf}_{\tau} \cup [m]::\text{hb}_{\tau_1} \cup [m]::\text{rf}_{\tau_1}$ $\Rightarrow \tau'.\tau_1 \approx \tau.\tau_1$

And similarly, $\tau'.\tau_1 = \tau.\tau_1 \Rightarrow [m]::\text{hb}_{\tau'} \cup [m]::\text{rf}_{\tau'} \cup [m]::\text{hb}_{\tau_1} \cup [m]::\text{rf}_{\tau_1} = [m]::\text{hb}_{\tau} \cup [m]::\text{rf}_{\tau} \cup [m]::\text{hb}_{\tau_1} \cup [m]::\text{rf}_{\tau_1} \Rightarrow [m]::\text{hb}_{\tau'} \cup [m]::\text{rf}_{\tau'} = [m]::\text{hb}_{\tau} \cup [m]::\text{rf}_{\tau} \Rightarrow \tau' \approx \tau$.

For property 7 consider two sets of relations.

$[m]::\text{ithb-r}_{\tau}$ (*ithb-reversible*): $e' \rightarrow_{\tau}^{[m]::\textbf{sw}} e \vee e' \rightarrow_{\tau}^{[m]::\textbf{dob}} e \Rightarrow e' \rightarrow_{\tau}^{[m]::\textbf{ithb-r}} e$

$[m]::\text{ithb-i}_{\tau}$ (*ithb-irreversible*): $e' \rightarrow_{\tau}^{[m]::\textbf{ithb}} e \wedge \neg e' \rightarrow_{\tau}^{[m]::\textbf{ithb-r}} e \Rightarrow e' \rightarrow_{\tau}^{[m]::\textbf{ithb-i}} e$

Finally, property 7 is satisfied as follows: $e_1 \rightarrow_{\tau.e_1.e_2}^{[m]::\textbf{hb}} e_2 \Rightarrow e_1 \rightarrow_{\tau.e_1.e_2}^{[m]::\textbf{sb}} e_2 \vee e_1 \rightarrow_{\tau.e_1.e_2}^{[m]::\textbf{ithb-r}} e_2$, if $e_1 \rightarrow_{\tau.e_1.e_2}^{[m]::\textbf{sb}} e_2$ then $e_1 \rightarrow_{\tau.e_1.e_3.e_2}^{[m]::\textbf{sb}} e_2$ (by definition of $[m]::\text{sb}_{\tau}$), and

if $e_1 \rightarrow_{\tau.e_1.e_2}^{[m]::\textbf{ithb-r}} e_2$ then $e_1 \rightarrow_{\tau.e_1.e_3.e_2}^{[m]::\textbf{ithb-r}} e_2$ (as $e_1 \nrightarrow_{\tau.e_1.e_3}^{[m]::\textbf{hb}} e_3$).

Note that, $e_1 \nrightarrow_{\tau.e_1.e_2}^{[m]::\textbf{ithb-i}} e_2$ because they are adjacent. □

Total-order on sc events for C11 restricted to MCA

Further, MoCA also maintains a total order relation $[m]::\text{to}_{\tau}$ on sc events. The $[m]::\text{to}_{\tau}$ relation is computed on the events of τ, as follows.

Definition 7. (sc total-order ($[m]::\text{to}_{\tau}$))

Let shw-to$_{\tau} \triangleq \{(e', e) \mid e', e \in \mathcal{E}_{\tau}^{\mathbb{R}(\textbf{sc})} \cup \mathcal{E}_{\tau}^{\mathbb{M}(\textbf{sc})} \cup \mathcal{E}_{\tau}^{\mathbb{F}(\textbf{sc})}, thr(e') \neq thr(e) \wedge e' <_{\tau} e\}$; then,
$[m]::\text{to}_{\tau} \triangleq \{(e', e) \mid e', e \in \mathcal{E}_{\tau}^{\mathbb{R}(\textbf{sc})} \cup \mathcal{E}_{\tau}^{\mathbb{W}(\textbf{sc})} \cup \mathcal{E}_{\tau}^{\mathbb{F}(\textbf{sc})}$ s.t. either $e' \rightarrow_{\tau}^{[m]::\textbf{sb}} e$, or $(\exists (e'_1, e_1) \in$ shw-to$_{\tau}$ s.t. $(e'_1 = e' \vee e'_1 = shw(e')) \wedge (e_1 = e \vee e_1 = shw(e)))\}$

Intuitively,

1. all `sc` events in $[m]$::sb_τ are also in $[m]$::to_τ relation, and
2. `sc` ordered events from different threads are in $[m]$::to_τ by their occurrence order in τ, except `write` events that are in $[m]$::to_τ by the occurrence order of their `shadow-write` event in τ.

Non-atomic data races

If a data race exists on `na` ordered events of an execution sequence τ (a.k.a `na` race) then the behavior of τ is considered *undefined* under the C11 model [20, 46, 62].

MoCA defines an `na` race in a sequence τ as:

Definition 8. (`na` race)

$$\text{if } \exists e', e \in \mathcal{E}_\tau^{(\texttt{na})} \text{ s.t. } e' <_\tau e \text{ but } e' \not\rightarrow_\tau^{[m]::\textbf{hb}} e \text{ then } \tau \text{ has an } \texttt{na} \text{ race on } e', e.$$

5.5.3 Trace coherence under C11 for MCA

The $[m]$::hb_τ relation, along with the $[m]$::mo_τ and $[m]$::rf_τ relations, is used in specifying a set of *C11-MCA coherence conditions*. The coherence conditions bound the set of traces feasible with `shadow-writes` and the $[m]$::hb_τ, $[m]$::mo_τ, and $[m]$::rf_τ relations to those valid under C11 and MCA.

C11-MCA coherence conditions

(mca-moWE): If a `write` e_w is $[m]$::ithb_τ or $[m]$::rf_τ; $[m]$::ithb_τ ordered with an event e from another thread, then either $shw(e_w)$ must occur before $shw(e)$ (if $e \in \mathcal{E}^{\mathbb{W}}$) or before e (if $e \notin \mathcal{E}^{\mathbb{W}}$).

(mca-moRR): If a `read` e_{r1} is hb-ordered before another read e_{r2}, then the `shadow-write` of e_{r1}'s source `write` must occur before the `shadow-write` of e_{r2}'s source `write` (where, source `write` is the `write` event that a `read` reads from);

(mca-moWR): `shadow-write` of a `read`'s source must occur after the `shadow-write` of all `writes` hb-ordered before the `read`.

(mca-cormw): To ensures atomicity, each `rmw` event must read-from the *immediately* ordered before event in the modification order.

(mca-co): A `read` must not read from a `write` hb-ordered after it.

Finally, through Theorem 7 demonstrates that traces generated by MoCA are indeed coherent under C11 by establishing that every C11 coherence condition is satisfied under MoCA.

Theorem 7. MoCA coherence $\Rightarrow$ C11 coherence.

Proof. Let $\twoheadrightarrow_\tau$ represent a relation *non-racing-hb* that represents hb-ordered events that do not race to access an object, that is, $e_1 \rightarrow_\tau^{[m]::\mathbf{hb}} e_2 \wedge \neg\,(e_1 \rightarrow_\tau^{[m]::\mathbf{sw}} e_2 \vee e_1 \rightarrow_\tau^{[m]::\mathbf{dob}} e_2) \Rightarrow e_1 \twoheadrightarrow_\tau e_2$.

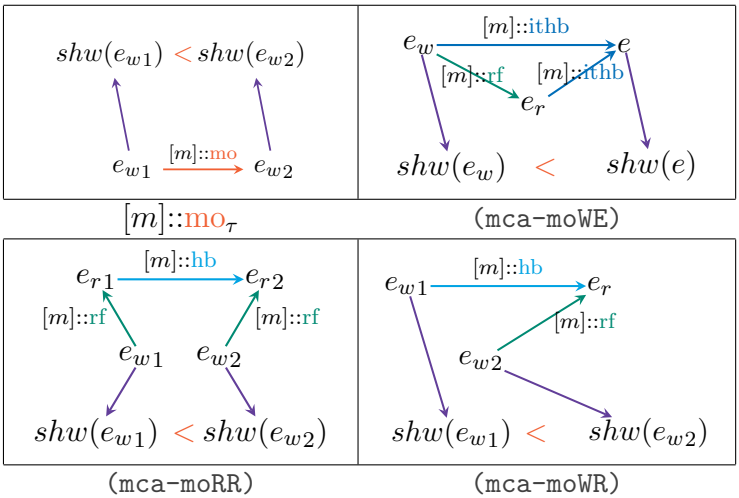
Recall, (coWW): $\forall w_1, w_2 \in \mathcal{E}_\tau^{\mathbb{W}},\ w_1 \rightarrow_\tau^{\mathbf{hb}} w_2 \Rightarrow w_1 \rightarrow_\tau^{\mathbf{mo}} w_2$.

Now, $w_1 \rightarrow_\tau^{[m]::\mathbf{hb}} w_2 \Rightarrow w_1 \rightarrow_\tau^{[m]::\mathbf{sb}} w_2 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} w_2$ but $\neg(w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} w_2 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} w_2)$ (as both events are `write` events).

If $w_1 \rightarrow_\tau^{[m]::\mathbf{sb}} w_2$ and $thr(shw(w_1)) = \mathbb{T}_i$ then $thr(shw(w_2)) = thr(shw(w_1)) = \mathbb{T}_i \wedge shw(w_1) <_{\mathbb{T}_i} shw(w_2)$.

Whereas, if $w_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} w_2$ then $w_1 \twoheadrightarrow_\tau w_2$ (since $\neg(w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} w_2 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} w_2) \Rightarrow shw(w_1) <_\tau shw(w_2)$ (by (mca-moWE))

Each of the two cases $\Rightarrow w_1 \rightarrow_\tau^{[m]::\mathbf{mo}} w_2$ (using (mca-mo)). inf(1).



(mca-co)	$\forall\ e_r \in \mathcal{E}_\tau^{\mathbb{R}},\ e_w \rightarrow_\tau^{[m]::\mathbf{rf}} e_r \Rightarrow e_r \nrightarrow_\tau^{[m]::\mathbf{hb}} e_w$
(mca-moWE)	$\forall\ e_w \in \mathcal{E}_\tau^{\mathbb{W}},\ e_r \in \mathcal{E}_\tau^{\mathbb{R}},\ e \in \mathcal{E}_\tau$, s.t. $thr(e) \neq thr(e_w)$, $e_w \rightarrow_\tau^{[m]::\mathbf{ithb}} e \vee e_w \rightarrow_\tau^{\mathbf{rf}} e_r \rightarrow_\tau^{[m]::\mathbf{ithb}} e \Rightarrow shw(e_w) <_\tau e$ (if $e \notin \mathcal{E}_\tau^{\mathbb{W}}$) $\wedge$ $shw(e_w) <_\tau shw(e)$ (if $e \in \mathcal{E}_\tau^{\mathbb{W}}$)
(mca-moRR)	$\forall\ e_{r1},\ e_{r2} \in \mathcal{E}_\tau^{\mathbb{R}}$ s.t. $e_{r1} \rightarrow_\tau^{[m]::\mathbf{hb}} e_{r2}$ if $\exists\ e_{w1}, e_{w2} \in \mathcal{E}_\tau^{\mathbb{W}}$ s.t. $e_{w1} \rightarrow_\tau^{[m]::\mathbf{rf}} e_{r1} \wedge e_{w2} \rightarrow_\tau^{[m]::\mathbf{rf}} e_{r2}$ where $e_{w1} \neq e_{w2}$ then $shw(e_{w1}) <_\tau shw(e_{w2})$
(mca-moWR)	$\forall\ e_{w1}, e_{w2} \in \mathcal{E}_\tau^{\mathbb{W}},\ e_r \in \mathcal{E}_\tau^{\mathbb{R}}$ s.t. $e_{w1} \rightarrow_\tau^{[m]::\mathbf{hb}} e_r \wedge e_{w2} \rightarrow_\tau^{[m]::\mathbf{rf}} e_r$ where $e_{w1} \neq e_{w2}$, $shw(e_{w1}) <_\tau shw(e_{w2})$
(mca-cormw)	$\forall\ e \in \mathcal{E}_\tau,\ act(e) = \mathtt{rmw},\ \exists\ e_w \rightarrow_\tau^{[m]::\mathbf{rf}} e$ s.t. $e_w \rightarrow_\tau^{[m]::\mathbf{mo}} e \wedge \nexists\ e'_w$ s.t. $e_w \rightarrow_\tau^{[m]::\mathbf{mo}} e'_w \rightarrow_\tau^{[m]::\mathbf{mo}} e$

Figure 5.11: Coherence conditions for MoCA traces

Recall, (coRR): $\forall\ r_1, r_2 \in \mathcal{E}_\tau^{\mathbb{R}},\ w_1 \in \mathcal{E}_\tau^{\mathbb{W}},\ r_1 \rightarrow_\tau^{\mathbf{hb}} r_2 \wedge w_1 \rightarrow_\tau^{\mathbf{rf}} r_1 \Rightarrow w_1 \rightarrow_\tau^{\mathbf{rf}} r_2 \vee (\exists\ w_2,\ w_2 \rightarrow_\tau^{\mathbf{rf}} r_2 \wedge w_1 \rightarrow_\tau^{\mathbf{mo}} w_2)$.

Now, $r_1 \rightarrow_\tau^{[m]::\mathbf{hb}} r_2 \wedge w_1 \neq w_2 \Rightarrow w_1 \rightarrow_\tau^{[m]::\mathbf{mo}} w_2$ (using rule (mca-moRR), (mca-mo)). inf(2).

Recall, (coRW): $\forall\ r_1 \in \mathcal{E}_\tau^{\mathbb{R}},\ w_1 \in \mathcal{E}_\tau^{\mathbb{W}},\ r_1 \rightarrow_\tau^{\mathbf{hb}} w_1 \Rightarrow \exists\ w_2 \rightarrow_\tau^{\mathbf{mo}} w_1$ s.t. $w_2 \rightarrow_\tau^{\mathbf{rf}} r_1$.

Now, if $r_1 \rightarrow_\tau^{[m]::\mathbf{hb}} w_1$, $\exists w_2$, $w_2 \rightarrow_\tau^{[m]::\mathbf{rf}} r_1$ (by definition of (mca-co))

then by definition of $[m]::\mathrm{rf}_\tau$, either $shw(w_2) <_\tau r_1 \Rightarrow shw(w_2) <_\tau r_1 <_\tau shw(w_1)$

or $thr(w_2) = thr(r_1) \Rightarrow w_2 \twoheadrightarrow_\tau r_1 \twoheadrightarrow_\tau w_1$ (as $r_1 \rightarrow_\tau^{[m]::\mathbf{hb}} w_1 \Rightarrow r_1 \rightarrow_\tau^{[m]::\mathbf{sb}} w_1 \vee r_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} w_1$ but $\neg\ (w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} w_2 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} w_2)) \Rightarrow shw(w_2) <_\tau shw(w_1)$ (by `(mca-moWE)`).

Each of the two cases $\Rightarrow w_2 \rightarrow_\tau^{\mathbf{mo}} w_1$ (using `(mca-mo)`). inf(3).

Recall, (`moWR`): $\forall\ w_1 \in \mathcal{E}_\tau^{\mathbb{W}},\ r_1 \in \mathcal{E}_\tau^{\mathbb{R}},\ w_1 \rightarrow_\tau^{\mathbf{hb}} r_1 \Rightarrow w_1 \rightarrow_\tau^{\mathbf{rf}} r_1 \vee (\exists\ w_2,\ w_2 \rightarrow_\tau^{\mathbf{rf}} r_1 \wedge w_1 \rightarrow_\tau^{\mathbf{mo}} w_2)$.

Now, $w_1 \rightarrow_\tau^{[m]::\mathbf{hb}} r_1 \Rightarrow w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} r_1 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} r_1 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{sb}} r_1 \vee (w_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} r_1$ but $\neg\ (w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} w_2 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} w_2))$ (by definition of $[m]::\mathrm{hb}_\tau$).

$w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} r_1 \Rightarrow w_1 = w_2$;

$w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} r_1 \Rightarrow w_2$ is in *release-sequence* of $w_1 \Rightarrow w_1 \rightarrow_\tau^{[m]::\mathbf{mo}} w_2$ (by definition of *release-sequence*);

$w_1 \rightarrow_\tau^{[m]::\mathbf{sb}} r_1 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{ithb}} r_1 \Rightarrow w_1 \twoheadrightarrow_\tau r_1$ (since $\neg\ (w_1 \rightarrow_\tau^{[m]::\mathbf{sw}} w_2 \vee w_1 \rightarrow_\tau^{[m]::\mathbf{dob}} w_2)$) $\Rightarrow shw(w_1) <_\tau r_1$ (using `(mca-moWE)`) then $w_2 \rightarrow_\tau^{[m]::\mathbf{rf}} r_1 \Rightarrow shw(w_1) <_\tau shw(w2) \Rightarrow w_1 \rightarrow_\tau^{[m]::\mathbf{mo}} w_2$ (using `(mca-mo)`).

$\Rightarrow (w_1 = w_2$ *i.e.* $w_1 \rightarrow_\tau^{\mathbf{rf}} r_1) \vee w_1 \rightarrow_\tau^{\mathbf{mo}} w_2$. inf(4).

Recall, (`coto`): $\forall sc_1, sc_2 \in \mathcal{E}_\tau^{\mathtt{sc}}$, $\mathrm{hb}_\tau|_{\mathtt{sc}} \cup \mathrm{mo}_\tau|_{\mathtt{sc}} \subseteq \mathrm{to}_\tau$

Now, $sc_1 \rightarrow_\tau^{[m]::\mathbf{to}} sc_2 \Rightarrow sc_1 <_\tau sc_2 \vee shw(sc_1) <_\tau sc_2 \vee sc_1 <_\tau shw(sc_2) \vee shw(sc_1) <_\tau shw(sc_2) \Rightarrow sc_2 \nrightarrow_\tau^{[m]::\mathbf{hb}} sc_1 \wedge sc_2 \nrightarrow_\tau^{[m]::\mathbf{mo}} sc_1$ (by definition of $[m]::\mathrm{to}_\tau$). inf(5).

Note that `(rfto1)`, `(rfto2)`, and `(tofen)` trivially hold by the definition of $[m]::\mathrm{rf}_\tau$ and $[m]::\mathrm{to}_\tau$. inf(6).

Further, `(corf)` and `(cormw)` trivially holds from `(mca-co)` and `(mca-cormw)` respectively. inf(7).

Inferences (1) to (7) $\Rightarrow$ all valid MoCA traces are valid C11 traces. Implying, that our technique does not explore a non-C11 behavior. □

5.6 Stateless model checking for C11 restricted to MCA

MoCA technique uses source-DPOR (refer to [1]) a state-of-the-art stateless model checking technique to explore the MCA outcomes of a C11 input program for detecting safety property (assert) violations and `na` races. MoCA bases the source-DPOR model checking technique on the event relations, $[m]::\mathrm{hb}_\tau$, $[m]::\mathrm{mo}_\tau$ and $[m]::\mathrm{rf}_\tau$, defined for C11 restricted to MCA (presented in §5.5), along with a set of relevant coherence conditions (refer to Figure 5.11).

The MoCA technique is,

1. *Coherent.* Each trace represented by the C11-MCA event relations, under the C11-MCA-coherence conditions, is a valid C11 outcome.

2. *Precise.* MoCA traces are equivalent to C11 traces feasible over MCA.

3. *Sound.* For each trace of C11 valid over MCA there exists a program execution explored by the MoCA technique[6].

The `shadow-writes`, as remarked before, enable reordering through interleaving. Note, however, that a `shadow-write` updates the memory for the corresponding `write` event at non-deterministic later timestamp. As a consequence, `shadow-writes` simulate the reordering of program `writes` with *later* events from the same thread.

To simulate the reordering of a `write` event with an *earlier* program event from the same thread, MoCA implicitly assumes that the `writes` are at the earliest feasible location in the program (where earlier refers to a lower event index).

To meet this requirement, MoCA performs a static *early-write transformation* from input program P to $\widehat{P}$.

[6]Since, the source-DPOR technique is sound [1], MoCA is also sound.

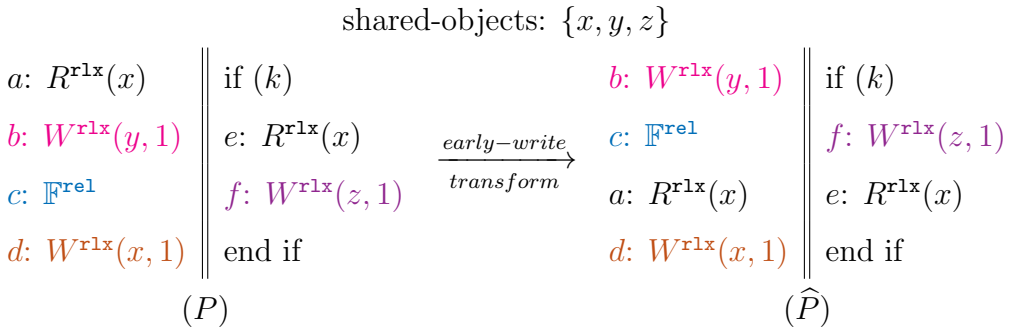


Figure 5.12: Early-write transformation

5.6.1 Early-write transformation

The transformation rules for each thread sequence $\mathbb{T}_i{:}\tau$ of the original program, P, are:

if there exists a corresponding thread sequence $\mathbb{T}_i{:}\tau'$ of $\widehat{P}$ then,

ewt1 $\mathcal{E}_{\mathbb{T}_i:\tau} = \mathcal{E}_{\mathbb{T}_i:\tau'}$
($i.e.$, the set of events of the original and transformed sequences remain same, however, their order of occurrence may vary);

ewt2 if $\exists e_1, e_2$ s.t. $e_1 <_{\mathbb{T}_i:\tau} e_2 \wedge e_2 <_{\mathbb{T}_i:\tau'} e_1$, then $e_2 \in (\mathcal{E}^{\mathbb{W}} \cup \mathcal{E}^{\mathbb{F}(\texttt{rel})})$, $e1 \notin \mathcal{E}^{\mathbb{F}(\texttt{rel})} \wedge (\nexists e_3 \in \mathcal{E}_{\mathbb{T}_i:\tau}$ s.t. $e_1 \leq_{\mathbb{T}_i:\tau} e_3 <_{\mathbb{T}_i:\tau} e_2 \wedge \texttt{dep}(e_3, e_2)$
($i.e.$ the `write` and `rel fences` that may form synchronizations with the `writes` (refer Section 5.5.2) are moved earlier than other events while maintaining their relative order and ensuring program dependence is not violated).

Consider the input program (P) in Figure 5.12 and its early-write transformation $(\widehat{P})$. The `write` event b and the `rel fence` c move earlier than a but event d does not because there is a data-dependence between a and d.

Further, assuming that control dependence is included in dep, f moves earlier than e but not earlier than the 'if' condition because of the control dependence.

Initially $x = 0, y = 0$

$(\mathbb{T}_1)$	$(\mathbb{T}_2)$	$(\mathbb{T}_3)$	$(\mathbb{T}_4)$
a: $W(x,1)$	b: $R(x,1)$	d: $W(y,1)$	e: $R(y,1)$
	c: $R(y,0)$		f: $R(x,0)$

Figure 5.13: IRIW

Early-write transformation along with `shadow-writes` simulate the reordering of `write` events with other events of a thread. To simulate the reordering of `read` events of a thread, MoCA proposes a static transformation called, ordered-reads transformation from the early-write transformed program $\widehat{P}$ to $\underline{\widehat{P}}$.

5.6.2 Ordered-reads transformation

Consider the outcome of the program IRIW shown in Figure 5.13 where the earlier `reads` of the threads $\mathbb{T}_2$ and $\mathbb{T}_4$ read from the `write` events a and d while the later `reads` read the initial values. The `shadow-writes` corresponding to the `writes` a and d do not perform any effective reordering and the outcome shown in Figure 5.13 is infeasible with `shadow-writes` and early-write transformation.

The outcome shown in Figure 5.13 is feasible under MCA if the `reads` are be reordered. To support such reordering consider variants $\widehat{P}^1, \cdots, \widehat{P}^m$ of the early-write transformed program $(\widehat{P})$ that vary on the order of `read` events of the input program. The variants are constructed s.t. each variant $\widehat{P}^k$ varies in the ordering of at least one pair of reads. Each of the variants $\widehat{P}^k$ are then verified separately.

Consider again the input program shown in Figure 5.13, the program has four read-order variants, characterized by:

(v1) no reordering,
(v2) reordering of $\mathbb{T}_2$ reads,
(v3) reordering of $\mathbb{T}_4$ reads, and

(v4) reordering of $\mathbb{T}_2$ and $\mathbb{T}_4$ reads.

Note that, the upper bound on the variants can be computed in $m = \prod_{\mathbb{T}_i \in P} |\mathcal{E}^{\mathbb{R}}_{\mathbb{T}_i}|!$.

The above discussed solution effectively reorders the `read` events. Combining the solution with early-write transformation and reordering with shadow-writes, *soundly* admit the permitted reordering of MCA. However, from the expression on the upper bound on the variants (m), we can ascertain that the solution is expensive to the verification process. If the program threads have a large number of read events, then the solution will inevitably lead to a combinatorial blow-up.

In order to overcome the expensive reordering of reads, consider an effective heuristic called *ordered-reads transformation*. The heuristic does not inflate the verification task and works well in practice.

The transformation chooses a fixed order of occurrence of `read` events of an object from all threads. The transformation ranks the objects of `read` events by program dependence *i.e.* if a `read` of object y in a thread is dependent on a `read` of object x and the `read` of object x has no such dependence on a `read` of y, then object y is ranked higher than object x. The transformation then places the `reads` in increasing order of object ranks; formally,

$escore(e) \triangleq |\{e' \text{ s.t. } \texttt{dep}(e', e)\}|$.
(size of the set of events that e is program dependent on).

$oscore(o) \triangleq \sum_{e \in \mathcal{E}^{\mathbb{R}}} escore(e)$ where $obj(e) = o$.
(score of object o based on the number of `reads` of o that are program dependent on other events).

$rank(o) \triangleq$ any arbitrary unique numeric value s.t. $\forall o'$ if $oscore(o') > oscore(o)$ then $rank(o') > rank(o)$
(each object o has a unique rank aligned with the object scores)

Consider the input program in Figure 5.13, $oscore(x) = 0$ (0+0), and $oscore(y) = 1$

(1+0) and accordingly, $rank(x) = 0$ and $rank(y) = 1$ (as a `read` of y from thread $\mathbb{T}_2$ is dependent on the `read` of x).

Given early-write transformed input program $\widehat{P}$, the rules of ordered-reads transformation (from $\widehat{P}$ to $\underline{\widehat{P}}$) for each thread sequence $\mathbb{T}_i{:}\tau$ of $\widehat{P}$ are:

if exists a corresponding ordered-reads transformed sequence $\mathbb{T}_i{:}\tau'$ of $\underline{\widehat{P}}$ then

ort1 $\mathcal{E}_{\mathbb{T}_i:\tau} = \mathcal{E}_{\mathbb{T}_i:\tau'}$
(the set of events of the original and transformed sequences remain same, however, their order of occurrence may vary);

ort2 if $\exists e_1, e_2$ s.t. $e_1 <_{\mathbb{T}_i:\tau} e_2 \wedge e_2 <_{\mathbb{T}_i:\tau'} e_1$, then $e_1, e_2 \in \mathcal{E}^{\mathbb{R}} \wedge rank(obj(e_2)) < rank(obj(e_1))$
$\wedge \nexists e_3 \in \mathcal{E}_{\mathbb{T}_i:\tau}$ s.t. $e_1 \leq_{\mathbb{T}_i:\tau} e_3 <_{\mathbb{T}_i:\tau} e_2 \wedge \mathtt{dep}(e_3, e_2)$
(`reads` e_1, e_2 can reorder if there is no intervening e_3 that introduces program dependence with e_2).

For the input program in Figure 5.13 the order of `read` events of $\mathbb{T}_4$ are reversed to bring the low ranking object's `read`, (f), above the higher ranking object's `read`, (e). The outcome shown in Figure 5.13, feasible under MCA, can be explored by MoCA with ordered-reads transformation.

Consequently, the ordered-reads transformation reorders the `read` events in effect. Further, with the transformation, MoCA needs to invoke stateless model checking exactly once.

Theorem 8. Early-write transformation and ordered-reads transformation (P to $\underline{\widehat{P}}$) are semantics preserving.

Proof. Consider the sequence of events of a thread $\mathbb{T}_i{:}\tau$ of the input program P, and its corresponding transformed sequence of events in a thread $\mathbb{T}_i{:}\tau'$ of $\underline{\widehat{P}}$. To ensure a semantic preserving reordering the conditions spr1-3 (refer to §5.4) must hold for corresponding threads of P and $\underline{\widehat{P}}$.

Let $\texttt{prev}\mathbb{W}_{[\mathbb{T}_i:\tau]}(e)$ represent the previous `write` event to e in a thread $\mathbb{T}_i$, formally, $e_w \in \mathcal{E}^{\mathbb{W}}_{\mathbb{T}_i:\tau} = \texttt{prev}\mathbb{W}_{[\mathbb{T}_i:\tau]}(e)$ if $thr(e_w) = thr(e) = \mathbb{T}_i$, $e_w <_{\mathbb{T}_i:\tau} e$ and $\nexists e'_w \in \mathcal{E}^{\mathbb{W}}_{\mathbb{T}_i:\tau}$ s.t. $thr(e'_w) = \mathbb{T}_i \wedge e_w <_{\mathbb{T}_i:\tau} e'_w <_{\mathbb{T}_i:\tau} e$.

spr1 The condition is met by definition of the transformations.

spr2 $\forall\, e', e$ to be executed by thread $\mathbb{T}_i$, $\texttt{dep}(e', e) \Rightarrow e' <_{\mathbb{T}_i:\tau} e \wedge e' <_{\mathbb{T}_i:\tau'} e$,
i.e., dependent events are not reordered (by definition of the transformations).

$\Rightarrow \forall e_r \in \mathcal{E}^{\mathbb{R}}_{\mathbb{T}_i:\tau}$, $\texttt{prev}\mathbb{W}_{[\mathbb{T}_i:\tau]}(e_r) = \texttt{prev}\mathbb{W}_{[\mathbb{T}_i:\tau']}(e_r)$.

$\Rightarrow \mathbb{T}_i{:}\tau'$ preserves the sequential semantics of $\mathbb{T}_i{:}\tau$.

spr3 $\forall o \in \mathcal{O}$, $\forall e', e \in \mathcal{E}$ of $\mathbb{T}_i{:}\tau$ s.t. $obj(e') = obj(e) = o$, if $e' <_{\mathbb{T}_i:\tau} e$ then $e' <_{\mathbb{T}_i:\tau'} e$,
i.e., events accessing same object are not reordered. (by definition of the transformations).

$\Rightarrow \mathbb{T}_i{:}\tau'$ preserves coherence-per-location.

Hence, early-write transformation and ordered-reads transformation are semantic preserving. □

Lemma 1. If e', e can reorder under MCA then they can reorder *in effect* under MoCA (with shadow-writes, early-write transformation, and ordered-reads transformation).

Proof. For all events e', e such that $e' \overset{R}{\Leftarrow} e_{\langle e' \rangle}$, there are four valid possibilities considered below.

Case(i). $e', e \in \mathcal{E}^{\mathbb{R}}$:

$\exists$ a variant $\widehat{P}_k$ of early-write transformed $\widehat{P}$ where e occurs before e'.

Case(ii). $e' \in \mathcal{E}^{\mathbb{R}}$, $e \in \mathcal{E}^{\mathbb{W}}$:

$(e' \overset{R}{\Leftarrow} e_{\langle e' \rangle}) \Rightarrow \neg\texttt{dep}(e', e) \Rightarrow e' <_{\mathbb{T}_i:\tau} e$ but $e <_{\mathbb{T}_i:\tau'} e'$ (the events will be reordered with early-write transformation) and then $shw(e)$ and e' can be interleaved.

Case(iii). $e' \in \mathcal{E}^{\mathbb{W}}$, $e \in \mathcal{E}^{\mathbb{R}}$:

$(e' \overset{R}{\Leftarrow} e_{\langle e' \rangle}) \Rightarrow$

(i) either $obj(e') \neq obj(e) \Rightarrow thr(shw(e')) \neq thr(e) \Rightarrow shw(e')$ and e can be interleaved; or

(ii) $obj(e') = obj(e)$ but $obj(e') \neq obj(e_{\langle e' \rangle})$ (case of interleaving with forwarding) $\Rightarrow$ $shw(e')$ and $e_{\langle e' \rangle}$ can be interleaved while e can still read from e' (by definition of $[m]::\mathrm{rf}_\tau$).

Case (iv). $e', e \in \mathcal{E}^{\mathbb{W}}$:

$(e' \overset{R}{\Leftarrow} e_{\langle e' \rangle}) \Rightarrow obj(e') \neq obj(e) \Rightarrow thr(shw(e')) \neq thr(shw(e)) \Rightarrow shw(e')$ and $shw(e)$ can be interleaved. □

5.6.3 Stateless model checking with source-DPOR

Central to MoCA is source-DPOR (Algorithm 1 of [1]). Source-DPOR is a state-of-the-art stateless model checking technique proposed as a near-optimal improvement over DPOR [35]. Source-DPOR was proposed for interleaving semantics and uses a novel representation of the set of events to be explored from a state called *source sets* [4] which is a much compact set of starts in comparison to the classic persistent sets [39] or ample sets [74]. It is noteworthy that the source-DPOR algorithm used in MoCA is *as is*, that is, without any modification, this was feasible because of several reasons:

- Design of a *valid* happens-before relation (Theorem 6) for restricting C11 under MCA is directly pluggable in source-DPOR,
- The proposal of `shadow-writes`, along with early-write transformation and ordered-reads transformation, makes it possible to avoid reordering instructions from a thread during exploration and rely on interleaving model of computation alone (that source-DPOR is designed for), and

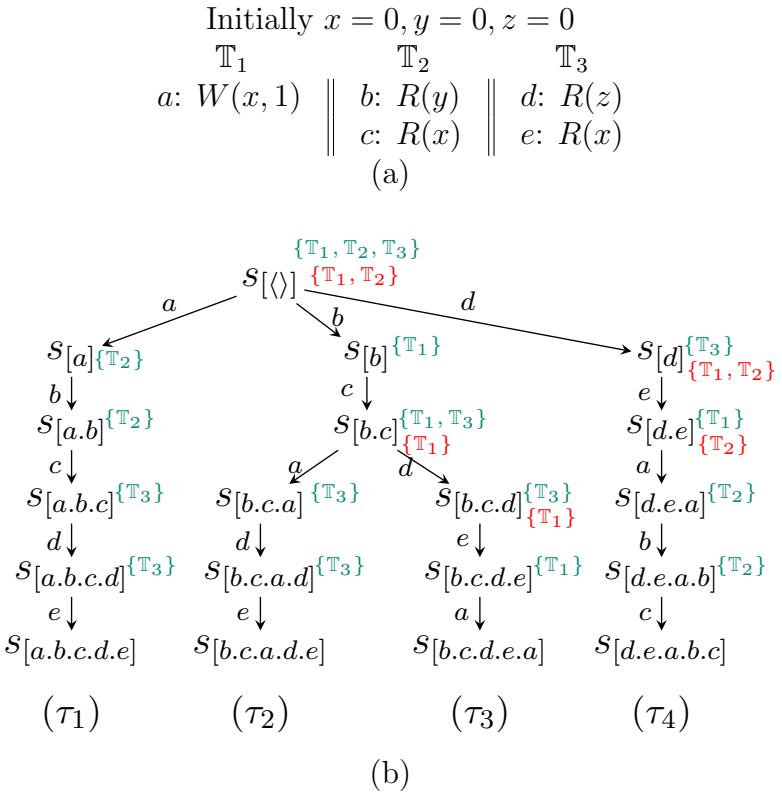


Figure 5.14: Writer-reader

- The parallel composition rule (`parcom`) (§5.4) satisfies the requirement of source-DPOR that only one thread executes at a time.

Brief overview of the Source-DPOR Algorithm

Source-DPOR is a non-chronological depth-first search of a directed acyclic graph of execution states. Much like the quintessential DPOR [35], source-DPOR maintains a set of threads that should be explored at each state and a set of *sleeping* threads.

Consider the program in Figure 5.14(a). Figure 5.14(b) shows its exploration by the source-DPOR algorithm. At each state of exploration, source-DPOR maintains a *backtrack* set of threads such that the next events from the threads in the backtrack

set must be explored from the state. At each state of exploration, source-DPOR also maintains a *sleep* set of threads representing the threads whose next event must not be explored from the state. While the backtrack set helps the algorithm to move forward for a sound and complete exploration, the sleep set is maintained to avoid redundant explorations.

Consider the exploration in Figure 5.14(b). The backtrack sets at each state are shown in green while the sleep sets are shown in red. The exploration starts at the initial state where thread $\mathbb{T}_1$ is randomly chosen for exploration and is added to the backtrack set. The algorithm similarly selects a random thread to explore at each state of the first execution sequence τ_1. From τ_1 source-DPOR recognizes the racing event pairs $a{\rightarrow}c$ and $a{\rightarrow}e$ and inserts $\mathbb{T}_2$ and $\mathbb{T}_3$ to the backtrack set at $s_{[\langle\rangle]}$ to reverse the order of the racing events. Source-DPOR then backtracks and finds a state that has unexplored threads in the backtrack set, *i.e.* $s_{[\langle\rangle]}$ where $\mathbb{T}_2$ and $\mathbb{T}_3$ are unexplored. It adds the previously explored thread, $\mathbb{T}_1$, to the sleep set at $s_{[\langle\rangle]}$ and continues to explore the next event of $\mathbb{T}_2$, and further the next event of $\mathbb{T}_3$.

Note that, the sleep set is carried to the next state of exploration until an event is explored which races with the next events of the sleeping thread. Hence, the sleep set at state $s_{[d]}$ is the same as the sleep set at $s_{[\langle\rangle]}$ *i.e.* $\{\mathbb{T}_1, \mathbb{T}_2\}$ since event d does not race with a, b, or c. Thread $\mathbb{T}_1$ is removed from the sleep set after exploration of e which races with a and a explores from $s_{[d.e]}$. Similarly, $\mathbb{T}_2$ is removed from the sleep set after exploration of a which races with c and b explores from $s_{[d.e.a]}$.

Refer to [1] for formal description and algorithm of source-DPOR.

Theorem 9. MoCA traces are equivalent to C11 traces valid over MCA.

Proof. *Case* $\rightarrow$:

$\forall e', e$ s.t. $\neg(e' \overset{R}{\Leftarrow} e) \Rightarrow \mathtt{dep}(e', e) \Rightarrow$ if $e'{<}_{\mathbb{T}_i:\tau}e$ then $e'{<}_{\mathbb{T}_i:\tau'}e$, for $\mathbb{T}_i : \tau$ of P and $\mathbb{T}_i : \tau'$ of $\widehat{P}$.

Secondly, if $\mathtt{dep}(e', e)$ where $e', e \in \mathcal{E}^{\mathbb{W}}$ then $obj(e') = obj(e) \Rightarrow shw(e'){<}_{\tau_1}shw(e)$ for every execution sequence τ_1 (by construction of shadow threads).

Further, MoCA supports forwarding (by definition of $[m]$::rf_τ, see Section 5.5.2 for details).

Thus, reordering restricted by C11 over MCA is also restricted by MoCA. inf(i)

Early-write transformation is semantic preserving.

Further, if $\exists e', e \in \mathcal{E}_{\mathbb{T}_i:\tau}$ s.t. $e' <_{\mathbb{T}_i:\tau} e$ and $e <_{\tau'} shw(e')$ for a trace τ' and $\mathtt{dep}(e', e)$ then e' is observable for e (by definition of $[m]$::rf_τ) and if effect of e is observed by another thread then so is effect of e' (using `(mca-co)`, `(mca-moRR)` and `(mca-WE)`).

Thus reordering allowed by MoCA is allowed by MCA. inf(ii)

Since, MoCA traces are coherent C11 traces (Theorem 7) and using inf(i) and inf(ii), the theorem can be restated as,
if a trace is valid under MoCA then it is valid under MCA.

Events $e_w \in \mathcal{E}^{\mathbb{W}}$ along with $shw(e_w)$ perform `(w-issue)` and `(w-update)` while preserving semantics (using `(shco)`); $e_r \in \mathcal{E}^{\mathbb{R}}$ perform `(r-shared)`. Source-DPOR algorithm ensures `(parcom)`. Sequences of operations performed by MoCA to generate traces are sequences of MCA operations. inf(iii)

Thus, from inf(i), (ii), and (iii) traces of MoCA have equivalent C11 MCA traces.

Case $\leftarrow$:

Assume MoCA restricts the reordering of events e', e then $e' <_{\mathbb{T}_i:\tau} e$ and $e' <_{\mathbb{T}_i:\tau'} e$ for $\mathbb{T}_i : \tau$ of P and $\mathbb{T}_i : \tau'$ of $\widehat{P}$.

Further if $e', e \in \mathcal{E}^{\mathbb{W}}$ then for all MoCA traces τ then $shw(e') <_\tau shw(e) \Rightarrow thr(shw(e')) = thr(shw(e)) \Rightarrow \mathtt{dep}(e', e)$ (by construction).

Hence, reordering restricted by MoCA is also restricted by C11 over MCA. inf(iv)

Reordering allowed by C11 over MCA are allowed by MoCA. (Lemma 1).

Since, MoCA traces are coherent C11 traces (Theorem 7) and using inf(iv) and Lemma 1, the theorem can be restated as,
if a trace is valid under MCA then it must be valid under MoCA.

`(r-shared)` is performed by $\mathcal{E}^{\mathbb{R}}$, `(w-issue)`, and `(w-update)` by $\mathcal{E}^{\mathbb{W}}$ and $\mathcal{E}^{\mathbb{M}}$, `(parcom)` is ensured by source-DPOR algorithm. Hence, sequences of operations produced by MCA semantics can be replicated by MoCA traces. inf(v)

Thus, from inf(iv) and (v) C11 MCA traces are valid MoCA traces. □

5.6.4 Time complexity analysis

The worst-case time complexity of the source-DPOR algorithm is $O(|\mathbb{T}|^2|\mathcal{E}|^2S)$, where S is the number of sequences explored[7].

The relation $[m]::\text{hb}_\tau$ (or $[t]::\text{hb}_\tau$ or $[a]::\text{hb}_\tau$) has the same computational complexity as its counterpart in the original source-DPOR work, *i.e.*, $O(|\mathcal{E}|^2)$.

The addition of shadow-threads, however, increases the number of processing elements and makes the worst-case complexity of MoCA $O(|\mathcal{O}|^2|\mathbb{T}|^2|\mathcal{E}|^2S)$, where $|\mathcal{O}|^2|\mathbb{T}|^2$ is the number of threads for MoCA including the program threads and a shadow-thread per program thread per object.

5.7 Implementation details of MoCA SMC

The implementation starts by statically converting the input program P to $\widehat{P}$, using early-write transformation, and then to $\underline{\widehat{P}}$, using ordered-reads transformation. The static transformation is performed once for every input program, and the stateless model checking is performed on the transformed program $\underline{\widehat{P}}$.

[7] Source-DPOR is not an optimal technique and may explore redundant sequences and sleep set blocked [1] sequences.

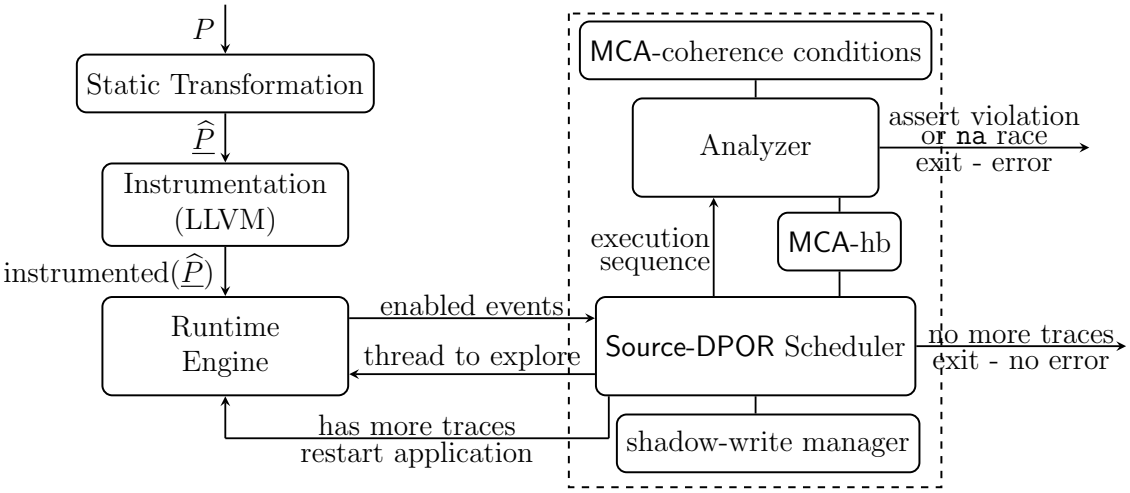


Figure 5.15: Structural overview of MoCA tool

The implementation of MoCA technique has two major components,

1. an *execution component* responsible for executing the input program for each maximal sequence explored by the technique, and
2. a *scheduler* that schedules the program executions and analysis the execution sequences for coherence and correctness.

The processing of MoCA tool starts by invoking the scheduler. The scheduler in turn starts a fresh execution of the statically transformed input program ($\widehat{P}$) and receives the first set of enabled events. The scheduler then applies the source-DPOR algorithm on the enabled events and informs the runtime engine of the next event to execute. Source-DPOR computes the event relations ($[m]::\mathrm{hb}_\tau$, $[m]::\mathrm{rf}_\tau$, and $[m]::\mathrm{mo}_\tau$) on the events of the sequence and relies on the MoCA coherence conditions (refer to Figure 5.11) to compute an execution sequence coherent under C11 restricted to MCA. On execution of an event, the runtime engine receives a set of newly enabled events that it forwards to the scheduler.

The process continues till a maximal sequence is executed, or an assert violation or `na` race is detected. Further, the scheduler restarts the transformed input program

($\underline{\widehat{P}}$) for the next execution sequence using the runtime engine when there are more traces to explore.

The key modules of the implementation design are presented in Figure 5.15 and discussed below.

- **Static Transformation.** As the first step, the implementation performs a source-to-source transformation of the input program P, using early-write transformation and ordered-reads transformation, to $\underline{\widehat{P}}$. The transformation is semantic preserving (shown formally with Theorem 8).

  After the transformation all valid program outcomes can be explored by scheduling various interleavings alone. The effects of feasible reorderings can be achieved with interleaving on $\underline{\widehat{P}}$.

- **Instrumentation.** The transformed input program ($\underline{\widehat{P}}$) is then instrumented by inserting code in $\underline{\widehat{P}}$ at relevant control points such as memory access, thread creation, thread join, assert condition etc.

  The instrumentation calls respective modules on reaching the relevant control points to supply data for the analyses and receive execution instructions. The instrumentation, thus, allows MoCA tool to take control of the interleaving order to realize the desired execution sequence. The instrumentation preserves program behaviors.

- **Runtime Engine.** The runtime engine starts the program $\underline{\widehat{P}}$, and executes the instrumented program as per the directed execution order.

- **Scheduler.** The scheduler is control center of the implementation that perform various tasks, such as,

  - It controls the interleaving order of program threads (by directing the runtime engine on next event to execute), and thus schedules various execution sequences. It also directs the runtime engine to restart the program if it computes that there are more traces to explore.

- It computes the event relations on the explored event sequence and uses the MoCA coherence conditions to help compute the next event of the sequence and the next execution sequence.
- It implements the source-DPOR algorithm based on the MoCA event relations and MoCA coherence conditions.
- It creates and maintains shadow-threads and `shadow-writes`.
- It invokes an *Analyzer* module on each maximal execution to analyze the program for correctness.

- **Analyzer.** Each maximal sequence is analyzed for assert violations and `na` races. On detection of either of the two conditions MoCA halts the stateless model checking and returns an error trace.

5.7.1 Tool description

The implementation of MoCA technique is done in `C++` language over `rInspect` [77] tool. The tool takes a `C` or `C++` program as input and uses the `pthread` library for multi-threading. The *intermediate representation* of the source program generated after compilation is instrumented using `LLVM`. The instrumentation performs a source-to-source transformation of the intermediate representation by inserting suitable calls to interpreter modules. The tool detects the violations of safety properties provided as assert statements in the input program.

The *scheduler* module controls the execution and analysis of program sequences. At each state of exploration the scheduler maintains two sets of events, called the source set and the sleep set (refer to [1]). The two sets contain events that must be explored and must not be explored from the state respectively.

If the tool explores a maximal sequence (no assert condition was violated and no `na` race was detected in the sequence) then the scheduler identifies a state ($s_{[\tau]}$) where the sleep set is a subset of the source set signifying that a different schedule has to be explored from the state. If such a state ($s_{[\tau]}$) exists then the scheduler restarts the

input program and schedules the execution of the event sequence, upto the state $s_{[\tau]}$, in the previously explored order. At the state $s_{[\tau]}$ the scheduler adds the previously explored event in the sleep set and executes the next event in the source set.

Exploring `shadow-writes`

For each program thread $\mathbb{T}_i$, the scheduler creates shadow-threads of $\mathbb{T}_i$ corresponding to each object. A program thread receives events from the runtime engine, while the events of a shadow-thread are created by the scheduler. The source-DPOR algorithm does not differentiate between program threads and shadow-threads.

`Shadow-write` events are created by the scheduler and added to the corresponding shadow-thread as follows: when an event $e \in \mathcal{E}^{\mathbb{W}}$ of thread $\mathbb{T}_i$ is executed from a state $s_{[\tau]}$, a corresponding `shadow-write` event $shw(e)$ is generated and added to the shadow-thread $\mathbb{T}_{si} \in sth(e)$ corresponding to $obj(e)$. Similar to program threads, execution of an enabled `shadow-write` event $shw(e)$ of a shadow-thread $\mathbb{T}_{si}$ from a state $s_{[\tau]}$ enables the next event of $\mathbb{T}_{si}$ at state $s_{[\tau.shw(e)]}$.

Supported data-types and operations

The tool supports atomic and non-atomic data-types and all data structures. The tool further supports the following operations on global variables (program events): read, write and the following `rmw` operations, *fetch-and-add*, *fetch-and-subtract*, *fetch-and-and*, *fetch-and-nand*, *fetch-and-or*, *fetch-and-xor*, *fetch-and-max*, *fetch-and-min*, *exchange* and *compare-and-exchange*. The tool supports all operations on local variables.

5.8 Experiments and Results on MoCA SMC

5.8.1 Experimental setup

The experiments are conducted on an Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz with 32GB RAM and 32 cores running Ubuntu 16.04.1 LTS. LLVM 3.3 is used for to perform the instrumentation due to the dependence of `rInspect` on this version.

The performance of MoCA is compared against two state-of-the-art stateless model checkers for C11 (and its variants), namely CDSChecker [72], and GenMC [53]; and a hardware model checker called HMC [54].

CDSChecker is a SMC for C11 programs. CDSChecker forms potential pairs of reads and its sources called *promises* and attempts to generate traces where the `read` can coherently read from the source `write` in the pair.

GenMC analysis programs under a variant of C11 called RC11. GenMC generates *execution graphs*, in essence, partial orders of event relations representing a trace, to perform the analysis. The design of execution graphs ensures GenMC does not explore any redundant sequences, also known as an optimal exploration.

HMC is a stateless model checker that takes C11 programs as input and analyzes them for a hardware memory model called IMM, a superset of hardware memory models such as POWER, RISC-V, and ARM. Like GenMC, HMC uses execution graphs to generate traces and claims optimality of exploration.

5.8.2 Coherence validation

To validate the coherence of the MoCA technique and its implementation, the MoCA tool is tested on a set of litmus tests for the MCA model and for the C11 model.

Coherence of MoCA with respect to MCA is validated the diy7 family of litmus tests [47]. Table 5.1 lists a sample set of the tests. Coherence of MoCA with respect to C11 is vali-

Table 5.1: MCA tests

Test	#Seq	Time
CoRR	3	0.02s
CO-RSDWI	6	0.02s
R+fn+fn	5	0.02s
RSDWI	22	0.11s
WRR+2W	29	0.12s
Luc17	12	0.07s
Luc10	MV	0.02s
S-popl	MV	0.01s

Table 5.2: C11 tests

Test	#Seq	Time	race?	#Rseq
simple-sw	3	0.006s	Y	2
simple-ithb	4	0.034s	Y	2
RS-blk	MV	0.07s	Y	8
CSE-no-blk	12	0.158s	N	-
no-fence-sync	MV	0.054s	Y	5
fib-no-assert	26	0.14	N	-
fmax-cas	31	0.21s	N	-
flipper	1628	9.29s	N	-

dated on a set of 56 synthesized litmus tests and multi-threaded benchmarks borrowed from SV-Comp benchmark suite [22]. The SV-Comp benchmarks are remodeled with the use of atomic data types and associated memory orders. Table 5.2 lists a sample set of the tests, where rows 1-4 depict results on the synthesized tests and rows 5-8 depict results on the SV-Comp benchmarks.

The column 'Time' shows the time of analysis and the column '#Seq' shows the number of maximal sequences explored, which includes *at least* one execution corresponding to each program trace and (possibly) a few redundant executions owing to the non-optimal nature of the underlying source-DPOR algorithm. The value 'MV' in the column '#Seq' signifies that an assert violation is detected in a C11 trace of the program that is valid over MCA. Further, in Table 5.2, a 'Y' in the column 'race?' signifies that the test contains an `na` races. In such a case the number of maximal sequences that contain `na` race(s) is reported in the column (column '#Rseq').

5.8.3 Litmus testing

To demonstrate the effectiveness of a precise analysis of C11 programs over MCA, we collect a set of litmus tests from SV-Comp benchmark suite and previous works [51, 54, 76] that produce a *strict subset* of C11 behaviors when restricted to MCA. The

Table 5.3: Comparative Results on litmus tests

Test(#MCA traces)	MoCA			CDSChecker			GenMC			HMC		
	M	N	Time	M	N	Time	M	N	Time	M	N	Time
WRC+addrs(7)	7	0	0.03s	7	1	0.01s	7	1	0.02s	7	1	0.03s
WR-ctrl(4)	7	0	0.03s	4	2	0.01s	4	2	0.02s	4	2	0.02s
Z6+poxxs(4)	18	0	0.12s	14	4	0.01s	4	4	0.03s	4	4	0.03s
IRIW+addrs(15)	15	0	0.07s	15	1	0.01s	15	1	0.02s	15	1	0.02s
WW+RR(15)	96	0	0.53s	15	66	0.02s	15	66	0.02s	15	66	0.02s

M: #MCA sequences, N: #non-MCA sequences

outcome of MoCA on such tests is compared against CDSChecker, GenMC and HMC.

The focus of litmus testing is on demonstrating that

1. MoCA indeed does not explore any program outcomes that represent non-MCA traces, while other existing verification techniques for C11 input programs explore MCA and non-MCA outcomes.

2. A significantly large number of program outcomes may be non-MCA in nature.

The litmus testing is performed on small tests with 15 or less MCA traces. Since the tests are small, the number of MCA outcomes for the input programs are computed manually.

Table 5.3 shows the results of the litmus testing. The number of C11 traces valid under MCA have been shown in bracket accompanying the name of the test. For instance, 'WRC+addrs(7)', shows that the test 'WRC+addrs' has 7 C11 traces valid over MCA (computed manually). Columns 'M' and 'N' represent, respectively, the number of MCA sequences and non-MCA sequences explored by the corresponding technique. Columns 'Time' represent the time of analysis.

Note that, MoCA is not an optimal technique (since the underlying source-DPOR algorithm is not an optimal algorithm). Thus, for tests where the explored MCA sequences

Table 5.4: MoCA performance analysis

Benchmark	MoCA		CDSChecker		GenMC		HMC	
	#Seq	Time	#Seq	Time	#Seq	Time	#Seq	Time
mutex	5	0.02s	2-NVs	0.01s	NV	0.03s	NV	0.02s
peterson	13	0.15s	666-NVs	2.73s	NV	0.02s	NV	0.02s
RW-lock	246	0.52s	193-NVs	0.38s	NV	0.02s	NV	0.04s
spinlock	506	16.98s	To	-	NV	0.08s	NV	0.15s
fibonacci-2	667	5.57s	To	-	NV	0.04s	NV	0.03s
fibonacci-3	10628	2m14s	To	-	NV	0.06s	NV	0.07s
fibonacci-4	92421	56m21s	To	-	NV	0.13s	NV	0.31s
counter-5	3599	39.78s	25-NVs	0.31s	NV	0.06s	NV	0.03s
counter-10	55927	12m53s	100-NVs	9.21s	NV	0.05s	NV	0.07s
counter-15	To	-	225-NVs	50.31s	NV	0.11s	NV	0.16s
flipper-5	2489	20.19s	201-NVs	3.26s	NV	0.03s	NV	0.04s
flipper-10	96737	6m12s	To	-	NV	0.04s	NV	0.02s
flipper-15	To	-	To	-	NV	0.03s	NV	0.03s
prod-cons-10	9373	1m23s	To	-	NV	0.04s	NV	0.04s
prod-cons-15	38593	6m46s	To	-	NV	0.02s	NV	0.02s
prod-cons-20	109838	20m28s	To	-	NV	0.02s	NV	0.02s

('M') is larger than the number of MCA traces, that is tests 'WR-ctrl', 'Z6+poxxs', and 'WW+RR', MoCA has explored redundant or sleep set blocked sequences (refer to [1]). Similarly, CDSChecker is not an optimal technique while GenMC and HMC are. Thus, tests where CDSChecker has explored more sequences than GenMC and HMC it implies that CDSChecker has explored redundant sequences.

As can be seen from Table 5.3, MoCA only explores MCA traces, while all other techniques explore non-MCA traces along with MCA traces. It can also be observed that the set of non-MCA outcomes can be significantly large.

5.8.4 Performance analysis

The performance analysis of MoCA is done over challenging benchmarks borrowed from SV-comp benchmark suite [22]. The performance is measured on various configurations of each benchmark, where the configurations vary on the problem size determined by program features such as the number of loop unrolls and the number of concurrent processing elements (or threads). In essence, higher configurations of benchmarks typically result in a higher number of program events.

The performance on the benchmarks is measured on three aspects.

1. *Time of analysis.* The time taken to verify a configuration of a benchmark input program.
2. *Scalability.* The highest configuration of a benchmark that can be verified within a reasonable time of analysis, also known as *Timeout of analysis* or simply *Timeout* (To), set at 3600 seconds.
3. *Number of non-MCA assert violations.* The number of execution sequences that report an assert violation which is infeasible on an MCA architecture.

The performance analysis of MoCA against that of techniques CDSChecker, GenMC, and HMC is presented in Table 5.4. The number of program outcomes of the benchmarks is much larger than that of the litmus tests. It is infeasible to compute the number of MCA outcomes manually. Therefore, appropriate assert conditions are added to the tests in Table 5.4 that are not violated under the MCA model but may violate under a non-MCA model. The goal of the testing is to ensure that MoCA does not report an assert condition while the other techniques do.

Configurations of benchmarks. The configurations of a benchmark vary the problem size such that higher configurations require a higher effort of analysis. The configurations typically vary on the number of concurrent elements (threads), the number of loop iterations, and the number of events. The configurations of the benchmarks in Table 5.4 vary on the following aspects.

The benchmark ‘fibonacci’ uses N which represents that the program can generate not larger than the Nth fibonacci term. The configurations of ‘fibonacci’ vary on N. The configurations of ‘counter’ vary on the number of times increment and decrement is performed on a shared object. Similarly configurations of ‘flipper’ vary on the number of times the value of a shared object is atomically flipped (between 0 and 1). Configurations of ‘prod-cons’ vary on the number of times the producer thread produces and the consumer thread consumes.

Observations. The value ‘NV’ in the column ‘#Seq’ indicates assert violation(s) in non-MCA execution sequences. On detecting an assert violation the techniques GenMC and HMC halt their verification process. However, CDSChecker continues the exploration after the detection of an assert violation and reports all execution sequences with assert violations. As a consequence, the total number of non-MCA violations are reported for CDSChecker, as ‘x-NVs’, where ‘x’ is the number of assert violations in non-MCA sequences. The number of assert violations is not known for GenMC and HMC (since they halt at the first violation). Hence, for GenMC and HMC the value ‘NV’ indicates that an assert violation is detected in a non-MCA execution sequence.

Note that, the time reported for GenMC and HMC is the time to encounter the first assert violation and is therefore much lower than the time reported by CDSChecker and MoCA that perform complete exploration. As the result, the reported time of analysis is incomparable and is added only for reference.

Consider again the bar graph shown in Figure 5.5. On the x-axis, the graph represents the number of MCA outcomes explored by MoCA over various tests of Table 5.4, and on the y-axis the graph shows the number of non-MCA assert violations explored by CDSChecker. Note that, the value ‘-’ on the x-axis represents the test ‘counter-15’ for which MoCA timed-out and the value for the x-axis is not known.

It can be observed from Table 5.4 and Figure 5.5 that benchmarks can produce hundreds of assert violations that may not be reproducible on an actual architecture. The result is witnessed especially for the benchmark ‘peterson’ where CDSChecker reports assert violations in 666 execution sequences that can never manifest on an

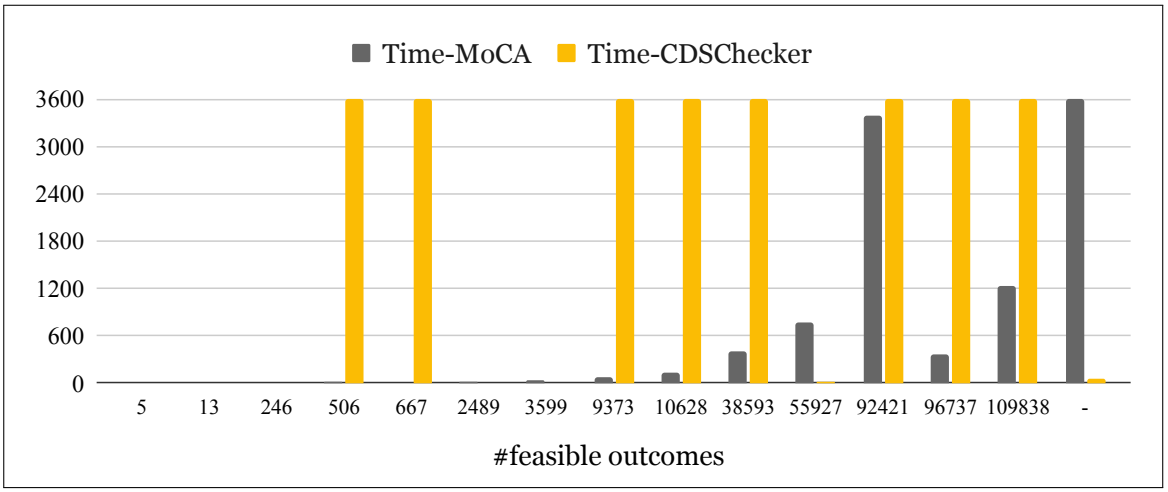


Figure 5.16: Time of analysis of CDSChecker vs Time of analysis of MoCA (seconds)

MCA architecture while the feasible outcomes is only 13. Thus, a precise technique for MCA such as MoCA can be useful.

A bar graph contrasting the performance of MoCA and CDSChecker, the two techniques that perform complete exploration, is shown in Figure 5.16 on the benchmarks of Table 5.4. The graph compares the time of analysis of MoCA (represented by yellow bars) against the time of analysis of CDSChecker (represented by gray bars).

The graph demonstrates that MoCA outperforms CDSChecker on the benchmarks of Table 5.4. CDSChecker performs analysis for the C11 traces while MoCA performs analysis for a restricted (for MCA) set of C11 traces. Hence, MoCA has to analyze a smaller set of equivalence classes and, thus, it outperforms CDSChecker.

The observation from the graph in Figure 5.16 empirically establishes that precise analysis restricts the number of equivalence classes reducing the analysis overhead (merit (M4), §5.3).

`na` race detection. The subset of execution sequences of the benchmarks `'mutex'`, and `'counter'` exhibit `na` data races. The `na` data races on these benchmarks were successfully detected by MoCA.

5.9 Scope, Limitations, and Future directions

Scope of C11-MCA happens-before relation and coherence conditions. The happens-before relation, presented for MoCA in §5.5.2, and its corresponding coherence conditions, presented in §5.5.3, admit traces valid under the memory model presented in the ISO 2011 standard of C/C++. The relation and the coherence conditions are not directly applicable to the memory models presented in the subsequent standards. However, the subsequent models are closely related to the C11 model and nominal modification to the happens-before relation and the coherence conditions can extend support for the subsequent models. The MoCA technique can then be extended to support the the subsequent models by plugging-in the respective happens-before relations and corresponding coherence conditions.

Threats to Validity

Scalability. SMCs are limited in scalability. There is a significant scope for improvement in performance of SMCs in terms of the time of analysis and scalability. MoCA does not scale to tests larger than those shown in §5.8.4.

Availability of benchmarks. There exist very few benchmarks of C11 programs that utilize C11 weak ordering guarantees, and there are even fewer benchmarks that highlight the difference in C11 outcomes and multi-copy atomic outcomes which are required for this work.

Memory orders for testing. The benchmark suites such as SV-Comp [22] and SCT [66] contain C programs without associated memory orders. The memory access operations are thus associated with the default memory order under C11, *i.e.* sc. To test the weaker behaviors the memory access operations are associated with appropriate weak memory orders which are chosen by the authors.

Establishing ground truth. The equivalence classes are not known to MoCA a priori. In this work, the ground truth is established by manually computing the set of outcomes of the input program.

Legacy limitations of `rInspect`. Due to low modularity in rInspect's design it invokes various legacy modules that may not be necessary for the analysis. The tool also has dependence on a considerably older version of LLVM (*i.e.* LLVM 3.3). `rInspect` legacy design adds a considerable analysis overhead on MoCA.

5.9.1 Future directions

Extension for other MCA models and other C/C++ models. The MCA model can be concretized for specific MCA architecture models by defining a concrete set of reordering rules. Accordingly, MoCA's support can be extended for specific architecture models by capturing their respective reordering rules through a corresponding happens-before relation.

Further, the modular design of the MoCA tool supports an easy extension for other memory models that follow MCA semantics. The support can be extended by appropriately replacing the MCA-hb component shown in Figure 5.15 for the respective happens-before relation.

MoCA's support may also be extended for newer versions of the C/C++ memory model, by appropriately adapting the $[m]$::hb_τ relation and the C11-MCA-coherence conditions (presented in §5.5.3).

Support for coarse grained synchronization mechanisms. The MoCA technique does not comprehend any coarse-grained synchronization primitives such as *locks*, in other words, it not *lock-aware*. MoCA's support may be extended for mutex locks or other synchronization primitives by appropriately extending the $[m]$::hb_τ relation.

5.10 Concluding remarks

This work presents a restriction of the C11 model for MCA. The restriction admits only those traces of the input program that are valid C11 trace and can manifest on

an architecture that supports MCA. The motivation of such a restriction is that it considers the developer specification along with the implicit ordering of the underlying architecture. Further, since MCA is supported my some of the most widely used architectures, traces of MCA relate to most systems in use.

This work recognizes the fundamental components the define a valid C11 trace as the event relations (hb_τ, mo_τ, and rf_τ), and a set of corresponding coherence conditions (refer to §3.2.3). Accordingly, the restriction of C11 for MCA is designed by redefining the two components to the C11-MCA event relations ($[m]::\text{hb}_\tau$, $[m]::\text{mo}_\tau$, and $[m]::\text{rf}_\tau$), and a set of corresponding coherence conditions (refer to §5.5.2 and §5.5.3 respectively). It is established with relevant theorems that the $[m]::\text{hb}_\tau$ relation is a valid happens-before relation and the restriction of C11 for MCA preserves coherence under C11.

Two key features allow a coherent restriction of C11 to MCA. The first feature is a novel event type called `shadow-writes` that update the shared memory for a `write` event at a later timestamp. The `shadow-write` events simulate *reordering through interleaving* by interleaving with the events in the corresponding `write` event's thread. `Shadow-writes` also simulate updates to a single shared memory. The second feature is a reads-from relation ($[m]::\text{rf}_\tau$), defined using the `shadow-writes`, that simulates the effect of reading from a single shared memory.

This work also presents a stateless model checking technique for precise verification of C11 traces that can manifest on an underlying MCA architecture. The technique is called MoCA. MoCA uses the source-DPOR algorithm based on the $[m]::\text{hb}_\tau$, , $[m]::\text{mo}_\tau$, and $[m]::\text{rf}_\tau$ relations and the C11-MCA-coherence conditions.

The technique uses two static transformations, called early-write transformation and ordered-reads transformation, performed prior to the model checking. Essentially, the transformations reshape the input program so that the use of `shadow-writes` can indeed simulate all feasible renderings permitted under C11 and MCA. It is shown that the transformations are semantic preserving, and that with the use of `shadow-writes` and the transformations, MoCA can capture the effect all feasible reorderings under C11 and MCA.

The MoCA technique, that includes the use of `shadow-writes`, C11-MCA event relations and coherence conditions, and the static transformations, is sound (explores a program execution for each trace), and precise (does not explore a trace that is not valid under C11 or MCA). This work also presents the worst-case time complexity analysis of the MoCA technique.

The MoCA technique is accompanied with an implementation for `C` and `C++` input programs. This work presents the corresponding implementation details including the (i) structural overview and key components, (ii) details of libraries and platforms used, and (iii) details of data-types and operations supported in the input program.

The MoCA tool is tested for coherence *wrt* C11 and coherence *wrt* MCA. The implementation is further tested on small litmus tests to highlight the imprecision of existing techniques *wrt* to MCA and the effectiveness of MoCA is precisely recognizing the C11 outcomes valid over MCA.

The performance of MoCA is compared, on competitive benchmarks, against SMCs that verify a C11 input program. The techniques are compared on the time of analysis, scalability, and precision for MCA. The results on the benchmarks highlight that a significantly large number of program outcomes may not be valid on MCA. Further, it is observed that the precise analysis (for a smaller set of equivalence classes) may result in a smaller time of analysis.

It is imperative to emphasize that the techniques CDSChecker and GenMC are designed for C11 (or its variants) and HMC is for a collection of hardware models subsuming MCA. Naturally, these techniques explore a larger set of traces, and the non-MCA violation(s) reported by them are indeed true violations under their respective models. However, some of the violations reported by them may never manifest on the underlying architecture.

Finally, this work discusses the scope of MoCA, focusing on the scope of the proposed event relations and coherence conditions. Based on the scope, this work also proposes worthy future directions.

Chapter 6

(fast)FenSying. Fence Synthesis for the C11 Memory Model

6.1 Background

The C11 memory model is classified as a weak memory model. Its semantics are weaker than most memory models associated with architectures and programming languages. Specifically, C11 falls into the non-multi-copy atomic class of memory models, as explained in §3.2.3.

Under C11, potentially every pair of events in an input program can be reordered, and `write` events can have staggered visibility to program threads. However, *coherence conditions* (refer to Figure 3.6) are used to restrict the possible *incoherent* outcomes resulting from such reordering or visibility.

Given the weak design of C11, the outcomes of a program executed under this model are more accurately represented as C11 traces - a tuple of $\langle \mathcal{E}_\tau, \text{hb}_\tau, \text{mo}_\tau, \text{rf}_\tau \rangle$. All program executions with the same set of events and event relations, hb_τ, mo_τ, and rf_τ, represent the same program trace or outcome. For further details on the C11 memory model, refer to §3.2.3.

Table 6.1: Number of buggy traces against size of input program

Benchmark name	#threads	LoC	#buggy traces
store-buffer(2)	2	56	6
store-buffer(4)	2	70	20
peterson(2,2)	2	68	30
peterson(2,3)	2	77	198
linuxrwlocks(2,1)	2	143	10
linuxrwlocks(3,8)	2	1462	353
seqlock(2,1,2)	3	81	500
seqlock(1,2,2)	3	88	592
dekker(2)	2	171	54
dekker(3)	2	242	1596
dekker-fen(3,2)	2	197	730
dekker-fen(3,4)	2	293	3076
burns(1)	2	46	36
burns(2)	2	64	10150
barrier(10)	3	78	416
barrier(100)	3	348	31106
bakery(4,3)	2	142	7272
bakery(4,4)	2	157	50402
burns-fen(2)	2	72	100708

C11 has been widely accepted in real-world applications such as Firefox and Chromium web browsers, Bitcoin-core, and Tensorflow. The memory ordering constructs in C11 are general in their scope and are also being adopted in programming APIs such as CUDA and OpenMP. Nonetheless, the ordering guarantees under C11 are known to have complex axiomatic semantics. Comprehending the set of feasible outcomes under C11 is formidable even for the experts of C/C++ language.

Due to its underlying complexity, the C11 model has become a subject of intense study. Efforts have been made to manage the complexity of the model by designing useful subsets of C11 [25, 50, 59, 60], that offer more intuitive reasoning and easier

programming. Effective SMCs have also been proposed for automated detection of the set of feasible outcomes under C11 [72], and its subsets [5, 51, 79, 81], including MoCA (refer to Chapter 5).

The large reachable state space of an input program under C11 poses a challenge in its analysis . Consequently, C11 *bugs* in input programs are tough to find, and it is equally laborious to fix them. Note that, a *bug* in the input program, or a *buggy outcome* of the input program, refers to a program trace that is valid under the coherence conditions, but is regarded undesirable and may violate a user (programmer) defined property.

Table 6.1 presents a list of tests and the corresponding number of buggy outcomes (detected by a model checker for C11 called CDSChecker [32]). The column 'Benchmark name' contains the names of the tests. Columns '#threads' and 'LoC' represent the size of the input program in terms of the number of threads and the lines of code respectively. Column '#buggy traces' represents the number of buggy traces detected by CDSChecker. It can be observed that C11 programs comprising of less than 100 lines of code (after loop unrolling) and involving just two threads can generate tens, and even hundreds, of thousands of buggy traces.

A buggy outcome may emerge in the absence of a necessary ordering between program events due to the weak semantics of C11. As a result, preserving the necessary ordering between program events is essential for eliminating such bugs.

6.1.1 Ordering with `fences`

Memory barriers and `fences` are used under weak memory models to preserve ordering between program events. Consequently, careful placement of `fences` in a buggy program may exclude the bugs. However, computing the correct combination of the type and location of `fences` is challenging. Too few or incorrectly placed `fences` may not preserve the necessary ordering, while too many `fences` can negatively impact the performance. Computing the correct combination of `fences` may be as tricky as comprehending the set of program outcomes. As a result, striking a balance between

preserving the correctness and obtaining performance with `fences` is non-trivial even for expert programmers. A recent study [73] suggests that memory order specification to ensure performance and correctness should not be left to humans.

Cleverly designed automated techniques for placing `fences` in the input program may effectively tackle the challenge. The process of placing additional `fences` in the input program to eliminate program bugs is called *fence synthesis*. Most existing techniques perform `fence` synthesis for a fixed architecture. The literature on `fence` synthesis is rich with techniques targeting the TSO [6, 11, 16, 17, 21] and PSO [10, 63] memory models or both [49, 55, 67]. Techniques have also been proposed for ARM (version 7) [21], sparc-RMO [55], Power [10, 17, 33], and IA-32 [33] memory models. However, the axiomatic definition of ordering varies with memory models. As a consequence, most existing techniques (such as those for TSO and PSO) may not detect C11 buggy traces due to a strong implicit ordering. Existing techniques, parametric in or oblivious to the memory model [16, 17, 83], also assume an ordering between pairs of events that is *globally visible* (to all threads). Such an ordering constraint is restrictive for the C11 model. Another, memory model oblivious technique [49] is proposed for outcomes resulting purely from interleaving with reordering. Program outcomes under C11 may not be feasible under such a restriction.

Furthermore, `C` and `C++` are portable languages and the C11 memory model is designed for the `C/C++` abstract machine. Synthesis of architecture specific memory barriers reduces the portability of a C11 input program.

C11 `fences`

The C11 model supports *C11 fences* that serve as tools for imposing ordering restrictions between program events. The semantics of C11 `fences` are defined in the C11 model and briefly discussed in §3.2.3. The C11 `fences` do not simulate the semantics of any architectural specific memory barriers. As a result, synthesis of C11 `fences` in a `C/C++` input program maintains its portability.

For the remainder of this chapter, the event type `fence` refers to a C11 `fence`.

6.2 Ordering with C11 fences

This work proposes the first fence synthesis technique for C11 memory model. The technique takes a buggy C11 program as input and synthesizes fences in the input program that introduce additional ordering between program events. The additional ordering extends the hb_τ ordering of a buggy trace τ, as explained in Figure 3.5. The C11 coherence conditions (presented in Figure 3.6), in essence, restrict the visibility of writes, or reads from writes in presence of certain inter-thread ordering. As a consequence, in the presence of the additional inter-thread ordering with fences, a read in the buggy trace may violate a coherence constraint under C11. The buggy trace τ is then *invalidated* (represented as $\tau^{\mathtt{inv}}$); in other words, given the input program with the synthesized fences the buggy trace cannot manifest as a program execution. Further, if the synthesis of fences in the input program P invalidates all buggy traces of P then the program is *fixed* or bug-free (represented as $P^{\mathtt{fx}}$).

C11 associates fences with memory orders, and hence, supports various degrees of ordering guarantees through fences. As a consequence, various combinations of fences may fix a buggy input program (program with buggy traces), where some fence combinations may add a higher performance overhead in the fixed program than others.

Consider the input program in Figure 6.1(a), where the superscripts of read and write events represents the associated memory orders. The program is buggy since the assert condition may violate in a program execution. Figure 6.1(b) represents a buggy trace of the program, where the set of events and the event relations are diagrammatically shown. The assert condition of Figure 6.1(a) is violated because the read events are not ordered before the write events of the same object, allowing reads from *later* writes.

Consider the following three sets of fences that can invalidate the trace in Figure 6.1(b): $c_1 = \{\mathbb{F}^{\mathtt{sc}}_{12}, \mathbb{F}^{\mathtt{sc}}_{22}\}$, $c_2 = \{\mathbb{F}^{\mathtt{rel}}_{12}, \mathbb{F}^{\mathtt{acq}}_{22}\}$ and $c_3 = \{\mathbb{F}^{\mathtt{rel}}_{12}\}$, where the superscripts indicate the memory orders and the subscripts represent the synthesis locations of the fences. Each such set of fences is called a *candidate solution*. The candidate

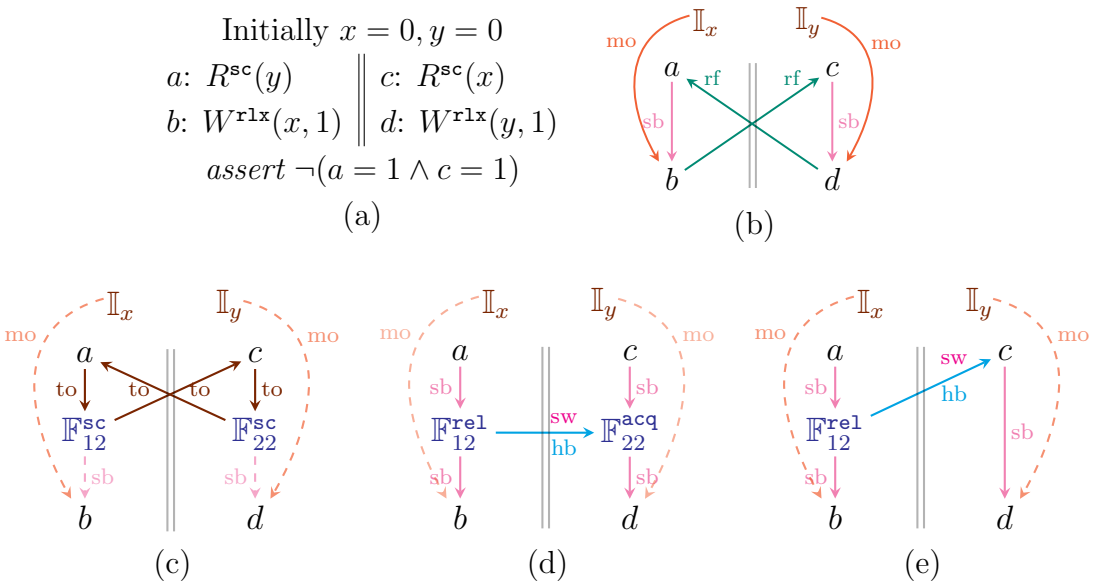


Figure 6.1: OTA. (a) input program, (b) buggy trace, (c)-(e) invalidated traces

solutions c_1, c_2 and c_3 are depicted in Figures 6.1(c), (d), and (e) respectively.

The candidate solution c_1 prevents a total order on the `sc` ordered events that does not violate the coherence condition `(coto)` of `(tosc)` (refer to Figure 3.6).

The `fences` of the candidate solution c_2 form a happens-before ordering $\mathbb{F}^{\mathtt{rel}}_{12} \rightarrow^{\mathbf{hb}} \mathbb{F}^{\mathtt{acq}}_{22}$ based on the condition depicted in Figure 3.5(d). The happens-before ordering in turn orders the `read` a before the `write` d of the same object, by forming $a \rightarrow^{\mathbf{ithb}} d$, as described in Figure 3.4. The ordering between a and d violates the coherence condition `(coRW)`, shown in Figure 3.6.

Candidate solution c_3 establishes a similar ordering $a \rightarrow^{\mathbf{ithb}} d$, however c_3 uses the strong memory order of c to form $\mathbb{F}^{\mathtt{rel}}_{12} \rightarrow^{\mathbf{hb}} c$.

All three candidate solutions can invalidate the buggy trace and fix the input program shown in Figures 6.1(a). However, c_2 and c_3 invalidate the buggy trace using weaker `fences` than c_1, and c_3 also uses the least number of `fences`. As a result, candidate solution c_3 adds the least performance overhead for fixing the input program.

6.2.1 Optimal fence synthesis

An optimal `fence` synthesis is the process of computing a candidate solution that fixes the input program while incurring the least performance overhead. The process involves finding solutions to two problems:

1. computing an optimal (minimal) set of locations to synthesize `fences`, and
2. computing an optimal (weakest) memory order to be associated with the `fences`.

The notion of optimality may vary with context. Consider two candidate solutions $\{\mathbb{F}_i^{\texttt{sc}}\}$ and $\{\mathbb{F}_j^{\texttt{rel}}, \mathbb{F}_k^{\texttt{acq}}\}$ where the superscripts represent the memory orders. The two solutions are incomparable under C11, since one solution uses a smaller set of `fences` while the other uses weaker `fences`. Further, the performance efficiency of the solutions is subject to the input program and the underlying architecture.

This work chooses a candidate solution c as an optimal solution if:

1. c has the smallest number of synthesized `fences`, and
2. each `fence` of c has the weakest memory order compared to other candidate solutions that satisfy the condition 1.

Formal definition of optimal synthesis

To formally define the optimality condition, assume that each candidate solution is assigned a weight $wt(c)$ such that, a weaker solution is assigned a lower weight. The weights are computed as the summation of the weights of `fences` in a solution, where

- a `fence` ordered `rel` or `acq` is assigned the weight 1,
- a `fence` ordered `acq-rel` is assigned the weight 2, and
- a `fence` ordered `sc` is assigned the weight 3.

Further, let $sz(c)$ represent the size of the candidate solution c, and given the set of all candidate solutions $\{c_1, ..., c_n\}$ to fix P, let $\underline{sz}(P) = \texttt{min}(sz(c_1), ..., sz(c_n))$.

Optimality for this work is formally defined as:

Definition 9. (Optimality of `fence` synthesis.)

Consider a set of candidate solutions $c_1, ..., c_n$.
A solution c_i (for $i \in [1, n]$) is considered optimal if:

1. $sz(c_i) = \underline{sz}(P) \wedge$
2. $\forall j \in [1, n]$ s.t. $sz(c_j) = \underline{sz}(P)$, $wt(c_i) \leq wt(c_j)$.

Consider the candidate solutions $c_1 = \{\mathbb{F}^{\texttt{sc}}_{12}, \mathbb{F}^{\texttt{sc}}_{22}\}$, $c_2 = \{\mathbb{F}^{\texttt{rel}}_{12}, \mathbb{F}^{\texttt{acq}}_{22}\}$ and $c_3 = \{\mathbb{F}^{\texttt{rel}}_{12}\}$, shown in Figures 6.1(c), (d) and (e) respectively. The weights of the solutions are computed as: $wt(c_1) = 3+3 = 6$, $wt(c_2) = 1+1 = 2$, $wt(c_3) = 1$, and their sizes are $sz(c_1) = 2$, $sz(c_2) = 2$, $sz(c_3) = 1$. Accordingly $\underline{sz}(P) = \texttt{min}(2,2,1) = 1 = sz(c_3)$. Hence, according to Definition 9, c_3 represents the optimal solution.

Further, consider again the two candidate solutions $c_i = \{\mathbb{F}^{\texttt{sc}}_i\}$ and $c_{jk} = \{\mathbb{F}^{\texttt{rel}}_j, \mathbb{F}^{\texttt{acq}}_k\}$; where, $wt(c_i) = 3$, $wt(c_{jk}) = 2$, and $sz(c_i) = 1$, $sz(c_{jk}) = 2$. Hence, according to Definition 9, c_i represents the optimal solution of the two (since, $\underline{sz}(P) = \texttt{min}(2,1) = 1 = sz(c_i)$).

Therefore, this work prefers candidate solutions that synthesize lesser number of `fences` over weakly ordered solutions with more `fences`.

Optimal `fence` synthesis is a computationally expensive process. Optimal `fence` synthesis with multiple types of `fences` (as with `C11 fences`) can be reduced from the minimum set cover problem. Using the reduction, it is shown that optimal `fence` synthesis problem with multiple types of `fences` is NP-hard even for straight-line programs [83].

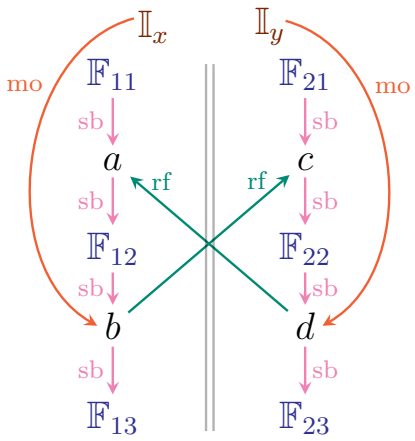


Figure 6.2: OTA-imm. Intermediate trace of the buggy trace in Figure 6.1(b)

6.3 Invalidating buggy trace with C11 `fences`

6.3.1 Intermediate trace

Given a buggy trace of the input program τ, consider its transformation into an *intermediate trace* ($\tau^{\texttt{imm}}$) generated by inserting *untyped* `fences` above and below each memory access event (`read`, `write`, and `rmw`) in the trace. The `fences` inserted in the program are called *candidate* ***`fences`***. For instance, Figure 6.2 represents the intermediate trace corresponding to the buggy trace shown in Figure 6.1(b), that contains six candidate `fences`.

The candidate `fences` inflate the relations sb_τ, sw_τ, and dob_τ with the additional ordering introduced with `fences` (assuming they are strongly ordered). The corresponding relations for the intermediate trace are represented by $\text{sb}_{\tau^{\texttt{imm}}}$, $\text{sw}_{\tau^{\texttt{imm}}}$, and $\text{dob}_{\tau^{\texttt{imm}}}$. The ithb_τ and hb_τ relations (defined over sb_τ, sw_τ, and dob_τ) are accordingly updated to $\text{ithb}_{\tau^{\texttt{imm}}}$ and $\text{hb}_{\tau^{\texttt{imm}}}$ (defined over $\text{sb}_{\tau^{\texttt{imm}}}$, $\text{sw}_{\tau^{\texttt{imm}}}$, and $\text{dob}_{\tau^{\texttt{imm}}}$). `Fences` do not contribute to the relations mo_τ and rf_τ, hence, $\text{mo}_{\tau^{\texttt{imm}}} = \text{mo}_\tau$ and $\text{rf}_{\tau^{\texttt{imm}}} = \text{rf}_\tau$. Additionally, `sc` ordered `fences` also contribute to the total order on `sc` events. The total order for the intermediate trace is represented as $\text{to}_{\tau^{\texttt{imm}}}$.

The intermediate version contains a `fence` at each feasible location of synthesis. As a result, the relations $\mathrm{hb}_{\tau^{\mathtt{imm}}}$, $\mathrm{mo}_{\tau^{\mathtt{imm}}}$, and $\mathrm{rf}_{\tau^{\mathtt{imm}}}$ represent the maximal ordering that can be introduced by synthesizing `fences`. Therefore, if it is feasible to invalidate a buggy trace with C11 `fences` then each candidate solution is within the corresponding intermediate trace.

6.3.2 Weak-FenSying. Invalidating buggy trace with weak `fences`

C11 coherence conditions that describe incoherence of hb_τ relation *wrt* the mo_τ and rf_τ can be violated with weak `fences`, where weak `fences` represents the set {`rel`, `acq`, `acq-rel`}. The coherence condition that constitute this category are `(corf)`, `(coWW)`, `(coWR)`, `(coRW)`, and `(coRR)`, presented formally in Figure 3.6.

Recall that, the conditions can be interpreted as the conjunction of the following constraints [60], (refer to §3.2.3).

hb_τ is irreflexive.	(`co-h`)
$\mathrm{rf}_\tau; \mathrm{hb}_\tau$ is irreflexive.	(`co-rh`)
$\mathrm{mo}_\tau; \mathrm{hb}_\tau$ is irreflexive.	(`co-mh`)
$\mathrm{mo}_\tau; \mathrm{rf}_\tau; \mathrm{hb}_\tau$ is irreflexive.	(`co-mrh`)
$\mathrm{mo}_\tau; \mathrm{hb}_\tau; \mathrm{rf}_\tau^{-1}$ is irreflexive.	(`co-mhi`)
$\mathrm{mo}_\tau; \mathrm{rf}_\tau; \mathrm{hb}_\tau; \mathrm{rf}_\tau^{-1}$ is irreflexive.	(`co-mrhi`)

Candidate solutions that violate the coherence conditions of this category can be computed by detecting reflexive ordering in the relation compositions of the above stated constraints. The problem can be viewed as a cycle detection problem in the compositions of relations $\mathrm{hb}_{\tau^{\mathtt{imm}}}$, $\mathrm{mo}_{\tau^{\mathtt{imm}}}$, and $\mathrm{rf}_{\tau^{\mathtt{imm}}}$ corresponding to the constraints. Weak-FenSying assumes the memory order `acq-rel` for candidate `fences` for computing the $\mathrm{hb}_{\tau^{\mathtt{imm}}}$ ordering with `fences`.

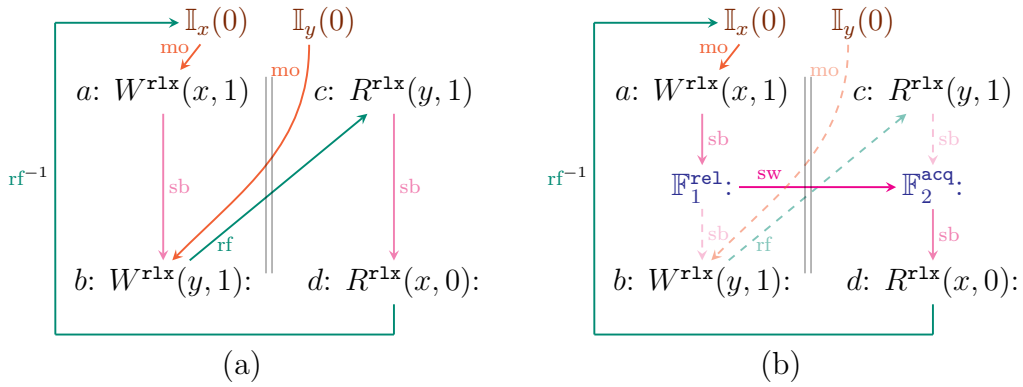


Figure 6.3: MP-invalidated. (a) buggy trace, (b) invalidated trace

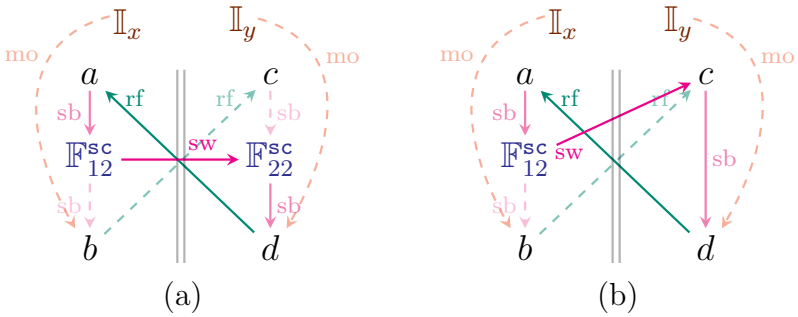


Figure 6.4: OTA-invalidated. (a), (b) invalidated traces of trace in Figure 6.1(a)

Consider the buggy trace, τ, in Figure 6.3(a), where $\mathbb{I}_x(0)$ and $\mathbb{I}_y(0)$ represent the initial values for the objects x and y. The corresponding intermediate trace $\tau^{\texttt{imm}}$ would form an inter-thread ordering using `fences` $\mathbb{F}_1^{\texttt{rel}}$ and $\mathbb{F}_2^{\texttt{acq}}$ at the locations shown in Figure 6.3(b). The ordering on the `fences` forms a reflexive ordering in $\mathbb{I}_x(0) \rightarrow^{\texttt{mo}}_{\tau^{\texttt{imm}}} a \rightarrow^{\texttt{hb}}_{\tau^{\texttt{imm}}} d \rightarrow^{rf^{-1}}_{\tau^{\texttt{imm}}} \mathbb{I}_x(0)$, where $a \rightarrow^{\texttt{hb}}_{\tau^{\texttt{imm}}} d$ ordering is formed using candidate `fences` as, $a \rightarrow^{\texttt{sb}}_{\tau^{\texttt{imm}}} \mathbb{F}_1^{\texttt{rel}} \rightarrow^{\texttt{sw}}_{\tau^{\texttt{imm}}} \mathbb{F}_2^{\texttt{acq}} \rightarrow^{\texttt{sb}}_{\tau^{\texttt{imm}}} d$ (since, $\text{sb}_{\tau^{\texttt{imm}}}$ and $\text{sw}_{\tau^{\texttt{imm}}}$ contribute to $\text{hb}_{\tau^{\texttt{imm}}}$). The reflexive ordering violates $\text{mo}_{\tau^{\texttt{imm}}}; \text{hb}_{\tau^{\texttt{imm}}}; \text{rf}^{-1}_{\tau^{\texttt{imm}}}$ irreflexivity (constraint `(co-mhi)`), invalidating the buggy trace τ, as shown in Figure 6.3(b). Note that, the intermediate trace has a `fence` at every feasible program location, however for readability, the figure only shows the `fences` relevant for the ordering.

Similarly, Figures 6.4(a) and (b) show two candidate solutions that invalidate the

buggy trace shown in Figure 6.1(b) using weak `fences`. Ordering between `fences` $\mathbb{F}^{\texttt{rel}}_{12}$ and $\mathbb{F}^{\texttt{acq}}_{22}$ forms a reflexive ordering in the $\text{rf}_{\tau^{\text{imm}}}; \text{hb}_{\tau^{\text{imm}}}$ relation composition (constraint `(co-rh)`), invalidating the buggy trace, as shown in Figure 6.4(a). Figure 6.4(b) shows another candidate solution that violates the `(co-rh)` coherence constraint with a single `fence`. Note that, the candidate solutions in Figures 6.4(a) and (b) represent the same solutions as in Figures 6.1(d) and (e) respectively, with additional details.

Lemma 2. Weak-FenSying is sound.
Let $\mathbb{C} = \{\ \text{hb}_{\tau^{\text{imm}}},\ \text{rf}_{\tau^{\text{imm}}}; \text{hb}_{\tau^{\text{imm}}},\ \text{mo}_{\tau^{\text{imm}}}; \text{rf}_{\tau^{\text{imm}}}; \text{hb}_{\tau^{\text{imm}}},\ \text{mo}_{\tau^{\text{imm}}}; \text{hb}_{\tau^{\text{imm}}},\ \text{mo}_{\tau^{\text{imm}}}; \text{hb}_{\tau^{\text{imm}}}; \text{rf}^{-1}_{\tau^{\text{imm}}},$
$\text{mo}_{\tau^{\text{imm}}}; \text{rf}_{\tau^{\text{imm}}}; \text{hb}_{\tau^{\text{imm}}}; \text{rf}^{-1}_{\tau^{\text{imm}}}\ \}$.
Given a buggy trace τ of input program P, let $\texttt{weakCycles}_{\tau}$ represent the set of cycles detected by Weak-FenSying for $\tau^{\texttt{imm}}$. The lemma can be formally stated as,

$$\exists \mathit{cond} \in \mathbb{C} \ \text{ s.t. } \mathit{cond} \text{ is reflexive} \Leftrightarrow \texttt{weakCycles}_{\tau} \neq \emptyset.$$

(There exists a violation of a coherence condition *if-and-only-if* Weak-FenSying detects a cycle in the corresponding relation compositions.)

Proof. Case $\Rightarrow$: $\exists \mathit{cond} \in \mathbb{C}$ s.t. *cond* is reflexive $\Rightarrow \texttt{weakCycles}_{\tau} \neq \emptyset$.

Assume that, given a set of ordered event pairs, the technique soundly detects all cycles. (A1)

Then, Weak-FenSying is sound if the event relations $\text{hb}_{\tau^{\text{imm}}}$, $\text{rf}_{\tau^{\text{imm}}}$, $\text{mo}_{\tau^{\text{imm}}}$, and $\text{rf}^{-1}_{\tau^{\text{imm}}}$ are correctly computed, *i.e.* $\nexists e_1, e_2 \in \mathcal{E}_{\tau^{\text{imm}}}$ s.t. a cycle would be formed containing an ordering of e_1, e_2 but the pair is not in the corresponding relation $\text{hb}_{\tau^{\text{imm}}}$ or $\text{rf}_{\tau^{\text{imm}}}$ or $\text{mo}_{\tau^{\text{imm}}}$ or $\text{rf}^{-1}_{\tau^{\text{imm}}}$.

Given a buggy trace τ, Weak-FenSying computes the relation $\text{hb}_{\tau^{\text{imm}}}$ using candidate `fences`, while $\text{rf}_{\tau^{\text{imm}}}$, $\text{mo}_{\tau^{\text{imm}}}$, and $\text{rf}^{-1}_{\tau^{\text{imm}}}$ are same as the corresponding relations of τ.

Since, computation of $\text{hb}_{\tau^{\text{imm}}}$ using `fences` is defined under C11 (refer to §3.2.3) , the soundness condition can be defined as:
Weak-FenSying soundly detects all weak cycle without recomputing $\text{rf}_{\tau^{\text{imm}}}$, $\text{mo}_{\tau^{\text{imm}}}$, and $\text{rf}^{-1}_{\tau^{\text{imm}}}$ relations for the events of $\tau^{\texttt{imm}}$.

rf_τ The relation is formed from `write` (or `rmw`) events to `read` (or `rmw`) events, since `fences` cannot be both, the rf_τ relations remains unchanged *i.e.* $\mathrm{rf}_{\tau^{\mathtt{imm}}} = \mathrm{rf}_\tau$.

rf_τ^{-1} The relation remains unchanged as rf_τ remains unchanged *i.e.* $\mathrm{rf}^{-1}_{\tau^{\mathtt{imm}}} = \mathrm{rf}^{-1}_\tau$.

mo_τ Assume $\exists w, w' \in \mathcal{E}_\tau$ s.t. as a consequence of synthesizing `fences` in the buggy trace τ to form $\tau^{\mathtt{imm}}$, w is *modification-ordered before* w'. However, $w \not\rightarrow^{\mathbf{mo}}_{\tau^{\mathtt{imm}}} w'$ since we consider $\mathrm{mo}_{\tau^{\mathtt{imm}}} = \mathrm{mo}_\tau$.

Consider the following four cases of coherence involving mo_τ (refer to Figure 3.6).

`(coWW)` Let $\exists w_1, w_2 \in \mathcal{E}^{\mathbb{W}}$ s.t. $w_1 \rightarrow^{\mathbf{hb}}_{\tau^{\mathtt{imm}}} w_2$.
If $w_1 \rightarrow^{\mathbf{mo}}_\tau w_2$ then there does not exist a violation.
However, if $w_2 \rightarrow^{\mathbf{mo}}_\tau w_1$ then a cycle is detected in $\mathrm{mo}_{\tau^{\mathtt{imm}}}; \mathrm{hb}_{\tau^{\mathtt{imm}}}$ violating the constraint `(co-mh)`.

`(coWR)`: Let $\exists r_1 \in \mathcal{E}^{\mathbb{R}}$, $\exists w_1, w_2 \in \mathcal{E}^{\mathbb{W}}$ s.t. $w_1 \rightarrow^{\mathbf{hb}}_{\tau^{\mathtt{imm}}} r_1$ and $w_2 \rightarrow^{\mathbf{rf}}_{\tau^{\mathtt{imm}}} r_1$.
If $w_1 \rightarrow^{\mathbf{mo}}_\tau w_2$ then there does not exist a violation.
However, if $w_2 \rightarrow^{\mathbf{mo}}_\tau w_1$ then a cycle is detected in $\mathrm{mo}_{\tau^{\mathtt{imm}}}; \mathrm{hb}_{\tau^{\mathtt{imm}}}; \mathrm{rf}^{-1}_{\tau^{\mathtt{imm}}}$ violating the constraint `(co-mhi)`.

`(coRW)`: Let $\exists r_1 \in \mathcal{E}^{\mathbb{R}}$, $\exists w_1, w_2 \in \mathcal{E}^{\mathbb{W}}$ s.t. $w_1 \rightarrow^{\mathbf{rf}}_{\tau^{\mathtt{imm}}} r_1$ and $r_1 \rightarrow^{\mathbf{hb}}_{\tau^{\mathtt{imm}}} w_2$.
If $w_1 \rightarrow^{\mathbf{mo}}_\tau w_2$ then there does not exist a violation.
However, if $w_2 \rightarrow^{\mathbf{mo}}_\tau w_1$ then a cycle is detected in $\mathrm{mo}_{\tau^{\mathtt{imm}}}; \mathrm{rf}_{\tau^{\mathtt{imm}}}; \mathrm{hb}_{\tau^{\mathtt{imm}}}$ violating the constraint `(co-mrh)`.

`(coRR)`: Let $\exists r_1, r_2 \in \mathcal{E}^{\mathbb{R}}$, $\exists w_1, w_2 \in \mathcal{E}^{\mathbb{W}}$ s.t. $w_1 \rightarrow^{\mathbf{rf}}_{\tau^{\mathtt{imm}}} r_1$, $w_2 \rightarrow^{\mathbf{rf}}_{\tau^{\mathtt{imm}}} r_2$ and $r_1 \rightarrow^{\mathbf{hb}}_{\tau^{\mathtt{imm}}} r_2$.
If $w_1 \rightarrow^{\mathbf{mo}}_\tau w_2$ then there does not exist a violation.
However, if $w_2 \rightarrow^{\mathbf{mo}}_\tau w_1$ then a cycle is detected in $\mathrm{mo}_{\tau^{\mathtt{imm}}}; \mathrm{rf}_{\tau^{\mathtt{imm}}}; \mathrm{hb}_{\tau^{\mathtt{imm}}}; \mathrm{rf}^{-1}_{\tau^{\mathtt{imm}}}$ violating the constraint `(co-mrhi)`.

Thus, Weak-FenSying does not miss a cycle in an irreflexive coherence constraint.

Case $\Leftarrow$: $\exists cond \in \mathbb{C}$ s.t. *cond* is reflexive $\Leftarrow$ $\texttt{weakCycles}_{\tau} \neq \emptyset$.

The computation of $\text{hb}_{\tau^{\texttt{imm}}}$ using `fences` is defined under C11 (refer to §3.2.3). Further, $\text{mo}_{\tau^{\texttt{imm}}}$ and $\text{rf}_{\tau^{\texttt{imm}}}$ are the same as the corresponding relations of τ.

Thus, if Weak-FenSying detects a cycle $e \rightarrow^{C} e$ where $e \in \mathcal{E}_{\tau^{\texttt{imm}}}$ and $C \in \mathbb{C}$ then the condition C is indeed reflexive.

Hence, if Weak-FenSying detects a weak cycle then a corresponding coherence condition is violated. □

6.3.3 Strong-FenSying. Invalidating buggy trace with strong `fences`

C11 coherence conditions that describe the necessary constraints of a total order on `sc` ordered events can be violated only with strong `sc` ordered `fences`. Hence, this technique assumes that all candidate `fences` have the memory order `sc`. The coherence condition that constitute this category are `(tosc)` (including `(coto)`, `(rfto1)`, and `(rfto2)`), and `(tofen)` , presented formally in Figure 3.6.

Intuitively, Strong-FenSying constructs an ordering on `sc` ordered events in coherence with the $\text{hb}_{\tau^{\texttt{imm}}}$, $\text{mo}_{\tau^{\texttt{imm}}}$ and $\text{rf}_{\tau^{\texttt{imm}}}$ relations in the intermediate trace $\tau^{\texttt{imm}}$. If the ordering is reflexive then it implies that a valid total order cannot be formed on the `sc` ordered events of $\tau^{\texttt{imm}}$. Thus, Strong-FenSying invalidates a buggy trace τ by introducing additional `sc` ordering with `sc fences` to prohibit a total order on the `sc` events.

Strong-FenSying introduces a relation $\text{so}_{\tau^{\texttt{imm}}}$ (called *sc-order*) such that a total order cannot be formed on the `sc` events of $\tau^{\texttt{imm}}$ iff a cycle exists in $\text{so}_{\tau^{\texttt{imm}}}$. Thus, Strong-FenSying can also be viewed as a cycle detection problem, in the relation $\text{so}_{\tau^{\texttt{imm}}}$.

Consider an irreflexive relation called *from-reads* for ordering `reads` with *later* `writes`, computed as

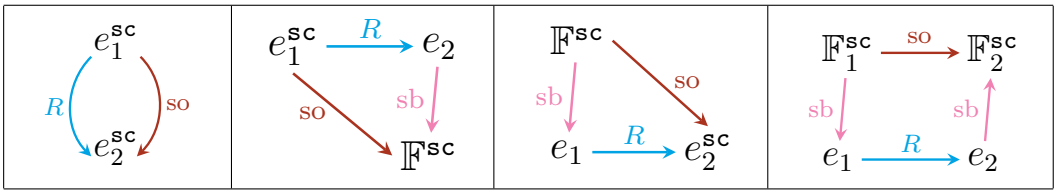


Figure 6.5: Conditions for construction $so_{\tau^{imm}}$ relation

$$fr_{\tau^{imm}} \triangleq rf^{-1}_{\tau^{imm}}; mo_{\tau^{imm}}$$

The relation $fr_{\tau^{imm}}$ supports the $hb_{\tau^{imm}}$, $mo_{\tau^{imm}}$, and $rf_{\tau^{imm}}$ relations in computing the $so_{\tau^{imm}}$ relation. The relation from-reads is typically used for stronger memory models such as the sequentially-consistent memory model (refer to §3.2.1) that relates all events of a trace by a total order. Strong-FenSying introduces the relation under C11, since $to_{\tau^{imm}}$ constitutes a similar requirement on the `sc` ordered events of an intermediate trace.

The rules for constructing the $so_{\tau^{imm}}$ relation are formally presented in Definition 10 and diagrammatically represented in Figure 6.5.

Definition 10. (`sc`-order ($so_{\tau^{imm}}$))

$$\forall e_1, e_2 \in \mathcal{E}_{\tau^{imm}} \text{ s.t. } (e_1, e_2) \in R, \text{ where } R = hb_{\tau^{imm}} \cup mo_{\tau^{imm}} \cup rf_{\tau^{imm}} \cup fr_{\tau^{imm}}$$

- if $e_1, e_2 \in \mathcal{E}^{(sc)}_{\tau^{imm}}$ then $e_1 \rightarrow^{so}_{\tau^{imm}} e_2$; (soee)
- if $e_1 \in \mathcal{E}^{(sc)}_{\tau^{imm}}$, $\exists \mathbb{F}^{sc} \in \mathcal{E}^{\mathbb{F}(sc)}_{\tau^{imm}}$ s.t. $e_2 \rightarrow^{sb}_{\tau^{imm}} \mathbb{F}^{sc}$ then $e_1 \rightarrow^{so}_{\tau^{imm}} \mathbb{F}^{sc}$; (soef)
- if $e_2 \in \mathcal{E}^{(sc)}_{\tau^{imm}}$, $\exists \mathbb{F}^{sc} \in \mathcal{E}^{\mathbb{F}(sc)}_{\tau^{imm}}$ s.t. $\mathbb{F}^{sc} \rightarrow^{sb}_{\tau^{imm}} e_1$ then $\mathbb{F}^{sc} \rightarrow^{so}_{\tau^{imm}} e_2$; (sofe)
- if $\exists \mathbb{F}^{sc}_1, \mathbb{F}^{sc}_2 \in \mathcal{E}^{\mathbb{F}(sc)}_{\tau^{imm}}$ s.t. $\mathbb{F}^{sc}_1 \rightarrow^{sb}_{\tau^{imm}} e_1$ and $e_2 \rightarrow^{sb}_{\tau^{imm}} \mathbb{F}^{sc}_2$ then $\mathbb{F}^{sc}_1 \rightarrow^{so}_{\tau^{imm}} \mathbb{F}^{sc}_2$. (soff)

Consider the buggy trace, τ, in Figure 6.6(a). Figure 6.6(b) shows a reflexive $so_{\tau^{imm}}$ ordering using `fences` $\mathbb{F}^{sc}_1$ and $\mathbb{F}^{sc}_2$: $a \rightarrow^{so}_{\tau^{imm}} \mathbb{F}^{sc}_1 \rightarrow^{so}_{\tau^{imm}} c \rightarrow^{so}_{\tau^{imm}} \mathbb{F}^{sc}_2 \rightarrow^{so}_{\tau^{imm}} a$, where $a \rightarrow^{so}_{\tau^{imm}} \mathbb{F}^{sc}_1$ and $c \rightarrow^{so}_{\tau^{imm}} \mathbb{F}^{sc}_2$ are formed from the condition (soee) (due to $a \rightarrow^{sb}_{\tau^{imm}} \mathbb{F}^{sc}_1$ and $c \rightarrow^{sb}_{\tau^{imm}} \mathbb{F}^{sc}_2$

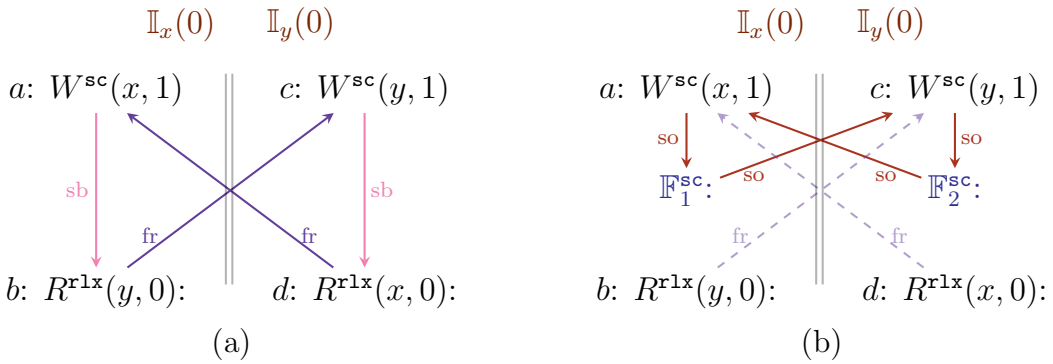


Figure 6.6: SB-invalidated. (a) buggy trace, (b) invalidated trace

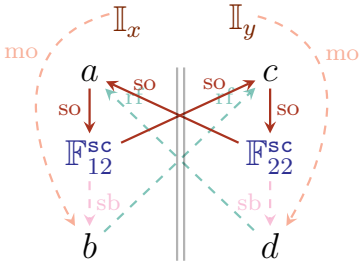


Figure 6.7: Invalidated traces of buggy trace in Figure 6.1(a)

respectively), and $\mathbb{F}^{\mathtt{sc}}_1 \rightarrow^{\mathtt{so}}_{\tau^{\mathtt{imm}}} c$ and $\mathbb{F}^{\mathtt{sc}}_2 \rightarrow^{\mathtt{so}}_{\tau^{\mathtt{imm}}} a$ are formed from the condition `(sofe)` (due to $b \rightarrow^{\mathtt{fr}}_{\tau^{\mathtt{imm}}} c$ and $d \rightarrow^{\mathtt{fr}}_{\tau^{\mathtt{imm}}} a$ respectively).

Similarly, Figure 6.7 shows a candidate solution that invalidates the buggy trace shown in Figure 6.1(b) using strong `fences`, where the $\mathrm{so}_{\tau^{\mathtt{imm}}}$ relations are formed from the conditions `(soee)` and `(sofe)`.

Observations on $\mathrm{so}_{\tau^{\mathtt{imm}}}$

1. Pairs of `sc` events that do not have a definite order are not ordered by $\mathrm{so}_{\tau^{\mathtt{imm}}}$.

   Consider the buggy trace in Figure 6.6(a), $a \rightarrow^{\mathtt{to}} c$ and $c \rightarrow^{\mathtt{to}} a$ are both valid total orders on the `sc` events of the trace. The set $\mathrm{so}_{\tau^{\mathtt{imm}}}$ does not contain either of the two event pairs.

The intuition for not ordering such pairs of events is that if the pair is involved in a cycle then their order can be freely flipped to eliminate the cycle. As a consequence, such pairs of events cannot contribute to the reflexivity of $\text{so}_{\tau^{\text{imm}}}$ and can be safely ignored.

The observation is formally presented in Lemma 4.

2. As a consequence of observation (1), $\text{so}^{+}_{\tau^{\text{imm}}} \subseteq \text{to}_{\tau^{\text{imm}}}$.

   The observation is formally presented as Lemma 3.

Lemma 3. $\text{so}^{+}_{\tau^{\text{imm}}} \subseteq \text{to}_{\tau^{\text{imm}}}$
For any valid C11 trace τ, each pair of events related by $\text{so}_{\tau^{\text{imm}}}$ are also ordered by $\text{to}_{\tau^{\text{imm}}}$.
As a consequence, $\text{so}_{\tau^{\text{imm}}}$ does not order events if the ordering violates `(to-sc)`.

Proof. Let $\exists e_1, e_2 \in \mathcal{E}_{\tau^{\text{imm}}}$ s.t. $e_1 \rightarrow^{\textbf{so}}_{\tau^{\text{imm}}} e_2$. Consider the exhaustive four conditions of constructing $e_1 \rightarrow^{\textbf{so}}_{\tau^{\text{imm}}} e_2$ (by definition of $\text{so}_{\tau^{\text{imm}}}$),

soee: $e_1, e_2 \notin \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\text{imm}}} \Rightarrow e_1 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_2$ (since $e_2 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_1$ violates `(coto)` or `(rfto1)`).

soef: $e_1 \notin \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\text{imm}}}, e_2 \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\text{imm}}} \Rightarrow e_1 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_2$ (since $e_2 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_1$ violates `(coto)` or `(rfto1)`).

sofe: $e_1 \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\text{imm}}}, e_2 \notin \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\text{imm}}} \Rightarrow e_1 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_2$ (since $e_2 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_1$ violates `(coto)` or `(rfto1)` or `(rfto2)`).

soff: $e_1, e_2 \in \mathcal{E}^{\mathbb{F}(\mathtt{sc})}_{\tau^{\text{imm}}} \Rightarrow e_1 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_2$ (since $e_2 \rightarrow^{\textbf{to}}_{\tau^{\text{imm}}} e_1$ violates `(coto)` or `(tofen)`).

Since, $\text{to}_{\tau^{\text{imm}}}$ is total, thus, $\text{so}^{+}_{\tau^{\text{imm}}} \subseteq \text{to}_{\tau^{\text{imm}}}$. □

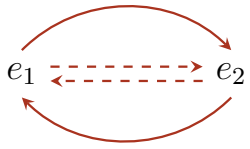
Lemma 4. Strong-FenSying is sound:
Given a buggy trace τ of input program P, let $\texttt{strongCycles}_{\tau}$ represent the set of cycles detected by Strong-FenSying in $\text{so}_{\tau^{\text{imm}}}$.
$\neg(\mathit{total}(\mathcal{E}^{(\mathtt{sc})}, \text{to}_{\tau^{\text{imm}}}) \wedge \mathit{order}(\mathcal{E}^{(\mathtt{sc})}, \text{to}_{\tau^{\text{imm}}})) \Leftrightarrow \texttt{strongCycles}_{\tau} \neq \emptyset$.
There does not exist a total order on the `sc` ordered events of an intermediate trace τ^{imm} *if-and-only-if* there exists a cycle in $\text{so}_{\tau^{\text{imm}}}$.

Proof. Assume that, given a set $\mathrm{so}_{\tau^{\mathtt{imm}}}$, the technique soundly detects all cycles. (A1)

Case $\Rightarrow$: $\neg(total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}}) \wedge order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}})) \Rightarrow \mathtt{strongCycles}_{\tau} \neq \emptyset$.
Consider $e_1, e_2 \in \mathcal{E}_{\tau}^{(\mathtt{sc})}$ s.t. both $e_1 \rightarrow_{\tau}^{\mathbf{to}} e_2$ and $e_2 \rightarrow_{\tau}^{\mathbf{to}} e_1$ do not violate the conditions `(coto)`, `(rfto1)`, `(rfto2)` and `(tofen)`. To form the total order we can assume either one of the two orders [46]. Assume $e_1 \rightarrow_{\tau}^{\mathbf{to}} e_2$.

Further, consider a total order cannot be formed on `sc` events of $\tau^{\mathtt{imm}}$ s.t. $e \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} \ldots \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_2 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} \ldots \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e$ then we simply flip $e_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_2$ to $e_2 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_1$ and eliminate the cycle.

Further, if a cycle in $\mathrm{to}_{\tau^{\mathtt{imm}}}$ includes $e_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_2$ and another cycle includes $e_2 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_1$ then there exists a cycle $e_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} \ldots \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_2 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} \ldots \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e_1$ (by by definition of $\mathrm{to}_{\tau^{\mathtt{imm}}}$), as shown in the figure below). Thus, pairs of `sc` ordered events that don't have a fixed $\mathrm{to}_{\tau^{\mathtt{imm}}}$ order cannot contribute to a strong cycle. inf(i).



Now, if there does not exist a total order on the `sc` ordered events of $\tau^{\mathtt{imm}}$ then

$\neg(total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}}) \wedge order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}}))$, *i.e.*

$\neg(total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}})) \vee \exists e \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{(\mathtt{sc})}$ s.t. $e \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e \vee \neg(\mathrm{to}_{\tau^{\mathtt{imm}}}^{+} \subseteq \mathrm{to}_{\tau^{\mathtt{imm}}})$

(by definition of $order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}})$, Figure 3.6).

By definition of $\mathrm{to}_{\tau^{\mathtt{imm}}}$, $\neg total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}})$ and $\neg(\mathrm{to}_{\tau^{\mathtt{imm}}}^{+} \subseteq \mathrm{to}_{\tau^{\mathtt{imm}}})$ are not feasible.

Thus, $\exists e \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{(\mathtt{sc})}$ s.t. $e \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e$
$\Rightarrow$ `sc` events of $\tau^{\mathtt{imm}}$ violate `(coto)`, `(rfto1)`, `(rfto2)` or `(tofen)`.

[coto] Let $\exists e^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{(\mathtt{sc})}$ s.t. $(e, e) \in (\mathrm{hb}_{\tau^{\mathtt{imm}}} \cup \mathrm{mo}_{\tau^{\mathtt{imm}}})$. Thus, `(coto)` is violated by $e^{\mathtt{sc}}$.

Since, $\mathrm{mo}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \cup \mathrm{hb}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \subseteq \mathrm{so}_{\tau^{\mathtt{imm}}}^{+}$ thus a cycle $e^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} \ldots \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} e^{\mathtt{sc}}$ is formed.

[rfto1] Let $\exists w_1^{\mathtt{sc}}, w_2^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{W}(\mathtt{sc})}$, $r_1^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{R}(\mathtt{sc})}$ s.t. $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} r_1^{\mathtt{sc}}$ and $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{rf}} r_1^{\mathtt{sc}}$.
Thus, `(rfto1)` is violated by $w_1^{\mathtt{sc}}$, $w_2^{\mathtt{sc}}$ and $r_1^{\mathtt{sc}}$.

Since, τ is a valid trace, $\neg w_1^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} w_2^{\mathtt{sc}} \vee \neg w_2^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} r_1^{\mathtt{sc}}$.

Further, since inserting `fences` only modifies the hb_{τ} relation, if $\neg w_1^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} w_2^{\mathtt{sc}}$ then $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} w_2^{\mathtt{sc}}$ (because $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} w_2^{\mathtt{sc}}$). Similarly, if $\neg w_2^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} r_1^{\mathtt{sc}}$ then $w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} r_1^{\mathtt{sc}}$.

Also, $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} w_2^{\mathtt{sc}} \Rightarrow w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{mo}} w_2^{\mathtt{sc}}$ (assuming `(co-mh)` is not violated) $\Rightarrow r_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{fr}} w_2^{\mathtt{sc}}$.

Since, $\mathrm{fr}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \cup \mathrm{hb}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \subseteq \mathrm{so}_{\tau^{\mathtt{imm}}}^{+}$ thus a cycle $r_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} r_1^{\mathtt{sc}}$ is formed.

[rfto2] Let $\exists w_1, w_2^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{W}}$, $r_1^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{R}(\mathtt{sc})}$ s.t. $ord(w_2^{\mathtt{sc}})$ is $\mathtt{sc}$, $w_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{rf}} r_1^{\mathtt{sc}}$, $w_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} w_2^{\mathtt{sc}}$ and $w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} r_1^{\mathtt{sc}}$.
Thus, `(rfto2)` is violated by w_1, $w_2^{\mathtt{sc}}$ and $r_1^{\mathtt{sc}}$.

Since, τ is a valid trace, $\neg w_1 \rightarrow_{\tau}^{\mathbf{hb}} w_2^{\mathtt{sc}} \vee \neg w_2^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} r_1^{\mathtt{sc}}$.

Further, since inserting `fences` only modifies the $\mathrm{hb}_{\tau^{\mathtt{imm}}}$ relation, if $\neg w_2^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} r_1^{\mathtt{sc}}$ then $w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} r_1^{\mathtt{sc}}$ (because $w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} r_1^{\mathtt{sc}}$).

Also, $w_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} w_2^{\mathtt{sc}} \Rightarrow w_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{mo}} w_2^{\mathtt{sc}}$ (assuming `(co-mh)` is not violated) $\Rightarrow r_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{fr}} w_2^{\mathtt{sc}}$.

Since, $\mathrm{fr}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \cup \mathrm{hb}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \subseteq \mathrm{so}_{\tau^{\mathtt{imm}}}^{+}$ thus a cycle $r_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} r_1^{\mathtt{sc}}$ is formed.

[tofen] Let $\exists w_1^{\mathtt{sc}}, w_2^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{W}(\mathtt{sc})}$, $r_1 \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{R}}$, $\mathbb{F}^{\mathtt{sc}} \in \mathcal{E}_{\tau^{\mathtt{imm}}}^{\mathbb{F}(\mathtt{sc})}$ s.t. $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} \mathbb{F}^{\mathtt{sc}}$, $\mathbb{F}^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{sb}} r_1$ and $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{rf}} r_1$.
Thus, `(tofen)` is violated by $w_1^{\mathtt{sc}}$, $w_2^{\mathtt{sc}}$, r_1 and $\mathbb{F}^{\mathtt{sc}}$.

Since, τ is a valid trace, $\neg w_1^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} w_2^{\mathtt{sc}} \vee \neg w_2^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} \mathbb{F}^{\mathtt{sc}}$.

Further, since inserting `fences` only modifies the $\mathrm{hb}_{\tau^{\mathtt{imm}}}$ relation, if $\neg w_1^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} w_2^{\mathtt{sc}}$ then $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} w_2^{\mathtt{sc}}$ (because $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} w_2^{\mathtt{sc}}$). Similarly, if $\neg w_2^{\mathtt{sc}} \rightarrow_{\tau}^{\mathbf{to}} \mathbb{F}^{\mathtt{sc}}$ then $w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{hb}} \mathbb{F}^{\mathtt{sc}}$.

Also, $w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} w_2^{\mathtt{sc}} \Rightarrow w_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{mo}} w_2^{\mathtt{sc}}$ (assuming `(co-mh)` is not violated) $\Rightarrow r_1 \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{fr}} w_2^{\mathtt{sc}} \Rightarrow \mathbb{F}^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} w_2^{\mathtt{sc}}$ (using `(sofe)`).

Since, $\mathrm{hb}_{\tau^{\mathtt{imm}}}|_{\mathtt{sc}} \subseteq \mathrm{so}_{\tau^{\mathtt{imm}}}$ thus a cycle $r_1^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} w_2^{\mathtt{sc}} \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}} r_1^{\mathtt{sc}}$ is formed.

Case $\Leftarrow$: $\neg(total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}}) \wedge order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}})) \Leftarrow \mathtt{strongCycles}_{\tau} \neq \emptyset$.
$\mathtt{strongCycles}_{\tau} \neq \emptyset \Rightarrow \exists e \in \mathcal{E}_{\tau^{\mathtt{imm}}}$ s.t. $e \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{so}^{+}} e \Rightarrow e \rightarrow_{\tau^{\mathtt{imm}}}^{\mathbf{to}} e$ (using Lemma 3).
$\Rightarrow$, $\neg(order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}})) \Rightarrow \neg(total(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}}) \wedge order(\mathcal{E}^{(\mathtt{sc})}, \mathrm{to}_{\tau^{\mathtt{imm}}}))$. □

```
1 Function synthesisCore(τ):                    /* τ = ⟨E_τ, hb_τ, mo_τ, rf_τ⟩ */
2 |  E_τ^imm := E_τ ∪ candidateFences(τ)
3 |  (hb_τ^imm, mo_τ^imm, rf_τ^imm, rf_τ^imm^-1, fr_τ^imm, so_τ^imm) := computeRelations(τ, E_τ^imm)
4 |  weakCycles_τ := weakFensying(τ^imm)
5 |  strongCycles_τ := strongFensying(τ^imm)
6 |  if weakCycles_τ = ∅ ∧ strongCycles_τ = ∅ then
7 |  |  abort                                  /* cannot stop τ with C11 fences */
8 |  return weakCycles_τ, strongCycles_τ
```

6.3.4 Computing candidate solutions using Weak-FenSying and Strong-FenSying

The detection of violation of C11 coherence conditions is formally presented as function `synthesisCore`. The function takes a buggy trace τ as input and computes the candidate solutions for invalidating τ using Weak-FenSying and Strong-FenSying. The function starts its processing by inserting candidate `fences` to obtain $\tau^{\texttt{imm}}$ (line 2). The function then computes the relations $\mathrm{hb}_{\tau^{\texttt{imm}}}$, $\mathrm{mo}_{\tau^{\texttt{imm}}}$, $\mathrm{rf}_{\tau^{\texttt{imm}}}$, $\mathrm{rf}^{-1}_{\tau^{\texttt{imm}}}$, $\mathrm{fr}_{\tau^{\texttt{imm}}}$, and $\mathrm{so}_{\tau^{\texttt{imm}}}$ on the candidate `fences` and program events (line 3). The function performs Weak-FenSying (line 4) and Strong-FenSying (line 5) as described in §6.3.2 and §6.3.3 respectively. *Johnson's* cycle detection algorithm [48] is used to detect cycles in the event relations for Weak-FenSying and Strong-FenSying.

As remarked before, the relations on the events of $\tau^{\texttt{imm}}$ represent the maximal ordering that can be introduced by synthesizing `fences`. Hence, if a buggy trace can be invalidated using `fences` then the function `synthesisCore` would detect candidate solutions as cycles using Weak-FenSying and Strong-FenSying. If no cycles are detected by the function then the process if *aborted* (line 6-7) signifying that the buggy trace cannot be invalidated with additional ordering from `fences`.

Note that, given a set of related event pairs, *Johnson's* cycle detection algorithm soundly detects all cycles, hence, the assumption (A1) of Lemmas 2 and 4 hold.

6.4 FenSying. Optimal fence synthesis for C11

This work proposes the first `fence` synthesis technique under the C11 memory model, called FenSying, and claims its optimality. This work is distinct from prior works in the following ways.

1. FenSying is the first technique that performs precise analysis for `fence` synthesis under the C11 model.
2. FenSying technique synthesizes C11 synchronization `fences` that are portable.

FenSying uses the Weak-FenSying and Strong-FenSying techniques to introduce additional ordering in the buggy traces of an input program using `fences`, such that, the additional orderings lead to the violation of a coherence condition under C11. The FenSying technique is,

1. *Sound.* If a buggy trace can be fixed with C11 `fences` then FenSying can find a candidate solution.
2. *Optimal.* FenSying synthesizes the smallest set of `fences` with weakest orders.

Algorithm 2 formally presents the FenSying technique. The steps of the algorithm are discussed below.

Buggy traces

FenSying takes a finite set of buggy traces of a C11 program P as input. FenSying relies on an external *buggy trace generator* (`BTG`) for the set of buggy traces of P (line 3). Note that, given the assumption in Chapter 2 that the input program has terminating executions, where each thread has deterministic computations, the set of buggy traces returned by the BTG would be finite.

Algorithm 2: FenSying (P)

```
1  Φ := ⊤                                   /* SAT query for all buggy traces of P   */
2  W := ∅; S := ∅                                    /* cycles of all traces */
3  forall τ ∈ buggyTraces(P) do          /* set of buggy traces from BTG  */
4  |  W_τ, S_τ := synthesisCore(τ)    /* Weak-FenSying and Strong-FenSying  */
5  |  Φ_τ := Q(W_τ, S_τ)                      /* SAT query using Equation 6.1 */
6  |  Φ := Φ ∧ Φ_τ          /* conjunct to retain a solution for each trace */
7  |  W :=W ∪ W_τ; S :=S ∪ S_τ
8  minΦ := minModel(Φ)      /* min-model across all traces for optimality */
9  F := assignMO(minΦ, W, S)          /* weakest order for optimal fences  */
10 return syn(P, F)                     /* synthesize optimal fences in P */
```

Detecting violation of trace coherence

For each buggy trace τ in the set returned by the BTG, FenSying invokes the function synthesisCore (refer to §6.3.4). The function inserts candidate fences above and below each event in τ to generate the intermediate trace τ^{imm}. The function then proceeds to compute cycles in event relations to violate coherence of the trace.

The function receives the set of weakCycles_τ and strongCycles_τ from synthesisCore that are computed from Weak-FenSying and Strong-FenSying respectively (line 4).

If for a buggy trace the set of cycles is empty then it implies that the buggy trace cannot be invalidated. Further, if a buggy trace of P cannot be invalidated then the program P cannot be fixed. As a result, the fence synthesis process is aborted on encountering such a buggy trace (lines 6-7, function synthesisCore).

Reduction for optimal set of fences using SAT

Given the set of candidate solutions to invalidate a buggy trace τ, the optimal set of fences is the smallest set of *candidate* ***fences*** that can form sufficient ordering to invalidate the buggy trace.

The set of candidate solutions are represented as a SAT query. A SAT solver is then

be used to compute the optimal set of candidate `fences`. Let $\mathtt{W}_\tau$ and $\mathtt{S}_\tau$ represent $\mathtt{weakCycles}_\tau$ and $\mathtt{strongCycles}_\tau$ of a buggy trace τ computed using the function `synthesisCore`. Further, for each $\mathtt{w} \in \mathtt{W}_\tau$ and $\mathtt{s} \in \mathtt{S}_\tau$, let $\mathtt{W}^{\mathbb{F}}$ and $\mathtt{S}^{\mathbb{F}}$ respectively represent the set of candidate `fences` in cycles `w` and `s`. The computation of the `SAT` query corresponding to a buggy trace τ is formally presented in Equation 6.1.

$$\mathcal{Q}(\mathtt{W}_\tau, \mathtt{S}_\tau) = (\bigvee_{\mathtt{W} \in \mathtt{W}_\tau} \bigwedge_{\mathbb{F}_\mathtt{w} \in \mathtt{W}^{\mathbb{F}}} \mathbb{F}_\mathtt{w}) \vee (\bigvee_{\mathtt{S} \in \mathtt{S}_\tau} \bigwedge_{\mathbb{F}_\mathtt{s} \in \mathtt{S}^{\mathbb{F}}} \mathbb{F}_\mathtt{s}) \tag{6.1}$$

Intuitively, the computation first conjuncts the candidate fences from each candidate solution. Further, to retain at least one solution corresponding to τ the computation takes a disjunction of the conjuncts (line 5).

Further, if a candidate solution is found for each buggy trace (the process is not *aborted* for any trace) then the corresponding `SAT` queries are conjuncted to represent that a solution must be retained for each buggy trace (line 6).

The final `SAT` query (Φ) is given to a `SAT` solver to compute the `min-model` of the query, that is, the smallest set fo `fences` that satisfy the query (line 8). The `min-model` returns the optimal (minimal) set of `fences` to invalidate each buggy trace of P.

Consider the three candidate solutions shown in Figures 6.1(c)-(e). The `SAT` query corresponding to the three solutions is $\Phi_\tau = (\mathbb{F}_{12}) \vee (\mathbb{F}_{12} \wedge \mathbb{F}_{22}) \vee (\mathbb{F}_{12} \wedge \mathbb{F}_{22})$. Further, assuming that Figures 6.1(b) represents the only buggy trace of the input program, $\Phi = \Phi_\tau$ and $\mathtt{min\text{-}model}(\Phi)$, $\mathtt{min}\Phi = \{\mathbb{F}_{12}\}$.

Type of `fences` in query. The `SAT` query is computed on untyped `fences`. The untyped `fences` do not differentiate between the use of a `fence` in a candidate solution computed by Weak-FenSying or Strong-FenSying. Thus, the computation of the `min-model` can select the smallest set of `fences` across weaker and stronger solutions.

Ignoring program `fences` in `SAT` query. The `SAT` query is formed on the set of candidate `fences` from each candidate solution. A candidate solution may also have `fences` that were present in the input program (and not inserted by `synthesisCore`),

called *program fences*. The program `fences` are not included in the `SAT` query to receive the smallest number of `fence` to be synthesized. For instance, consider the set of `fences` in two candidate solutions, $c_1 = \{\mathbb{F}_1, \mathbb{P}_1, \mathbb{P}_2\}$ and $c_2 = \{\mathbb{F}_1, \mathbb{F}_2\}$, where $\mathbb{F}_i$ represent synthesized `fences` while $\mathbb{P}_i$ represent program `fences`. If program `fences` are considered along with synthesized `fences` in the `SAT` query, then `min-model` would return the solution c_2. However, in such a case the fixed program would have four `fences` (two synthesized and two program `fences`). On the other hand, solution c_1 would result in the synthesis of a single `fence`, and hence, a total of three `fences` in the fixed program. Considering only synthesized `fences` in the `SAT` query results in the solution c_1 as the min-model.

Weakest type for optimal `fences`

The `min-model` computed using a `SAT` solver returns a solution `min`Φ that is optimal in the number of `fences`. The `fences` in `min`Φ are further assigned the weakest memory order that can invalidate each buggy trace of P, using function `assignMO` (line 9). Let `min-cycles` represent a set of cycles such that each candidate `fence` in the cycles belongs to `min`Φ.

Weakest memory orders to invalidate individual cycles. Recall that, synthesized `fences` form the relations $\mathrm{sb}_{\tau^{\mathtt{imm}}}$, $\mathrm{sw}_{\tau^{\mathtt{imm}}}$, and $\mathrm{dob}_{\tau^{\mathtt{imm}}}$ with the events of $\tau^{\mathtt{imm}}$. Further, since a buggy trace τ is a valid `C11` trace, a coherence condition cannot be violated on the events of a thread, even with the addition of candidate `fences`. Thus, the candidate `fences` of a candidate solution must form a $\mathrm{sw}_{\tau^{\mathtt{imm}}}$ or a $\mathrm{dob}_{\tau^{\mathtt{imm}}}$ ordering (the inter-thread relations formed with `fences`).

Let $R = \mathrm{sw}_{\tau^{\mathtt{imm}}} \cup \mathrm{dob}_{\tau^{\mathtt{imm}}}$.
Weakest memory orders for `fences` of a cycle in `min-cycles` is computed as:

- Consider a cycle $c \in$ `min-cycles` $\cap$ `weakCycles`$_\tau$,
  - if a `fence` $\mathbb{F}$ in c is related to an event e in c as $e \rightarrow^R \mathbb{F}$, then $\mathbb{F}$ is assigned the memory order `acq`;

  - if a `fence` $\mathbb{F}$ in c is related to an event e in c as $\mathbb{F} \rightarrow^{R} e$, then $\mathbb{F}$ is assigned the memory order `rel`;
  - if a `fence` $\mathbb{F}$ in c is related to events e', e in c as $e' \rightarrow^{R} \mathbb{F} \rightarrow^{R} e$, then $\mathbb{F}$ is assigned the memory order `acq-rel`.

- Consider a cycle $c \in$ `min-cycles` $\cap$ `strongCycles`$_{\tau}$, each `fence` in c is assigned the memory order `sc`.

Consider a cycle c: $e \rightarrow^{\mathbf{sb}}_{\tau_1\texttt{imm}} \mathbb{F}_1 \rightarrow^{\mathbf{sw}}_{\tau_1\texttt{imm}} \mathbb{F}_2 \rightarrow^{\mathbf{sw}}_{\tau_1\texttt{imm}} \mathbb{F}_3 \rightarrow^{\mathbf{sb}}_{\tau_1\texttt{imm}} e' \rightarrow^{\mathbf{rf}}_{\tau_1\texttt{imm}} e$ representing a violation of $\mathrm{rf}_{\tau_1\texttt{imm}}; \mathrm{hb}_{\tau_1\texttt{imm}}$ irreflexivity (coherence constraint `(co-rh)`). Let `min`$\Phi = \{\mathbb{F}_1, \mathbb{F}_2, \mathbb{F}_3\}$. According to the rules discussed above, the `fences` $\mathbb{F}_1$, $\mathbb{F}_2$ and $\mathbb{F}_3$ are assigned the memory orders `rel`, `acq-rel` and `acq` respectively.

Consider the three candidate solutions shown in Figures 6.1(c)-(e); the computed `min`$\Phi = \{\mathbb{F}_{12}\}$. Since, $\mathbb{F}_{12}$ forms the ordering $\mathbb{F}_{12} \rightarrow^{\mathbf{sw}} c$ in Figure 6.1(e), thus, the memory order assigned to $\mathbb{F}_{12}$ is `rel`.

Weakest memory orders to invalidate all buggy traces of P. After computing the memory orders for each cycle in `min-cycles`, the function `assignMO` iterates over all buggy traces and detects the sound weakest memory order for each `fence` across traces. For each `fence` $\mathbb{F}$ in `min-model`, `assignMO` assigns the weakest memory order that is at least as strong as the order of $\mathbb{F}$ computed for individual `min-cycles`.

Assume a cycle c_1 in ${\tau_1}^{\texttt{imm}}$ and a cycle c_2 in ${\tau_2}^{\texttt{imm}}$. The function computes a union of the `fences` in c_1 and c_2 while choosing the stronger memory order for each `fence` that is present in both the cycles. In doing so, both τ_1 and τ_2 are invalidated. If two candidate solutions have the same set of fences, the function selects the solution with the lower weight. Computation of the weight of a cycle is discussed in §6.2.1.

Consider the cycles $\tau_1 c_1$ and $\tau_1 c_2$ of buggy trace τ_1, and $\tau_2 c_1$ of buggy trace τ_2 shown in Figure 6.8. Let `min`$\Phi = \{\mathbb{F}_1, \mathbb{F}_2, \mathbb{F}_3\}$. The memory orders of the `fences` for each cycle are shown with superscripts and the weights of the cycles are written against the name of the cycles. The candidate solutions $\tau_1 c_1$ and $\tau_1 c_2$ are combined with $\tau_2 c_1$ to form $\tau_{12} c_{11}$ and $\tau_{12} c_{21}$ of weights 5 $(2+2+1)$ and 4 $(1+1+2)$, respectively. The solution $\tau_{12} c_{11}$ is of higher weight and is discarded. Memory orders `rel`, `acq`

$\tau_1 c_1(4)$:	$\mathbb{F}_1^{\texttt{acq-rel}} \wedge \mathbb{F}_2^{\texttt{acq-rel}}$
$\tau_1 c_2(4)$:	$\mathbb{F}_1^{\texttt{rel}} \wedge \mathbb{F}_2^{\texttt{acq}} \wedge \mathbb{F}_3^{\texttt{acq-rel}}$
	cycles of τ_1
$\tau_2 c_1(3)$:	$\mathbb{F}_1^{\texttt{rel}} \wedge \mathbb{F}_2^{\texttt{acq}} \wedge \mathbb{F}_3^{\texttt{acq}}$
	cycle of τ_2
$\tau_{12} c_{11}(5)$:	$\mathbb{F}_1^{\texttt{acq-rel}} \wedge \mathbb{F}_2^{\texttt{acq-rel}} \wedge \mathbb{F}_3^{\texttt{acq}}$
$\tau_{12} c_{21}(4)$:	$\mathbb{F}_1^{\texttt{rel}} \wedge \mathbb{F}_2^{\texttt{acq}} \wedge \mathbb{F}_3^{\texttt{acq-rel}}$

Figure 6.8: Weakest memory orders of optimal `fences`

and `acq-rel` assigned to `fences` $\mathbb{F}_1$, $\mathbb{F}_2$ and $\mathbb{F}_3$ respectively represent the optimal (weakest) memory orders for the `fences`.

Fixed input program

The optimal (minimal) set of `fences` (computed using `min-model` in line 8), with the optimal (weakest) memory orders (computed using `assignMO` in line 9) are synthesized in P at the program locations corresponding to the synthesis locations (line 10). The transformed program with the synthesized `fences` is fixed or bug free.

Lemma 5. `AssignMO` is sound:
$\forall$ cycles $c = e \rightarrow^{R_1} ...e_1 \rightarrow^{R_2} \mathbb{F} \rightarrow^{R_3} e_2... \rightarrow^{R_4} e \in$ `min-cycles` (where $R_i \in \{\text{hb}_{\tau^{\text{imm}}}, \text{mo}_{\tau^{\text{imm}}}, \text{rf}_{\tau^{\text{imm}}}, \text{rf}^{-1}_{\tau^{\text{imm}}}, \text{so}_{\tau^{\text{imm}}}\}$), if `fence` $\mathbb{F}$ is assigned the memory order m then $e_1 \rightarrow^{R_2} \mathbb{F}^m \rightarrow^{R_3} e_2$.
If a min-cycle c is formed due to event relations introduced by a `fence` $\mathbb{F}$ then after assigning a memory order m for $\mathbb{F}$ using `assignMO` the event relations still hold; *i.e.* FenSying does not assign a memory order to a `fence` that is too weak to stop the buggy trace.

Proof. As defined by the computation of event relations using `fences` under C11, if there exists a weak cycle in the intermediate trace τ^{imm} then $\exists\ e \rightarrow^{\text{sw}}_{\tau^{\text{imm}}} e' \vee e \rightarrow^{\text{dob}}_{\tau^{\text{imm}}} e'$ s.t. $\neg e \rightarrow^{\text{sw}}_{\tau} e' \wedge \neg e \rightarrow^{\text{dob}}_{\tau} e'$ (where $e, e' \in \mathcal{E}_\tau$).

By definitions of $e \rightarrow^{\mathbf{sw}}_{\tau^{\mathtt{imm}}} e'$ and $e \rightarrow^{\mathbf{dob}}_{\tau^{\mathtt{imm}}} e'$ (refer to Figure 3.4) if e is a `fence` then its memory order must be `rel` or stronger, if e' is a `fence` then its memory order must be `acq` or stronger.

Since, `assignMO` assigns `rel` for e and `acq` for e', thus, the locally assigned memory orders are sufficiently strong. *inf*(i)

If there exists a `fence`, $\mathbb{F}$, that was locally assigned a memory order m and after coalescing with other buggy trace the final memory order of $\mathbb{F}$ is m' then either $m' = m$ or m' is stronger than m (by construction of coalesced candidate solutions).

Since, we know that m was sufficiently strong (using *inf*(i)) then the final memory order m' is also sufficiently strong. □

Lemma 6. FenSying is sound for 1 trace.
Given an input program P s.t. $\mathtt{buggyTraces}(P) = \{\tau\}$.
$\exists \mathcal{E}'$ s.t. $\mathtt{buggyTraces}(\mathtt{syn}(P, \mathcal{E}')) = \emptyset \Rightarrow$ FenSying can construct $\tau^{\mathtt{inv}}$.

Proof. Firstly, using Lemma 2 and Lemma 4 we can state that (i) each weak and strong cycle is detected by FenSying, and (ii) a cycle detected by FenSying is indeed represents a true violation under C11.

Secondly, the `fences` introduced for at least 1 of the violations exist in the final solution (by construction of `SAT` query).

Thirdly, The memory order assigned to the `fences` in `min`Φ is sufficiently strong to stop the buggy trace (Lemma 5(inf(i))).
Hence, FenSying is sound for 1 trace □

Theorem 10. FenSying is sound.
Given an input program P s.t. $\mathtt{buggyTraces}(P) \neq \emptyset$. $\exists \mathcal{E}'$ s.t. $\mathtt{buggyTraces}(\mathtt{syn}(P, \mathcal{E}'))$ $= \emptyset \Rightarrow \forall \tau \in \mathtt{buggyTraces}(P)$ FenSying can construct $\tau^{\mathtt{inv}}$.
If P can be fixed by synthesizing C11 `fences` then FenSying fixes P.

Proof. Let $\texttt{BT} = \texttt{buggyTraces}(P)$. Consider induction on $|\texttt{BT}|$.

Base Case: Consider $|\texttt{BT}| = 1$.
Using Lemma 6, FenSying is sound for 1 trace.

Induction Hypothesis: Assume that FenSying is sound for $|\texttt{BT}| = \text{N}$.

Induction Step: Consider $|\texttt{BT}| = \text{N}+1$.

Since, we take a conjunction on the SAT formulas from various traces thus at least 1 cycle from each trace exists in the min-model (by construction of SAT query).
Further, we know from Lemma 5 that FenSying assigns memory orders that can invalidate all buggy traces.
Thus, FenSying is sound for N+1 buggy traces. □

Lemma 7. min-model returns the optimal number of fences.
Let $\mathcal{E}^{\mathbb{F}^{\texttt{fx}}}$ represent the set of events of $P^{\texttt{fx}}$. min-model returns the optimal number of fences if $\nexists \mathcal{E}^o$ s.t. $|\mathcal{E}^o| < |\mathcal{E}^{\mathbb{F}^{\texttt{fx}}}|$ and $\texttt{buggyTraces}(\texttt{syn}(P, \mathcal{E}^o)) = \emptyset$.

Proof. Let $\mathcal{F}$ represent the set of fences returned by min-model and let $\mathcal{F}^o$ represent an optimal set of fences. Assume $|\mathcal{F}^o| < |\mathcal{F}|$.

The min-model is computed using a SAT solver and the computation is assumed to be correct. As the consequence, $|\mathcal{F}^o| < |\mathcal{F}| \Rightarrow$ the optimal result was not a part of the SAT query.

Using Lemma 2 and Lemma 4 we know that FenSying does not miss any weak or strong cycle $\Rightarrow$ every set of fences that forms a correct solution, including the optimal solution, is contained in the SAT query.

Thus, $|\mathcal{F}| = |\mathcal{F}^o|$ *i.e.*min-model returns the optimal number of fences. □

Theorem 11. FenSying synthesizes the optimal number of fences with the optimal memory orders.
Let $\mathcal{E}^{\mathbb{F}^{\texttt{fx}}}$ represent the set of events of $P^{\texttt{fx}}$ then $\nexists \mathcal{E}^o$ s.t. $|\mathcal{E}^o| < |\mathcal{E}^{\mathbb{F}^{\texttt{fx}}}|$ or $(\exists e^o \in \mathcal{E}^o, e \in \mathcal{E}^{\mathbb{F}^{\texttt{fx}}}$ s.t. $thr(e^o) = thr(e)$, $idx(e^o) = idx(e)$, $act(e^o) = act(e)$, $obj(e^o) = obj(e)$ and

$loc(e^o) = loc(e)$ but $ord(e^o) \sqsubseteq ord(e)$) and $\texttt{buggyTraces}(\texttt{syn}(P, \mathcal{E}^o)) = \emptyset$.

Proof. `AssignMO` iterates over cycles in `min-cycles` and takes union over `fences` of cycles from `min-cycles`. As `min`Φ consists of the optimal number of `fences` (Lemma 7) then union over cycles of `min-cycles` is the same set of `fences` as `min`Φ. Thus, FenSying is optimal in the number of `fences`.

Let `BT` = `buggyTraces`(P). Consider induction on |`BT`|.

Base Case: Consider |`BT`| = 1.
Each `fence` is locally assigned the weakest memory order that is sound (Lemma 5). Thus, FenSying is optimal in the memory order of `fences` for 1 buggy trace.

Induction Hypothesis: Assume, FenSying is optimal in the memory order of `fences` when |`BT`| = N.

Induction Step: Consider |`BT`| = N+1.
Let $s_1, ..., s_M$ represent the M coalesced solutions for buggy traces $\tau_1, ..., \tau_N$ and $\tau_{N+1}c_i$ for $i \in \{1, ..., q\}$ represent the q cycles of $(N+1)^{th}$ trace.

Every coalesced solution $\tau_{N+1}s_j$ has the same number of `fences` = `fences` of `min`Φ because `min`Φ returns the minimum number of `fences` required to stop $\tau_1, ..., \tau_{N+1}$.

If there exists a `fence` $\mathbb{F}$ with memory order m in a cycle $\tau_{N+1}c_i$ but the final solution of FenSying assigns memory order m' to $\mathbb{F}$ s.t. m' is stronger than m
then, $\exists s_j$ where memory order of $\mathbb{F}$ is m' (by construction of coalesced solutions),

Further, $\nexists s_k$ where memory order of $\mathbb{F}$ is m s.t. $wt(\tau_{N+1}c_i\text{-}s_k) < wt(\tau_{N+1}c_i\text{-}s_j)$ (where $wt(x\text{-}y)$ represents the weight of the solution formed by coalescing cycle x with candidate solution y).

Thus, FenSying is optimal in the memory order of `fences` for N+1 buggy traces. □

6.4.1 Time complexity analysis

Time complexity of `synthesisCore`. The complexity of detecting all cycles for a trace is $\mathscr{C} = \mathcal{O}((|\mathcal{E}_\tau|+\texttt{E}).(\texttt{C}+1))$ where `C` represents the number of cycles of a buggy trace τ and `E` represents the number of pairs of events in $\mathcal{E}_\tau$. Note that `E` is in $O(|\mathcal{E}_\tau|^2)$ and `C` is in $O(|\mathcal{E}_\tau|!)$. Thus, Weak-FenSying and Strong-FenSying have exponential complexities in the number of traces and the number of events per trace.

Time complexity of `SAT` solving. The complexity of constructing the query Φ_τ is $\mathscr{S} = \mathcal{O}(\texttt{C.F})$, where `C` is the number of cycles of τ and `F` is the number of `fences` per cycle. The structure of the query Φ corresponds to the *Head-cycle-free* (HCF) class of `CNF` theories; hence, the min-model computation falls in the `FP` complexity class [18].

Time complexity of `assignMO`. Determining the optimal memory orders has a complexity $\mathscr{A} = \mathcal{O}(\texttt{BT.C.F}+\texttt{M}^{\texttt{BT}})$, where `BT` if the number of buggy traces of P, `C` and `F` are defined as before, and `M` is the number of min-cycles per trace.

Thus, the time complexity of FenSying is computed as $\mathcal{O}(\texttt{BT}.(\mathscr{C}+\mathscr{S})+\mathscr{A})$.

6.5 fastFenSying. Efficient `fence` synthesis for C11

Optimal `fence` synthesis problem with multiple types of `fences` is NP-hard even for straight-line (refer to §6.2.1 for details). The hardness is witnessed empirically with proposed optimal `fence` synthesis solution even for the simplest `C11` programs. Experiments with FenSying (§6.7) show an exponential increase in the analysis time with the increase in the program size.

To address scalability, this work proposes a *near-optimal* `fence` synthesis technique called fastFenSying. fastFenSying fixes one buggy trace at a time optimally. Note that, fixing one buggy trace optimally may not guarantee optimality across all buggy traces. In the process, this technique may add a small number of extra `fences` than what an optimal solution would compute. Experiments with fastFenSying show

Algorithm 3: fastFenSying (P)

```
1 if ∃τ ∈ buggyTraces(P) then                    /* next buggy trace from BTG  */
2     W_τ, S_τ := synthesisCore(τ)               /* Weak-FenSying and Strong-FenSying  */
3     Φ := Q(W_τ, S_τ)                           /* SAT query using Equation 6.1 */
4     minΦ := minModel(Φ)                        /* optimal fences to invalidate τ */
5     F := assignMO(minΦ, W_τ, S_τ)              /* weakest order for optimal fences  */
6     P' := syn(P, F)                            /* synthesize fences optimal for τ */
7     return fastFenSying (P')                   /* fences synthesis on transformed P */
8 else return P                                  /* no more buggy traces */
```

that it performs exponentially better than FenSying, in terms of the time of analysis and scalability. Further, fastFenSying computes the optimal synthesis result in over 99.5% of the experiments. The fastFenSying technique is,

1. *Sound.* If a buggy trace can be fixed with C11 `fences` then fastFenSying can find a candidate solution.

2. *Optimal for one buggy trace.* fastFenSying synthesizes the smallest set of `fences` with weakest orders for a single buggy trace.

Similar to FenSying, fastFenSying performs precise analysis for `fence` synthesis under the C11 model, and synthesizes portable C11 `fences`. Algorithm 3 formally presents the fastFenSying technique. The steps of the algorithm are discussed below.

Unlike the FenSying technique, fastFenSying takes any one buggy traces of a C11 program P from `BTG` as input (line 1). fastFenSying invokes the function `synthesisCore` (refer to §6.3.4) on the buggy trace τ, received from `BTG`, and obtains $\texttt{weakCycles}_\tau$ and $\texttt{strongCycles}_\tau$ (line 2). Similar to FenSying if the set of $\texttt{weakCycles}_\tau$ and $\texttt{strongCycles}_\tau$ is empty then the `fence` synthesis process is aborted, since τ cannot be invalidated and, hence, P cannot be fixed (lines 6-7, function `synthesisCore`).

fastFenSying forms a `SAT` query on $\texttt{weakCycles}_\tau$ and $\texttt{strongCycles}_\tau$ using Equation 6.1 (line 3) and invokes a `SAT` solver to compute the `min-model`, $\texttt{min}\Phi$, of the

cycles in τ_1 ($\mathtt{C}_{\tau_1}$):
$\{\mathbb{F}_1, \mathbb{F}_2, e_1\}$ and $\{\mathbb{F}_1, \mathbb{F}_3, \mathbb{F}_4\}$
$\Phi_{\tau_1} = (\mathbb{F}_1 \wedge \mathbb{F}_2) \vee (\mathbb{F}_1 \wedge \mathbb{F}_3 \wedge \mathbb{F}_4)$

cycles in τ_2 ($\mathtt{C}_{\tau_2}$): $\{\mathbb{F}_3, \mathbb{F}_4\}$
$\Phi_{\tau_2} = (\mathbb{F}_3 \wedge \mathbb{F}_4)$

Figure 6.9: Example of non-optimal computation of fastFenSying

query (line 4). The set $\mathtt{min}\Phi$ represents the smallest set of `fences` that can invalidate a single buggy trace τ. On the set $\mathtt{min}\Phi$, fastFenSying invokes `assignMO` that computes the weakest memory orders for the `fences` in $\mathtt{min}\Phi$ to invalidate τ (line 5). `assignMO` first computes the weakest memory orders to invalidate individual cycles and chooses the cycle with the weakest memory orders for `fences` (as described under 'Weakest type for optimal `fences`' in §6.4).

The optimal (minimal) set of `fences` (computed using `min-model` in line 4), with the optimal (weakest) memory orders (computed using `assignMO` in line 5) are synthesized in P at the program locations corresponding to the synthesis locations (line 6).

The transformed program P' is then passed as input to the `BTG` to receive any one buggy trace of P' (line 7). The process repeats till fastFenSying generates a program by synthesis that has no buggy traces (line 8).

Non-optimality of fastFenSying

The non-optimality of fastFenSying is a result of the reduction in information that the technique has to work with (from all buggy traces to just one at a time).

Consider the example in Figure 6.9. The figure shows cycles in two buggy traces τ_1 and τ_2 of an input program.

FenSying provides the formula $\Phi_{\tau_1} \wedge \Phi_{\tau_2}$ to the `SAT` solver and the optimal solution obtained is $(\mathbb{F}_1 \wedge \mathbb{F}_3 \wedge \mathbb{F}_4)$. On the other hand, fastFenSying considers the formulas

Φ_{τ_1} and Φ_{τ_2} in separate iterations. As a result, fastFenSying may compute the set $\mathbb{F}_1 \wedge \mathbb{F}_2$ in one iteration and the set $\mathbb{F}_3 \wedge \mathbb{F}_4$ in the next iteration, and thus, synthesize four fence ($\{\mathbb{F}_1, \mathbb{F}_2, \mathbb{F}_3, \mathbb{F}_4\}$).

Theorem 12. fastFenSying is sound
Given an input program P, and $\tau \in$ buggyTraces(P).
$\exists \mathcal{E}'$ s.t. buggyTraces(syn($prg(P, \mathcal{E}')$)$= \emptyset \Rightarrow$ fastFenSying can compute τ^{inv}.

Note that, the proof of Theorem 12 follows from the proof of Lemma 6.

Theorem 13. fastFenSying is optimal for one buggy trace.

Proof. fastFenSying is optimal in the number of fences (using Lemma 7).

Let min-cycles represent a set of cycles such that each candidate fences in the cycles belongs to minΦ. For each cycle c in min-cycles, assignMO computes the weakest memory order for the fences in c (*inf*(i), Lemma 5) . Since, assignMO chooses the cycle with fences of weakest memory orders (by definition of assignMO), fastFenSying is optimal for one buggy trace. □

6.5.1 Time complexity analysis

Time complexity of synthesisCore and SAT solving. Similar to FenSying (refer to 6.4.1), the time complexity of synthesisCore is $\mathscr{C} = \mathcal{O}((|\mathcal{E}_\tau|+\text{E}).(\text{C}+1))$ where C represents the number of cycles of the buggy trace τ and E represents the number of pairs of events in $\mathcal{E}_\tau$, and SAT solving is $\mathscr{S} = \mathcal{O}(\text{C.F})$, where F is the number of fences per cycle of τ.

Time complexity of assignMO. Since, fastFenSying does not coalesce memory orders across traces the time complexity of assignMO is reduced to $\mathscr{A}' = \mathcal{O}(\text{C.F}+\text{M})$, where C and F are defined as before, and M is the number of min-cycles per trace.

Assuming BT if the number of buggy traces of P, the time complexity of fastFenSying is computed as $\mathcal{O}(\text{BT}.(\mathscr{C} + \mathscr{S} + \mathscr{A}'))$.

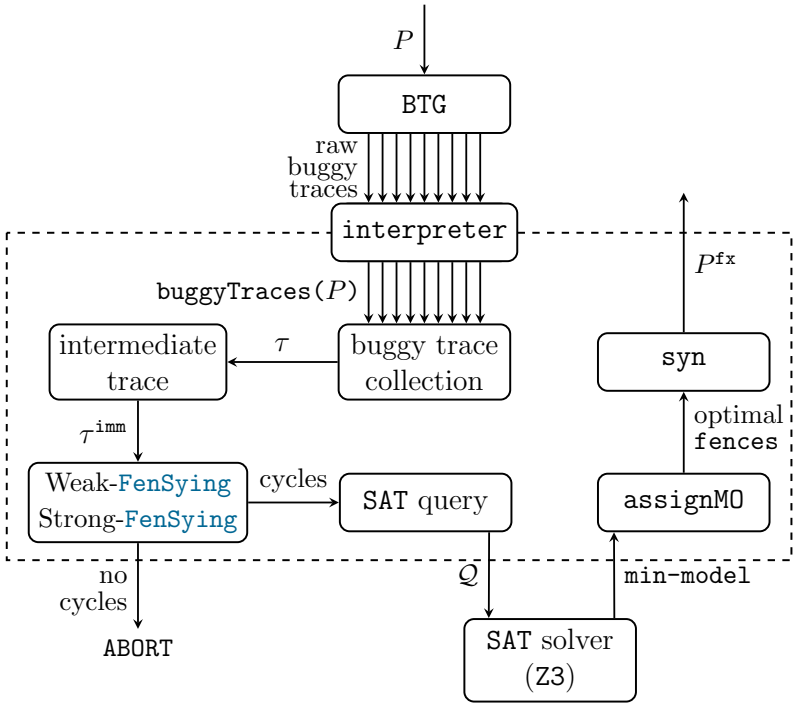


Figure 6.10: Structural overview of FenSying tool

Although, the time complexity of fastFenSying is similar to that of FenSying, empirically it is observed (refer to §6.7) that the number of buggy traces analyzed by fastFenSying is significantly lesser than |BT|. Therefore, in practice, the complexity of various steps of fastFenSying, that are dependent on |BT|, reduces exponentially by a factor of |BT|.

6.6 Implementation details of (fast)FenSying

The implementations of the FenSying and fastFenSying techniques have internal modules that execute the steps of the respective Algorithms 2 and 3, and external modules of preexisting techniques used by Algorithms 2 and 3. The preexisting techniques used by FenSying and fastFenSying include a BTG and a SAT solver.

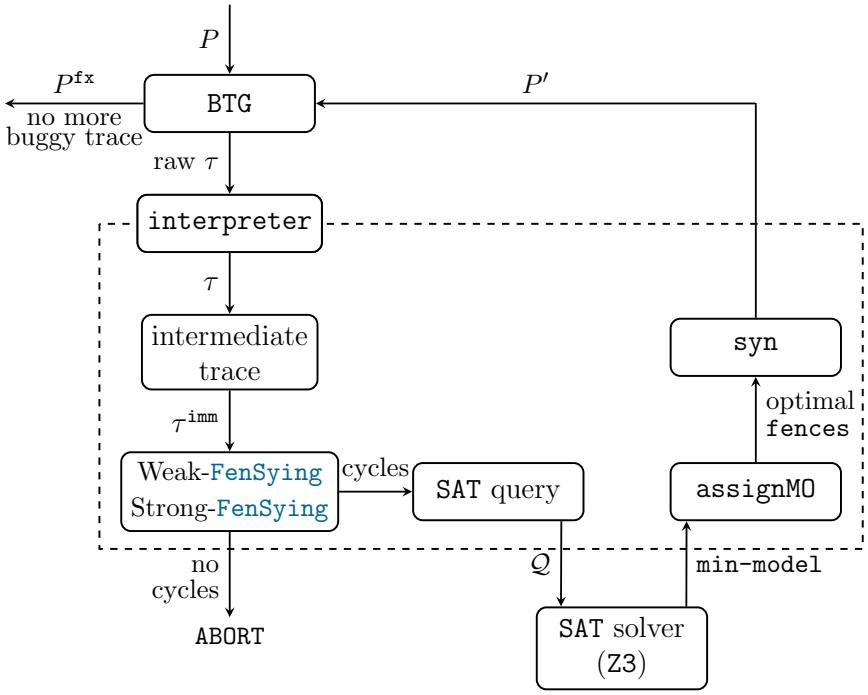


Figure 6.11: Structural overview of fastFenSying tool

Figure 6.10 represents the overview of the FenSying implementation where the internal modules are shown within the dashed box. FenSying receives the set of buggy traces of the input program P from an external BTG and sends one trace at a time to the synthesisCore. Modules 'intermediate trace' and 'Weak-FenSying Strong-FenSying' in Figure 6.10 together represent the synthesisCore. synthesisCore computes the set of cycles using *Johnson's* cycle detection and sends them to the module 'SAT query' to generate the query $\mathcal{Q}$. The query $\mathcal{Q}$ is given to an external 'SAT solver' module that computes the min-model. FenSying computes the weakest memory orders for the fences in min-model using the internal module 'assignMO' and synthesizes the optimal set of fences back in the input program using the module 'syn' to generate the fixed program $P^{\texttt{fx}}$.

Figure 6.11 represents the overview of the fastFenSying implementation where again the internal modules are shown within the dashed box and external modules are outside the dashed box. In contrast to FenSying, fastFenSying receives a single

buggy trace from BTG. It then follows similar interaction between the internal and external modules to generate the transformed program P' with fences synthesized to invalidate the input buggy trace. fastFenSying invokes BTG repeatedly by sending the transformed P' as input. The process continues till P' is buggy. When a bug free P' is generated, fastFenSying returns the bug free P' as $P^{\texttt{fx}}$.

Interpreter. An 'interpreter' module takes a set of raw buggy traces and interprets them in the format readable for FenSying and fastFenSying. Given an external BTG, the raw format of buggy traces refers to the output format used by the BTG for the bug trace dump. Interpreter converts the dump into a tuple of $\langle \mathcal{E}_\tau, \text{hb}_\tau, \text{mo}_\tau, \text{rf}_\tau \rangle$ *i.e.* the format of a trace that is recognized by the techniques. If a relation, hb_τ, mo_τ, or rf_τ, is not provided by the BTG, then the interpreter computes the relation from the available data in the dump. Therefore, to use a tool as the BTG module a suitable interpreter is created for the tool.

6.6.1 Strengthening of program fences

Consider that the fence synthesis, by FenSying or fastFenSying, computes the synthesis of a fence $\mathbb{F}$ with a memory order m at a program location l in the input program P. If there exists a program fence $\mathbb{P}$ (fence already in the input program) at l with a memory order weaker than m, then as a design choice the techniques strengthen the memory order of $\mathbb{P}$ to m instead of synthesizing $\mathbb{F}$. Such a choice reduces the performance overhead of an additional fence in the fixed program.

6.6.2 Tool description

The techniques FenSying and fastFenSying are implemented in Python3. Weak-FenSying and Strong-FenSying use *Johnson's* cycle detection algorithm in the *networkx* library version 2.6.3 [45]. The Z3 theorem prover is used as the SAT solver to find the *min-model* of SAT queries. As a BTG, a model checker called CDSChecker [32] is preferred for the following reasons;

1. CDSChecker is a model checker and thus it can soundly compute the complete set of buggy traces (unlike a testing or simulation alternative).
2. CDSChecker supports the C11 semantics. Most other techniques are designed for a variant [51] or a subset [5, 9, 81] of C11.
3. CDSChecker returns buggy traces along with the corresponding hb_τ, mo_τ, and rf_τ relations.
4. CDSChecker does not halt at the detection of the first buggy trace; instead, it continues to provide all buggy traces as required by FenSying.
5. CDSChecker is open-source and is actively maintained [38].

CDSChecker's source code is modified to accept program location as an attribute of the program events and to halt at the first buggy trace when specified (for fastFenSying). The modifications do not alter the core technique, and thus, are semantic preserving. The data types and data structures supported by an input program is limited to CDSChecker support. Since, CDSChecker is a dynamic technique it supports typical atomic and non-atomic memory accesses and C11 `fences`.

CDSChecker takes a C/C++ program as input and dumps the buggy traces as a structured list that contains the set of events, and the hb_τ and rf_τ relations. The relation mo_τ is separately dumped as a graph in the format of a *dot* file. The *interpreter* module accordingly reads the dumps and generates the tuple of events and relations representing a C11 trace.

6.7 Experiments and Results on FenSying and fastFenSying

6.7.1 Experimental setup

The experiments are conducted on an Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz with 32GB RAM and 32 cores running Ubuntu 20.04.1 LTS. The experiments are run on Python3 version 3.6.9 and Z3 version 4.4.1.

FenSying and fastFenSying are the first techniques that perform fence synthesis under the C11 memory model. The existing fence synthesis technique work on a different set of buggy traces (buggy under their memory model) and accordingly synthesize fences assuming the implicit ordering under their memory model. Consequently, the outcome of FenSying and fastFenSying is incomparable with any existing techniques.

6.7.2 Litmus testing

FenSying and fastFenSying are tested on 1,389 litmus tests of buggy C11 programs (programs where the assert condition in violated in some program run), with the focus on,

(i) **soundness**: if a buggy program can be fixed with C11 fences then both FenSying and fastFenSying can do so;

(ii) **optimality of** FenSying: FenSying synthesizes the smallest number of weakest fences.

Identifying failure of soundness. The fence synthesis process, of FenSying or fastFenSying, is *aborted* for a program P when the set of $\texttt{weakCycles}_\tau$ and $\texttt{strongCycles}_\tau$ for a buggy trace τ of P is empty (refer to §6.3.4). A failure of soundness indicates that the buggy trace could be invalidated with C11 fences and the technique wrongly aborted. The following process is followed to detect failure of soundness for the buggy trace τ that could not be invalidated.

1. For each candidate fence in $\tau^{\texttt{imm}}$, assign memory order sc.
2. Synthesize the fences in P at the locations indicated by $\tau^{\texttt{imm}}$ to form a transformed program P'.
3. Invoke BTG on P'.

Table 6.2: Details of litmus tests

Tests	min-BT	max-BT	avg-BT	min-syn	max-syn	avg-syn	min-str	max-str	avg-str
1389	1	9	1.05	1	4	2.25	0	0	0

Table 6.3: Litmus test results

	completed (syn+no fix)	TO	NO	Tbtg (total)	TF (total)	Ttotal
FenSying	1333 (1185+148)	56	0	50453.19	36896.06	87266.09
fastFenSying	1355 (1207+148)	34	0	30703.71	49068.61	79772.32

The strong order of the synthesized `fences` in P' form the maximal ordering feasible with C11 `fences`. If the `BTG` returns a buggy trace still, then the program indeed cannot be fixed with C11 `fences`.

Identifying failure of optimality for FenSying. Failure of optimality occurs when there exists a weaker version of the fixed program $P^{\texttt{fx}}$ that is bug free. Here, a weaker version refers to a transformation of $P^{\texttt{fx}}$ with lesser or weaker `fences` than $P^{\texttt{fx}}$. The following processes is followed to detect failure of optimality for a fixed program $P^{\texttt{fx}}$.

1. For each synthesized `fence` $\mathbb{F}$ in $P^{\texttt{fx}}$, for each memory order weaker than the order of $\mathbb{F}$, create a transformed program $P^{\texttt{fx}}(i)$ with the weaker orders for $\mathbb{F}$.
2. Create a transformed program from $P^{\texttt{fx}}$ by removing $\mathbb{F}$.
3. Invoke `BTG` on each version created in the previous two steps.

If the `BTG` returns a buggy trace for each of the weaker versions of $P^{\texttt{fx}}$ then there does not exist a weaker fix for the input program P.

Litmus tests. The set of 1,389 litmus tests is borrowed from a previous work [9]. The work proposes a larger set of tests but apart from the 1,389 tests used, the others are bug free. The details of the litmus tests are give in Table 6.2, where 'BT' represents the number of buggy traces, and 'syn', 'str' represent the number of `fences` synthesized, strengthened respectively. Prefixes 'min', 'max', and 'avg' refer to minimum, maximum, and average respectively.

As shown in the table, the number of buggy traces for the litmus tests range between 1 to 9 with an average of 1.05, while the number of `fences` synthesized range between 2 to 4. None of the litmus tests contained `fences` in the input program. Hence, no `fences` were strengthened for any of the litmus tests. Additionally, the average length of litmus tests is 70.46 lines of code.

The results of FenSying and fastFenSying on the litmus tests are presented in Table 6.3.

The column 'completed' represents the number of tests successfully completed for the respective technique. A successful completion signifies that the technique could complete within a timeout of analysis (set at 900s for `BTG`, and an additional 900s for FenSying or fastFenSying), and that the outcome of the synthesis process is sound and optimal (for FenSying). Further, the values in the bracket represent the number of tests that could be fixed ('syn') and the number of tests that the techniques detected cannot be fixed with C11 `fences` ('no fix').

The column 'TO' represents the number of tests that the corresponding techniques timed out for, and 'NO' represents the number of tests for which the techniques computed a non-optimal result. Note that, the non-optimal fastFenSying technique also computed the optimal solution for all the litmus tests that it did not timeout for. Further, observe that the sum total of the number of tests completed and the number of tests that the techniques timed out for is equal to the number of litmus tests, indicating that the techniques were not *unsound* for any test. The columns 'Tbtg', 'TF', and 'Ttotal' represent the time of analysis (in seconds) of the `BTG`, the technique (FenSying or fastFenSying) and the total of the two. The time of analysis is recorded over 5 runs with a timeout of analysis set at 900s each for the `BTG` and the technique.

Note that, the time of analysis of `BTG` is significant in comparison to the time of analysis of the FenSying or fastFenSying technique itself. Therefore, it is reported separately for an accurate indication of the time of analysis of the techniques. The time of analysis of the `SAT` solver is negligible, and therefore, it is included in 'TF'.

6.7.3 Performance analysis

The performance analysis of FenSying and fastFenSying is done over challenging benchmarks, that produce buggy traces under C11, borrowed from previous works on model checking under the C11 memory model and its variants [5, 9, 72, 81].

The performance is measured on various configurations of each benchmark, where the configurations vary on the problem size determined by program features such as the number of loop unrolls and the number of concurrent processing elements (or threads). In essence, higher configurations of the benchmarks result in a higher number of program events.

The performance on the benchmarks is measured on four aspects.

1. *Number of fences synthesized + strengthened.* The number of candidate `fences` synthesized and the number of program `fences` strengthened.
2. *Time of analysis of BTG.* The time taken by the BTG to generate the set of buggy traces for FenSying and the total time across various iterations to generate the next buggy trace for fastFenSying.
3. *Time of analysis of the technique.* The time of analysis of the internal modules of FenSying and fastFenSying. This includes the time of computing the `min-model` using SAT solver.
4. *Number of iterations to reach the result.* Relevant for fastFenSying.

The performance analysis of FenSying and fastFenSying is presented in Table 6.4.

The columns 'Tbtg', 'TF', and 'Ttotal' represent the time of analysis (in seconds) of the BTG, the technique (FenSying or fastFenSying), and the total of the two. The time of analysis is recorded over 5 runs with a timeout of analysis set at 900s each for the BTG ('BTo') and the technique ('FTo'). Column '#BT' shows the number of buggy traces in the input program. A '?' in '#BT' signifies that BTG could

Table 6.4: Comparative performance analysis

			FenSying				fastFenSying				
Id	Name	#BT	syn+str	Tbtg	TF	Ttotal	iter	syn+str	Tbtg	TF	Ttotal
1	peterson(2,2)	30	1+0	2.63	54.31	56.94	1:1	1:1+0:0	0.18	2.07	2.25
2	peterson(2,3)	198	1+0	29.96	594.34	624.3	1:1	1:1+0:0	0.53	3.58	4.11
3	peterson(4,5)	?	–	BTo	–	–	1:1	1:1+0:0	397.51	21.07	418.58
4	peterson(5,5)	?	–	BTo	–	–	1:1	1:1+0:0	BTo	31.52	*931.52
5	barrier(5)	136	1+0	1.09	207.74	208.83	1:1	1:1+0:0	0.13	1.40	1.53
6	barrier(10)	416	1+0	3.37	565.44	568.81	1:1	1:1+0:0	0.2	2.70	2.9
7	barrier(100)	31106	–	–	FTo	–	1:1	1:1+0:0	34.2	198.54	232.74
8	barrier(150)	?	–	BTo	–	–	1:1	1:1+0:0	117.09	399.20	516.29
9	barrier(200)	?	–	BTo	–	–	–	–	–	FTo	–
10	store-buffer(2)	6	2+0	0.08	0.91	0.99	1:1	2:2+0:0	0.04	0.05	0.09
11	store-buffer(4)	20	2+0	1.61	195.35	196.96	1:1	2:2+0:0	1.20	0.05	1.25
12	store-buffer(5)	30	–	–	FTo	–	1:1	2:2+0:0	14.07	0.22	14.29
13	store-buffer(6)	42	–	–	FTo	–	1:1	2:2+0:0	171.09	0.15	171.24
14	store-buffer(10)	?	–	BTo	–	–	1:1	2:2+0:0	BTo	0.05	*900.05
15	dekker(2)	54	2+0	0.17	0.27	0.44	1:1	2:2+0:0	0.26	0.04	0.3
16	dekker(3)	1596	–	–	FTo	–	1:1	2:2+0:0	586.46	1.34	587.8
17	dekker-fen(2,3)	54	1+1	0.15	0.29	0.44	1:1	1:1+1:1	0.25	0.05	0.3
18	dekker-fen(3,2)	730	–	–	FTo	–	1:1	1:1+1:1	159.84	5.56	165.4
19	dekker-fen(3,4)	3076	–	–	FTo	–	1:1	1:1+1:1	BTo	6.06	*906.06
20	burns(1)	36	–	–	FTo	–	7:8	8:10+2:2	0.61	4.69	5.3
21	burns(2)	10150	–	–	FTo	–	6:7	8:10+0:1	71.53	554.6	626.13
22	burns(3)	?	–	BTo	–	–	–	–	–	FTo	–
23	burns-fen(2)	100708	–	–	FTo	–	5:7	4:6+3:3	329.41	43.96	373.37
24	burns-fen(3)	?	–	BTo	–	–	5:7	4:6+3:3	BTo	70.14	*970.14
25	linuxrwlocks(2,1)	10	–	–	FTo	–	1:1	2:2+0:0	0.13	0.12	0.25
26	linuxrwlocks(3,8)	353	–	–	FTo	–	2:2	3:4+0:0	686.52	0.41	*686.93
27	seqlock(2,1,2)	500	–	–	FTo	–	1:1	1:1+0:0	341.54	2.38	343.92
28	seqlock(1,2,2)	592	–	–	FTo	–	1:2	1:2+0:0	119.88	27.69	147.57
29	seqlock(2,2,3)	?	–	BTo	–	–	1:2	1:2+0:0	BTo	88.52	*988.52
30	bakery(2,1)	6	1+0	0.25	25.42	2.88	1:1	1:1+0:0	0.07	0.18	0.25
31	bakery(4,3)	7272	–	–	FTo	–	1:1	1:1+0:0	166.11	5.68	171.79
32	bakery(4,4)	50402	–	–	FTo	–	1:1	1:1+0:0	BTo	18.17	*918.17
33	lamport(1,1,2)	1	No fix.	0.06	0.05	0.11	1:1	No fix.	0.04	0.05	0.09
34	lamport(2,2,1)	1	No fix.	411.94	0.05	411.99	1:1	No fix.	53.34	0.05	53.39
35	lamport(2,2,3)	?	–	BTo	–	–	1:1	No fix.	389.77	0.05	389.82
36	flipper(5)	297	2+0	6.22	254.18	260.40	1:1	2+0	2.51	0.02	2.53
37	flipper(7)	4493	–	–	FTo	–	1:1	2+0	119.21	0.02	119.23
38	flipper(10)	?	–	BTo	–	–	1:1	2+0	BTo	0.03	*900.03

not scale for the test, so the number of buggy traces is unknown. Column 'iter' shows the minimum:maximum number of iterations performed by fastFenSying across five runs. The column 'syn+str' under FenSying report the number of fences synthesized+strengthened. Under fastFenSying 'syn+str' reports the minimum: maximum number of fences synthesized+strengthened across five runs. A '*' against the total time (column 'Ttotal') under fastFenSying signifies that the input program is fixed but the BTG timed out in detecting that the program has no more buggy traces. The assessment that the program is fixed is drawn from manual analysis with help from the fixed programs for lower configurations[1].

Configurations of benchmarks. The configurations of a benchmark vary the problem size such that higher configurations require a higher effort of analysis. The configurations typically vary on the number of concurrent elements (threads), the number of loop iterations, and the number of events. The configurations of the benchmarks in Table 6.4 vary on the following aspects.

The configurations of the benchmark 'barrier' (test IDs 5-9) vary on the number of loop iterations and the number of *writer threads*. Similarly, the configurations of the benchmark 'seqlock' (test IDs 27-29) vary on the number of loop iterations and the number of *reader threads*. The configurations of all other benchmarks vary on the number of loop iterations of various loops in the program.

Observations. A graph contrasting the performance of FenSying and fastFenSying is shown in Figure 6.12. The graph represents the time of analysis of FenSying and fastFenSying, on the y-axis, for the tests in Table 6.4, indicated on x-axis in increasing order of time of analysis of FenSying and fastFenSying, ('*' on the x-axis represents tests that both techniques timeout for). It can be observed that the time of analysis of FenSying is notably higher than that of fastFenSying.

A similar trend can also be observed from the Table 6.4 where fastFenSying significantly outperforms FenSying in terms of the time of analysis and scalability (indicated

[1]Typically, increasing the problem size between configurations increases the set of events not the set of program locations (such as with increasing the number of loop iterations). For such tests the synthesis solution remains same across configurations and so the solution for a lower configuration can be used as an indicator to determine whether the tests marked with '*' are indeed fixed.

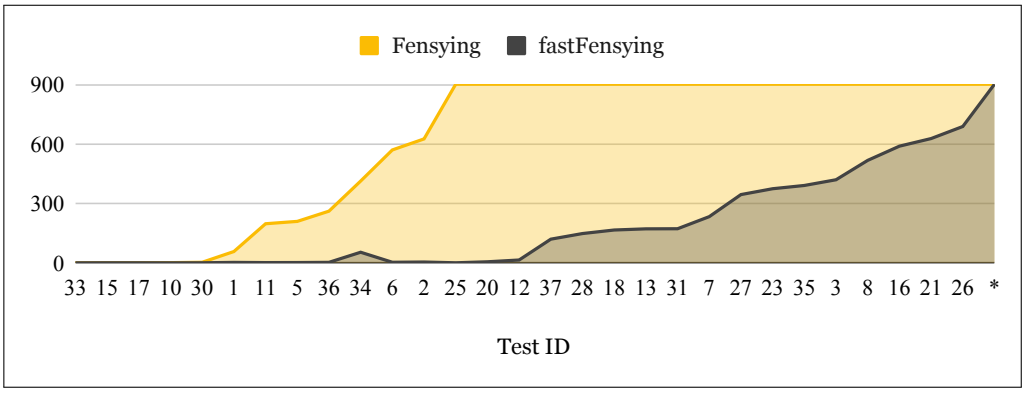


Figure 6.12: Time of analysis of FenSying against fastFenSying (seconds)

by lesser number of 'FTo'). Another noteworthy observation is that fastFenSying adds extra `fences` in only 7 tests with an average of 1.57 additional `fences`.

With the increase in the number of buggy traces, FenSying's time of analysis grows exponential leading to FTo; except in case of test IDs 12, 13, 20, and 25, where FenSying times out with as low as 10 traces. The tests time-out in *Johnson's* cycle detection due to a high density of the number of related events or the number of cycles. The event relations of some buggy traces form as many as 2-3 millions cycles for test IDs 12-13, 14 million for test ID 20, and over 36 million for test ID 25, eventually leading to the timeout.

fastFenSying analyzes a remarkably smaller number of buggy traces in comparison to '#BT' ($\leq$2 traces for $\sim$85% of tests), as shown in column 'iter'. Thus, it can be concluded that a solution corresponding to a single buggy trace fixes more than one buggy traces. As a result, fastFenSying scales to tests with thousands of buggy traces with an average speedup of over 67x, with over 100x speedup in $\sim$41% of tests, against FenSying. fastFenSying could also scale for the tests with traces of millions of cycles because such traces were not encountered by the technique.

Consider test ID 16, `BTG` times out in 3/5 runs and completes in $\sim$100s in the remaining 2 runs. A `fence` is synthesized between two events that are inside a loop. Additionally, the *later* event is within a condition. Depending on where the `fence` is synthesized

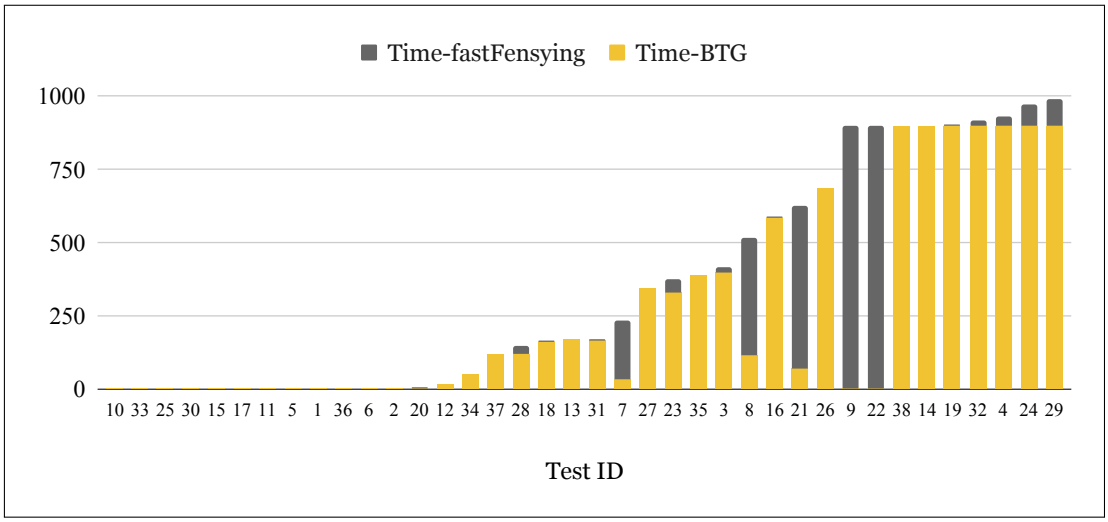


Figure 6.13: Time of analysis of `BTG` against `fastFenSying` internal modules (seconds)

(within the condition or outside), `BTG` either runs out of time or finishes promptly.

`BTG` times out in 3/5 runs for test ID 26 as well. However, the reason here is the additional non-optimal `fences` synthesized that increase the analysis overhead of the chosen `BTG` (CDSChecker).

Note that, for most benchmarks, `fastFenSying`'s scalability is limited by BTo. Observably, `fastFenSying`'s time of analysis is much lesser than FTo for such cases. Consider the bar graph in Figure 6.13 where the bars indicate the total time of analysis of `fastFenSying` on various tests in Table 6.4, ordered in increasing order of the total time of analysis (column 'Ttotal' of `fastFenSying`). The yellow components represent the portion of 'Ttotal' corresponding to 'Tbtg', while the black components represent 'TF'. It is evident from the graph that `BTG` takes up the major share of the total time of analysis. As a consequence, higher configurations of benchmarks could not be tested as `BTG` did not scale for the current configurations.

6.8 Scope, Limitations, and Future directions

Applicability under other version C/C++ memory model. The Weak-FenSying and Strong-FenSying techniques presented in §6.3.2 and §6.3.3 respectively are designed for the ISO 2011 standard of C/C++. The analyses are not directly applicable to the memory models presented in the subsequent standards. By defining suitable analysis corresponding to Weak-FenSying and Strong-FenSying, FenSying and fastFenSying techniques can be extended to support the the subsequent models of C/C++.

fence synthesis to restore sequential consistency. Some earlier works synthesize fences to restrict the program outcomes to those allowed under sequential consistency [6, 17, 33] or its variant [11]. Such restriction provides compatibility with existing analysis techniques for sequential consistency. However, most fences synthesis techniques [10, 49, 55, 63, 67], attempt to remove traces violating a safety property specification under their respective axiomatic definition of memory model, similar to FenSying and fastFenSying.

Alternate definition of optimality. Technique [16] assigns weights to various types of fences (similar to this work) and defines optimality on the weights of candidate solutions. Consequently, out of the candidate solutions $\{\mathbb{F}_i^{\texttt{sc}}\}$ and $\{\mathbb{F}_j^{\texttt{rel}}, \mathbb{F}_k^{\texttt{acq}}\}$, discussed in §6.2.1, a similar definition of optimality as in [16] would prefer the solution $\{\mathbb{F}_j^{\texttt{rel}}, \mathbb{F}_k^{\texttt{acq}}\}$, whereas FenSying and fastFenSying prefer $\{\mathbb{F}_i^{\texttt{sc}}\}$.

The definition of optimality in [16] is incomparable with that presented in §6.2.1, and their efficiency is subject to the input program and the underlying architecture. No prior work establishes the advantage of one definition over the other.

Modifying memory orders against synthesis of fences. A recent technique [73] fixes a buggy C11 program by strengthening memory access events instead of synthesizing fences. This work synthesizes C11 fences and stands fundamentally different from techniques that modify the memory orders of program events.

sc fences cannot restore sequential consistency [60], hence, strengthening memory

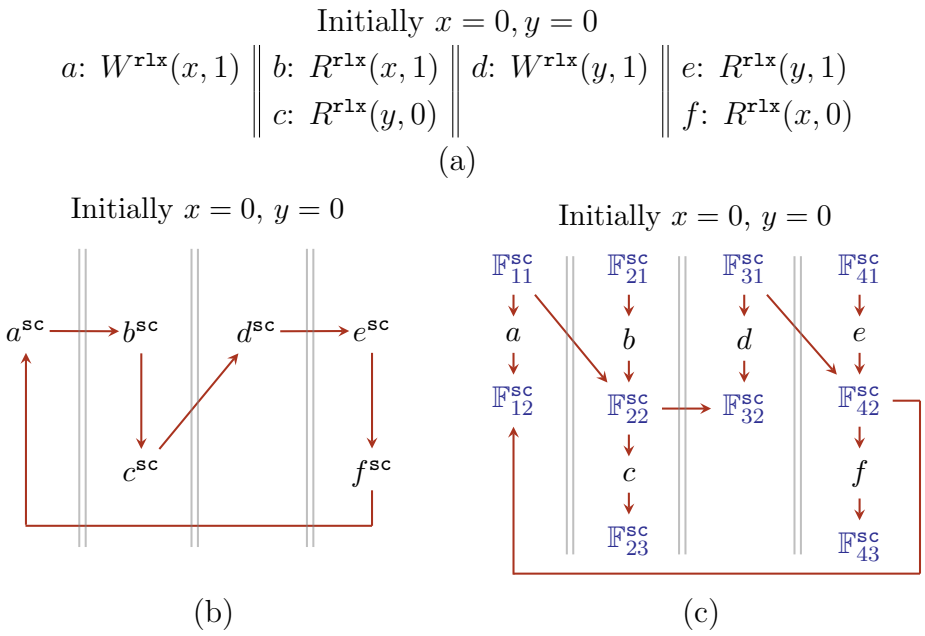


Figure 6.14: IRIW-rlx. (a) buggy trace (b) with strong memory orders, (c) with strong `fences`

orders may invalidate buggy traces that the strongest C11 `fences` cannot.

Consider the outcome IRIW with `rlx` memory orders for the `read` and `write` events, shown in Figure 6.14(a) (where $\rightarrow$ depicts the total-order on `sc` events). Assume that the outcome is buggy. Figure 6.14(b) and (c) show two transformations of the program. In Figure 6.14(b), let the memory order `sc` as superscripts represent that the memory orders of the respective `read` and `write` events are strengthened to `sc`. In Figure 6.14(c) the strongest kind of C11 `fences` are synthesized at all feasible synthesis locations.

The `fences` of Figure 6.14(c) do not introduce an ordering between the `write` events and the corresponding `read` events thereby allowing `reads` from initial events. Thus, the trace shown in Figure 6.14(c) cannot be invalidated by C11 `fences`. Whereas, changing the memory order of `read` and `write` events achieves the desired ordering and invalidates the outcome, as shown in Figure 6.14(b). `FenSying` and `fastFenSying` invalidate traces by synthesizing C11 `fences`, thus, the outcome in Figure 6.14(a)

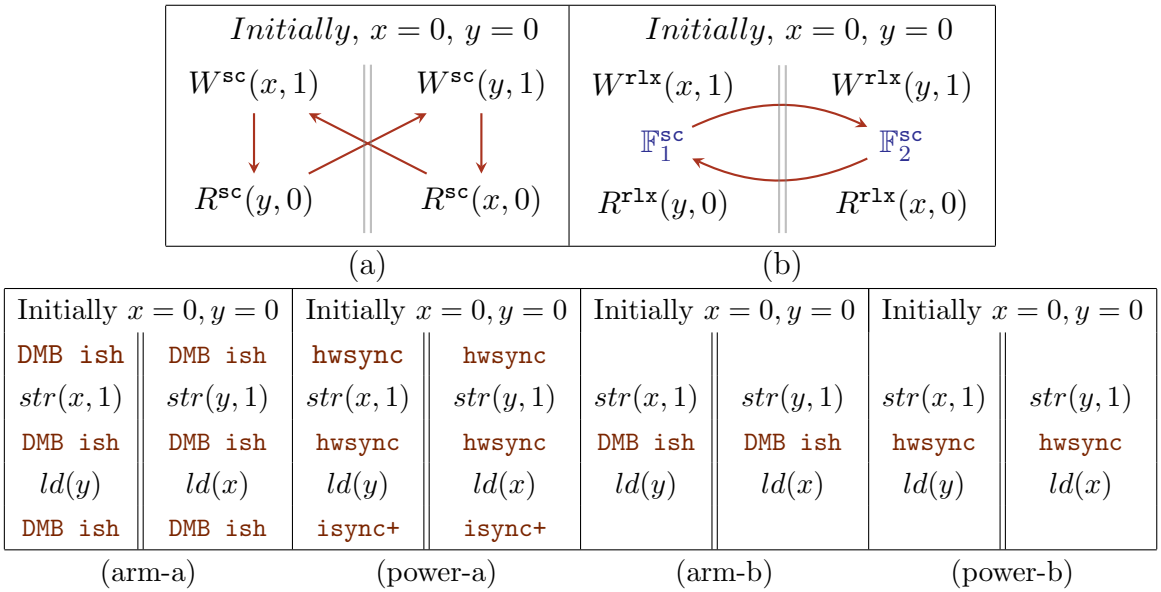


Figure 6.15: SB. (a) with strong memory orders, (b) with strong `fences`

cannot be invalidated by the techniques.

On the other hand, architectures translate the strongly ordered memory access events to memory access operation and supporting barriers. The translation of strongly ordered events to barriers may be sub-optimal.

Consider the two versions of the program store buffer (Figure 3.3(a)) shown in Figures 6.15(a) and (b), (where $\rightarrow$ depicts the total-order on `sc` events). In Figure 6.15(a) the memory orders of the `read` and `write` events are strengthened, while in Figure 6.15(b) C11 `fences` are synthesized. The translation of Figure 6.15(a) on ARM-v7 and Power is depicted in Figures 6.15(arm-a) and (power-a) respectively. The translation of Figure 6.15(b) ARM-v7 and Power is depicted in Figures 6.15(arm-b) and (power-b) respectively. Clearly each of the two architectures place additional (and unnecessary) barriers on interpreting barrier requirement from the memory orders of program events; while, `fences` present a better indication of the necessary barrier requirement.

[1] `isync+` refers to `cmp; bc; isync`

As a result, the two alternative approaches of strengthening memory orders of memory access events and synthesizing `fences` are incomparable and subjectively superior.

Threats to Validity

Scalability. Since SMC limited in scalability, the dependence of FenSying and fastFenSying on an SMC limits their scalability as well. There is a significant scope for improvement in performance of the techniques in terms of the time of analysis and scalability while ensuring near-optimality of the resulting fence synthesis.

Availability of benchmarks. There exist very few benchmarks of C11 programs that utilize C11 weak ordering guarantees, and there are even fewer benchmarks with bugs emerging from the weak ordering guarantees of C11.

Memory orders for testing. The benchmark suites such as SV-Comp [22] and SCT [66] contain `C` programs without associated memory orders. To test the weaker behaviors the memory access operations are associated with appropriate weak memory orders which are chosen by the authors.

6.8.1 Future directions

Efficient synthesis. The time complexity of FenSying and fastFenSying is dominated by the cycle detection, that is, $\mathcal{O}((|\mathcal{E}_\tau|+\mathtt{E}).(\mathtt{C}+1))$ where `E` is in $O(|\mathcal{E}_\tau|^2)$ and `C` is in $O(|\mathcal{E}_\tau|!)$. Hence, reducing the size of $\mathcal{E}$ can reduce the complexity of cycle detection.

The size of $\mathcal{E}$ can be reduced by reducing the number of candidate `fences` in the intermediate trace. Clever heuristics may be used to compute a well suited set of candidate `fences` instead of inserting a candidate `fence` at each feasible location. Further, bounding techniques , such as depth bounding on the number of events or the number of event relations may also reduce the complexity of cycle detection.

Alternate `BTG`. As depicted by the graph in Figure 6.13, `BTG` takes up the major share

of the time of analysis of fastFenSying. Further, it can be observed from Table 6.4 that there exist tests for which the BTG (CDSChecker) can compute the complete set of buggy traces but cannot generate them one at a time from the transformed programs of each iteration of fastFenSying (such as test IDs 19 and 32). It may be concluded that CDSChecker's performance decreases with fences in the input program.

An alternate BTG may significantly improve the scalability of fastFenSying. Alternatively, collecting the set of buggy traces from CDSChecker and internally computing the set of invalidated and remaining traces in each iteration can also be explored.

Eliminating fences for efficient fixed programs. Fence elimination is out of the scope of this work. For future, elimination of program fences can be explored aiming at generating fixed programs with lower performance overhead. A similar approach can be used to extend FenSying and fastFenSying to take bug free programs as input for computing performance-efficient versions of the programs.

Support for program fence elimination can be extended primarily by including program fences to compute the min-model using the SAT solver.

Support for richer constructs. The technique may be extended to support richer constructs such as coarse-grained locking. The modules Weak-FenSying and Strong-FenSying may be suitably extended for detecting cycles in the presence of locks.

6.9 Concluding remarks

This work presents the first fence synthesis techniques for the C11 memory model, called FenSying and fastFenSying. The techniques provide an automated solution for fixing a C/C++ program that violates a user specified (assert) property. The techniques take buggy traces of a C/C++ program as input and synthesize C11 fences in the program to make the program bug free.

The FenSying technique is shown to be sound and optimal. However, the optimal computation is NP-hard and empirically the technique is seen to suffer in scalability.

The fastFenSying technique is sound. The technique is not optimal but can be perceived as *near-optimal* for two reasons; first, it is shown that the technique is optimal for a single buggy trace, and second, empirically it is observed that fastFenSying computes optimally in over 99.5% tests. This work also presents the worst case time complexity analysis for the FenSying and fastFenSying algorithms.

The FenSying and fastFenSying techniques are accompanied with an implementation for C and C++ buggy input programs. This work presents the corresponding implementation details including the structural overview and key components.

The implementation of the FenSying and fastFenSying techniques are tested over 1300+ litmus tests from previous works,with the focus on soundness and optimality (for FenSying). FenSying and fastFenSying timeout for 4% and 2.5% of the tests, respectively. For the remaining tests, the techniques perform a sound analysis. Both the techniques also produce an optimal solution for the remaining tests.

To determine the effectiveness of FenSying and fastFenSying techniques and their corresponding implementations, the presented tool is tested on challenging benchmarks. The tests compare the techniques on the time of analysis, the number of fences synthesized and strengthened, and scalability. The comparative results on benchmarks highlight that fastFenSying significantly outperforms FenSying in terms of the time of analysis and scalability, while incurring a small number of extra fences in a small number of tests.

Finally, this work discusses the scope of FenSying and fastFenSying, and highlights differences with some popular and similar notions in related fields. Based on the scope of the techniques this work also proposes worthy future directions.

Chapter 7

Conclusion

This work presents techniques for the ease of development of efficient programs that produce expected outcomes under relaxed memory consistency models. The focus of this work is on two objectives, (i)*verification efficiency* and (ii) *developer productivity* and *runtime efficiency*.

The work proposes the techniques,

- `ViEqui`, under the objective of verification efficiency, presented in Chapter 4;
- `MoCA`, under the objectives of verification efficiency and developer productivity, presented in Chapter 5; and,
- `FenSying` and `fastFenSying`, under the objectives of developer productivity and runtime efficiency, presented in Chapter 6.

`ViEqui` is an efficient stateless model checker that is based on a novel equivalence relation for trace partitioning called view-equivalence, that is as coarse as any existing equivalence relation. It is shown with relevant theorems that `ViEqui` is sound, complete, and optimal in its exploration.

`MoCA` is a precise stateless model checker for a restriction of the C11 memory model

over MCA. This work presents the restriction of the C11 happens-before relation and proposes coherence conditions precisely for MCA. The work introduces novel type of events called shadow-writes that simulate reordering through interleaving to achieve the restriction. It is shown with relevant theorems that `MoCA` is sound, coherent under C11 and precise for C11 restricted to MCA.

`FenSying` and `fastFenSying` are the first `fence` synthesis techniques for automated fixing of buggy C11 programs. This work presents the approach of invalidating buggy outcomes by breaking their coherence with `fences`. The `FenSying` technique performs optimal (minimal) `fence` synthesis by introducing the smallest number of weakest `fences` to fix a buggy program but suffers in the time of analysis and scalability. The `fastFenSying` technique efficiently performs `fence` synthesis while performing optimally in most cases. It is shown with relevant theorems that `FenSying` is sound, and optimal and `fastFenSying` is sound and *near*-optimal.

Each technique is accompanied by an implementation to empirically validate the claims and show the effectiveness of the techniques. The techniques are validated on relevant litmus tests and `ViEqui` and `MoCA` are further compared against state-of-the-art comparative techniques. Since, `FenSying` and `fastFenSying` are the first techniques in their scope, their performance is compared against each other. Each technique is also presented with a worst case time complexity analysis.

This document briefly presents the relevant background for this work in Chapter 3. Further, a detailed background corresponding to each technique is presented in their corresponding chapters (Chapters 4, 5 and 6). The background includes details on the existing related work and the difference of the existing works with the proposed techniques. Each technique is also presented with the scope of the work, comparison with other similar notions and representations, and worthy future directions.

In summary, this work proposes provably correct techniques for the said objectives. Each technique is further supported with a useful implementation, and empirical study of the techniques establish relevance and efficacy of each technique and utility of the accompanying implementation.

Appendix A

Glossary

ARM	Memory model of ARM architecture for versions 8 and later
`acq`	memory order *acquire* (`memory_order_acquire`)
`acq-rel`	memory order *acquire-release* (`memory_order_acq_rel`)
C11	`C/C++` ISO 2011 memory model
bug	A valid program trace that violates a user specified (assert) property
DPOR	Dynamic Partial Order Reduction
`(fast)FenSying`	`FenSying` and `fastFenSying`
MCA	Multi-copy atomics
`na`	non-atomic event type
non-MCA	Non-multi-copy atomics
5 POR	Partial order reduction
PSO	Partial store order
`rel`	memory order *release* (`memory_order_release`)
`rlx`	memory order *relaxed* (`memory_order_relaxed`)
`sc`	memory order *sequentially consistent* (`memory_order_seq_cst`)
SMC	Stateless Model Checker
TSO	Total store order

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*, 49(1):373–384, 2014.

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for tso and pso. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, 2015.

[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for tso and pso. *Acta Informatica*, 54(8):789–818, 2017.

[4] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: a foundation for optimal dynamic partial order reduction. *Journal of the ACM (JACM)*, 64(4):1–49, 2017.

[5] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1117–1132, 2019.

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *International Static Analysis Symposium*, pages 164–180. Springer, 2012.

[7] 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. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[8] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for power. In *International Conference on Computer Aided Verification*, pages 134–156. Springer, 2016.

[9] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[10] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under pso. In *International Conference on Networked Systems*, pages 32–47. Springer, 2015.

[11] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for tso. In *European Symposium on Programming Languages and Systems*, pages 308–332. Springer, 2015.

[12] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. In *International Conference on Computer Aided Verification*, pages 341–366. Springer, 2021.

[13] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J Stuckey. Context-sensitive dynamic partial order reduction. In *International Conference on Computer Aided Verification*, pages 526–543. Springer, 2017.

[14] Elvira Albert, Maria Garcia De La Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J Stuckey. Optimal context-sensitive dynamic partial order reduction with observers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 352–362, 2019.

[15] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained dynamic partial order reduction. In *International Conference on Computer Aided Verification*, pages 392–410. Springer, 2018.

[16] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence. In *International Conference on Computer Aided Verification*, pages 508–524. Springer, 2014.

[17] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *International Conference on Computer Aided Verification*, pages 258–272. Springer, 2010.

[18] Fabrizio Angiulli, Rachel Ben-Eliyahu-Zohary, Fabio Fassetti, and Luigi Palopoli. On the tractability of minimal model computation for some cnf theories. *Artificial Intelligence*, 210:56–77, 2014.

[19] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–248. Springer, 2018.

[20] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *ACM SIGPLAN Notices*, 46(1):55–66, 2011.

[21] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. *ACM SIGPLAN Notices*, 50(10):367–385, 2015.

[22] Dirk Beyer. Software verification: 10th comparative evaluation (sv-comp 2021). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422. Springer, 2021.

[23] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

[24] Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. The reads-from equivalence for the tso and pso memory

models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[25] Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[26] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[27] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. Value-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[28] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*, pages 368–371. IEEE, 2003.

[29] Edmund M Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[30] Edmund M Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30, 2012.

[31] Robert J Colvin and Graeme Smith. A wide-spectrum language for verification of programs on weak memory models. In *International Symposium on Formal Methods*, pages 240–257. Springer, 2018.

[32] computersforpeace. model-checker. https://github.com/computersforpeace/model-checker, 2021.

[33] Xing Fang, Jaejin Lee, and Samuel P Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 285–294, 2003.

[34] Cormac Flanagan, Stephen N Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, pages 262–277. Springer, 2002.

[35] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1):110–121, 2005.

[36] Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the armv8 architecture, operationally: Concurrency and isa. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 608–621, 2016.

[37] Vojtech Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. Precise predictive analysis for discovering communication deadlocks in mpi programs. In *International Symposium on Formal Methods*, pages 263–278. Springer, 2014.

[38] gabriel araujjo. model-checker. https://github.com/gabriel-araujjo/model-checker, 2021.

[39] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[40] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, 1997.

[41] Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[42] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *International Conference on Computer Aided Verification*, pages 438–449. Springer, 1993.

[43] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[44] Ashutosh Gupta, Thomas A Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. Succinct representation of concurrent trace sets. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 433–444, 2015.

[45] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[46] ISO/IEC. Programming languages – c. international standard. www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf, 2011.

[47] Alglave Jade and Maranget Luc. diy7 tool suite. http://diy.inria.fr/doc/index.html, 2017. Accessed: 2020-02-12.

[48] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[49] Saurabh Joshi and Daniel Kroening. Property-driven fence insertion using reorder bounded model checking. In *International Symposium on Formal Methods*, pages 291–307. Springer, 2015.

[50] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *ACM SIGPLAN Notices*, 52(1):175–189, 2017.

[51] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for c/c++ concurrency. *Proceedings of the ACM on Programming Languages*, 2(POPL):17, 2017.

[52] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022.

[53] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 96–110, 2019.

[54] Michalis Kokologiannakis and Viktor Vafeiadis. Hmc: Model checking for hardware memory models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1157–1171, 2020.

[55] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.

[56] Robert Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 345–357. Springer, 1998.

[57] M Kusano and C.Wang. Thread-modular static analysis for relaxed memory models. In *Proceedings of 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[58] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 799–809, 2016.

[59] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. *ACM SIGPLAN Notices*, 51(1):649–662, 2016.

[60] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++ 11. *ACM SIGPLAN Notices*, 52(6):618–632, 2017.

[61] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers c-28*, 9:690–691, 1979.

[62] Christopher Lidbury and Alastair F Donaldson. Dynamic race detection for c++ 11. In *ACM SIGPLAN Notices*, volume 52, pages 443–457. ACM, 2017.

[63] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in pso memory systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 339–353. Springer, 2013.

[64] ARM Ltd. Arm architecture reference manual (armv8, for armv8-a architecture profile). https://developer.arm.com/documentation/ddi0487/aa, 2017.

[65] Antoni Mazurkiewicz. Trace theory. In *Advanced course on Petri nets*, pages 278–324. Springer, 1986.

[66] Mc-imperial. Sctbench. https://github.com/mc-imperial/sctbench, 2022.

[67] Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *International Static Analysis Symposium*, pages 237–252. Springer, 2014.

[68] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, 42(6):446–455, 2007.

[69] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, 2008.

[70] Huyen TT Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. In *International Conference on Computer Aided Verification*, pages 354–371. Springer, 2018.

[71] Nidhugg. Nidhugg. https://github.com/nidhugg/nidhugg, 2021.

[72] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with c/c++ atomics. In *ACM SIGPLAN Notices*, volume 48, pages 131–150. ACM, 2013.

[73] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, et al. Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 530–545, 2021.

[74] Doron Peled. All from one, one for all: on model checking using representatives. In *International Conference on Computer Aided Verification*, pages 409–423. Springer, 1993.

[75] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of c/c++ concurrency. *arXiv preprint arXiv:1606.01400*, 2016.

[76] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[77] rinspect. rinspect. https://github.com/rinspect/rinspect, 2017.

[78] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *International Conference on Concurrency Theory*, page 456–469, 2015.

[79] Divyanjali Sharma and Subodh Sharma. Thread-modular analysis of release-acquire concurrency. In *International Static Analysis Symposium*, pages 384–404. Springer, 2021.

[80] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Scalable dynamic partial order reduction. In *International Conference on Runtime Verification*, pages 19–34. Springer, 2013.

[81] Sanjana Singh, Divyanjali Sharma, and Subodh Sharma. Dynamic verification of c11 concurrency over multi copy atomics. In *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 39–46. IEEE, 2021.

[82] Thibault Suzanne and Antoine Miné. Relational thread-modular abstract interpretation under relaxed memory models. In *Asian Symposium on Programming Languages and Systems*, pages 109–128. Springer, 2018.

[83] Mohammad Taheri, Arash Pourdamghani, and Mohsen Lesani. Polynomial-time fence insertion for structured programs. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[84] Abhishek Udupa, Ankush Desai, and Sriram Rajamani. Depth bounded explicit-state model checking. In *International SPIN Workshop on Model Checking of Software*, pages 57–74. Springer, 2011.

[85] 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 *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, 2015.

[86] Antti Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, pages 491–515. Springer, 1989.

[87] Chao Wang, Sudipta Kundu, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. *Formal aspects of computing*, 23(6):781–805, 2011.

[88] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M Kirby. Efficient stateful dynamic partial order reduction. In *International SPIN Workshop on Model Checking of Software*, pages 288–305. Springer, 2008.

[89] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation*, pages 250–259, 2015.

List of Publications

Conference publications and submissions under review

[1] Sanjana Singh, Divyanjali Sharma, Ishita Jaju, and Subodh Sharma. Fence synthesis under the c11 memory model. In *Automated Technology for Verification and Analysis: 20th International Symposium, ATVA 2022, Virtual Event, October 25–28, 2022, Proceedings*, pages 83–99. Springer, 2022.

[2] Sanjana Singh, Divyanjali Sharma, and Subodh Sharma. Dynamic verification of C11 concurrency over multi copy atomics. In *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 39–46. IEEE, 2021.

[3] Sanjana Singh, and Subodh Sharma. Optimal Stateless Model Checking based on View-equivalence. *Invited to resubmit a revised version in OOPSLA 2023 with an 'Accept' and three 'Weak accepts'.*

Extended Versions

[1] Sanjana Singh, Divyanjali Sharma, Ishita Jaju, and Subodh Sharma. Fence synthesis under the c11 memory model. arXiv preprint arXiv:2208.00285, 2022.

[2] Sanjana Singh, Divyanjali Sharma, and Subodh Sharma. Dynamic Verification of C/C++11 Concurrency over Multi Copy Atomics. arXiv preprint arXiv:2103.01553, 2021

Biography

Sanjana Singh
Ph.D.
Computer Science and Engineering
Indian Institute of Technology Delhi

Research Interests

Concurrency, Program Analysis, Verification, Model Checking, Weak memory models

Work Experience and Education

- Indian Institute of Technology Delhi 2016-2023

  Ph.D. and Teaching Assistant, Department of Computer Science and Engineering. Completed Ph.D. under Subodh Sharma, Department of Computer Science and Engineering, IITD, on Program analysis under relaxed memory concurrency.

- Kobayashi Lab at University of Tokyo, Japan 2017

  3 weeks under Japan-Asia Youth (Sakura Science) Exchange Program Research exchange leading to collaborations

- Jaypee University of Information Technology, Solan 2013-2016

Designation: Assistant Professor

Additionally, T&P CSE Head adn Faculty in charge of CSE Technical Club

- Bharti Airtel Ltd., Gurgaon 2011

  Designation: Summer Intern

- Jaypee Institute of Information Technology, Noida 2008-2013

  Integrated B.Tech + M.Tech (Computer Science and Engineering)

  Dissertation title: Bug localization for Exception Handling and Multithreading through source code mutation

Ongoing Research Projects

- Automated Bug Detection and Repair in C11 Programs: Insights and Experiences (under peer review)
- Lock-aware extension of Optimal Stateless Model Checking based on a view-equivalence.

Non-research Projects

- Software Source-Code Analysis Related to Plagiarism Dispute

  A report developed for the Hon'ble Delhi High Court.
- Website design for Vertecs 2 research group at IIT Delhi
- Website design for The Second Indian SAT+SMT School, Mysore, India

Technical Presentations / Talks

- Formal Methods Update Meeting 2023 Jul 2023

  Title: Optimal Stateless Model Checking based on View-equivalence

- Software Engineering Research in India (SERI) Update Meeting Jun 2023
  Title: Fence Synthesis under the C11 Memory Model
- Automated Technology for Verification and Analysis Conference Oct 2022
  Title: Fence Synthesis under the C11 Memory Model
- Theoretical Aspects of Software Engineering Conference Aug 2021
  Title: Dynamic verification of C11 concurrency over multi-copy atomics

Conference and Workshops

- FMU, Goa, India Jul 2023
- SERI, Goa, India Jun 2023
- ATVA 2022, Beijing, China (attended virtually) Oct 2022
- TASE 2021, Shanghai, China (attended virtually) Aug 2021
- Japan-Asia Youth (Sakura Science) Exchange Program for 3 weeks at the University of Tokyo, Japan. Jun-Jul 2017
- Winter School in Software Engineering, Pune, India Dec 2017
- The Third Indian SAT+SMT School, Hyderabad, India Dec 2018
- The Second Indian SAT+SMT School, Mysore, India Dec 2017
- The First Indian SAT+SMT School, Mumbai, India Dec 2016
- IEEE INDICON 2015, New Delhi, India Dec 2015
- IEEE ICIIP 2015 (Organizer), Solan, India Dec 2015
- IEEE PDGC 2014 (Organizer), Solan, India Dec 2014
- IEEE ICIIP 2013 (Organizer), Solan, India Dec 2013

Tools

1. FenSying and fastFenSying:
   singhsanjana. fensying. https://github.com/singhsanjana/fensying, 2022.

2. ViEqui artifact (VM):
   https://zenodo.org/record/7589364#.Y9k3HtJBxH4, 2023

   Also available as a docker image at:
   https://www.dropbox.com/sh/ld34pj8setdkzb8/AABObLd6nxC8iJAcCeCwD-sda?dl=0