Soundness of Predictive Concurrency Analyses

SHUYANG LIU, University of California at Los Angeles, USA
DOUG LEA, SUNY Oswego, USA
JENS PALSBERG, University of California at Los Angeles, USA

A predictive analysis takes an execution trace as input and discovers concurrency bugs without accessing the program source code. A sound predictive analysis reports no false positives, which sounds like a property that can be defined easily, but which has been defined in many different ways in previous work. In this paper, we unify, simplify, and generalize those soundness definitions for analyses that discover concurrency bugs that can be represented as a consecutive sequence of events. Our soundness definition is graph based, separates thread-local properties and whole-execution properties, and works well with weak memory executions. We also present a three-step proof recipe, and we use it to prove six existing analyses sound. This includes the first proof of soundness for a predictive analysis that works with weak memory.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Concurrency, Dynamic Program Analysis, Soundness, Weak Memory Models

ACM Reference Format:
Shuyang Liu, Doug Lea, and Jens Palsberg. 2025. Soundness of Predictive Concurrency Analyses. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 101 (April 2025), 44 pages. https://doi.org/10.1145/3720435

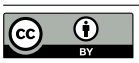
1 Introduction

Concurrent programs are often error-prone due to the non-deterministic nature of their executions. Over the last two decades, various techniques have been proposed to catch bugs in concurrent programs. Among them, *dynamic predictive analysis* has become a promising research area. A dynamic predictive analysis takes an execution trace as input and discovers concurrency bugs without accessing the program source code. Recent works in the area of predictive analysis [4–8, 11, 18, 19, 21, 27, 28] support *soundness* as one of their most important properties. A sound predictive analysis reports no false positives, which sounds straightforward but has been defined in many different ways in previous work [8, 11, 18, 19, 21, 25, 27, 28].

Three problems emerge from the use of varying soundness criteria in previous work. First, there is no single recipe for proving soundness, so existing proofs cannot be directly applied or adapted to a new algorithm. Second, some previous proofs later turned out to be flawed when counterexamples emerged, without revealing whether the algorithm or the soundness proof were wrong. Third, existing soundness criteria and proof techniques have no support for weak memory models. This is in part due the focus on trace-based soundness definitions in previous work, which is a poor fit for weak memory models.

To address these issues, we propose a modular framework for soundness of predictive analyses that unifies, simplifies, and generalizes existing approaches and handles weak memory behaviors.

Authors' Contact Information: Shuyang Liu, University of California at Los Angeles, Los Angeles, USA, sliu44@cs.ucla.edu; Doug Lea, SUNY Oswego, Oswego, USA, dl@cs.oswego.edu; Jens Palsberg, University of California at Los Angeles, Los Angeles, USA, palsberg@ucla.edu.



ACM 2475-1421/2025/4-ART101
https://doi.org/10.1145/3720435

Instead of traces, our framework is based on *execution graphs*, which are widely used in the field of axiomatic memory models [1–3, 22, 23]. Compared to traces, execution graphs distinguish the sequential semantics per thread versus global weak memory semantics as two separate aspects of validity. The former is affected by the source language semantics, and the latter is determined by the axiomatic memory model. As a result, this feature enables a modular structure of soundness, which also leads to a simple recipe for constructing a proof.

Our contributions are the following:

- §4 We propose a modular soundness definition parameterized by a memory model. The definition is more general than existing soundness definitions, reflects a closer connection with the language semantics, and can be applied with weak memory models. Specifically, we derive an executability property from a language semantics and augment existing multicopy-atomic memory models with lock semantics. We analyze the execution spaces represented by the existing soundness definitions and explain their relationships with our soundness definition.
- §5 We provide a three-step recipe for constructing a soundness proof by constructing a witness execution graph that satisfies our soundness definition. We also provide a set of reusable lemmas that can be applied in the construction of proofs under other memory models or semantic constraints.
- §6 We use our recipe to prove the soundness of six data race predictive analyses [8, 9, 15, 18, 19, 21]. Among them, the proof for MCR-TSO [9] is the first proof of soundness for a predictive analysis that works with weak memory. In addition, we extend the approach of Huang and Huang [9] and define a new data race predictor based on a transformation that can discover more data races under TSO. We show that the extended data race predictor is sound following the same recipe in Appendix F.

§2 reviews the past works and the soundness definitions they have used; §3 introduces the formal definitions; §7 concludes the paper.

Scope and Limitations. In this paper we focus on the *strong* soundness theorem, i.e., every reported bug indicates a valid execution that witnesses the bug. On the other hand, soundness definitions for data races may be extended to state that every reported data race indicates either the presence of a race or a predictable deadlock, i.e., the *weak* soundness theorem. Our treatment does not currently incorporate the deadlock provision covered by the weak soundness theorem, which is used in some partial-order-based data race predictive analyses [6, 11, 27]. While one can modify our soundness definition with a disjunction to accommodate the weak soundness theorem, precisely capturing a predictable deadlock pattern with respect to the reported data race requires further exploration.

We represent the notion of a *bug* as a consecutive sequence of events (the formal definition can be found in Section 3). This corresponds to most common definitions of data race errors, but not necessarily to violations of higher-level properties such serializability. We leave as future work techniques for transforming these into forms that are amenable to our approach.

We restrict attention to multicopy-atomic memory models. Extending coverage to non-multicopy atomic models would introduce additional potential executions that are not possible on other platforms. Some well-known multicopy-atomic models include x86-TSO [20], ARMv8 [1, 23], and RISC-V, while ARMv7[17] and the PowerPC [24] models are non-multicopy-atomic. Java and C/C++ have non-multicopy atomic models to provide multi-platform supports. The primary challenge in accommodating non-multicopy-atomic models is integration with the semantics of lock operations, which we leave as future work. Further, we omit Read-Modify-Write (RMW) operations. Accommodating them would add some cases to our soundness definition. Adopting these restrictions allows us to omit coverage of issues that add complexity without bearing on the basic logic of our approach.

Name	Memory Model	Bug Type	Sound Witness Definition	Ref.
HB (FastTrack)	SC	Data Race	Correct Reordering	[5, 15]
CP	SC	Data Race	Correct Reordering	[27]
WCP	SC	Data Race	Correct Reordering	[11]
SHB	SC	Data Race	Relaxed CR	[18]
SyncP	SC	Data Race	Sync-Preserving CR	[19]
SPD	SC	Deadlock	Sync-Preserving CR	[28]
OSR	SC	Data Race	Optimistic CR	[26]
M2	SC	Data Race	Correct Reordering	[21]
RVPredict	SC	Data Race	Feasible Closure	[8]
SeqCheck	SC	Bugs represented as a sequence of events	Feasible Closure	[4]
MCR-tso	TSO	Data Race	Correct Reordering informally relaxed with TSO semantics	[9]

Fig. 1. Various Criteria for a Sound Witness from Past Work

Acknowledgement. We thank the anonymous reviewers for helping to clarify limitations and opportunities for future work. This material is based upon work supported by the National Science Foundation under Grant No. 1815496.

2 Motivation

A concurrency bug is a sequence of events that occur in some specific order. For example, a *data race* is a pair of two conflicting events ordered consecutively. Since a predictive analysis predicts whether a bug can occur in some execution of the program that produced the input execution, the soundness theorem of an analysis is defined by the existence of a *witness* execution that exhibits the bug. Therefore, characterizing the valid witness executions and showing that each reported bug corresponds to a witness execution that satisfies all the characteristics become critical in the soundness proofs of existing works in predictive analysis. Fig. 1 shows a summary of the soundness criteria used in existing works.

FastTrack [5] is a data race analysis that builds a partial order, the *happens-before* (HB) order [15], among the events in a given input trace. While FastTrack with HB can report multiple data races in a single run, only the first data race is guaranteed to be sound. The soundness theorem of FastTrack states that the algorithm correctly implements the HB order using vector clocks. On the other hand, the soundness of the HB order was assumed in the paper.

After FastTrack, CP [27] and WCP [11] built weaker partial orders than HB and used Correct Reordering (CR) to characterize valid witness traces. Correct Reordering requires each read event that appears in the witness execution to maintain the same values as in the input execution. The soundness theorems of both algorithms state that the first race reported by the algorithms is a HB-race or there is a deadlock in a correct reordering of the input trace. Both the soundness of Correct Reordering and the HB order were implicitly assumed.

Mathur et al. [18] shows an improvement over HB that ensures the soundness of *all* reported data races. Their idea is to build a strictly stronger partial order, *schedulable-happens-before* (SHB), which orders the unsound HB-races after the first race, so that all reported data races are sound. The soundness of SHB is based on a relaxed version of Correct Reordering, denoted as Relaxed

CR in Fig. 1. Relaxed CR has the same requirements of Correct Reordering except that if a read is the last event of a thread in the witness execution, then it does not have to maintain the same value. In addition to the soundness of SHB, Mathur et al. [18] also formally proved that HB is sound under the definition of Relaxed CR. But the soundness of Relaxed CR and Correct Reordering were implicitly assumed.

Mathur et al. [19] and Tunç et al. [28] used a more restricted version of Correct Reordering called Sync-Preserving CR. In addition to the requirements of Correct Reordering, the critical sections that appear in a Sync-Preserving CR follow the same synchronization order as in the input trace. The proofs of both works were done by linearizing an event closure called SRFClosure (or SPClosure in [28]) using the same trace order of the input trace. The soundness of Sync-Preserving CR relied on Correct Reordering, which was implicitly assumed.

Shi et al. [26] used Optimistic CR, which relaxes the notion of Sync-Preserving CR to capture a slightly different set of data races. Comparing to Sync-Preserving CR, Optimistic CR allows certain critical sections to be reordered. On the other hand, since the Optimistic lock-closure sometimes includes more events than the Sync-preserving closure, as demonstrated in their example [26, Example 7], Optimistic CR and Sync-Preserving CR are not comparable in general. Similar to Sync-Preserving CR, the soundness of Optimistic CR relied on Correct Reordering, of which the correctness was implicitly assumed.

Pavlogiannis [21] used Correct Reordering as the soundness criteria for a witness trace. The algorithm computes a partial order P over a subset of events from the input trace and used it to determine if a pair of events forms a data race. The key correctness result is their Theorem 3.1, which showed that for a trace-closed partial order P computed from the input trace, the Max-Min algorithm that solves a particular linearization problem based on P always produce a correct reordering [21, Theorem 3.1]. But the soundness of Correct Reordering was implicitly assumed.

Huang et al. [8] used the notion of Feasible Closure of traces derived from the causal model of ŞerbănuŢă et al. [25]. Their main correctness result [8, Theorem 1] showed the existence of a mapping from symbolic feasible traces to concrete feasible traces, but did not formally show that the set of concrete feasible traces can be generated by the same program of the input trace.

Similarly, although with a different approach to pretict concurreny bugs, Cai et al. [4] developed SeqCheck using Feasible Closure as their correctness criteria as well. The main soundness theorem [4, Theorem 1] stated that their algorithm returns a trace that is an element of the Feasible Closure given the input trace. The proof was done by induction on the structure of the trace that is produced by their algorithm and showing that it has the same structure as a trace in the Feasible Closure.

Huang and Huang [9] used a restricted version of Feasible Closure that preserves all read values, but relaxed with a memory operation constraint, Φ_{mem}, to capture the weak behavior under x86-TSO. Effectively, their soundness criteria is the same as Correct Reordering relaxed with the TSO write-buffer semantics. Instead of requiring all program orders to be preserved, it does not require write-to-read program orders to be preserved. However, the soundness of the constraints was left unproved.

From the review above, we can see that the soundness criteria from existing works all specify properties that a witness execution has to satisfy given an input execution. For dynamic predictive analyses, the program source code is kept unknown. Therefore, all existing soundness definitions focus on characterizing a set of valid executions that can be produced by *all* programs that can generate the input execution. In other words, existing soundness definitions focus on executions in an *intersection* space of all the programs that can generate the input execution. Fig. 2 shows a hierarchy of the sets of valid executions specified by the soundness criteria from past works. Each rectangle represents a set of executions that satisfy the corresponding soundness definition.

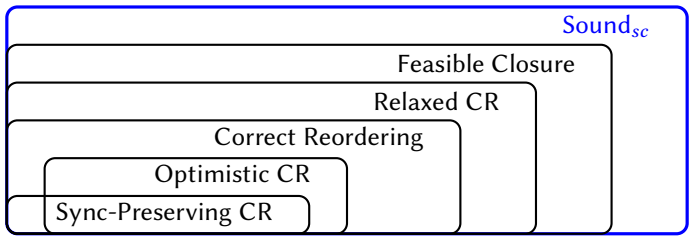


Fig. 2. Hierarchy of Existing Sound Definitions under Sequential Consistency

However, since all existing definitions focus on an intersection execution space, the size of this execution space depends on the amount of information about the program semantics captured in the input execution. Indeed, the analyses that used Feasible Closure [7, 8, 25] record branch events in the input execution whereas other analyses [11, 18, 19, 21, 27, 28] do not. This difference enables Feasible Closure to subsume other soundness definitions in Fig. 2. A better approach is to define soundness independently of how much information the input execution captures. We focus on characterizing the set of valid executions that can be produced by a single program that generates the input execution. In other words, our soundness definition is *not based on an intersection* of execution sets, but is based on the complete execution set of each program that generates each input execution. Obviously, this set of executions subsumes all other sets specified by existing definitions, and is shown as the blue rectangle in Fig. 2 annotated with Sound$_{sc}$. The set inclusion relations shown in Fig. 2 is the following.

$$\begin{aligned} \text{Sound}_{sc}\ \text{[this paper]} &\supseteq \text{Feasible Closure [8]} \\ &\supseteq \text{Relaxed CR [18]} \\ &\supseteq \text{Correct Reordering [11, 21, 27]} \\ &\supseteq \text{Sync-Preserving CR [19, 28]} \end{aligned}$$

The subscript *sc* in the name of our soundness stands for sequential consistency. While sequential consistency is subsumed by other memory models, the sets of valid executions under different memory models are not comparable in general. In §4, we give a general definition of soundness, $\text{Sound}_{\mathcal{M}}$, under a memory model $\mathcal{M}$.

The dynamic nature of the analyses, that is, having no access to the program source code, makes this set of executions unknown to the analysis tools. As a result, the traditional approach, which requires one to step through the source code of the program to show an execution is valid, cannot be used to prove a predictive analysis sound. Instead, we leverage the existing input execution and provide a three-step proof technique in Section 5. Note that the restriction of having no access to the program source code does not affect the soundness definition, but only the proof technique.

In Appendix A, we provide the formal definitions of the existing soundness criteria and show their relationships with our soundness definition.

3 Preliminaries

In this section, we recall standard definitions of programs and execution graphs [22], which we extend with acquire and release events [21], and a straightforward sync order over such events. Our operational semantics of programs uses the standard notion of a thread state (Definition 3). We instrument thread states with symbolic book-keeping information that plays no role in the semantics but helps in proofs of soundness.

We use standard notation for relations and functions. For a relation R, we use $R^?$, R^+, and R^* to denote the reflexive, transitive, and reflexive-transitive closures of R, respectively. We use R^{-1} for

$$
\begin{array}{lll}
i \in \mathsf{Instr} & ::= & r := e \\
& & \mid \texttt{if } e \texttt{ goto } n \\
& & \mid [x] := e \\
& & \mid r := [x] \\
& & \mid \texttt{lock}(l) \mid \texttt{unlock}(l) \\
e \in \mathsf{Expr} & ::= & r \mid e + e \mid e - e \mid e * e \mid v
\end{array}
\qquad
\begin{array}{r}
x \in \mathsf{Loc} \\
v \in \mathsf{Val} \\
r \in \mathsf{Reg} \\
l \in \mathsf{Lock} \\
n \in \mathbb{N}
\end{array}
$$

Fig. 3. Instructions of the Language

the inverse of R. Given a set A, $[A]$ is the identity relation on A, $\wp(A)$ is the power set of A. The composition of two relations R_1 and R_2 is written as $R_1; R_2$. $R|_A$ stands for R restricted to the set A. We say a set A is downward-closed with respect to a relation R if for each element $e \in A$, if there is $\langle e', e\rangle \in R$, then $e' \in A$. We use the following domains: Loc is a set of shared memory locations; Lock is a set of locks; Val is a set of concrete integer values; Sym is a set of symbols; Reg is a set of thread-local registers; and Thrd is a set of natural numbers for thread identifiers. In addition, we assume all memory locations are fixed.

3.1 Programs

A concurrent program consists of a set of threads, each containing a list of instructions. Formally, a *program* is a map from thread identifiers to sequential programs $P : \mathsf{Tid} \to \mathsf{Sprog}$, where $\mathsf{Sprog} = \mathbb{N} \to \mathsf{Instr}$ and the set of instructions is defined in Fig. 3.

Our sequential programs use standard instructions, including `lock` and `unlock` instructions for the acquire and release operations of locks. These two instructions allow us to reason about lock operations without worrying about the implementations of locks. We assume the implementation of `lock` and `unlock` is correct and guarantees lock-fairness [13, 16] so that each lock-acquiring request is eventually fulfilled. As a result of this assumption, the set of lock events generated from the lock instructions is definite.

3.2 Execution Graphs

Each program generates a set of *execution graphs*. In this section, we formally define events and execution graphs.

An *event* is a tuple of form $\langle tid, eid, typ, val, loc\rangle$ where $tid \in \mathbb{N}$ is the identifier of the thread of the event; $eid \in \mathbb{N}$ is a unique identifier for the event; $typ \in \{r, w, \mathsf{acq}, \mathsf{rel}, \mathsf{br}\}$ is the event type with r standing for read events, w for write events, acq for lock acquire events, rel for lock release events, and br for branch events; $val \in \mathsf{Val}$ is the value of the event; and $loc \in \mathsf{Loc} \cup \mathsf{Lock}$ is the memory location or lock that the event accesses.

An execution graph consists of a set of events and the relations over the events.

Definition 1 (Execution Graph). An *execution graph* $G = \langle \mathsf{Evt}, \mathsf{po}, \mathtt{rf}, \mathtt{co}, \mathtt{sync}\rangle$ with each component defined below:

- Events is a finite set of events $G.\mathsf{Evt}$. We use $G.T$ where $T \in \{\mathsf{Rd}, \mathsf{Wrt}, \mathsf{Acq}, \mathsf{Rel}, \mathsf{Br}\}$ to denote subsets of events based on their types. In addition, $G.\mathsf{Init}$ is a set of initialization writes to each memory location. $G.\mathsf{Init} \cap G.\mathsf{Evt} = \varnothing$.
- Program Order (po) is a partial order $G.\mathsf{po} \subseteq G.\mathsf{Evt} \times G.\mathsf{Evt}$. $\langle e_1, e_2\rangle \in G.\mathsf{po}$ iff $e_1.tid = e_2.tid$ and $e_1.eid < e_2.eid$.
- Reads-from Order (`rf`) is a binary relation $G.\mathtt{rf} \subseteq G.\mathsf{Wrt} \times G.\mathsf{Rd}$.

- Coherence Order (co) is a binary relation $G.\mathtt{co} \subseteq G.\mathsf{Wrt} \times G.\mathsf{Wrt}$.
- Synchronization Order (sync) is a binary relation $G.\mathtt{sync} \subseteq G.\mathsf{Rel} \times G.\mathsf{Acq}$.

In addition, the from-read order [3] (fr) is defined as $\mathtt{fr} = \mathtt{rf}^{-1}; \mathtt{co}$. We use com to denote the union of the communication orders, $\mathtt{com} = \mathtt{fr} \cup \mathtt{rf} \cup \mathtt{co}$. If an execution graph is sequentially consistent, then there is also a linear order trace among all events of the execution graph. For each event e, we use $\mathsf{LocksHeld}(e)$ to denote the set of locks that are acquired but not released at the point of e in its thread.

Furthermore, a *symbolic execution graph* $\hat{G}$ is an execution graph with an event set of which event values are either concrete values $v \in \mathsf{Val}$ or symbols $\hat{s} \in \mathsf{Sym}$. For each read event $r \in \hat{G}.\mathsf{Rd}$, if r is a read event with a concrete value, then either there exists a unique concrete write event $\langle w, r\rangle \in \hat{G}.\mathtt{rf}$ such that $w.loc = r.loc$ and $w.val = r.val$, or $r.val$ is the initial value. Note that for a read event with symbolic value, we do not require it to be mapped to a unique write event. Intuitively, symbolic execution graphs are used to accomodate the approaches of Huang et al. [8] and Huang and Huang [9], where some of the events becomes symbolic due to changes of the rf-map in the prediction result. On the other hand, later in Section 5.3, we show that there exists a concrete execution graph that a symbolic execution graph can be mapped to, given a set of conditions are satisfied.

Using definitions from [22], an execution graph G is a *plain* execution graph, if $G.\mathtt{rf} = G.\mathtt{co} = G.\mathtt{sync} = \emptyset$. An execution graph G with well-formed rf, co, and sync relations is called a *complete* execution graph.

Definition 2 (Well-formed Complete Execution Graph). A complete execution graph $G = (\mathsf{Evt}, \mathsf{po}, \mathtt{rf}, \mathtt{co}, \mathtt{sync})$ is well-formed if

- Each read event must be justified by a write event in the same execution graph. That is, for each read event $r \in G.\mathsf{Rd}$ either there is a unique write event $w \in G.\mathsf{Wrt}$ such that $\langle w, r\rangle \in G.\mathtt{rf}$, or $r.val = w_{init}.val$ where w_{init} is the initialization write of $r.loc$. For each pair $\langle w, r\rangle \in G.\mathtt{rf}$, $r.val = w.val$, and $r.loc = w.loc$.
- For each pair of distinct writes w_1 and w_2, if $w_1.loc = w_2.loc$, then either $\langle w_1, w_2\rangle \in G.\mathtt{co}$ or $\langle w_2, w_1\rangle \in G.\mathtt{co}$ but not both. In addition, $\langle w_0, w\rangle \in G.\mathtt{co}$ for each initial write w_0 and $w \in G.\mathsf{Wrt}$ such that $w_0.loc = w.loc$.
- There is a function $\mathsf{match} : G.\mathsf{Rel} \to G.\mathsf{Acq}$ such that for each $\mathsf{rel} \in G.\mathsf{Rel}$, $\mathsf{match}(\mathsf{rel}).loc = \mathsf{rel}.loc$, and $\langle \mathsf{match}(\mathsf{rel}), \mathsf{rel}\rangle \in G.\mathsf{po}$. There is a function $Open : G.\mathsf{Lock} \to G.\mathsf{Acq}$ such that $Open(l) = \mathsf{acq}$ iff $\mathsf{acq}.loc = l$ and for all $\mathsf{rel} \in G.\mathsf{Rel}$, $\mathsf{match}(\mathsf{rel}) \neq \mathsf{acq}$. If no such acquire event exists for lock l, $Open(l) = \bot$. For each $\langle \mathsf{rel}, \mathsf{acq}\rangle \in G.\mathtt{sync}$, $\mathsf{rel}.loc = \mathsf{acq}.loc$. For each lock $l \in \mathsf{Lock}$, let $\mathtt{cs} \in linear(G.\mathsf{Rel}_l)$ be a linear order among all the release events of l. Then for each $\langle \mathsf{rel}_1(l), \mathsf{rel}_2(l)\rangle \in \mathtt{cs}$, we have $\langle \mathsf{rel}_1(l), \mathsf{match}(\mathsf{rel}_2(l))\rangle \in \mathtt{sync}$.

3.3 From Programs to Execution Graphs

Given the formal definitions of programs and execution graphs, we explain how execution graphs are generated from programs.

An execution graph is generated from a program by starting with an empty execution graph G_0. Given a value map $loadVal : Load \to \mathsf{Val}$ for each load instruction of the form $r := [x]$, use the sequential operational semantics of instructions and the $loadVal()$ function to generate a chain of events for each thread by adding one event at a time. At this stage, only the program order po is added to the graph while other relations are left empty. The result of this stage is a plain execution graph. Note that we do not consider the consistency of the graph at this stage. Lastly, rf, co, and sync relations are added to the plain execution graph according to the values and

$$\hat{e} \in \mathsf{SymExpr} ::= \hat{s} \mid (\hat{e}) \mid v \mid \hat{e} + \hat{e} \mid \hat{e} - \hat{e} \mid \hat{e} * \hat{e} \qquad \hat{s} \in \mathsf{Sym},\ v \in \mathsf{Val}$$

Fig. 4. Symbolic Expressions

the well-formedness conditions, which we will define later. The result of this stage is a complete execution graph ready for the consistency check.

We next define the notion of thread state. Here we use the same definition from [22] with an additional *symbolic* register state Φ^θ and a function $\theta : G.\mathsf{Rd} \to \mathsf{Sym}$. These two new components do *not* replace any functionality of other components as described in Podkopaev et al. [22]. They are only added for additional book-keeping purposes. Later in the proof of Lemma 1 in §4.3, they are used to re-construct state transition paths of prediction results.

In addition, we assume memory fairness [13] so that each $\mathtt{lock}(l)$ instruction will eventually succeed and produce an acq event.

Definition 3 (Thread State). A thread state $st \in \mathsf{State}$ for a thread t is a tuple

$$st = \langle sprog, pc, \Phi, G, \Psi, ctrl, \theta, \Phi^\theta \rangle$$

with each component defined as follows:

- $sprog : \mathbb{N} \to \mathsf{Instr}$ is the instructions in thread t, $sprog = P(t)$.
- $pc \in \mathbb{N}$ is the program counter pointing to the next instruction
- $\Phi : \mathsf{Reg} \to \mathsf{Val}$ is map recording the value of each register
- G is the execution graph that has been constructed so far for thread t
- $\Psi : \mathsf{Reg} \to \wp(G.\mathsf{Rd})$ maps each register to a set of read events in G such that the value in the register depends on the set of reads.
- $ctrl \subseteq G.\mathsf{Rd}$ is a set of read events that has a control dependency with the current program point.
- $\theta : G.\mathsf{Rd} \to \mathsf{Sym}$ is a map recording the symbolic value of each read event. When a read event is generated from executing a load instruction, in addition to the concrete value that it receives via the $loadVal$ function, a fresh new symbol is assigned to the read event and an entry is added in θ.
- $\Phi^\theta : \mathsf{Reg} \to \mathsf{SymExpr}$ is a map recording the symbolic expression used to calculate the current (concrete) value on each register. The grammar of symbolic expressions is defined in Fig. 4. In other words, the concrete value on each register can be calculated by plugging the concrete values of the read events into its symbolic expression.

The initial state for a thread $t \in \mathsf{Tid}$ is $st_0^t = \langle sprog, 0, \lambda r.0, G_0, \lambda r.\varnothing, \varnothing, \lambda rd.null, \lambda r.0 \rangle$ where $sprog = P(t)$ and G_0 is an empty execution graph such that $G_0.\mathsf{Evt} = G_0.\mathsf{po} = G_0.\mathtt{rf} = G_0.\mathtt{co} = G_0.\mathtt{sync} = \varnothing$.

An important invariant for a state to be valid is $\forall r \in \mathsf{Reg},\ subst(\Phi^\theta(r), val^\theta(\Psi(r))) = \Phi(r)$ where the helper functions $subst : \mathsf{SymExpr} \to (\mathsf{Sym} \to \mathsf{Val}) \to \mathsf{Expr}$ replaces each symbol in an symbolic expression with a concrete value given a mapping from symbols to concrete values, and $val^\theta : \wp(G.\mathsf{Rd}) \to (\mathsf{Sym} \to \mathsf{Val})$ uses θ to provide a set of such mapping. In words, evaluating the symbolic expressions that each register is mapped to with the concrete value of each read events should have the same result as the value stored in Φ. Note that in order for this invariant to hold, a register cannot occur on both sides of an assignment.

For two thread states $st_1, st_2 \in \mathsf{State}$ of a thread t, we write $st_1 \to_t st_2$ if st_1 steps to st_2 in a single step and $st_1 \to_t^* st_2$ if st_1 steps to st_2 in zero or more steps. We provide the sequential operational semantics for our language in Appendix B.

We borrow the following definition from [22]. For an execution graph G and $t \in \mathsf{Tid}$, $G|_t$ is a thread of events such that $G|_t.\mathsf{Evt} = G.\mathsf{Evt}|_t$, $G|_t.\mathsf{po} = G.\mathsf{po} \cap (G|_t.\mathsf{Evt} \times G|_t.\mathsf{Evt})$, and $G|_t.\mathtt{rf} = G|_t.\mathtt{co} = G|_t.\mathtt{sync} = \emptyset$

Definition 4 (Plain Program Executions [22]). An execution graph G is an *execution graph of a program* P if for every $t \in \mathsf{Tid}$, there exists a state st such that $st.G = G|_t$ and $st_0^t \rightarrow_t^* st$.

We write $G \in [\![P]\!]$ for such an execution graph G and program P. We say a plain symbolic graph $\hat{G} \in [\![P]\!]$ if there exists a $st_0 \rightarrow_t^* st_n$ transition path for each thread t that produces the resulting graph. The only difference from Definition 4 is the invariant condition that each state has to maintain because $val^\theta(\Psi(r))$ may not be defined for every register in the presence of events with symbolic values. We relax the requirement and let the invariant condition only apply to registers with concrete values, i.e., $subst(\Phi^\theta(reg), val^\theta(\Psi(reg))) = \Phi(reg)$ if $\Phi(reg) \in \mathsf{Val}$. Moreover, if a write event has a concrete value, it has to be computed from concrete values as well. In other words, for each state st_i such that $st_i.sprog(st_i.pc) = [x] := e$, if w is the write event emitted after st_i and $w.val \in \mathsf{Val}$, then for each $reg \in \mathsf{Reg}$ used in e, $\Phi(reg) \in \mathsf{Val}$.

Given a plain execution graph, the rf, co, and sync relations are added to the graph according to Definition 2 to obtain a complete execution graph.

3.4 Bug Sequence

Each instance of bug is represented as a sequence of events:

$$b = e_1 \dots e_n$$

The basic well-formedness requirement for a bug sequence is for it to be sequentially consistent. We give three common examples below.

Data Races. In the context of *data race* prediction, each reported bug is in the form of a sequence $e_1 e_2$ such that $e_1.tid \neq e_2.tid \wedge e_1.loc = e_2.loc \wedge \{e_1.typ, e_2.typ\} \cap \{\mathsf{Wrt}\} \neq \emptyset$. We write $e_1 \bowtie e_2$ for such pair of events. Note that the value of e_1 or e_2 may not be the same as occurred in the input execution G_σ.

Deadlocks. In the context of *deadlock* prediction, Tunç et al. [28] provided a necessary pattern consisting a set of acquire events that imposes a cyclic resource dependency. However, note that this pattern itself violates memory consistency because a lock cannot be acquired again while it is held. The essential issue here is that the acquire events in a sound execution represent successful aquisition of the locks, whereas the acquire events in the deadlock pattern of [28] represent *requests* of the locks. Therefore, for deadlock prediction, the lock request events have to be distinguished from lock acquire events, following the approach of Kalhauge and Palsberg [10]. The rest of the definition stays the same as in [28]. A bug sequence of size k is defined as a sequence of request events $e_0, \dots, e_{k-1}$ on k distinct threads $t_0, \dots, t_{k-1}$ and k distinct locks $l_0, \dots, l_{k-1}$ such that $e_i.tid = t_i$, e_i is a request of lock l_i, where $l_i \in \mathsf{LocksHeld}(e_{(i+1)\%k})$. In addition, $\mathsf{LocksHeld}(e_i) \cap \mathsf{LocksHeld}(e_j) = \emptyset$ for $i \neq j$. Note that using this definition, the composition of a sound witness execution and the deadlock sequence is still a sound execution.

Atomicity Violations. Atomicity violation patterns can be represented as sequences of events in multiple ways. Each pattern consists of an atomic pair, i.e., a pair of events (e_1, e_2) on the same thread that is expected to be executed atomically, and a third event e_3 accessing the same memory location from another thread. Huang et al. [7] provided one pattern in their example: $r_1 w_1 w_2$ where $r_1 \in \mathsf{Rd}$, $w_1, w_2 \in \mathsf{Wrt}$, $r_1.loc = w_1.loc = w_2.loc$, and $\langle r_1, w_2 \rangle$ is an atomic pair. Cai et al. [4] provided another example: $w_2 w_1 r_1$ where $\langle w_2, r_1 \rangle$ is an atomic pair.

4 Soundness

We begin this section by asking a question.

What does it mean for a predictive analysis to be sound?

A sound predictive analysis only reports a bug if it can be exposed by a valid witness execution. As shown in Fig. 5, let P be a program with a reported bug to be fixed. G_σ is an execution graph of P captured from running P. In the rest of this paper, we assume the captured G_σ is sequentially consistent and the events of G_σ all have concrete values. A predictive analysis algorithm analyzes G_σ without inspecting P to spot the existence of any concurrent bug. Note that the predictive analysis can report bugs that are not necessarily exposed in the recorded execution G_σ, but in some other execution of the same program P. Let G_ρ be such a *witness* execution where the bug is exposed. Then a sound predictive analysis should ensure that G_ρ is indeed a valid execution of P. But what does it mean for an execution to be valid?

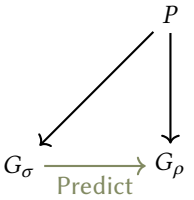


Fig. 5. Predictive Analyses

We identify two important but separate aspects in answering this question:

- **Executability** The program that generates the input execution must be able to generate the witness execution. From Definition 4, we can see that this is a *local* property. That is, an execution can be generated by a program if each of its threads can be generated by the program. In an execution graph, threads are formed by events related by the program orders po. In other words, executability means the events on each thread can be generated by the program in the order specified by po.
- **Memory Consistency** Another factor is the memory consistency model under which the execution is executed. Axiomatic memory models specify consistent executions under weak memory contexts by characterizing the ordering relations among the events, including the inter-thread communications. Therefore, this is a *global* property that concerns the orders among the events in an execution graph.

In addition, the well-formedness of the witness and its relationship with the reported bug are required to ensure that the composition of the witness with the reported bug sequence is a sound execution. As a result, we define the overall soundness of a predictive analysis as a conjunction of four major components.

Formally, a predictive analysis is a function $\mathsf{Predict} : \mathcal{G} \to \wp(\mathcal{B})$ that takes a recorded execution as an input and reports a set of bugs predicted from the execution. Soundness of such a predictive analyses is defined as the following.

$$\mathsf{Sound}_{\mathcal{M}} \triangleq \forall P \in \mathsf{Prog},\ \forall G_\sigma \in [\![P]\!] \wedge \mathsf{SC\text{-}consistent}(G_\sigma),\ \forall b \in \mathsf{Predict}(G_\sigma),$$
$$\exists G_\rho,\ G_\rho \rhd b \wedge \mathsf{wf}(G_\rho)\ \wedge\ G_\rho \in [\![P]\!]\ \wedge\ \mathcal{M}\text{-consistent}(G_\rho)$$

The above soundness definition states that in order to prove a predictive analysis is sound, one needs to show that for each bug reported, there is a *witness* execution G_ρ such that:

§4.1 $G_\rho \rhd b$	G_ρ is composable with the reported bug b	[Definition 6]
§4.2 $\mathsf{wf}(G_\rho)$	G_ρ is well-formed	[Definition 7]
§4.3 $G_\rho \in [\![P]\!]$	G_ρ is executable	[Definition 4]
§4.4 $\mathcal{M}$-consistent(G_ρ)	G_ρ is $\mathcal{M}$-consistent	[Definition 9]

The first three parts of the conjunction states properties that the witness execution must satisfy as a *plain* execution graph, whereas the last part is about the orders among the events in the witness

execution as a *complete* execution graph. In the rest of this section, we discuss each part in detail. The proofs of the propositions and lemmas can be found in the Appendix C.

4.1 Composability with bug sequences

Intuitively, witness G_ρ is the execution that must occur before the bug sequence b occurs. Hence, the composition of them should also be a valid execution. We start by formally defining the composition of an execution graph with an event sequence.

Definition 5 (Composition). Let G_ρ be a well-formed execution graph and b be a bug sequence. Then the composition

$$G = G_\rho \circ b$$

is an execution graph such that:

- $G.\mathsf{Evt} = G_\rho.\mathsf{Evt} \cup b.\mathsf{Evt}$
- $G.\mathsf{po} = G_\rho.\mathsf{po} \cup b.\mathsf{po} \cup \{\langle e_1, e_2\rangle \mid e_1 \in G_\rho.\mathsf{Evt} \wedge e_2 \in b.\mathsf{Evt} \wedge e_1.tid = e_2.tid\}$
- $G.\mathtt{co} = G_\rho.\mathtt{co} \cup b.\mathtt{co} \cup \{\langle w_1, w_2\rangle \mid w_1 \in G_\rho.\mathsf{Wrt} \wedge w_2 \in b.\mathsf{Wrt} \wedge w_1.loc = w_2.loc\}$
- $G.\mathtt{rf} = G_\rho.\mathtt{rf} \cup b.\mathtt{rf} \cup \{\langle w, r\rangle \mid r \in b.\mathsf{Rd} \wedge r \notin range(b.\mathtt{rf}) \wedge w \in G_\rho.\mathsf{Wrt} \wedge (\forall w' \in G_\rho.\mathsf{Wrt}, (w'.loc = w.loc \wedge w' \neq w) \Rightarrow \langle w', w\rangle \in G_\rho.\mathtt{co}) \wedge w.loc = r.loc \wedge w.val = r.val\}$
- $G.\mathtt{sync} = G_\rho.\mathtt{sync} \cup b.\mathtt{sync} \cup \{\langle \mathsf{rel}(l), \mathsf{acq}(l)\rangle \mid \mathsf{rel}(l) \in G_\rho.\mathsf{Rel} \wedge \mathsf{acq}(l) \in b.\mathsf{Acq}\}$

To ensure that the composition $G_\rho \circ b$ is executable, we require G_ρ to be *composable* with the bug sequence b.

Definition 6 (Composability). Let G_σ be an input execution that is sequentially consistent, G_ρ be a witness execution with an event map $\delta_\rho : G_\rho.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$, and b be a well-formed bug sequence with an event map $\delta_b : b.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$.

We say an execution graph G_ρ witnesses a bug sequence b, written $G_\rho \rhd b$, if $G_\rho.\mathsf{Evt} \cap b.\mathsf{Evt} = \emptyset$ and

- No Skipping. For each thread $t \in b.\mathsf{Thrd}$, let $e \in b|_t.\mathsf{Evt}$ be the first event occur in $b|_t$. For any $e' \in G_\sigma.\mathsf{Evt}$ such that $\langle e', \delta_b(e)\rangle \in G_\sigma.\mathsf{po}$, there is an event $e'' \in G_\rho.\mathsf{Evt}$ such that $\delta_\rho(e'') = e'$, and
- Same Control Flow. For each $r \in G_\rho.\mathsf{Rd}$ and $e \in b.\mathsf{Evt}$, if $\langle \delta_\rho(r), \delta_b(e)\rangle \in G_\sigma.\mathtt{ctrl}$, then $\delta_\rho(r) = r$.

The conditions above ensure the composition $G_\rho \circ b$ inherits executability from G_ρ.

Proposition 1. Let G_ρ be a well-formed execution graph such that $G_\rho \in [\![P]\!]$, and b be a bug sequence. If $G_\rho \rhd b$, then $(G_\rho \circ b) \in [\![P]\!]$.

On the other hand, $\mathcal{M}$-consistency (which will be defined in §4.4) is inherited by construction.

Proposition 2. Let G_ρ be a well-formed execution graph such that G_ρ is $\mathcal{M}$-consistent, and b be a sequence of events. Then $(G_\rho \circ b)$ is $\mathcal{M}$-consistent.

4.2 Well-formedness of Plain Execution

The second requirement is well-formedness requirements for a plain execution $\langle \mathsf{Evt}, \mathsf{po}\rangle$.

Definition 7 (Well-Formed Plain Execution). A plain execution $G_\rho = (\mathsf{Evt}, \mathsf{po})$ is well-formed if:

- $G_\rho.\mathsf{po}$ is a partial order over Evt that orders each pair of events $\langle e_1, e_2\rangle$ iff $e_1.tid = e_2.tid$.

- Read Feasible. For each read event included in G_ρ.Evt, either there exists a write event included in G_ρ.Evt with the same value and location, or the value of the read is the same as the initial value of the memory location.
- Lock Feasible. For each lock $l \in \mathsf{Lock}$, there is at most one open critical section protected by l. An open critical section is defined as sequence of events totally ordered by G_ρ.po where the minimal event in the sequence is an acquire event $\mathsf{acq}(l)$ for some lock l and the matching release event $\mathsf{rel}(l)$ is not included in G_ρ.Evt.

4.3 Executability

Now we turn our attention to the most important part of the soundness definition, *executability*. Given a witness execution graph G_ρ, we want to make sure that G_ρ is indeed an execution graph of the input program P, namely, $G_\rho \in [\![P]\!]$, via the semantics defined in Appendix B.

Typically, in order to show that an execution graph G_ρ is generated by a program P, one must start from the initial state (as seen in §3) and determine whether there is a reachable state containing G_ρ from the initial state via a path of transitions. However, in the setting of dynamic analysis, one cannot inspect the source code of P and hence cannot follow the semantic rules to determine whether such a state is reachable.

In the setting of dynamic analysis, what we have is the input execution G_σ, which is obtained by running the program P, i.e., $G_\sigma \in [\![P]\!]$. Therefore, for each thread t in G_σ, there exists a path $st_0 \to st_1 \to \ldots \to st_k$ with each $st_i \in \mathsf{State}$ and $st_0.sprog = P(t)$. We use $Path(G_\sigma|_t)$ to denote this path. Again, due to the nature of dynamic analyses, the states on this path are opaque. To show that $G_\rho \in [\![P]\!]$, the key is to *reuse* these states and identify a *similar* transition path that generates each thread of G_ρ from the initial state. By Definition 4, it means we have to show that for each thread t in G_ρ, there exists a state st'_m such that $st'_m.G = G_\rho|_t$ and $st'_0 \to^*_t st'_m$ with $st'_0.sprog = P(t)$. Note that st'_m is not necessarily a terminal state.

We first borrow the notion of *data-abstract equivalence* from [8] and lift it to execution graphs.

Definition 8 (Data-Abstract Equivalent Graph). A plain execution graph G is *data-abstract equivalent* to another plain execution graph G' if there is a map $\delta : G.\mathsf{Evt} \to G'.\mathsf{Evt}$ such that for each event $e \in G.\mathsf{Evt} = \langle tid, eid, typ, val, loc\rangle$, there is $\delta(e) \in G'.\mathsf{Evt} = \langle tid, eid, typ, val', loc\rangle$ for some val'. In addition, for each $\langle e_1, e_2\rangle \in G.\mathsf{po}$, $\langle \delta(e_1), \delta(e_2)\rangle \in G'.\mathsf{po}$ and vice versa.

We write $G \approx G'$ if G is data-abstract equivalent to G.

The data-abstract equivalence relation can be extended naturally to the map Ψ and the set $ctrl$ in a state. Now we use this notion to define a similarity relation over states. Let $st, st' \in \mathsf{State}$ where $st = \langle sprog, pc, \Phi, G, \Psi, ctrl, \theta, \Phi^\theta\rangle$ and $st' = \langle sprog', pc', \Phi', G', \Psi', ctrl', \theta', \Phi^{\theta'}\rangle$. We say $st' \approx st$, if $sprog = sprog'$, $pc = pc'$, $G \approx G'$, $\Psi \approx \Psi'$, $ctrl \approx ctrl'$, $\theta \approx \theta'$, and $\Phi^\theta = \Phi^{\theta'}$. For each read event $r \in G.\mathsf{Rd}$, $\theta(r) = \theta(\delta(r))$. Essentially, two states are data-abstract equivalent if they only differ by the concrete values on each read events. It's easy to see that if two states are identical, then they are data-abstract equivalent, i.e., $st = st' \Rightarrow st \approx st'$.

As illustrated in Fig. 6, for each thread t, if we can construct a similar state transition path $st'_0 \to \ldots \to st'_n$ that generates $G_\rho|_t$ from an empty state containing the same source program as $P(t)$, then by Definition 4 we can conclude that $G_\rho \in [\![P]\!]$.

We can use the following lemma to prove that an execution graph is executable.

Lemma 1. Let G_ρ, G_σ be well-formed execution graphs and $G_\sigma \in [\![P]\!]$. If for each thread $t \in G_\rho.\mathsf{Thrd}$, $t \in G_\sigma.\mathsf{Thrd}$ and there is a $\sqsubseteq$-ordered set $\{st'_0, \ldots, st'_m\}$ such that for $i \in 0 \ldots m$,

- each state st'_i satisfies the invariant $subst(\Phi^\theta(r), val^\theta(\Psi(r))) = \Phi(r)$ for $r \in \mathsf{Reg}$,

$$\begin{array}{ccccccc} st_0 & \longrightarrow & st_1 & \longrightarrow \ldots \longrightarrow & st_m & \longrightarrow \ldots \longrightarrow & st_k \\ \| & & \approx & & \approx & & \\ st'_0 & \longrightarrow & st'_1 & \longrightarrow \ldots \longrightarrow & st'_m & & \end{array}$$

Fig. 6. Similar State Transitions

- for each st'_i there exists a state $st_i \in Path(G_\sigma|_t)$ with $st_i \approx st'_i$ and $st_0 = st'_0$,
- for each st'_i if $st'_i.sprog(st'_i.pc) = \texttt{if } expr \texttt{ goto } k \texttt{ then } st_i.\Phi(expr) = st'_i.\Phi(expr)$,
- $st'_m.G = G_\rho|_t$,

then $G_\rho \in [\![P]\!]$.

Lemma 1 uses the notion of *states* to reason about the relation between the input execution and the witness execution. In addition, it requires each event in the witness graph to be concrete. The following lemma simplifies the requirements of executability, focusing the reasoning process on *events* and allowing events with symbolic values.

Lemma 2. Let $\hat{G_\rho}$ be a well-formed symbolic plain execution graph and G_σ be a concrete input execution graph such that $G_\sigma \in [\![P]\!]$. If $\hat{G_\rho}$ satisfies the following conditions:

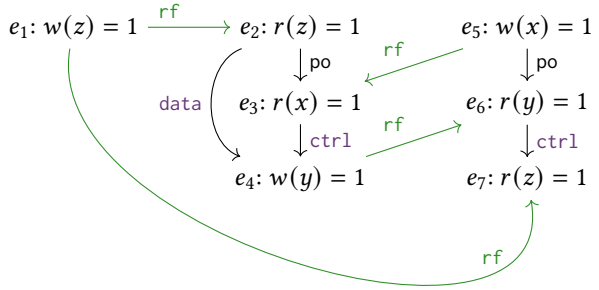
- there is a map $\delta : \hat{G_\rho}.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$ such that for each event $e \in \hat{G_\rho}.\mathsf{Evt}$, $\delta(e) \approx e$ and if $e.val \in \mathsf{Val}$ (i.e., $e.val$ is concrete), then $\delta(e) = e$.
- if $\langle e_1, e_2\rangle \in G_\sigma.\mathsf{po}$ and $e_2 = \delta(e'_2)$ for some $e'_2 \in \hat{G_\rho}.\mathsf{Evt}$, then there is an event $e'_1 \in \hat{G_\rho}.\mathsf{Evt}$ such that $e_1 = \delta(e'_1)$ and $\langle e'_1, e'_2\rangle \in \hat{G_\rho}.\mathsf{po}$,
- for each thread $t \in \hat{G_\rho}.\mathsf{Thrd}$ and each event $e \in \hat{G_\rho}|_t.\mathsf{Evt}$, if there is a read event $r \in \hat{G_\rho}|_t.\mathsf{Rd}$, such that $\langle \delta(r), \delta(e)\rangle \in G_\sigma.\texttt{ctrl}$, then $r.val \in \mathsf{Val}$ (i.e., $r.val$ is concrete),
- for each write event $w \in \hat{G_\rho}.\mathsf{Wrt}$, if $w.val \in \mathsf{Val}$ (i.e., $w.val$ is concrete), then for all $r \in \hat{G_\rho}.\mathsf{Rd}$ such that $\langle \delta(r), \delta(w)\rangle \in G_\sigma.\texttt{data}$, $r.val \in \mathsf{Val}$ (i.e., $r.val$ is concrete).

then $\hat{G_\rho} \in [\![P]\!]$.

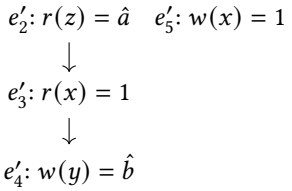
Example. We use an example to demonstrate the intuition behind Lemma 2.

Fig. 7 shows three execution graphs. In particular, G_σ in Fig. 7a is a concrete execution graph, $\hat{G_{\rho 1}}$ in Fig. 7b is a symbolic execution graph with e'_2 and e'_4 having symbolic values, and $\hat{G_{\rho 2}}$ in Fig. 7c is a symbolic execution graph with e''_7 having a symbolic value. For better readability, we annotate events in $\hat{G_{\rho 1}}$ with single primes and events in $\hat{G_{\rho 2}}$ with double primes. Assuming $G_\sigma \in [\![P]\!]$, it's easy to check that we can apply Lemma 2 to show that $\hat{G_{\rho 1}} \in [\![P]\!]$ and $\hat{G_{\rho 2}} \in [\![P]\!]$. Indeed, the map $\delta_1 = \{e'_i \mapsto e_i\}$ for $i \in 2 \ldots 4$ satisfies the requirements of Lemma 2, which shows $\hat{G_{\rho 1}}$ is executable, and the map $\delta_2 = \{e''_i \mapsto e_i\}$ for $i \in 1 \ldots 7$ satisfies the requirements of Lemma 2, which shows $\hat{G_{\rho 2}}$ is executable. Note that both $\hat{G_{\rho 1}}$ and $\hat{G_{\rho 2}}$ are *plain* symbolic execution graphs, which means they do not inherit memory orders from G_σ other than a subset of po. Intuitively, the requirements in Lemma 2 are the minimal conditions that ensure the control and data flows included in $\hat{G_{\rho 1}}$ and $\hat{G_{\rho 2}}$ are the same as in G_σ.

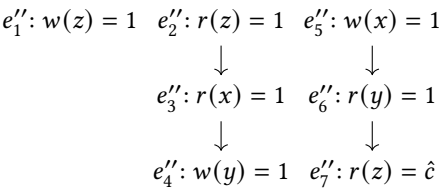
Specifically, $\hat{G_{\rho 1}}$ is a symbolic execution graph with two symbolic events. The value of e'_4 is a symbolic value $\hat{b}$, since the concrete value of e_4 is not critical to the control flow included in $\hat{G_{\rho 1}}$. On the other hand, there is a control dependency from e_3 to e_4 in G_σ. In order for a write event accessing y to occur, regardless its value, the concrete value of e_3 must be preserved. Therefore, in



(a) A Concrete Execution Graph G_σ



(b) A Symbolic Execution Graph $\hat{G_{\rho 1}}$



(c) A Symbolic Execution Graph $\hat{G_{\rho 2}}$

Fig. 7. An example of using Lemma 2 to show executability of symbolic execution graphs

$\hat{G_{\rho 1}}$, e'_3 stays as a concrete event with the same value as e_3. Note that in order to be well-formed, each read event with a concrete value must be justified by a concrete write event accessing the same location with the same value. From G_σ, we can see that e_5 justifies the value of e_3. Therefore, the corresponding event e'_5 also has the same concrete value. Although there is a data dependency from e_2 to e_4, since e'_4 now becomes a symbolic event, e'_2 does not need to preserve the concrete value of e_2. Hence, e'_2 with a symbolic value $\hat{a}$ does not impact the executability of $\hat{G_{\rho 1}}$.

On the other hand, $\hat{G_{\rho 2}}$ includes more events than $\hat{G_{\rho 1}}$, which leads to more restrictions on the values of the events. e''_7 is a symbolic event for the same reason as e'_4 being symbolic. There is a control dependency from e_6 to e_7. Hence e_6's value must be preserved to make sure the control flow is not altered. From G_σ, we can see that e_4 justifies the value of e_6. Hence, the corresponding event in $\hat{G_{\rho 2}}$, e''_4, must have the same concrete value as e_4. Now e''_2 must have the same concrete value as e_2 as well, since there is a data dependency from e_2 to e_4 and e_4's value is preserved. Again, to justify the concrete values of e''_2 and e''_3, e''_1 and e''_5 are also included as concrete events. As a result, only e''_7 is a symbolic event and all other events included in $\hat{G_{\rho 2}}$ have to preserve their original values from G_σ to ensure executability.

4.4 Memory Consistency

The second major component of soundness is memory consistency. Most existing work in predictive analysis focuses on sequentially consistent executions (except for [9], which focused on TSO and PSO models). Under sequential consistency, an execution can be treated as a linear sequence of events, i.e., a *trace*. However, under relaxed memory models, executions may not be linearizable by a global order. Instead, the validity of an execution is determined by a *memory consistency model*, which can be represented as a pair of constraints: emptiness and irreflexivity constraints over event orders [12].

To keep the presentation manageable, we consider Multicopy-Atomic (MCA) models. It remains our future work to integrate the semantics of lock operations into non-MCA models. Under an

MCA model, if a write event is visible to a thread that is not its issuing thread, then it is visible to all other threads as well. This property simplifies the axiomatic model because all cross-thread communications can be treated as global in MCA models. The rest of the section relies on MCA models that are defined in the following way.

An execution is consistent under an MCA memory model if

$$\text{irreflexive } (\text{ppo} \cup \text{com})^+$$

$$\text{irreflexive } (\text{po-loc} \cup \text{com})^+$$

where the ppo stands for *preserved program order*. Each MCA model provides its own definition of ppo. The second requirement corresponds to *SC-per-location*.

In addition, we omit treatment of Read-Modify-Write (RMW) events, and consider only fully fenced lock events. All of these could be accommodated without impacting the logic of our approach.

Under sequential consistency (SC), preserved program order is given by all po orders.

$$\text{ppo} = \text{po}$$

Under x86-TSO [20], preserved program order is given by

$$\text{ppo} = \text{po} \cap ((\text{Wrt} \times \text{Wrt}) \cup (\text{Rd} \times \text{Wrt}) \cup (\text{Rd} \times \text{Rd}))$$

Under ARMv8, preserved program order is given by the *locally-ordered-before* (lob) order from [1].

We now augment these memory models with lock operations. We first define a new relation sync order. For each lock l, there is a linear order among the critical sections protected by l. For each two ordered critical sections of the same lock CS_1 and CS_2, where $CS_1 \longrightarrow CS_2$, sync_l is a relation from the release event of CS_1 to the acquire event of CS_2. sync is the union of sync_l for all locks $l \in \text{Lock}$.

We do not allow events to move into or out of a critical section. Therefore, each lock operation also has a fence-like effect. In practice, this is typically given by the use of fence instructions inside lock implementations. We augment preserved program order with the program orders where a lock operation occurs in-between. We use L to denote the union of lock acquires and releases.

$$\text{ppo} \cup (\text{po}; [\text{L}]) \cup ([\text{L}]; \text{po})$$

The most important property that locks should provide is mutual exclusion. Given two critical sections protected by the same lock, the events in one critical section should be all finished before the events in another critical section start. In other words, there should not be any interleaving among the events from two critical sections.

We can rule out this behavior by augmenting MCA models with:

$$\text{irreflexive } (\text{ppo} \cup (\text{po}; [\text{L}]) \cup ([\text{L}]; \text{po}) \cup \text{com} \cup \text{sync})^+$$

It's easy to see that moving events into a critical section while preserving all other program orders does not introduce new behavior as it only monotonically adds more ppo order into the execution graph.

Finally, there may be open critical sections in an execution graph. By well-formedness of an execution graph, there is at most one open critical section per lock. The open critical sections should always be ordered as the last critical section in the linear order of that lock. Therefore, we add one more restriction to the memory model. For each lock l, if there is an open critical section, let $\text{acq}(l)$ be the acquire event of that open critical section. The for every release event $\text{rel}(l)$ of the same lock occur in the same execution, we have $\langle \text{rel}(l), \text{acq}(l) \rangle \in \text{sync}$.

Overall, we have the following definition for an MCA memory model augmented with lock operations.

Definition 9 ($\mathcal{M}$-consistency). Given a definition of preserved program order ppo from an memory model $\mathcal{M}$ for an MCA architecture, a complete execution graph G_ρ is $\mathcal{M}$-consistent if

- $(\mathsf{po\text{-}loc} \cup \mathsf{com})^+$ is irreflexive, and
- $(\mathsf{ppo} \cup (\mathsf{po}; [\mathsf{L}]) \cup ([\mathsf{L}]; \mathsf{po}) \cup \mathsf{com} \cup \mathsf{sync})^+$ is irreflexive, and
- for each lock $l \in \mathsf{Lock}$, if $G_\rho.Open(l) = \mathsf{acq}(l)$, then for each release event $\mathsf{rel}(l) \in G_\rho.\mathsf{Rel}$, $\langle \mathsf{rel}(l), \mathsf{acq}(l) \rangle \in G_\rho.\mathsf{sync}$.

As a sanity check, we prove a property that is a weaker variation of the DRF-SC theorem. The following proposition states that, if every pair of conflicting access is protected by some lock, which means there is no data race in the program, then the program is guaranteed to be sequentially consistent.

Proposition 3. Let P be a program. For each sequentially consistent execution graph of P, if for each conflicting memory event $e_1 \bowtie e_2$, $e_1, e_2 \in CS_l$ for some $l \in \mathsf{Lock}$, then every sound execution of P is sequentially consistent.

Given the definition of soundness, in the rest of this paper, we explain how one can prove that a given predictive analysis is sound.

5 A Recipe to Prove Soundness

The soundness definition from §4 has an existential quantifier over witness execution graphs. Hence, a proof of soundness for a given predictive analysis should provide a scheme to construct a witness execution graph for each reported concurrent bug and show that the witness satisfies each of the four requirements of soundness.

Note that the soundness definition asks for a complete execution graph where the value of each event is concrete. In some cases where the rf order of the witness execution is altered from that of the input execution, the values of some events are not computable without knowing the program source code. On the other hand, there is a subset of events whose values are critical for the execution control flow, and therefore have to be preserved. For events that do not affect the control flow, their concrete values are unimportant to the soundness of the algorithm. Therefore, we use a *symbolic* execution graph as an intermediate form, which allows the values of a subset of events to be symbolic if they do not need to be preserved before memory orders are inserted.

To prove soundness for a predictive analysis, a witness execution can be constructed in the following steps:

- §5.1 **Constructing a Symbolic Plain Execution.** Construct a symbolic plain execution graph $\hat{G}_\rho$ with an event map $\delta : \hat{G}_\rho.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$, such that $\hat{G}_\rho \rhd b$, and $\hat{G}_\rho$ is well-formed and executable.
- §5.2 **Inserting Consistent Memory Orders.** Insert rf, co, and sync memory orders so that $\hat{G}_\rho$ is $\mathcal{M}$-consistent up to concrete events and well-formed.
- §5.3 **Mapping to a Concrete Execution.** Map the symbolic execution $\hat{G}_\rho$ to a concrete execution graph G_ρ with a complete rf-map while preserving all the properties.

In the rest of this section, we discuss each step in details. The proofs of the propositions and lemmas can be found in Appendix D.

5.1 Constructing a Symbolic Plain Execution

Recall that a symbolic plain execution graph $\hat{G}_\rho$ is a tuple $(\mathsf{Evt}, \mathsf{po})$ where some of the events have symbolic values. To ensure executability, the program order po from G_σ has to be preserved in $\hat{G}_\rho$.

Therefore, the task of constructing $\hat{G_\rho}$ is essentially finding a set of events to be included in $\hat{G_\rho}$.Evt and determine the concrete values of a subset of events in the set.

The set of events Evt to be included in the witness execution is determined by dependencies and lock semantics. We identify two types of dependencies, *control* and *data* dependencies. While modern architectures define other types of dependencies as well, control and data dependencies are two most fundamental dependencies that determines the soundness of a predictive analysis in the language of this paper.

- **Control dependencies**. Control dependencies determine the control flow of the program. Formally, it is a subset of po whose domain is a set of read events. For each $\langle r, e\rangle \in \texttt{ctrl}$ in an execution, the value of r determines whether the instruction that generates e is eventually executed at some point. In each execution state, the field $ctrl$ is a set of read events that is used to compute control dependency.
- **Data dependencies**. Data dependencies determine the data flow of the program. Formally, it is a subset of po whose domain is a set of read events and whose range is a set of write events. For each $\langle r, w\rangle \in \texttt{data}$ in an execution, the value of r determines the value of w.

The two dependencies encapsulate the sequential and control properties from any memory model considered in this paper. They are sufficient to determine a set of events in the input execution that each bug sequence depends on. As hinted from the reasoning process in the example of Fig. 7, we can use the two dependencies to find two sets of events, a set of events to be mapped to the events in the witness execution graph and a set of events of which the values need to be preserved. In other words, there is a set of events $S_\sigma \subseteq G_\sigma$.Evt with a subset $C_\sigma \subseteq S_\sigma$ such that the existence of events in S_σ and the values of the events in C_σ determine the control flow of the execution that leads to the bug sequence. These events can then be used to construct a plain execution graph, i.e. the witness execution. To ensure the witness execution satisfy the first three soundness requirements, S_σ and C_σ have to satisfy the following properties.

Definition 10. Let $\langle S_\sigma, C_\sigma\rangle$ be two event sets such that $C_\sigma \subseteq S_\sigma \subseteq G_\sigma$.Evt. We say $\langle S_\sigma, C_\sigma\rangle$ *enables* a bug sequence b if

I. For each event $e \in b$, if $\langle e', e\rangle \in G_\sigma$.po, then $e' \in S_\sigma$
II. S_σ is downward-closed w.r.t. G_σ.po
III. S_σ is lock-feasible (Definition 7)
IV. For each event $e \in b$, if $\langle r, e\rangle \in G_\sigma.\texttt{ctrl}$, then $r \in C_\sigma$
V. For each $e \in S_\sigma$, if $\langle r, e\rangle \in G_\sigma.\texttt{ctrl}$, then $r \in C_\sigma$
VI. For each $e \in C_\sigma$, if $\langle r, e\rangle \in G_\sigma.\texttt{data}$, then $r \in C_\sigma$
VII. For each $r \in C_\sigma$, there exists a write $w \in C_\sigma$ such that $r.val = w.val$ and $r.loc = w.loc$
VIII. $S_\sigma \cap b = \emptyset$

In the examples in Fig. 7, if $b = e_4$, then $S_\sigma = \{e_2, e_3, e_5\}$ and $C_\sigma = \{e_3, e_5\} \subseteq S_\sigma$; if $b = e_7$, then $S_\sigma = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ and $C_\sigma = S_\sigma = \{e_1, e_2, e_3, e_4, e_5, e_6\}$.

Given such a pair of event sets $\langle S_\sigma, C_\sigma\rangle$, one can construct a plain execution graph $\hat{G_\rho} = (\text{Evt}, \text{po})$ by the following steps. Let S_ρ be an event set such that $S_\rho \approx S_\sigma$ and $C_\sigma \subseteq S_\rho$. In other words, there is a bijective event map $\delta : S_\rho.\text{Evt} \to S_\sigma.\text{Evt}$ where $\delta(e) \approx e$ for each $e \in S_\rho$ and $\delta(e) = e$ for each $e \in C_\sigma$. Then we set the following components of $\hat{G_\rho}$:

- $\hat{G_\rho}.\text{Evt} = S_\rho$.
- $\hat{G_\rho}.\text{po} = \delta^{-1}(G_\sigma.\text{po} \cap (\delta(\hat{G_\rho}.\text{Evt} \times \hat{G_\rho}.\text{Evt})))$.

In the examples in Fig. 7, the constructed graph is $\hat{G_{\rho 1}} \setminus \{e_4'\}$ if $b = e_4$; the constructed execution graph is $\hat{G_{\rho 2}} \setminus \{e_7''\}$ if $b = e_7$.

Given a pair of event set $\langle S_\sigma, C_\sigma \rangle$ that enables the bug sequence b, the constructed witness plain execution $\hat{G}_\rho$ can be shown to satisfy the first three soundness requirements.

Proposition 4. If $\langle S_\sigma, C_\sigma \rangle$ enables a bug sequence b, then $\hat{G}_\rho$ is well-formed up to concrete events and $\hat{G}_\rho \rhd b$.

Proposition 5. If $\langle S_\sigma, C_\sigma \rangle$ enables a bug sequence b, and $G_\sigma \in [\![P]\!]$, then $\hat{G}_\rho \in [\![P]\!]$.

Since the soundness of the witness execution depends on the pair $\langle S_\sigma, C_\sigma \rangle$, one essentially has to provide such a pair and show that it enables the reported bug sequence.

In practice, the precise information of control and data dependencies are rarely known to the predictive analysis. In addition, if the analysis does not record the concrete values of events in the input execution, then it'd be hard to determine which write event has the same value of a read event, as required by one of the conditions above. One way to overcome these challenges is tracing the $\mathsf{po} \cup \mathtt{rf}$ orders of the input execution G_σ and leverage the well-formed properties that they provide. The following lemma shows that any plain execution graphs of which the event set is downward-closed with respect to po and a subset of rf is read-feasible and executable.

Lemma 3. Let $\hat{G}_\rho = (\mathsf{Evt}, \mathsf{po})$ be a plain execution graph such that $\hat{G}_\rho.\mathsf{Evt}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{rf}|_C)$ where C is the set of concrete read and write events of $\hat{G}_\rho$ such that $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ for a bug sequence b, and $\delta(e) = e$ for each $e \in C$. Then $\hat{G}_\rho$ is read-feasible up to C and executable.

Lemma 3 does not guarantee lock feasibility. Lock feasibility may be ensured by tracing a partial order from G_σ, as stated in the following lemma.

Lemma 4. Let $\hat{G}_\rho = (\mathsf{Evt}, \mathsf{po})$ be a plain execution graph such that $\hat{G}_\rho.\mathsf{Evt}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \mathtt{sync})$, then $\hat{G}_\rho$ is lock-feasible.

As we will see in §6, analyses that over-approximate the control and data dependencies can apply Lemma 3 and Lemma 4 to show the well-formedness and the executability requirements.

5.2 Inserting Consistent Memory Orders

After a symbolic plain execution graph is determined, the next step is providing a memory order insertion scheme so that the complete execution graph is memory consistent. The goal in this step is to obtain an $\mathcal{M}$-consistent symbolic graph such that rf-map is defined and well-formed for all concrete read events, co is a total order among all write events to the same location, and sync is well-formed for each lock. While the insertion scheme is, in general, specific to the predictive algorithm, in some cases memory orders of G_σ can be reused since it is sequentially consistent, which automatically ensures $\mathcal{M}$-consistency.

We begin by inserting rf orders among concrete events in $\hat{G}_\rho$. For each concrete read in $\hat{G}_\rho$, by Lemma 2, we know that its value is inherited from the correspondent event in G_σ. From (VII) of Definition 10, there is a write event whose value is also preserved. Therefore, we can insert the same rf order between the preserved write and preserved reads.

Moreover, note that the only case where a cycle would potentially occur is when critical sections are reordered, due to the second requirement for $\mathcal{M}$-consistency: open critical sections should be ordered after all other critical sections of the same lock. Hence, if any of the events occur in the bug sequence is in an open critical section, all other critical sections of the same lock in $\hat{G}_\rho$ have to be ordered before the open critical section. For critical sections that originally occurred after some

events from the bug sequence, it means they would need to be reordered with the bug events in $\hat{G}_\rho$. In order to show that $\hat{G}_\rho$ is $\mathcal{M}$-consistent, one has to show that such reordering can never cause a cycle that is forbidden by $\mathcal{M}$ to occur.

On the other hand, if the set of events in $\hat{G}_\rho$ is guaranteed *not* to include such critical sections, then the memory orders from G_σ can be reused and the resulting execution graph is still sequentially consistent. The following lemma demonstrates this idea.

Lemma 5. Let G_σ be an input execution such that G_σ is sequentially consistent, equipped with a linear `trace` order. Let $\hat{G}_\rho$ be a symbolic plain execution that is well-formed, and $\hat{G}_\rho \rhd b$ where b is a reported bug. If for all acquire event $\mathsf{acq}(l) \in \hat{G}_\rho.\mathsf{Evt}$ such that $l \in \mathsf{LocksHeld}(e)$, $\langle \mathsf{acq}(l), e\rangle \in G_\sigma.\texttt{trace}$ for each event $e \in b.\mathsf{Evt}$ that is in a critical section where the acquire event of the critical section $\mathsf{acq}(l) \in \hat{G}_\rho$, then there exists a memory order insertion scheme over $\hat{G}_\rho.\mathsf{Evt}$ such that $\hat{G}_\rho$ is sequentially consistent.

As we will see in §6, predictive analyses that preserve synchronization orders [18, 19] use this lemma to insert memory orders when constructing the witness execution.

5.3 Mapping to a Concrete Execution

Finally, once the memory orders are inserted while maintaining $\mathcal{M}$-consistency, the following lemma shows that there exists a well-formed concrete graph, i.e., G_ρ, that can be obtained by concretizing the symbolic events in $\hat{G}_\rho$ and all properties from the previous steps are preserved.

Lemma 6. Let $\hat{G}_\rho$ be a symbolic execution with a well-formed `rf`-map over concrete events, a total `co` order over write events to the same location, and a well-formed `sync` over lock events. If $\hat{G}_\rho$ is $\mathcal{M}$-consistent and $\hat{G}_\rho \in [\![P]\!]$ with $e.val \in \mathsf{Val}$ for each $e \in \mathsf{preserve}(b)$, then there exists a map $\Theta : \mathsf{Sym} \to \mathsf{Val}$ such that the concrete execution $G_\rho = \Theta(\hat{G}_\rho)$ with a complete `rf`-map is $\mathcal{M}$-consistent and $G_\rho \in [\![P]\!]$.

The result of applying this lemma is a complete and concrete execution graph that satisfies the four requirements of soundness, which finishes the soundness proof.

6 Proving Race Prediction Algorithms Sound

In §2, we reviewed a set of existing predictive analyses that used various soundness definitions as their correctness criteria. Fig. 8 shows six of the nine analyses from §2 that focus on predicting data races with their race reporting criteria. Because the soundness definitions used in their papers were different, their soundness proofs are hard to compare with each other. In this section, we take a closer look at these algorithms and use our recipe on each of them to show their soundness. The point of this section is to showcase a unified and structured way of proving soundness of existing algorithms.

In the rest of the section, we assume $e_1, e_2 \in G_\sigma$ are two events from the input execution and $e_1 \bowtie e_2$. In addition, to keep the discussions concise, $\langle e_1, e_2\rangle \in G_\sigma.\texttt{trace}$.

6.1 M2

M2 [21] determines data races by computing linearizable closures. In particular, a set of events, $\mathsf{RCone}_\sigma(e_1, e_2)$, is first computed. Then a partial order is inserted among the events in this set until a linearizable state is reached. If e_1 or e_2 is included in $\mathsf{RCone}_\sigma(e_1, e_2)$, or a cycle occurs during the process of inserting the partial order, then $\langle e_1, e_2\rangle$ is not a data race. Otherwise, $\langle e_1, e_2\rangle$ is reported as a data race. Formally, $\mathsf{RCone}_\sigma(e_1, e_2)$ is defined inductively as the following:

	Analysis	Race Reporting Criterion
[21]	M2	$\mathtt{P}$ is a strict partial order over $\mathrm{RCone}_\sigma(e_1, e_2)$
[8]	RVPredict	$\exists\rho \models \Phi^\sigma_{\mathsf{mhb}} \wedge \Phi^\sigma_{\mathsf{lock}} \wedge \Phi^\sigma_{\mathsf{race}}(e_1, e_2)$
[9]	MCR-TSO	$\exists\rho \models \Phi^\sigma_{\mathsf{ppo}} \wedge \Phi^\sigma_{\mathsf{lock}} \wedge \Phi^\sigma_{\mathsf{race}}(e_1, e_2)$
[5]	HB	the first $e_1 \parallel_{\mathsf{hb}} e_2$
[18]	SHB	$e_1 \parallel_{\mathsf{shb}} e_2$
[19]	SyncP	$\{e_1, e_2\} \cap \mathrm{SPIdeal}_\sigma(e_1, e_2) = \emptyset$
§F	Enhanced-MCR-TSO	$\exists\rho \models \Phi^\sigma_{\mathsf{ppo}} \wedge \Phi^\sigma_{\mathsf{lock}} \wedge \Phi^\sigma_{\mathsf{race}}(e_1, e_2)$ after Read Elimination

Fig. 8. Race Reporting Criterion of Various Race Prediction Algorithms

- $\{\mathrm{prev}_\sigma(e_1), \mathrm{prev}_\sigma(e_2)\} \subseteq \mathrm{RCone}_\sigma(e_1, e_2)$, where $\langle \mathrm{prev}_\sigma(e), e\rangle \in G_\sigma.\mathsf{po}|_{imm}$ for all $e \in G_\sigma.\mathrm{Evt}$,
- for each event $e \in G_\sigma.\mathrm{Evt}$, if $\langle e, e'\rangle \in G_\sigma.(\mathsf{po} \cup \mathtt{rf})$ for some event $e' \in \mathrm{RCone}_\sigma(e_1, e_2)$, then $e \in \mathrm{RCone}_\sigma(e_1, e_2)$,
- for each acquire event $\mathrm{acq}(l) \in \mathrm{RCone}_\sigma(e_1, e_2)$, if $\mathrm{acq}(l).tid \neq e_1.tid$ and $\mathrm{acq}(l).tid \neq e_2.tid$, then there is a release event $\mathrm{rel}(l) \in \mathrm{RCone}_\sigma(e_1, e_2)$ such that $\mathrm{match}(\mathrm{rel}(l)) = \mathrm{acq}(l)$.

Then a closure algorithm is applied over events in $\mathrm{RCone}_\sigma(e_1, e_2)$. The algorithm inserts a strict partial order $\mathtt{P}$ based on the following closure rules.

1. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})|_{\mathrm{RCone}} \subseteq \mathtt{P}$,
2. For every $\mathrm{acq}(l) = \mathit{Open}_l(\mathrm{RCone}_\sigma(e_1, e_2))$ and every $\mathrm{rel}(l) \in \mathrm{RCone}_\sigma(e_1, e_2).\mathrm{Rel}$, $\mathrm{rel}(l) \xrightarrow{\mathtt{P}} \mathrm{acq}(l)$,
3. If $w' \xrightarrow{\mathtt{P}} r$ and $w \xrightarrow{\mathtt{rf}} r$ then $w' \xrightarrow{\mathtt{P}} w$ for each $w, w' \in \mathrm{RCone}_\sigma(e_1, e_2).\mathrm{Wrt}$ and $r \in \mathrm{RCone}_\sigma(e_1, e_2).\mathrm{Rd}$ where $w'.loc = w.loc = r.loc$
4. If $w \xrightarrow{\mathtt{P}} w'$ and $w \xrightarrow{\mathtt{rf}} r$ then $r \xrightarrow{\mathtt{P}} w'$ for each $w, w' \in \mathrm{RCone}_\sigma(e_1, e_2).\mathrm{Wrt}$ and $r \in \mathrm{RCone}_\sigma(e_1, e_2).\mathrm{Rd}$ where $w'.loc = w.loc = r.loc$
5. For $\mathrm{acq}_1(l) = \mathrm{match}(\mathrm{rel}_1(l))$, $\mathrm{acq}_2(l) = \mathrm{match}(\mathrm{rel}_2(l))$, if $\mathrm{acq}_1(l) \xrightarrow{\mathtt{P}} \mathrm{rel}_2(l)$, then $\mathrm{rel}_1(l) \xrightarrow{\mathtt{P}} \mathrm{acq}_2(l)$

Moreover, if there exists any pair of events $e \bowtie e' \in \mathrm{RCone}_\sigma(e_1, e_2)$ such that $e_i.tid \notin \{e.tid, e'.tid\}$ for a non-deterministically chosen $i \in \{1, 2\}$, $e \xrightarrow{\mathtt{trace}} e'$, and $e \parallel_{\mathtt{P}} e'$, then $\langle e, e'\rangle$ is added into $\mathtt{P}$ and the closure rules above are applied to reach a fixed point.

The soundness of M2 is stated as the following.

Theorem 1. If $\mathrm{LocksHeld}(e_1) \cap \mathrm{LocksHeld}(e_2) = \emptyset$, $\{e_1, e_2\} \cap \mathrm{RCone}_\sigma(e_1, e_2) = \emptyset$, and $\mathtt{P}$ computed as described by closure rule 1-5 is a strict partial order such that $\forall \bar{e_1}, \bar{e_2} \in \mathrm{RCone}_\sigma(e_1, e_2) \setminus G_\sigma.\mathrm{Evt}|_{e_i.tid}$, $\bar{e_1} \bowtie \bar{e_2} \Rightarrow \bar{e_1} \not\parallel \bar{e_2}$ where $i \in \{1, 2\}$, then $\langle e_1, e_2\rangle$ is a sound data race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \mathrm{RCone}_\sigma(e_1, e_2) \quad C_\sigma = G_\sigma.(\mathrm{Rd} \cup \mathrm{Wrt}) \cap S_\sigma$$

From the definition, we know that S_σ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})$. Let G_ρ be a plain execution graph such that $G_\rho.\mathrm{Evt} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathrm{Evt} \to G_\sigma.\mathrm{Evt}$ is the identity function.

We first show that $G_\rho \rhd b$ where $b = e_1 e_2$. To start with, $\{e_1, e_2\} \cap \mathrm{RCone}_\sigma(e_1, e_2) = \emptyset$ comes from the assumption. In addition, No Skipping is satisfied by the base condition of the definition, i.e., $\{\mathrm{prev}(e_1), \mathrm{prev}(e_2)\} \subseteq \mathrm{RCone}_\sigma(e_1, e_2)$. Since δ is the identity function, for each $r \in G_\sigma.\mathrm{Rd}$ such that $\langle r, e_1\rangle \in G_\sigma.\mathtt{ctrl}$ or $\langle r, e_2\rangle \in G_\sigma.\mathtt{ctrl}$, $\delta(r) = id(r) = r \in G_\rho.\mathrm{Evt}$. Hence, Same Control Flow is also satisfied.

We now show that G_ρ is well-formed and executable. Since $G_\rho.\mathrm{Evt}$ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})$, by Lemma 3, G_ρ is read feasible and executable. From that last condition of RCone's definition, we know that every critical section on thread t, where $e_1.tid \neq t \neq e_2.tid$, is closed. Hence, open critical sections can only occur on thread t_1 and t_2 where $t_1 = e_1.tid$ and $t_2 = e_2.tid$. By well-formedness of G_σ, we know that at most one open critical section of each lock can occur on each thread. Given that $\mathrm{LocksHeld}(e_1) \cap \mathrm{LocksHeld}(e_2) = \emptyset$, we can conclude that at most one open critical section is included in $\mathrm{RCone}_\sigma(e_1, e_2)$ for each lock. Thus, $\mathrm{RCone}_\sigma(e_1, e_2)$ is lock feasible.

► *Inserting Memory Orders.* Lastly, we show that there exists a memory order insertion scheme such that G_ρ is sequentially consistent. First, we set $G_\rho.(\mathsf{po} \cup \mathtt{rf}) = G_\sigma.(\mathsf{po} \cup \mathtt{rf})|_{\mathrm{RCone}}$. From 1., we know that $G_\rho.(\mathsf{po} \cup \mathtt{rf}) \subseteq \mathsf{P}$. Next, for every $\mathrm{acq}(l) = Open_l(\mathrm{RCone}_\sigma(e_1, e_2))$ and every $\mathrm{rel}(l) \in \mathrm{RCone}_\sigma(e_1, e_2)$, we set $\langle \mathrm{rel}(l), \mathrm{acq}(l)\rangle \in G_\rho.\mathtt{sync}$. We call this subset of $\mathtt{sync}$ order as $\mathtt{sync}|_{open}$. Then from 2., it's easy to see that $\mathtt{sync}|_{open} \subseteq \mathsf{P}$. In addition, the third requirement for $\mathcal{M}$-consistency in Definition 9, where $\mathcal{M}$ is sequential consistency, is satisfied. We now insert $\mathtt{co}$ orders.

For each pair of write events accessing the same location, $\langle w_1, w_2\rangle$, $\mathtt{co}$ is inserted in the following way,

- if $w_1 \xrightarrow{\mathsf{P}} w_2$, then $w_1 \xrightarrow{\mathtt{co}} w_2$; if $w_2 \xrightarrow{\mathsf{P}} w_1$, then $w_2 \xrightarrow{\mathtt{co}} w_1$
- otherwise, if $w_1.tid = e_i.tid$, then $w_1 \xrightarrow{\mathtt{co}} w_2$; if $w_2.tid = e_i.tid$, then $w_2 \xrightarrow{\mathtt{co}} w_1$

For each $\mathrm{rel}_1(l)$, $\mathrm{rel}_2(l)$ and $\mathrm{acq}_1(l)$, $\mathrm{acq}_2(l)$ such that $\mathrm{match}(\mathrm{rel}_1(l)) = \mathrm{acq}_1(l)$, and $\mathrm{match}(\mathrm{rel}_2(l)) = \mathrm{acq}_2(l)$, $\mathtt{sync}$ is inserted in the following way,

- if $\mathrm{rel}_1(l) \xrightarrow{\mathsf{P}} \mathrm{acq}_2(l)$, then $\mathrm{rel}_1(l) \xrightarrow{\mathtt{sync}} \mathrm{acq}_2(l)$; if $\mathrm{rel}_2(l) \xrightarrow{\mathsf{P}} \mathrm{acq}_1(l)$, then $\mathrm{rel}_2(l) \xrightarrow{\mathtt{sync}} \mathrm{acq}_1(l)$,
- otherwise, if $\mathrm{rel}_1(l).tid = e_i.tid$ then $\mathrm{rel}_1(l) \xrightarrow{\mathtt{sync}} \mathrm{acq}_2(l)$; if $\mathrm{rel}_2.tid = e_i.tid$, then $\mathrm{rel}_2(l) \xrightarrow{\mathtt{sync}} \mathrm{acq}_1(l)$

Note that after inserting the ordered as described above, $\mathtt{co}$ and $\mathtt{sync}$ are well-formed in G_ρ.

We now show that this insertion scheme guarantees sequential consistency. Suppose, towards contradiction, that there is a $(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})^+$ cycle in G_ρ after we finish inserting the orders. First, since $(\mathsf{po} \cup \mathtt{rf}) \subseteq \mathsf{P}$ and P is a strict partial order, one of the edges forming this cycle must be a $(\mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})$ edge from some event e_a to e_b such that $e_a.tid = e_i.tid \neq e_b.tid$ and $e_a \parallel_\mathsf{P} e_b$. In addition, there is also a P edge in this cycle from an event e_c to some event e_d such that $e_d.tid = e_i.tid \neq e_c.tid$. Since they form a cycle, we can infer that $e_d \xrightarrow{\mathsf{po}}{}^* e_a$, which also means $e_d \xrightarrow{\mathsf{P}}{}^* e_a$. In other words, the cycle is structure as

$$e_d \xrightarrow{\mathsf{P}}{}^* e_a \xrightarrow{\mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync}} e_b \xrightarrow{\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync}}{}^* e_c \xrightarrow{\mathsf{P}} e_d$$

Since $e_b.tid \neq e_i.tid \neq e_c.tid$, for any conflicting events $e \bowtie e'$, we know that $e \not\parallel_\mathsf{P} e'$. Given this and that $(\mathsf{po} \cup \mathtt{rf}) \subseteq \mathsf{P}$, the $(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})^+$ path between e_b and e_c must also be a P path (note that all communication edges relates conflicting events), i.e., $e_b \xrightarrow{\mathsf{P}}{}^* e_c$. Now we analyze each possible case.

- $e_d \xrightarrow{\mathsf{P}}{}^* e_a \xrightarrow{\mathtt{co}} e_b \xrightarrow{\mathsf{P}}{}^* e_c \xrightarrow{\mathsf{P}} e_d$. But $e_b \xrightarrow{\mathsf{P}} e_a$ implies that the $e_b \xrightarrow{\mathtt{co}} e_a$, which contradicts with $e_a \xrightarrow{\mathtt{co}} e_b$.

- $e_d \xrightarrow{\mathsf{P}}{}^* e_a \xrightarrow{\mathtt{fr}} e_b \xrightarrow{\mathsf{P}}{}^* e_c \xrightarrow{\mathsf{P}} e_d$. Then there exists a write event $w \xrightarrow{\mathtt{rf}} e_a$ and $w \xrightarrow{\mathtt{co}} e_b$. From $e_b \xrightarrow{\mathsf{P}} e_a$ and $w \xrightarrow{\mathtt{rf}} e_a$, we know $e_b \xrightarrow{\mathsf{P}} w$ because of the closure rule 3., which contradicts with $w \xrightarrow{\mathtt{co}} e_b$.
- $e_d \xrightarrow{\mathsf{P}}{}^* e_a \xrightarrow{\mathtt{sync}} e_b \xrightarrow{\mathsf{P}}{}^* e_c \xrightarrow{\mathsf{P}} e_d$. Then $e_a \in G_\rho.\mathsf{Rel}$, $e_b \in G_\rho.\mathsf{Acq}$, and there exists $\mathsf{acq}_a = \mathsf{match}(e_a)$ and $\mathsf{match}(\mathsf{rel}_b) = e_b$. By the closure rule 5., we have $\mathsf{rel}_b \xrightarrow{\mathtt{sync}} \mathsf{acq}_a$, which contradicts with $e_a \xrightarrow{\mathtt{sync}} e_b$.

Since each case gives us a contradiction, we can conclude that $(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{sync})^+$ is irreflexive. That is, G_ρ is sequentially consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

6.2 RVPredict

RVPredict [8] is an SMT-based approach to predicting data races in a program. Given an input execution G_σ with a `trace` order, the algorithm maps each event from the input execution to an integer variable that represents its order in a potential witness along with a formula generated from the input execution. A pair of conflicting events is reported as a data race if the set of constraints is satisfiable, i.e., there exists a map from the variables to integers that solves the contraints. Formally, for each event $e \in G_\sigma.\mathsf{Evt}$, $O_e \in \mathcal{O}$ is an order variable for e. A formula Φ is generated from G_σ:

$$\Phi = \Phi_{\mathsf{mhb}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}$$

where each sub-formula is defined as the following.

$$\Phi_{\mathsf{mhb}} = \bigwedge_{\langle e,e'\rangle \in G_\sigma.\mathsf{po}} O_e < O_{e'}$$

$$\Phi_{\mathsf{lock}} = \bigwedge_{\mathsf{rel}_1(l),\mathsf{rel}_2(l) \in G_\sigma.\mathsf{Rel}} \left(O_{\mathsf{rel}_1(l)} < O_{\mathsf{acq}_2(l)} \vee O_{\mathsf{rel}_2(l)} < O_{\mathsf{acq}_1(l)}\right)$$
$$\text{where } \mathsf{acq}_i(l) = \mathsf{match}(\mathsf{rel}_i(l))$$

$$\Phi_{\mathsf{race}} = (O_{e_2} - O_{e_1} = 1) \wedge \Phi^{\approx}_{\mathsf{cf}}(e_1) \wedge \Phi^{\approx}_{\mathsf{cf}}(e_2)$$

$$\Phi^{\approx}_{\mathsf{cf}}(e) = \Phi_{\mathsf{cf}}(br) \text{ where } br \in G_\sigma.\mathsf{Br} \text{ is the last branch event such that } \langle br, e\rangle \in G_\sigma.\mathsf{po}$$

$$\Phi_{\mathsf{cf}}(e) = \bigwedge_{r \in G_\sigma.\mathsf{Rd}} \Phi_{\mathsf{cf}}(r) \text{ where } \langle r, e\rangle \in G_\sigma.\mathsf{po} \text{ and } e \in G_\sigma.(\mathsf{Br} \cup \mathsf{Wrt})$$

$$\Phi_{\mathsf{cf}}(e) = \Phi_{\mathsf{cf}}(w) \wedge O_w < O_e \bigwedge_{w' \in G_\sigma.\mathsf{Wrt}} (O_{w'} < O_w \vee O_e < O_{w'})$$
$$\text{where } w.loc = e.loc = w'.loc, \langle w, e\rangle \in G_\sigma.\mathtt{rf}, w \neq w', \text{ and } e \in G_\sigma.\mathsf{Rd}$$

The soundness theorem for RVPredict is the following.

Theorem 2. If there exists a map $\rho : \mathcal{O} \to \mathsf{Int}$ such that Φ is satisfiable for (e_1, e_2), i.e., $\exists \rho \models \Phi_{\mathsf{mhb}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}(e_1, e_2)$, then $\langle e_1, e_2\rangle$ is a sound race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a subset of events defined as the following.

$$S_\sigma = \{e \in G_\sigma \mid \rho(O_e) < \rho(O_{e_1})\} \quad C_\sigma = R \cup W \text{ where}$$
$$R = \{r \in G_\sigma.\mathsf{Rd} \mid (\langle r, br\rangle \in G_\sigma.\mathsf{po} \text{ for some branch event } br \in S_\sigma.\mathsf{Brs})$$
$$\vee\ (\exists w \in C_\sigma, \langle r, w\rangle \in G_\sigma.\mathsf{po})\}$$

$$W = \{w \in G_\sigma.\mathsf{Wrt} \mid \exists r \in C_\sigma, \langle w, r\rangle \in G_\sigma.\mathtt{rf}\}$$

Since $\rho(O_{e_2}) - \rho(O_{e_1}) = 1$, we know that $S_\sigma \cap \{e_1, e_2\} = \emptyset$.

By Φ_{mhb}, we have that S_σ is po-closed.

Let S_ρ be a set of data-abstract equivalent events of S_σ where $e' \in S_\rho$ iff there is $e \in S_\sigma$ such that $e' \approx e$. We now define a bijective map $\delta : S_\rho.\mathsf{Evt} \to S_\sigma.\mathsf{Evt}$ such that $\delta(e) \approx e$. If there is a last branch event $br_i \in G_\sigma.\mathsf{Br}$ that po-ordered before e_i, we have $br_i \in S_\sigma$ for $i \in \{1, 2\}$. Then for all read event such that $\langle r, br_i\rangle \in G_\sigma.\mathsf{po}$ (note that $r \in S_\sigma$), set $\delta^{-1}(r) = r$. For each write event $w \in S_\sigma.\mathsf{Wrt}$, if for all $\langle r, w\rangle \in G_\sigma.\mathsf{po}$ (note that $r \in S_\sigma$), $r = \delta^{-1}(r)$, then set $w = \delta^{-1}(w)$. For each read event $r \in S_\sigma.\mathsf{Rd}$, if there is $\langle w, r\rangle \in G_\sigma.\mathtt{rf}$, then by $\Phi_{\mathsf{cf}}(r)$ we have $w \in S_\sigma$. If $\delta^{-1}(w) = w$, then set $\delta^{-1}(r) = r$. Otherwise, we assign a distinct symbolic value $\delta^{-1}(e).val = \hat{s}$ for read or write event $e \in S_\sigma$. Observe that the set of concrete events $C \subseteq S_\rho$ is $(\mathsf{po} \cup \mathtt{rf})$-closed by construction.

Let $\hat{G_\rho}$ be a symbolic plain execution graph such that $\hat{G_\rho}.\mathsf{Evt} = S_\rho$ and $\hat{G_\rho}.\mathsf{po} = \delta(G_\sigma.\mathsf{po}|_{S_\sigma})$.

We first show that $\hat{G_\rho} \rhd b$ where $b = e_1 e_2$. For $i \in \{1, 2\}$ and each event $e \in G_\sigma$ such that $\langle e, e_i\rangle \in G_\sigma.\mathsf{po}$, from Φ_{mhb}, we know that $\rho(O_e) < \rho(O_{e_i})$. Hence $e \in S_\sigma$ and there is an event $e' \in \hat{G_\rho}$ such that $\delta(e') = e$. Therefore, No Skipping is satisfied. In addition, if $\langle r, e_i\rangle \in G_\sigma$, then there is a branch event $\langle br_i, e_i\rangle, \langle r, br_i\rangle \in G_\sigma.\mathsf{po}$. From the definition of δ above, $\delta^{-1}(r) = r$. Hence there is $r' \in \hat{G_\rho}.\mathsf{Rd}$ such that $\delta(r') = r = r'$ and $r' \in C$. Therefore, Same Control Flow is satisfied.

We now show that $\hat{G_\rho}$ is well-formed and executable. Observe that $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ in $\hat{G_\rho}$ since $\mathtt{ctrl} \subseteq \mathsf{po}; [\mathsf{Br}]; \mathsf{po}$ and $\mathtt{data} \subseteq \mathsf{po}$. By Lemma 3, $\hat{G_\rho}$ is read feasible and executable. For each lock l, if two acquire events $\mathsf{acq}_1(l), \mathsf{acq}_2(l) \in S_\sigma$, then by Φ_{lock}, either $\mathsf{rel}_1(l) \in S_\sigma$ or $\mathsf{rel}_2 \in S_\sigma$. Since $\hat{G_\rho}.(\mathsf{Acq} \cup \mathsf{Rel}) = S_\sigma.(\mathsf{Acq} \cup \mathsf{Rel})$, we have if two acquire events $\mathsf{acq}_1(l), \mathsf{acq}_2(l) \in \hat{G_\rho}.\mathsf{Acq}$, then either $\mathsf{rel}_1(l) \in \hat{G_\rho}.\mathsf{Rel}$ or $\mathsf{rel}_2 \in \hat{G_\rho}.\mathsf{Rel}$. Therefore, $\hat{G_\rho}$ is lock feasible.

► *Inserting Memory Orders.* We now show that there is a memory order insertion scheme such that $\hat{G_\rho}$ is sequentially consistent up to C. For each read event $r \in C \subseteq \hat{G_\rho}.\mathsf{Rd}$, from the definition of δ, we know that there is a write event $w \in C$ such that $\langle \delta(w), \delta(r)\rangle \in G_\sigma.\mathtt{rf}$. Insert $\langle w, r\rangle \in \hat{G_\rho}.\mathtt{rf}$. For each write events $w, w' \in \hat{G_\rho}.\mathsf{Wrt}$, if $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$, then $\langle w, w'\rangle \in \hat{G_\rho}.\mathtt{co}$. For each lock l, if $\rho(O_{\mathsf{rel}(l)}) < \rho(O_{\mathsf{acq}(l)})$ for some release event $\mathsf{rel}(l)$ and acquire event $\mathsf{acq}(l)$, then $\langle \mathsf{rel}(l), \mathsf{acq}(l)\rangle \in \hat{G_\rho}.\mathtt{sync}$. Since ρ maps order variables to integers, which are linearly ordered, $\hat{G_\rho}.\mathtt{co}$ is a linear among writes to the same location and $\hat{G_\rho}.\mathtt{sync}$ is well-formed. We argue $\hat{G_\rho}$ is sequentially consistent after inserting the orders. First, from Φ_{lock}, we can infer that open critical sections are ordered last. For each $\langle r, w'\rangle \in \hat{G_\rho}.\mathtt{fr}$, there is a write w such that $\langle w, r\rangle \in \hat{G_\rho}.\mathtt{rf}$ and $\langle w, w'\rangle \in \hat{G_\rho}.\mathtt{co}$. By the order insertion scheme above, it means $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$. From $\Phi_{\mathsf{cf}}(r)$, we can infer that $\rho(O_{\delta(r)}) < \rho(O_{\delta(w')})$. From Φ_{mhb}, for each $\langle e, e'\rangle \in \hat{G_\rho}.\mathsf{po}$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$. From $\Phi_{\mathsf{cf}}(r)$, for each $\langle w, r\rangle \in \hat{G_\rho}.\mathtt{rf}$, we have $\rho(O_{\delta(w)}) < \rho(O_{\delta(r)})$. Therefore, for any $\langle e, e'\rangle \in \hat{G_\rho}.(\mathsf{po} \cup \mathtt{rf} \cup \mathtt{fr} \cup \mathtt{co} \cup \mathtt{sync})^+$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$. Thus, $\hat{G_\rho}$ is sequentially consistent.

► *Mapping to Concrete Execution Graph.* Lastly, by Lemma 6, there exists a concrete execution G_ρ inherits all the properties of $\hat{G_\rho}$, i.e., $G_\rho \rhd b$, and G_ρ is well-formed, executable, and sequentially consistent. □

6.3 MCR-TSO

Here we provide a formal proof for a race predictor built based on the constraint encoding from Huang and Huang [9]. While the paper of MCR [9] did not provide any example of new data races discovered under TSO, we observe that the non-SC-race example from Fig. 8 of Pavlogiannis [21] is such a data race. We provide a detailed explanation of the example in Appendix G.

Given an input execution G_σ with a `trace` order, the algorithm maps each event from the input execution to an integer variable that represents its order in a potential witness along with a formula generated from the input execution. A pair of conflicting events is reported as a data race if the set of constraints is satisfiable, i.e., there exists a map from the variables to integers that solves the constraints. Formally, for each event $e \in G_\sigma.\text{Evt}$, $O_e \in \mathcal{O}$ is an order variable for e. A formula Φ is generated from G_σ:

$$\Phi = \Phi_{\text{ppo}} \wedge \Phi_{\text{lock}} \wedge \Phi_{\text{race}}$$

where each sub-formula is defined as the following.

$$\Phi_{\text{ppo}} = \bigwedge_{\langle e,e'\rangle \in G_\sigma.\text{ppo}} O_e < O_{e'} \bigwedge_{\langle e,e'\rangle \in G_\sigma.\text{po-loc}} O_e < O_{e'} \bigwedge_{\langle e,\text{rel}(l)\rangle,\langle \text{acq}(l),e'\rangle \in G_\sigma.\text{po}} O_e < O_{\text{rel}(l)} \wedge O_{\text{acq}(l)} < O_{e'}$$

$$\Phi_{\text{lock}} = \bigwedge_{\text{rel}_1(l),\text{rel}_2(l) \in G_\sigma.\text{Rel}} \left(O_{\text{rel}_1(l)} < O_{\text{acq}_2(l)} \vee O_{\text{rel}_2(l)} < O_{\text{acq}_1(l)}\right)$$

where $\text{acq}_i(l) = \text{match}(\text{rel}_i(l))$

$$\Phi_{\text{race}} = (O_{e_2} - O_{e_1} = 1) \wedge \Phi_{\text{obs}}$$

$$\Phi_{\text{obs}} = \bigwedge_{\langle w,r\rangle \in G_\sigma.\text{rf}} (O_w < O_r) \bigwedge_{w' \in G_\sigma.\text{Wrt}} (O_{w'} < O_w \vee O_r < O_{w'})$$

where $w.loc = r.loc = w'.loc$, $w \neq w'$, and $r \in G_\sigma.\text{Rd}$

The soundness theorem is the following.

Theorem 3. If there exists a map $\rho : \mathcal{O} \to \text{Int}$ such that Φ is satisfiable for (e_1, e_2), i.e., $\exists \rho \models \Phi_{\text{ppo}} \wedge \Phi_{\text{lock}} \wedge \Phi_{\text{race}}(e_1, e_2)$, then $\langle e_1, e_2\rangle$ is a sound race.

Proof.

► *Constructing a Plain Execution Graph.* Let S_σ be a lock-feasible event set that is downward-closed w.r.t. $G_\sigma.\text{po} \cup \text{rf}$ from $b = e_1 e_2$. Let S_ρ be a set of data-abstract equivalent events of S_σ where $e' \in S_\rho$ iff there is $e \in S_\sigma$ such that $e' \approx e$. We now define a bijective map $\delta : S_\rho.\text{Evt} \to S_\sigma.\text{Evt}$ such that $\delta(e) \approx e$. Let $C_\sigma = G_\sigma.(\text{Wrt} \cup \text{Rd}) \cap S_\sigma$. If $\delta(e) \in C_\sigma$, then we set $\delta(e) = e$. Let $\hat{G_\rho}$ be a symbolic plain execution graph such that $\hat{G_\rho}.\text{Evt} = S_\rho$ and $\hat{G_\rho}.\text{po} = \delta(G_\sigma.\text{po}|_{S_\sigma})$.

By Lemma 3 and the fact that S_σ is lock-feasible, we get $\hat{G_\rho}$ is well-formed and executable. In addition, since S_σ is downward-closed from b, $\hat{G_\rho} \rhd b$.

► *Inserting Memory Orders.* We now provide a memory insertion scheme such that $\hat{G_\rho}$ is TSO-consistent. First, for each read event $r \in C_\sigma$, from the definition of δ and C_σ, we know that there exists a write event $w \in C_\sigma$ such that $\langle \delta(w), \delta(r)\rangle \in G_\sigma.\text{rf}$. Insert $\langle w, r\rangle \in \hat{G_\rho}$. From Φ_{obs}, we know that $\rho(O_w) < \rho(O_r)$. For each write events $w, w' \in \hat{G_\rho}.\text{Wrt}$, if $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$, then $\langle w, w'\rangle \in \hat{G_\rho}.\text{co}$. For each lock l, if $\rho(O_{\text{rel}(l)}) < \rho(O_{\text{acq}(l)})$ for some release event $\text{rel}(l)$ and acquire event $\text{acq}(l)$, then $\langle \text{rel}(l), \text{acq}(l)\rangle \in \hat{G_\rho}.\text{sync}$. Since ρ maps order variables to integers, which are linearly ordered, $\hat{G_\rho}.\text{co}$ is a linear among writes to the same location and $\hat{G_\rho}.\text{sync}$ is well-formed. We argue $\hat{G_\rho}$ is TSO-consistent after inserting the orders. First, from Φ_{lock}, we can infer that open critical sections are ordered last. For each $\langle r, w'\rangle \in \hat{G_\rho}.\text{fr}$, there is a write w such that $\langle w, r\rangle \in \hat{G_\rho}.\text{rf}$ and $\langle w, w'\rangle \in \hat{G_\rho}.\text{co}$. By the order insertion scheme above, it means $\rho(O_{\delta(w)}) < \rho(O_{\delta(w')})$. From Φ_{obs}, we have $\rho(O_{\delta(r)}) < \rho(O_{\delta(w')})$. From Φ_{ppo}, for each $\langle e, e'\rangle \in \hat{G_\rho}.\text{ppo}$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$, and for each $\langle e, e'\rangle \in \hat{G_\rho}.\text{po-loc}$, we have $\rho(O_{\delta(e)}) < \rho(O_{\delta(e')})$. Hence, we can infer

that $(\mathsf{ppo} \cup (\mathsf{po}; [\mathtt{L}]) \cup ([\mathtt{L}]; \mathsf{po}) \cup \mathtt{com} \cup \mathsf{sync})^+$ and $(\mathtt{po\text{-}loc} \cup \mathtt{com} \cup \mathsf{sync})^+$ are both irreflexive. That is, $\hat{G}_\rho$ is TSO-consistent.

► *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

6.4 Happens-Before (HB)

Happens-before (HB) is a partial order that is commonly used for partial-order-based dynamic race analyses. A partial-order-based race prediction algorithm predicts data races based on whether two conflicting events are ordered by a partial order built by the algorithm from the input execution. Formally, let D be a partial order built by the algorithm given an input trace σ. For an event pair $\langle e_1, e_2\rangle \in G_\sigma.\mathsf{Evt}$ such that $e_1 \bowtie e_2$, if $e_1 \parallel_D e_2$, then $\langle e_1, e_2\rangle$ is reported as a predictable race.

The happens-before (hb) order is defined as the following.

$$\mathsf{hb} = (\mathsf{po} \cup \mathsf{sync})^+$$

In addition, Happens-before only guarantees soundness of the first pair of conflicting events that is not ordered by hb. In other words, an extra assumption, i.e., every conflicting pair of events before e_1 and e_2 are ordered by hb, is added to its soundness theorem.

Theorem 4. If $\langle e_1, e_2\rangle$ is the first reported race such that $e_1 \parallel_{\mathsf{hb}} e_2$, i.e., for all events $e \xrightarrow{\mathtt{trace}}_\sigma e' \xrightarrow{\mathtt{trace}}_\sigma e_2$ such that $e \bowtie e'$, $\langle e, e'\rangle \in G_\sigma.\mathsf{hb}$, then $\langle e_1, e_2\rangle$ is a sound data race.

6.5 Schedulable Happens-before (SHB)

Schedulable Happens-before (SHB) [18] is an extension of Happens-before (HB) in which the race reported by the SHB algorithm does not have to be the first race to be sound. Like HB, it is also a partial order based algorithm that builds an shb order, which is defined as the following.

$$\mathsf{shb} = (\mathsf{po} \cup \mathsf{sync} \cup (\mathtt{rf} \setminus \{\langle e_1, e_2\rangle\}))^+$$

The soundness theorem is stated as the following.

Theorem 5. If $e_1 \parallel_{\mathsf{shb}} e_2$, then $\langle e_1, e_2\rangle$ is a sound race.

6.6 SyncP

SyncP [19] is a data race prediction algorithm that computes the event set of a potential witness and determine if $\langle e_1, e_2\rangle$ is a data race by checking whether they are included in the set. Formally, the sync-reversal-free closure of (e_1, e_2), $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$, is a set of events defined inductively by the following rules:

- $\{\mathsf{prev}_\sigma(e_1), \mathsf{prev}_\sigma(e_2)\} \subseteq \mathsf{SRFIdeal}_\sigma(e_1, e_2)$, where $\langle \mathsf{prev}_\sigma(e), e\rangle \in G_\sigma.\mathsf{po}|_{imm}$ for all $e \in G_\sigma.\mathsf{Evt}$,
- for each event $e \in G_\sigma.\mathsf{Evt}$, if $\langle e, e'\rangle \in G_\sigma.(\mathsf{po} \cup \mathtt{rf})$ for some event $e' \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$, then $e \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$,
- for any two acquire events $\mathsf{acq}_1(l)$ and $\mathsf{acq}_2(l) \in G_\sigma.\mathsf{Acq}$, if $\mathsf{acq}_1(l) \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and $\mathsf{acq}_2(l) \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and $\langle \mathsf{acq}_1(l), \mathsf{acq}_2(l)\rangle \in G_\sigma.\mathtt{trace}$, then there is a release event $\mathsf{rel}_1(l) \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ such that $\mathsf{match}(\mathsf{rel}_1(l)) = \mathsf{acq}_1(l)$.

The soundness theorem is stated as the following.

Theorem 6. If $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \emptyset$, then $\langle e_1, e_2\rangle$ is a sound race.

7 Conclusion

We proposed a new modular soundness definition for predictive analyses that takes account of the language semantics and the memory model. The new soundness definition subsumes all existing definitions and comes with a simple recipe for constructing a proof.

8 Data-Availability Statement

We do not have any data.

References

[1] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats. *ACM Transactions on Programming Languages and Systems* 43 (6 2021), 1–54. Issue 2. doi:10.1145/3458926

[2] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The semantics of power and ARM multiprocessor machine code. *Proceedings of the 4th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP'09* (2009), 13–24. doi:10.1145/1481839.1481842

[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in weak memory models (extended version). *Formal Methods in System Design* 40 (4 2012), 170–205. Issue 2. doi:10.1007/s10703-011-0135-z

[4] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and efficient concurrency bug prediction. *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* 21 (8 2021), 255–267. doi:10.1145/3468264.3468549

[5] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.* 44, 6 (jun 2009), 121–133. doi:10.1145/1543135.1542490

[6] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 179 (oct 2019), 30 pages. doi:10.1145/3360605

[7] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic Predictive Concurrency Analysis. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 1 (5 2015), 847–857. doi:10.1109/ICSE.2015.96

[8] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* 49 (6 2014), 337–348. Issue 6. doi:10.1145/2594291.2594315

[9] Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 447–461. doi:10.1145/2983990.2984025

[10] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *Proceedings of the ACM on Programming Languages* 2 (11 2018). Issue OOPSLA. doi:10.1145/3276516

[11] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 157–170. doi:10.1145/3062341.3062374

[12] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. *Proc. ACM Program. Lang.* 7, POPL, Article 19 (jan 2023), 29 pages. doi:10.1145/3571212

[13] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2020. Making Weak Memory Models Fair. *arXiv preprint arXiv:2012.01067* (12 2020). http://arxiv.org/abs/2012.01067

[14] Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 479–495.

[15] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (jul 1978), 558–565. doi:10.1145/359545.359563

[16] Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. 2023. Fair Operational Semantics. *Proc. ACM Program. Lang.* 7, PLDI, Article 139 (jun 2023), 24 pages. doi:10.1145/3591253

[17] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. 50 pages. https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf Draft available from http://www. cl. cam. ac. uk/ pes20/ppc-supplemental/test7.pdf.

[18] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proceedings of the ACM on Programming Languages* 2 (10 2018), 1–29. Issue OOPSLA. doi:10.1145/3276515

[19] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal prediction of synchronization-preserving races. *Proceedings of the ACM on Programming Languages* 5 (1 2021). Issue POPL. doi:10.1145/3434317

[20] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. doi:10.1007/978-3-642-03359-9_27

[21] Andreas Pavlogiannis. 2020. Fast, Sound, and effectively complete dynamic race prediction. *Proceedings of the ACM on Programming Languages* 4 (1 2020), 1–29. Issue POPL. doi:10.1145/3371085

[22] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proceedings of the ACM on Programming Languages* 3 (1 2019). Issue POPL. doi:10.1145/3290382

[23] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages* 2 (1 2018), 1–29. Issue POPL. doi:10.1145/3158107

[24] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, 175. doi:10.1145/1993498.1993520

[25] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. 2013. Maximal Causal Models for Sequentially Consistent Systems. In *Runtime Verification*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–150.

[26] Zheng Shi, Umang Mathur, and Andreas Pavlogiannis. 2024. Optimistic Prediction of Synchronization-Reversal Data Races. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 134, 13 pages. doi:10.1145/3597503.3639099

[27] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. *SIGPLAN Not.* 47, 1 (jan 2012), 387–400. doi:10.1145/2103621.2103702

[28] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proceedings of the ACM on Programming Languages* 7 (6 2023), 1733–1758. Issue PLDI. doi:10.1145/3591291

A Relationship with Existing Soundness Definitions

In this section, we provide the formal definitions of the existing soundness definitions reviewed in §2 and show their relationships with our soundness definition. Since all of the definitions from existing works are based on sequential traces, we lift the definitions to execution graphs by treating each well-formed trace as a well-formed execution graph equipped with a `trace` order, which is a total order among all events in the execution graph. Note that we are focusing on the validity of each execution itself here, without considering its relationship with the reported bug. Therefore, we drop the requirement of composability.

Feasible Closure. Feasible closure is defined in [8] over sequential traces. We lift the definition to sequentially consistent execution graphs and use the same composition operation from Definition 5.

Definition 11 (Feasible Closure). Given a well-formed execution graph G_σ that is sequentially consistent with a linear order $G_\sigma.\texttt{trace}$ over all events, the Feasible Closure of G_σ, $\mathsf{feasible}(G_\sigma)$, is the smallest set of well-formed execution graphs that includes G_σ and is closed under the following operations:

- Prefixes. If G_1 is an execution graph such that $G_1.\mathsf{Evt} \subseteq G_2.\mathsf{Evt}$ is closed with respect to $G_2.\texttt{trace}$ for some execution $G_2 \in \mathsf{feasible}(G_\sigma)$, then $G_1 \in \mathsf{feasible}(G_\sigma)$
- Local Determinism. Assume that $G_1 \circ e_1, G_2 \in \mathsf{feasible}(G_\sigma)$, $e_1.tid = t$, and $G_1|_t \approx G_2|_t$. Then
  - Branch. If $e_1.typ = \mathsf{br}$ and $G_1|_t \cap G_1.\mathsf{Rd} = G_2|_t \cap G_2.\mathsf{Rd}$, then $G_2 \circ e_1 \in \mathsf{feasible}(\tau)$.
  - Read. If $e_1.typ = r$ and e_2 is a read event such that $e_2 \approx e_1$, then $G_2 \circ e_2 \in \mathsf{feasible}(G_\sigma)$.
  - Write. If $e_1.typ = w$ then $G_2 \circ e_2 \in \mathsf{feasible}(G_\sigma)$ with some write event e_2 such that $e_2 \approx e_1$. If $G_1|_t \cap G_1.\mathsf{Rd} = G_2|_t \cap G_2.\mathsf{Rd}$, then $e_2.val = e_1.val$. Otherwise, $e_2.val = \hat{s}$ for some $\hat{s} \in \mathsf{Sym}$.
  - Otherwise, $G_2 \circ e_1 \in \mathsf{feasible}(G_\sigma)$.

Proposition 6. Let G_σ be a sequentially consistent input execution. Then for any $G_\rho \in \text{feasible}(G_\sigma)$, G_ρ is well-formed, executable, and sequentially consistent.

The well-formedness of G_ρ is given by the definition. The executability of G_ρ can be shown by applying Lemma 3 with $C = W \cup R$ where

$$\begin{aligned} R = \{r \in G_\sigma.\mathsf{Rd} \mid (\langle r, br\rangle \in G_\sigma.\mathsf{po} \text{ for some branch event } br \in G_\rho.\mathsf{Brs}) \\ \vee (\exists w \in C, \langle r, w\rangle \in G_\sigma.\mathsf{po})\} \\ W = \{w \in G_\sigma.\mathsf{Wrt} \mid \exists r \in C, \langle w, r\rangle \in G_\sigma.\mathtt{rf}\} \end{aligned}$$

Notice that this definition of C satisfies $((\mathtt{data} \cup \mathtt{rf})^*; \mathtt{ctrl})^+ \subseteq C$ for event $e \in G_\rho$, because $\mathtt{ctrl} \subseteq \mathsf{po}; [\mathsf{Br}]; \mathsf{po}$ and $\mathtt{data} \subseteq [\mathsf{Rd}]; \mathsf{po}; [\mathsf{Wrt}]$. By Proposition 2, the the composition operation ($\circ$) preserves memory consistency. Since G_σ is sequentially consistent, we can inductively show that G_ρ is also sequentially consistent.

Relaxed CR. We use the definition from [18] for Relaxed CR, again lifted to execution graphs.

Definition 12 (Relaxed CR). Given a well-formed execution graph G_σ that is sequentially consistent, a well-formed execution G_ρ is a Relaxed CR of G_σ, if:

- G_ρ is sequentially consistent
- for each event $e \in G_\rho.\mathsf{Evt}$, if $\langle e', e\rangle \in G_\sigma.\mathsf{po}$, then $e' \in G_\rho.\mathsf{Evt}$ and $G_\rho.\mathsf{po} \subseteq G_\sigma.\mathsf{po}$
- for each read event $r \in G_\rho.\mathsf{Evt}$, if r is not the last event in its thread, and $\langle w, r\rangle \in G_\sigma.\mathtt{rf}$, then $w \in G_\rho.\mathsf{Evt}$ and $\langle w, r\rangle \in G_\rho.\mathtt{rf}$ for each pair of such read and write events.

Proposition 7. Let G_σ be a sequentially consistent input execution. Then for any Relaxed CR G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

The well-formedness of G_ρ is given by the definition. The executability of G_ρ can be shown by applying Lemma 3 with $C = W \cup R$ where

$$\begin{aligned} R = \{r \in G_\sigma.\mathsf{Rd} \mid \langle r, e\rangle \in G_\rho.\mathsf{po} \text{ for some non-branch event } e \in G_\rho.\mathsf{Evt}\ \} \\ W = G_\sigma.\mathsf{Wrt} \end{aligned}$$

Notice that this definition of C satisfies $C_\sigma \subseteq C$ for event $e \in G_\rho$, since all of the rules (IV, V, VI in Definition 10) that add a read event r into C_σ require some non-branch event e' to be po-ordered after r in G_ρ. Lastly, G_ρ is sequentially consistent by definition.

It's easy to see that the concrete set C of Feasible Closure is a subset of the concrete set C of Relaxed CR, while both require the execution to be well-formed and sequentially consistent. Hence, any Relaxed CR is also in the Feasible Closure of the input execution.

Correct Reordering. Correct Reordering is defined as the following.

Definition 13 (Correct Reordering). Given a well-formed execution G_σ that is sequentially consistent, a well-formed execution G_ρ is a Correct Reordering of G_σ if:

- G_ρ is sequentially consistent,
- for each event $e \in G_\rho.\mathsf{Evt}$, if $\langle e', e\rangle \in G_\sigma.(\mathsf{po} \cup \mathtt{rf})$, then $e' \in G_\rho.\mathsf{Evt}$,
- $G_\rho.\mathtt{rf} = G_\sigma.\mathtt{rf} \cap (G_\rho.\mathsf{Wrt} \times G_\rho.\mathsf{Rd})$,
- $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po} \cap (G_\rho.\mathsf{Evt} \times G_\rho.\mathsf{Evt})$.

Proposition 8. Let G_σ be a sequentially consistent input execution. Then for any Correct Reordering G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

The well-formedness of G_ρ is given by the definition. The executability of G_ρ can be shown by applying Lemma 3 with $C = G_\sigma.\mathsf{Wrt} \cup G_\sigma.\mathsf{Rd}$. It is obvious that this definition of C satisfies $C_\sigma \subseteq C$ for each event $e \in G_\rho$. Finally, G_ρ is sequentially consistent by definition.

Sync-Preserving CR. The formal definition of Sync-Preserving CR is given by

Definition 14. Given a well-formed execution G_σ that is sequentially consistent, a well-formed execution G_ρ is a Sync-Preserving CR of G_σ if:

- G_ρ is a Correct Reordering of G_σ
- $G_\rho.\mathtt{sync} \subseteq G_\sigma.\mathtt{sync}$

By definition, any Sync-Preserving CR is a Correct Reordering of the input execution, hence the following proposition.

Proposition 9. Let G_σ be a sequentially consistent input execution. Then for any Sync-Preserving CR G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

Interestingly, if we want to show a Sync-Preserving CR is executable without considering the obvious relation with Correct Reordering, the concrete set C for Sync-Preserving CR is the same as Correct Reordering. This is because Sync-Preserving CR only differ from Correct Reordering in the memory orders among the events, whereas executability is *not* affected by memory orders other than po.

Optimistic CR. The formal definition of Optimistic CR [26] is given by

Definition 15. Given a well-formed execution G_σ that is sequentially consistent, a well-formed execution G_ρ is a Optimistic CR of G_σ if:

- G_ρ is a correct reordering of G_σ
- $G_\rho.(\mathtt{co}^?;\mathtt{rf}) = G_\sigma.(\mathtt{co}^?;\mathtt{rf}) \cap (G_\rho.\mathsf{Wrt} \times G_\rho.\mathsf{Rd})$
- $G_\rho.\mathtt{co} = G_\sigma.\mathtt{co} \cap (G_\rho.\mathsf{Wrt} \times G_\rho.\mathsf{Wrt})$
- $G_\rho.\mathtt{fr} = G_\sigma.\mathtt{fr} \cap (G_\rho.\mathsf{Rd} \times G_\rho.\mathsf{Wrt})$
- for each $\mathsf{acq}_1(l), \mathsf{acq}_2(l) \in G_\rho.\mathsf{Acq}$, if $\mathsf{match}(\mathsf{acq}_1), \mathsf{match}(\mathsf{acq}_1) \in G_\rho.\mathsf{Evt}$, then $\langle \mathsf{match}(\mathsf{acq}_1), \mathsf{acq}_2\rangle \in G_\rho.\mathtt{sync}$ iff $\langle \mathsf{match}(\mathsf{acq}_1), \mathsf{acq}_2\rangle \in G_\sigma.\mathtt{sync}$

By definition, any Optimistic CR is a Correct Reordering of the input execution, hence the following proposition.

Proposition 10. Let G_σ be a sequentially consistent input execution. Then for any Optimistic CR G_ρ of G_σ, G_ρ is well-formed, executable, and sequentially consistent.

An important difference between Optimistic CR and Sync-preserving CR lies at the sync order. Sync-preserving CR preserves all sync orders that are included in the witness execution, whereas Optimistic CR only preserves the sync orders among critical sections that are completely included in the witness execution. On the other hand, the Optimistic Lock-closure [26], which is used to compute the set of events in the witness execution for OSR [26], may include more events than the Sync-Preserving Closure [19]. As a result, the race coverages of Mathur et al. [19] and Shi et al. [26] are not comparable.

B Operational Semantics

In this section we provide the operational semantics for our language. The semantics are intended to be straightforward and unsurprising.

Recall that a thread state $st \in \text{State}$ for a thread t is a tuple of the form

$$st = \langle sprog, pc, \Phi, G, \Psi, ctrl, \theta, \Phi^\theta \rangle$$

Helper Functions. We start by defining the helper functions.

Given a map $\theta : G.\text{Rd} \to \text{Sym}$, we define the function $val^\theta : \wp(G.\text{Rd}) \to (\text{Sym} \to \text{Val})$ as the following.

$$val^\theta[] \triangleq \lambda s.0$$

$$val^\theta(r :: tl) \triangleq \lambda s.\ \texttt{if}\ s = \theta(r)\ \texttt{then}\ r.val\ \texttt{else}\ val^\theta(tl)(s)$$

The substitution function $subst : \text{SymExpr} \to (\text{Sym} \to \text{Val}) \to \text{Expr}$ replaces symbols with concrete values in an expression.

In addition, the function $mkExpr : \text{Expr} \to (\text{Reg} \to \text{SymExpr}) \to \text{SymExpr}$ converts a concrete expression to a symbolic expression using a given map $\Phi^\theta : \text{Reg} \to \text{SymExpr}$.

The function $newSym : \wp(\text{Sym}) \to \text{Sym}$ returns a fresh new symbol that does not occur in the given set of existing symbols.

Finally, we use a the same function add_G defined from [22] but without read-modify-write operations and address dependencies (we assume all memory locations are fixed).

Semantic Rules. Recall that we are given a map $loadVal : Load \to \text{Val}$ for each load instruction of the form $r := [x]$ with $r \in \text{Reg}$ and $x \in \text{Loc}$. In this section, we use the helper functions defined previously and the $loadVal$ map to define the semantic rules for our language.

We use the notation $st_1 \xrightarrow{instr}_t st_2$ to represent a state transition in thread t from st_1 to st_2 where $st_1.sprog(st_1.pc) = instr \in \text{Instr}$ and $st_1.sprog = st_2.sprog$.

$$\frac{\begin{array}{c} st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi[r \mapsto st_1.\Phi(e)] \\ st_2.G = st_1.G \\ st_2.\Psi = st_1.\Psi[r \mapsto st_1.\Psi(e)] \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta[r \mapsto mkExpr(e, st_1.\Phi^\theta)] \end{array}}{st_1 \xrightarrow{r:=e}_t st_2} \text{(Assignment)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\text{Evt}|, \text{Wrt}, st_1.\Phi(e), x \rangle \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \Psi(e), st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{[x]:=e}_t st_2} \text{(Store)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\text{Evt}|, \text{Rd}, loadVal(i), x \rangle \\ \hat{s} = newSym(codomain(st_1.\theta)) \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi[r \mapsto loadVal(i)] \\ st_2.G = add(st_1.G, evt, \emptyset, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi[r \mapsto \{evt\}] \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta[evt \mapsto \hat{s}] \\ st_2.\Phi^\theta = st_1.\Phi^\theta[r \mapsto \hat{s}] \end{array}}{st_1 \xrightarrow{r:=[x]}_t st_2} \text{(Load)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\text{Evt}|, \text{Br}, -, - \rangle \\ st_1.\Phi(e) = 0 \Rightarrow st_2.pc = st_1.pc + 1 \\ st_1.\Phi(e) \neq 0 \Rightarrow st_2.pc = n \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \emptyset, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \cup st_1.\Psi(e) \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{\texttt{if}\ e\ \texttt{goto}\ n}_t st_2} \text{(If-Goto)}$$

$$\frac{\begin{array}{c} evt = \langle t, |st_1.G.\text{Evt}|, \text{Acq}, -, l\rangle \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \emptyset, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{\texttt{lock}(l)}_t st_2} \text{(Lock)} \qquad \frac{\begin{array}{c} evt = \langle t, |st_1.G.\text{Evt}|, \text{Rel}, -, l\rangle \\ st_2.pc = st_1.pc + 1 \\ st_2.\Phi = st_1.\Phi \\ st_2.G = add(st_1.G, evt, \emptyset, st_1.ctrl) \\ st_2.\Psi = st_1.\Psi \\ st_2.ctrl = st_1.ctrl \\ st_2.\theta = st_1.\theta \\ st_2.\Phi^\theta = st_1.\Phi^\theta \end{array}}{st_1 \xrightarrow{\texttt{unlock}(l)}_t st_2} \text{(Unlock)}$$

C Proofs for Soundness Properties in §4

In this section, we present the proofs of §4.

Proposition 1. Let G_ρ be a well-formed execution graph such that $G_\rho \in [\![P]\!]$, and b be a bug sequence. If $G_\rho \rhd b$, then $(G_\rho \circ b) \in [\![P]\!]$.

Proof. Note that $(G_\rho \circ b)$ is a symbolic execution graph since all events in b have symbolic values. From $G_\rho \in [\![P]\!]$, we know that for each thread $t \in G_\rho.\text{Thrd}$, there is a state transition path $st'_0 \to \cdots \to st'_m$ generating $G_\rho|_t$ where $st_0 = st'_0$ and $st_i \approx st'_i$ for $st_i \in Path(G_\sigma|_t)$. Let e_1 be the first event and e_n be the last event in $b|_t$. Then there is a set $\{\delta_b(e_1), \ldots, \delta_b(e_n)\} \subseteq G_\sigma|t.\text{Evt}$. Let $st_j \to \cdots \to st_n$ be the subsequence of transition path that generates $\{\delta_b(e_1), \ldots, \delta_b(e_n)\}$. We construct a set of states $\{st'_j, \ldots, st'_n\}$ as follows. For each $i \in j \ldots n$, we first set $st'_i.sprog = st_i.sprog$, $st'_i.pc = st_i.pc$, and $st'_i.\Phi^\theta = st_i.\Phi^\theta$. For each read event $r' \in b|_t.\text{Rd}$, there exists a read event $r = \delta_b(r')$ in $G_\sigma|_t.\text{Rd}$ such that $r' \approx r$. Set $st'_i.\theta$ be a symbol map such that $st'_i.\theta(r') = st_i.\theta(r)$ for each read event r'. Then $st'_i.\Psi$ and $st'_i.ctrl$ are derived from $st_i.\Psi$ and $st_i.ctrl$ respectively by replacing each r with r'. Finally, $st'_i.G$ is derived from $st_i.G$ by replacing every event e with e', where $e = \delta_b(e')$. By this construction, since b is sequentially consistent, the set $\{st'_j, \ldots, st'_n\}$ is *sqsubseteq*-ordered. In addition, from the No Skipping condition, we know that there is $e_q \in G_\rho$ such that $\langle \delta_\rho(e_q), \delta_b(e_1)\rangle \in G_\sigma.\text{po}$. Hence, the set $\{st'_0, \ldots, st'_m, st'_j, \ldots, st'_n\}$ is *sqsubseteq*-ordered as well. For state st'_{m-1}, if $st'_{m-1}.sprog(st'_{m-1}.pc) = \texttt{if } expr \texttt{ goto } n$, for each $r \in st_{m-1}.\Psi(expr)$, we have $\langle \delta_\rho(r), \delta_b(e_k)\rangle \in G_\sigma.\texttt{ctrl}$ for all $k \in 1 \ldots n$. From the Same Control Flow condition, we know that $\delta_\rho(r) = r$. Hence, $st_{m-1}.\Phi(expr) = st_{m-1}.\Phi(expr)$. From the assumption that $G_\rho \in [\![P]\!]$ and that there is no concrete event in b, we can infer that $st'_0 \to \cdots \to st'_m \to st'_j \to \cdots \to st'_n$ is a valid transition path generating $(G_\rho \circ b)$. Thus, $(G_\rho \circ b) \in [\![P]\!]$. □

Proposition 2. Let G_ρ be a well-formed execution graph such that G_ρ is $\mathcal{M}$-consistent, and b be a sequence of events. Then $(G_\rho \circ b)$ is $\mathcal{M}$-consistent.

Proof. Towards contradiction, suppose there is a $\texttt{ppo} \cup \texttt{com} \cup (\texttt{po}^?; [L]; \texttt{po}^?) \cup \texttt{sync}$ cycle in $(G_\rho \circ b)$. Since G_ρ is $\mathcal{M}$-consistent and b is a sequentially consistent, there must be an ordering edge from an event of b to an event of G_ρ. Since $(G_\rho \circ b).\text{po}$ orders all events of G_ρ before events of b on the same threads, this back edge is not a po edge. By the definition of composition, we can further rule out sync, co, and rf edge for similar reasons. If it is an fr edge, then it means there is a write event $w \in b.\text{Wrt}$ such that $\langle w, r\rangle \in b.\texttt{rf}$ for some read event $r \in b.\text{Rd}$ and $\langle w, w'\rangle \in (G_\rho \circ b).\texttt{co}$. However, it contradicts with the definition of $(G_\rho \circ b).\texttt{co}$. Thus, $(G_\rho \circ b)$ is $\mathcal{M}$-consistent. □

Lemma 1 is proved using two helper lemmas, Lemma 7 and Lemma 8. We begin by providing their proofs.

Lemma 7. Let $st_1, st_2, st_1', st_2' \in \text{State}$ be four valid states and $st_1.sprog(st_1.pc) \neq \texttt{if } e \texttt{ goto } n$. If $st_1 \approx st_1'$, $st_2 \approx st_2'$ via the same map δ and $st_1 \longrightarrow st_2$, then $st_1' \longrightarrow st_2'$.

Proof. Let $st_1 = \langle sprog_1, pc_1, \Phi_1, G_1, \Psi_1, ctrl_1, \theta_1, \Phi_1^\theta \rangle$, $st_2 = \langle sprog_2, pc_2, \Phi_2, G_2, \Psi_2, ctrl_2, \theta_2, \Phi_2^\theta \rangle$, $st_1' = \langle sprog_1', pc_1', \Phi_1', G_1', \Psi_1', ctrl_1', \theta_1', \Phi_1^{\theta\prime} \rangle$, and $st_2' = \langle sprog_2', pc_2', \Phi_2', G_2', \Psi_2', ctrl_2', \theta_2', \Phi_2^{\theta\prime} \rangle$.

Recall that each valid state satisfies the invariant $\forall r \in \text{Reg},\ subst(\Phi^\theta(r), val^\theta(\Psi(r))) = \Phi(r)$.

It's obvious that $sprog_1 = sprog_2 = sprog_1' = sprog_2'$.

Due to the data-abstract equivalence relations and the assumption that $st_1.sprog(st_1.pc) \neq \texttt{if } e \texttt{ goto } n$, we have $pc_1 = pc_1'$ and $pc_2 = pc_2'$. Because $sprog_1(pc_1) \neq \texttt{if } e \texttt{ goto } n$, we have $pc_2 = pc_1 + 1$. Hence, $pc_2' = pc_1' + 1$.

For the execution graph, we have $G_2 = add(G_1, e)$ for some event e. Therefore, for each $e' \in G_2$ that $e' \neq e$, we know that $e' \in G_1$, which implies that $\delta(e') \in G_1$. For e, we have $e \notin G_1$ which implies that $\delta(e) \notin G_1'$. In addition, because $G_2 \approx G_2'$, we know that $\delta(e) \in G_2'$. Therefore, we can infer that $G_2' = add(G_1', \delta(e))$.

For the $ctrl$ set, because $sprog(pc_1) \neq \texttt{if } e \texttt{ goto } n$, we have $ctrl_1 = ctrl_2$. Because $st_2 \approx st_2'$ for each read event $e \in ctrl_2$, we have $\delta(e) \in ctrl_2'$. Similarly, for each read event $e \in ctrl_1'$, we have $\delta^{-1}(e) \in ctrl_1$. Hence, for each read event $e \in ctrl_1'$, $\delta(\delta^{-1}(e)) = e \in ctrl_2'$. Similarly, for each read event $e \in ctrl_2'$, $\delta(\delta^{-1}(e)) = e \in ctrl_1'$. Therefore, $ctrl_1' = ctrl_2'$.

Let $i = sprog_1(pc_1)$. We do a case analysis on i to establish the relations on Φ, Φ^θ and Ψ.

- $i = \texttt{lock}(l)$ or $i = \texttt{unlock}(l)$. Then we have $\Phi_2 = \Phi_1$ and $\Psi_1 = \Psi_2$. It's obvious that $\Phi_1^\theta = \Phi_1^{\theta\prime} = \Phi_2^\theta = \Phi_2^{\theta\prime}$. Because the data-abstract equivalence is established via the same function δ, for each register $r \in \text{Reg}$, we have $\Phi_2'(r) = subst(\Phi_1^{\theta\prime}(r), val^\theta(\Psi_1'(r))) = \Phi_1'(r)$. Hence, $\Phi_1' = \Phi_2'$. Given that $\Psi_1 = \Psi_2$, for each register $r \in \text{Reg}$, $\Psi_1(r) = \Psi_2(r)$. Lifting the function δ, we have $\delta(\Psi_1(r)) = \delta(\Psi_2(r))$ for each $r \in \text{Reg}$. Therefore, $\Psi_1' = \Psi_2'$.
- $i = r \texttt{ := } e$. Then we have $\Phi_2 = \Phi_1[r \mapsto \Phi_1(e)]$ and $\Phi_2^\theta = \Phi_1^\theta[r \mapsto \hat{e}]$ where $\hat{e} = mkExpr(e, \Phi_1^\theta)$. In addition, we have $\Psi_2 = \Psi_1[r \mapsto \Psi_1(e)]$. From $st_2 \approx st_2'$, we have $\Phi_2^\theta = \Phi_2^{\theta\prime}$. From $st_1 \approx st_1'$, we have $\Phi_1^\theta = \Phi_1^{\theta\prime}$. Since for each read event $r \in G_1.\text{Rd}$, $\theta_1(r) = \theta_1'(\delta(r))$, we have $\Phi_2^{\theta\prime} = \Phi_1^{\theta\prime}[r \mapsto \hat{e}]$. For each $r' \neq r$, it's obvious that $\Phi_2'(r') = \Phi_1'(r')$. We know that r does not occur on the right-hand-side of the assignment, therefore, $\Psi_1'(e) = \Psi_2'(e)$ and hence $\Phi_2'(r) = subst(\Phi_2^{\theta\prime}(r), val^\theta(\Psi_1'(e))) = subst(\Phi_1^{\theta\prime}[r \mapsto \hat{e}](r), val^\theta(\Psi_1'(e))) = subst(\hat{e}, val^\theta(\Psi_1'(e))) = \Phi_1'(e)$. Therefore, $\Phi_2' = \Phi_1'[r \mapsto \Phi_1'(e)]$. Given that $\Psi_2 = \Psi_1[r \mapsto \Psi_1(e)]$, for registers $r' \neq r$, $\Psi_2'(r') = \delta(\Psi_2(r')) = \delta(\Psi_1(r')) = \Psi_1'(r')$. For the register r, we have $\Psi_2(r) = \Psi_1(e)$. Therefore, $\Psi_2'(r) = \delta(\Psi_2(r)) = \delta(\Psi_1(e)) = \Psi_1'(e)$. Thus, $\Psi_2' = \Psi_1'[r \mapsto \Psi_1'(e)]$.
- $i = r \texttt{ := } [x]$. Then we have $\Phi_2 = \Phi_1[r \mapsto v]$ for some concrete value v and $\Phi_2^\theta = \Phi_1^\theta[r \mapsto \hat{v}]$ for some symbolic value $\hat{v}$. From $st_2 \approx st_2'$ and $st_1 \approx st_1'$, we have $\Phi_2^{\theta\prime} = \Phi_1^{\theta\prime}[r \mapsto \hat{v}]$. Now we want to show that there exists a concrete value v' such that $\Phi_2' = \Phi_1'[r \mapsto v']$. Note that at this stage of generating execution graph, we do not restrict the value of each event to be consistent with the memory model or well-formed. Therefore, any value from the domain Val is eligible to be given to a read event. Since the domain Val is not empty, we can be sure that there exists a value $v' \in \text{Val}$ such that $\delta(e).val = v'$, where e is the read event generated from st_1 to st_2 with $G_2 = add(G_1, e)$. Because we have already established that $\Phi_2^{\theta\prime} = \Phi_1^{\theta\prime}[r \mapsto \hat{v}]$ and $\theta_2 \approx \theta_2'$, we have $\Phi_2' = \Phi_1'[r \mapsto v']$ for some $v' \in \text{Val}$. From the data-abstract equivalence relation among states, we can easily see that $\Psi_2' = \Psi_1'[r \mapsto \{\delta(e)\}]$ via a similar set of reasonings for other cases.
- $i = [x] \texttt{ := } e$. Then we have $\Phi_1 = \Phi_2$, $\Phi_1^\theta = \Phi_2^\theta$ and $\Psi_1 = \Psi_2$. We immediately have $\Phi_1^\theta = \Phi_1^{\theta\prime} = \Phi_2^\theta = \Phi_2^{\theta\prime}$. In addition, from the data-abstract equivalence relations, we get that $\Psi_2' = \Psi_1'$. Because

the data-abstract equivalence relations are established via the same function δ, for each register $r \in \text{Reg}$, $val^{\theta}(\Psi_2'(r)) = val^{\theta}(\Psi_1'(r))$. Therefore, $\Phi_2'(r) = subst(\Phi_2^{\theta\prime}(r), val^{\theta}(\Psi_1'(r))) = \Phi_1'(r)$.

With the relations on each of the components of st_1' and st_2' established, we can conclude that $st_1' \rightarrow st_2'$. □

As we have seen in the proof, we are able to establish the state transition relation with only the data-abstract equivalence relations among states when there is no if-goto instruction because the control flow of the program does not depend on the concrete values in those cases. On the other hand, there is one extra condition needed on states with if-goto instructions because this is where the control flow becomes dependent on the values. We have the following lemma.

Lemma 8. Let $st_1, st_2, st_1', st_2' \in \text{State}$ be four valid states and $st_1.sprog(st_1.pc) = \texttt{if } e \texttt{ goto } n$. If $st_1 \approx st_1'$, $st_2 \approx st_2'$ via the same bijiective function δ, $st_1 \rightarrow st_2$ and $\Phi_1(e) = \Phi_1'(e)$, then $st_1' \rightarrow st_2'$.

Proof. If $\Phi_1(e) = \Phi_1'(e) = 0$, then $pc_2 = pc_2' = pc_1 + 1 = pc_1' + 1$. If $\Phi_1(e) = \Phi_1'(e) \neq 0$, then $pc_2 = pc_2' = n$. Therefore, we can establish that $\Phi_1' = 0 \Rightarrow pc_2' = pc_1' + 1$ and $\Phi_1' \neq 0 \Rightarrow pc_2' = n$. We can establish the relations of other components of st_1' and st_2' using a similar proof as the one for Lemma 7. Thus, $st_1' \rightarrow st_2'$. □

We are now ready to proof Lemma 1.

Lemma 1. Let G_ρ, G_σ be well-formed execution graphs and $G_\sigma \in [\![P]\!]$. If for each thread $t \in G_\rho.\text{Thrd}$, $t \in G_\sigma.\text{Thrd}$ and there is a $\sqsubseteq$-ordered set $\{st_0', \ldots, st_m'\}$ such that for $i \in 0 \ldots m$,

- each state st_i' satisfies the invariant $subst(\Phi^{\theta}(r), val^{\theta}(\Psi(r))) = \Phi(r)$ for $r \in \text{Reg}$,
- for each st_i' there exists a state $st_i \in Path(G_\sigma|_t)$ with $st_i \approx st_i'$ and $st_0 = st_0'$,
- for each st_i' if $st_i'.sprog(st_i'.pc) = \texttt{if } expr \texttt{ goto } k$ then $st_i.\Phi(expr) = st_i'.\Phi(expr)$,
- $st_m'.G = G_\rho|_t$,

then $G_\rho \in [\![P]\!]$.

Proof. Induction on the size of the set $\{st_0', \ldots, st_m'\}$. For the base case, since $st_0' = st_0$ is the initial state, it's obvious that the lemma holds. For the inductive step, assume that the lemma holds for $\{st_0', \ldots, st_i'\}$. That is, given such set of states, we have $st_0' \rightarrow_t^* st_i'$. We want to prove the same property holds for st_{i+1}'. There are two cases to analyze. If $st_i'.sprog(st_i'.pc) = \texttt{if } e \texttt{ goto } n$, then we can apply Lemma 8. Otherwise, we can apply Lemma 7. By the transitivity of state transition relation, we have $st_0' \rightarrow_t^* st_{i+1}'$. Thus, we can conclude that $st_0' \rightarrow_t^* st_m'$. Hence, $G_\rho \in [\![P]\!]$ by Definition 4. □

Lemma 2. Let $\hat{G_\rho}$ be a well-formed symbolic plain execution graph and G_σ be a concrete input execution graph such that $G_\sigma \in [\![P]\!]$. If $\hat{G_\rho}$ satisfies the following conditions:

- there is a map $\delta : \hat{G_\rho}.\text{Evt} \rightarrow G_\sigma.\text{Evt}$ such that for each event $e \in \hat{G_\rho}.\text{Evt}$, $\delta(e) \approx e$ and if $e.val \in \text{Val}$ (i.e., $e.val$ is concrete), then $\delta(e) = e$.
- if $\langle e_1, e_2 \rangle \in G_\sigma.\text{po}$ and $e_2 = \delta(e_2')$ for some $e_2' \in \hat{G_\rho}.\text{Evt}$, then there is an event $e_1' \in \hat{G_\rho}.\text{Evt}$ such that $e_1 = \delta(e_1')$ and $\langle e_1', e_2' \rangle \in \hat{G_\rho}.\text{po}$,
- for each thread $t \in \hat{G_\rho}.\text{Thrd}$ and each event $e \in \hat{G_\rho}|_t.\text{Evt}$, if there is a read event $r \in \hat{G_\rho}|_t.\text{Rd}$, such that $\langle \delta(r), \delta(e) \rangle \in G_\sigma.\texttt{ctrl}$, then $r.val \in \text{Val}$ (i.e., $r.val$ is concrete),
- for each write event $w \in \hat{G_\rho}.\text{Wrt}$, if $w.val \in \text{Val}$ (i.e., $w.val$ is concrete), then for all $r \in \hat{G_\rho}.\text{Rd}$ such that $\langle \delta(r), \delta(w) \rangle \in G_\sigma.\texttt{data}$, $r.val \in \text{Val}$ (i.e., $r.val$ is concrete).

then $\hat{G_\rho} \in [\![P]\!]$.

Proof. For each thread $t \in \hat{G_\rho}.\text{Thrd}$, we know that $t \in G_\sigma.\text{Thrd}$ because the map $\delta : \hat{G_\rho}.\text{Evt} \to G_\sigma.\text{Evt}$ maps each event $e \in G_\rho.\text{Evt}$ to some event $\delta(e) \in G_\sigma.\text{Evt}$ and $\delta(e) \approx e$, which implies $\delta(e).tid = e.tid$. Because $G_\sigma \in [\![P]\!]$, there is a valid transition path generating $G_\sigma|_t$. Let st_m be the state right after e_n is generated (note that $n \leq m$ because there may be internal transitions that do not emit any memory event). Then there is a set $\{st_0, \ldots, st_m\}$ that is $\sqsubseteq$-ordered. The goal is to derive a set of states $\{st'_0, \ldots, st'_m\}$ from $\{st_0, \ldots, st_m\}$ such that the derived states satisfy the requirements in Lemma 1.

We start by setting $st'_0 = st_0$, where st_0 is the initial state. For each state st_i where $i \in 1 \ldots m$, there is a state st'_i constructed in the following way.

For each state st_i, we first set $st'_i.sprog = st_i.sprog$, $st'_i.pc = st_i.pc$, and $st'_i.\Phi^\theta = st_i.\Phi^\theta$. That is, the sequential program instructions, program counter, and the symbolic expressions recorded for each register on each state are directly copied from the original states. For each read event $r' \in G_\rho|_t.\text{Rd}$, there exists a read event $r = \delta(r')$ in $G_\sigma|_t.\text{Rd}$ such that $r' \approx r$. Set $st'_i.\theta$ be a symbol map such that $st'_i.\theta(r') = st_i.\theta(r)$ for each read event r'. If r' is a concrete event, we can then use $st'_i.\theta$ to compute the concrete values of registers by plugging in $r'.val$. Then $st'_i.\Psi$ and $st'_i.ctrl$ are derived from $st_i.\Psi$ and $st_i.ctrl$ respectively by replacing each r with r'. Finally, $st'_i.G$ is derived from $st_i.G$ by replacing every event e with e', where $e = \delta(e')$. By this construction, for each state st_i and st'_i, we have $st_i \approx st'_i$ and the invariant $subst(\Phi^\theta(reg), val^\theta(\Psi(reg))) = \Phi(reg)$ for $reg \in \text{Reg}$ holds if $\Phi(reg) \in \text{Val}$. In addition, for each $\langle e_1, e_2 \rangle \in \hat{G_\rho}.\text{po}$, there is $\langle \delta(e_1), \delta(e_2) \rangle \in G_\sigma.\text{po}$. Therefore, $\{st'_0, \ldots, st'_m\}$ is $\sqsubseteq$-ordered.

For each state st'_i such that $st'_i.sprog(st'_i.pc) = \texttt{if}\ expr\ \texttt{goto}\ n$, let e'_j be the event generated right before st'_i. Then by the language semantics, for each $r \in st'_i.\Psi(expr)$, we have $\langle \delta(r), \delta(e'_k) \rangle \in G_\sigma.\texttt{ctrl}$ for all $k > j$. From the assumption, we know that $r.val \in \text{Val}$ and $\delta(r) = r$. Hence, $st'_i.\Phi(expr)$ is concrete and $st'_i.\Phi(expr) = st_i.\Phi(expr)$.

Lastly, for each state st'_i such that $st'_i.sprog(st'_i.pc) = [x]\ :=\ expr$, let w be the write event generated after st'_i. If $w.val \in \text{Val}$, then by the semantics of store instruction, $st'_i.\Phi(expr) \in \text{Val}$ has to be concrete. Hence, for all $r \in st'_i.\Psi(expr)$, $r.val \in \text{Val}$ has to be concrete. Note that for all $r \in st'_i.\Psi(expr)$, there is $\langle \delta(r), \delta(w) \rangle \in G_\sigma.\texttt{data}$, and we have the assumption that $\delta(r) = r$. Therefore, $st'_i.\Phi(expr) = st_i.\Phi(expr) \in \text{Val}$. □

Proposition 3. Let P be a program. For each sequentially consistent execution graph of P, if for each conflicting memory event $e_1 \bowtie e_2$, $e_1, e_2 \in CS_l$ for some $l \in \text{Lock}$, then every sound execution of P is sequentially consistent.

Proof. Towards contradiction, suppose not. Then there is an execution of P that is not sequentially consistent but is M-consistent for some MCA memory model M. Hence, there is a $(\texttt{po} \cup \texttt{com} \cup \texttt{sync})^+$ cycle in the execution where at least one of the edges is not ppo edge. By the transitive nature of po order, for each event in the cycle is either a source or a target or both of a communication com edge.

First, note that com relates conflicting events. Hence, for each two events related by com, there is a lock protecting the two events. In other words, for each $\langle e_1, e_2 \rangle \in \texttt{com}$, there are lock events $\texttt{acq}_1(\texttt{l})$ and $\texttt{rel}_1(\texttt{l})$ enclosing e_1 and $\texttt{acq}_2(\texttt{l})$ and $\texttt{rel}_2(\texttt{l})$ enclosing e_2. In addition, we can also infer that $\texttt{rel}_1(\texttt{l}) \xrightarrow{\texttt{sync}} \texttt{acq}_2(\texttt{l})$ since critical sections of the same lock are linearly ordered and the other direction would immediately yield a contradiction.

In addition, for each two events $e_1 \xrightarrow{\texttt{po}} e_2$ but not $e_1 \xrightarrow{\texttt{ppo}} e_2$, they are enclosed in the same one or more critical sections. If not, then either they are not in the same critical section of some

lock or they are both not in any critical section. In the former case, there would be a lock event po ordered between e_1 and e_2. By our new definition for ppo, this would result in $e_1 \xrightarrow{\text{ppo}} e_2$, which is a contradiction. In the latter case, note that each event in this cycle is either a source or a target or both of a communication com edge relating conflicting events. The hypothesis states that each conflicting events are protected by the same lock, which implies the existence of a critical section enclosing every event in the cycle.

Since for each two events $e_1 \xrightarrow{\text{po}} e_2$ but not $e_1 \xrightarrow{\text{ppo}} e_2$, they are enclosed in the same one or more critical sections, from the new definition of ppo, we can infer that $\texttt{acq} \xrightarrow{\text{ppo}} \texttt{rel}$ for *any* acquire event and release event (they may be lock events for different locks) enclosing e_1 and e_2.

Hence, if there is a $(\text{po} \cup \texttt{com} \cup \text{sync})^+$ cycle in the execution graph, there is also a $(\text{ppo} \cup \texttt{com} \cup \text{sync})^+$ cycle in the same graph, contradicting with the assumption that the execution graph is M-consistent. Thus, any sound execution of the same program must be sequentially consistent. □

D Proofs of §5

Proposition 4. If $\langle S_\sigma, C_\sigma \rangle$ enables a bug sequence b, then $\hat{G}_\rho$ is well-formed up to concrete events and $\hat{G}_\rho \rhd b$.

Proof. Since $G_\sigma.\text{po}$ is well-formed, it follows that $\hat{G}_\rho.\text{po}$ is also well-formed. For each read event $r \in \hat{G}_\rho.\text{Rd}$, if $r.val \in \text{Val}$, then it means $r = \delta(r) \in C_\sigma$. Since $\langle S_\sigma, C_\sigma \rangle$ enables the bug sequence b, by (VII), there exists a write event $w \in \hat{G}_\rho.\text{Wrt}$ such that $\delta(w) = w \in C_\sigma$, $w.loc = r.loc$, and $w.val = r.val$. Hence, $\hat{G}_\rho$ is read feasible. In addition, (III) ensures $\hat{G}_\rho$ is lock feasible.

(VIII) ensures the premise for composability. No Skipping is satisfied by (I). Same Control Flow is satiesfied by (IV). Hence $\hat{G}_\rho \rhd b$. □

Proposition 5. If $\langle S_\sigma, C_\sigma \rangle$ enables a bug sequence b, and $G_\sigma \in [\![P]\!]$, then $\hat{G}_\rho \in [\![P]\!]$.

Proof. It's obvious that the map δ satisfies the requirement $\delta(e) \approx e$ for each $e \in \hat{G}_\rho.\text{Evt}$. For each event $e_1 \in \hat{G}_\rho.\text{Evt}$, let $e_1' = \delta(e)$. Then $e_1' \in S_\sigma$. By (II), if there is $\langle e_2', e_1' \rangle \in G_\sigma.\text{po}$, then $e_2' \in S_\sigma$. Hence, there is $e_2 \in \hat{G}_\rho.\text{Evt}$ such that $\delta(e_2) = e_2'$ and $\langle e_2, e_1 \rangle \in G_\rho.\text{po}$. For each thread $t \in \hat{G}_\rho.\text{Thrd}$, let e_t be the last event on thread t. By (V), if there is a read event $r \in G_\sigma.\text{Evt}$ such that $r = \delta(r')$ for some $r' \in \hat{G}_\rho.\text{Rd}$ and $\langle r, \delta(e_t) \rangle \in G_\sigma.\texttt{ctrl}$, then $r \in C_\sigma$. Then $r = \delta(r') = r'$. A write event w has a concrete value if and only if $\delta(w)$ is included in C_σ. From (VI), for each $\delta(w) \in C_\sigma$, if there is $\langle \delta(r), \delta(w) \rangle \in G_\sigma.\texttt{data}$, then $\delta(r) \in C_\sigma$, which means $\delta(r) = r \in \hat{G}_\rho.\text{Rd}$. Therefore, by Lemma 2, $\hat{G}_\rho \in [\![P]\!]$. □

Lemma 3. Let $\hat{G}_\rho = (\text{Evt}, \text{po})$ be a plain execution graph such that $\hat{G}_\rho.\text{Evt}$ is downward-closed with respect to $G_\sigma.(\text{po} \cup \texttt{rf}|_C)$ where C is the set of concrete read and write events of $\hat{G}_\rho$ such that $((\texttt{data} \cup \texttt{rf})^*;\texttt{ctrl})^+ \subseteq C$ for a bug sequence b, and $\delta(e) = e$ for each $e \in C$. Then $\hat{G}_\rho$ is read-feasible up to C and executable.

Proof. It is clear that $\hat{G}_\rho$ is read feasible by the fact that $\hat{G}_\rho$ is downward-closed with respect to $G_\sigma.(\text{po} \cup \texttt{rf}|_C)$ for all concrete events $e \in C$. Because $\hat{G}_\rho$ is downward-closed with respect to $G_\sigma.\text{po}$, for each $e_2 \in \hat{G}_\rho.\text{Evt}$, if there is $\langle e_1', \delta(e_2) \rangle \in G_\sigma.\text{po}$, then there is $e_1 \in \hat{G}_\rho.\text{Evt}$ such that $e_1' = \delta(e_1)$ and $\langle e_1, e_2 \rangle \in \hat{G}_\rho.\text{po}$.

In addition, $(\texttt{data} \cup \texttt{rf})^*;\texttt{ctrl} \subseteq C$ ensures that there is a pair $\langle S_\sigma, C_\sigma \rangle$ that satisfies (II), (V), (VI), (VII) of Definition 10 and $C_\sigma \subseteq C$. Let $S_\sigma = \delta(\hat{G_\rho}.\mathsf{Evt})$ and $C_\sigma = G_\sigma.((\texttt{data} \cup \texttt{rf})^*;\texttt{ctrl}) \cap \delta(\hat{G_\rho}.\mathsf{Evt})$.

- II S_σ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$. This can be shown by noticing that $\hat{G_\rho}.\mathsf{Evt}$ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$.
- V For each $e \in S_\sigma$, if $\langle r, e \rangle \in G_\sigma.\texttt{ctrl}$, then $r \in C_\sigma$. This can be shown by noticing $\texttt{ctrl} \subseteq ((\texttt{data} \cup \texttt{rf})^*;\texttt{ctrl})^+$.
- VI For each $e \in C_\sigma$, if $\langle r, e \rangle \in G_\sigma.\texttt{data}$, then $r \in C_\sigma$. By definition of $\texttt{data}$, we know that e is a write. Then for $e \in C_\sigma$, there must be a read $r' \in C_\sigma$ such that $e \xrightarrow{\texttt{rf}} r'$. Given that $(\texttt{data};\texttt{rf})^*;\texttt{ctrl} \subseteq ((\texttt{data} \cup \texttt{rf})^*;\texttt{ctrl})^+$, we have $r \in C_\sigma$.
- VII For each $r \in C_\sigma$, there exists a write $w \in C_\sigma$ such that $r.val = w.val$ and $r.loc = w.loc$. This is ensured by the well-formedness of $G_\sigma.\texttt{rf}$.

The similar reasoning steps as in the proof of Proposition 5, (II), (V), and (VI) ensures executability of $\hat{G_\rho}$. (VII) ensures that $\hat{G_\rho}$ is read-feasible.

□

Lemma 4. Let $\hat{G_\rho} = (\mathsf{Evt}, \mathsf{po})$ be a plain execution graph such that $\hat{G_\rho}.\mathsf{Evt}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \texttt{sync})$, then $\hat{G_\rho}$ is lock-feasible.

Proof. Suppose, towards contradiction, that $\hat{G_\rho}$ is not lock feasible. Then there exists a lock l such that there are two open critical sections in $\hat{G_\rho}$ with acquire events a_1 and a_2. Then there are release events $r_1 = \mathsf{match}(a_1)$ and $r_2 = \mathsf{match}(a_2)$ in G_σ such that either $r_1 \xrightarrow{\texttt{sync}} a_2$ or $r_2 \xrightarrow{\texttt{sync}} a_1$ and neither of them are included in $\hat{G_\rho}$. This immediately contradicts with the hypothesis that $\hat{G_\rho}$ is downward-closed with respect to $G_\sigma.(\mathsf{po} \cup \texttt{sync})$, which implies that at least one of r_1 and r_2 is included in $\hat{G_\rho}$. Thus, $\hat{G_\rho}$ is lock-feasible. □

Lemma 5. Let G_σ be an input execution such that G_σ is sequentially consistent, equipped with a linear $\texttt{trace}$ order. Let $\hat{G_\rho}$ be a symbolic plain execution that is well-formed, and $\hat{G_\rho} \rhd b$ where b is a reported bug. If for all acquire event $\mathsf{acq}(l) \in \hat{G_\rho}.\mathsf{Evt}$ such that $l \in \mathsf{LocksHeld}(e)$, $\langle \mathsf{acq}(l), e \rangle \in G_\sigma.\texttt{trace}$ for each event $e \in b.\mathsf{Evt}$ that is in a critical section where the acquire event of the critical section $\mathsf{acq}(l) \in \hat{G_\rho}$, then there exists a memory order insertion scheme over $\hat{G_\rho}.\mathsf{Evt}$ such that $\hat{G_\rho}$ is sequentially consistent.

Proof. Set

$$\hat{G_\rho}.\texttt{rf} = G_\sigma.\texttt{rf}|_C$$
$$\hat{G_\rho}.\texttt{co} = \delta^{-1}(G_\sigma.\texttt{co}|_{\delta(\hat{G_\rho}.\mathsf{Wrt})})$$
$$\hat{G_\rho}.\texttt{sync} = G_\sigma.\texttt{sync} \cap (\hat{G_\rho}.\mathsf{Rel} \times \mathsf{Acq})$$

where C is the set of concrete events in $\hat{G_\rho}$. Since $\hat{G_\rho}$ is read feasible, $\hat{G_\rho}.\texttt{rf}$ is well-formed over concrete events.

Since G_σ is sequentially consistent, the inserted orders in $\hat{G_\rho}$ do not form a $(\mathsf{po} \cup \texttt{sync} \cup \mathsf{com})^+$ cycle.

For each $e \in b.\mathsf{Evt}$ such that e is in a critical section, let the acquire event of the critical section be $\mathsf{acq}(l) \in \hat{G_\rho}.\mathsf{Acq}$. Then this critical section is an open critical section in $\hat{G_\rho}$. Since $\hat{G_\rho}$ is lock feasible, all other critical sections of l are closed, i.e., each $\mathsf{acq}(l)' \in \hat{G_\rho}.\mathsf{Acq}$ has a

matching $\mathrm{rel}(l)'$. Since $\hat{G}_\rho.\mathtt{sync} = G_\sigma.\mathtt{sync} \subseteq G_\sigma.\mathtt{trace}$, we have $\langle \mathrm{rel}(l)', \mathrm{acq}(l)\rangle \in \hat{G}_\rho.\mathtt{sync}$ for all $\mathrm{rel}(l)' \neq \mathrm{match}(\mathrm{acq}(l))$. Hence, all open critical sections are ordered last in $\hat{G}_\rho$.

With both requirements satisfied, we can conclude that $\hat{G}_\rho$ is sequentially consistent. □

Lemma 6. Let $\hat{G}_\rho$ be a symbolic execution with a well-formed $\mathtt{rf}$-map over concrete events, a total $\mathtt{co}$ order over write events to the same location, and a well-formed $\mathtt{sync}$ over lock events. If $\hat{G}_\rho$ is $\mathcal{M}$-consistent and $\hat{G}_\rho \in [\![P]\!]$ with $e.val \in \mathsf{Val}$ for each $e \in \mathsf{preserve}(b)$, then there exists a map $\Theta : \mathsf{Sym} \to \mathsf{Val}$ such that the concrete execution $G_\rho = \Theta(\hat{G}_\rho)$ with a complete $\mathtt{rf}$-map is $\mathcal{M}$-consistent and $G_\rho \in [\![P]\!]$.

Proof. We begin by inserting $\mathtt{rf}$ orders so that for each (symbolic) read event $r \in \hat{G}_\rho.\mathsf{Rd}$, there is a unique write event $w \in \hat{G}_\rho.\mathsf{Wrt}$ such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{rf}$ and $w.loc = r.loc$. In the rest of the proof, we define the global happens-before as $\mathtt{ghb} = \mathtt{com} \cup \mathtt{sync} \cup \mathtt{ppo} \cup (\mathtt{po}; [L]) \cup ([L]; \mathtt{po})$.

Claim: Let $t \in \hat{G}_\rho.\mathsf{Thrd}$ and $r \in \hat{G}_\rho.\mathsf{Rd}$ be the first read event with symbolic value in thread t. Let $\hat{G}'_\rho$ be the resulting graph after inserting $\langle w, r\rangle$ to $\hat{G}_\rho.\mathtt{rf}$, where w is the $\mathtt{co}$-maximal write such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{ghb}$. If there is no such write event, then w is the initial write to the memory location $r.loc$. Then $\hat{G}'_\rho$ is still $\mathcal{M}$-consistent.

Proof of Claim: Comparing to $\hat{G}_\rho$, $\hat{G}'_\rho$ has the following memory orders inserted: $\langle w, r\rangle \in \mathtt{rf}$, and $\langle r, w'\rangle \in \mathtt{fr}$ for all w' such that $\langle w, w'\rangle \in \hat{G}_\rho.\mathtt{co}$. Since w is the $\mathtt{co}$-maximal write event such that $\langle w, r\rangle \in \hat{G}_\rho.\mathtt{ghb}$, we know that $\langle w', r\rangle \notin \hat{G}_\rho.\mathtt{ghb}$. Hence, there is no new $\mathtt{ghb}$ cycle formed with the newly inserted orders. Hence, $\hat{G}'_\rho$ is also $\mathcal{M}$-consistent.

For each thread $t \in \hat{G}_\rho.\mathsf{Thrd}$, we start from the beginning of t and insert $\mathtt{rf}$ orders for each symbolic read events according to the previous claim. Since $\mathcal{M}$-consistency is maintained at each step, the resulting execution graph is $\mathcal{M}$-consistent.

Let $\hat{G}'_\rho$ be the resulting execution graph after the previous step of inserting $\mathtt{rf}$ orders. We now show that there is a map $\Theta : \mathsf{Sym} \to \mathsf{Val}$ that maps $\hat{G}'_\rho$ to a concrete execution G_ρ.

Claim: For each event $e \in \hat{G}'_\rho$ such that $e.val \in \mathsf{Sym}$, then there exists a value $v \in \mathsf{Val}$ such that the resulting execution graph $\hat{G}''_\rho$ after substituting $e.val$ by v is executable, i.e., $\hat{G}''_\rho \in [\![P]\!]$, and well-formed.

Proof of Claim: Induction on the events on $(\mathtt{rf} \cup \mathtt{data})^+$ chain. Let $e \in \hat{G}'_\rho.\mathsf{Evt}$. If there is no event $(\mathtt{rf} \cup \mathtt{data})^+$-ordered before e or all events $(\mathtt{rf} \cup \mathtt{data})^+$-ordered before e are concrete, then set $e.val = \delta(e).val$. Then obviously the resulting execution graph $\hat{G}''_\rho \in [\![P]\!]$.

If $e \in \hat{G}'_\rho.\mathsf{Rd}$, then there exists a unique write event $w \in \hat{G}'_\rho.\mathsf{Wrt}$ such that $\langle w, e\rangle \in \hat{G}'_\rho.\mathtt{rf}$. If $w.val \in \mathsf{Val}$, then set $e.val = w.val$. If $w.val \in \mathsf{Sym}$, then by induction hypothesis, there exists a value $v \in \mathsf{Val}$ for w such that the resulting execution graph is executable and well-formed. Set $e.val = v$. The resulting graph is executable because $e \notin C_\sigma$, and well-formed since $e.val = w.val$.

If $e \in \hat{G}'_\rho.\mathsf{Wrt}$, since $\hat{G}'_\rho \in [\![P]\!]$, there exists a state st_i right before e is generated such that $st_i.sprog(pc) = [x] := expr$. By induction hypothesis, for each read event r such that $\langle \delta(r), \delta(e)\rangle \in G_\sigma.\mathtt{data}$, there exists a value such that the resulting graph is executable and well-formed. Set $e.val = subst(st_i.\Phi^\theta(expr), val(st_i.\Psi(expr)))$ where $val : \mathsf{Sym} \to \mathsf{Val}$ is a value map for those reads. The resulting graph is executable and well-formed because $e \notin C_\sigma$ and each read that e depends on is mapped to a concrete value before $e.val$ is mapped.

Then G_ρ can be obtained by following the $(\mathtt{rf} \cup \mathtt{data})^+$ chain and replace the value of each symbolic event by a concrete value according to the above description. By previous claims, G_ρ is $\mathcal{M}$-consistent, executable, and well-formed.

□

E Proofs of §6

Theorem 4. If $\langle e_1, e_2\rangle$ is the first reported race such that $e_1 \parallel_{\mathsf{hb}} e_2$, i.e., for all events $e \xrightarrow{\mathtt{trace}}_\sigma e' \xrightarrow{\mathtt{trace}}_\sigma e_2$ such that $e \bowtie e'$, $\langle e, e'\rangle \in G_\sigma.\mathsf{hb}$, then $\langle e_1, e_2\rangle$ is a sound data race.

Proof.

► *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \{e \in G_\sigma.\mathsf{Evt} \mid \langle e, e_1\rangle \in G_\sigma.\mathsf{hb} \ \vee \ \langle e, e_2\rangle \in G_\sigma.\mathsf{hb}\}$$
$$C_\sigma = G_\sigma.(\mathsf{Rd} \cup \mathsf{Wrt}) \cap S_\sigma$$

Because hb is transitive, S_σ is downward-closed w.r.t. hb. In addition, note that for each $\langle w, r\rangle \in G_\sigma.\mathtt{rf}$ and $w \xrightarrow{\mathtt{trace}} r \xrightarrow{\mathtt{trace}} e_1$, since $w \bowtie r$ by definition, then $\langle w, r\rangle \in G_\sigma.\mathsf{hb}$. Similarly, for each $r \in G_\sigma.\mathsf{Rd}$ such that $\langle e_1, r\rangle \in G_\sigma.\mathtt{rf}$ and $e_1 \xrightarrow{\mathtt{trace}} r \xrightarrow{\mathtt{trace}} e_2$, $\langle e_1, r\rangle \in G_\sigma.\mathsf{hb}$. Hence, for every reads $r \in S_\sigma.\mathsf{Rd}$, if there is $\langle w, r\rangle \in G_\sigma.\mathtt{rf}$, then $w \in S_\sigma$, i.e., S_σ is downward-closed w.r.t. $G_\sigma.\mathsf{po} \cup \mathtt{rf} \cup \mathsf{sync}$. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evt} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$ is the identity function. Therefore, we omit the application of δ in the rest of the proof for better readability.

We first show that $G_\rho \rhd b$. Because $e_1 \parallel_{\mathsf{hb}} e_2$, we know that $\{e_1, e_2\} \cap S_\sigma = \emptyset$. Then No Skipping is satisfied because $G_\sigma.\mathsf{po} \subseteq G_\sigma.\mathsf{hb}$. Since δ is the identity function, for each $r \in G_\sigma.\mathsf{Rd}$ such that $\langle r, e_1\rangle \in G_\sigma.\mathtt{ctrl}$ or $\langle r, e_2\rangle \in G_\sigma.\mathtt{ctrl}$, $\delta(r) = id(r) = r \in G_\rho.\mathsf{Evt}$. Hence, Same Control Flow is also satisfied.

We now show that G_ρ is well-formed and executable. By Lemma 3, G_ρ is executable. Since $G_\rho.\mathsf{Evt} \subseteq G_\sigma.\mathsf{Evt}$ and $G_\sigma.\mathtt{rf}|_{S_\sigma} \subseteq G_\sigma.\mathsf{hb}$, G_ρ is reads-from feasible. Since $G_\sigma.(\mathsf{po} \cup \mathsf{sync}) \subseteq G_\sigma.\mathsf{hb}$, by Lemma 4, G_ρ is lock feasible.

► *Inserting Memory Orders.* Finally, we show that there exists a memory insertion scheme such that G_ρ is sequentially consistent. Observe that for all $\mathsf{acq}(l) \in G_\rho.\mathsf{Acq}$, where $l \in \mathsf{LocksHeld}(e_1)$, we have $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_1$ because either $\langle \mathsf{acq}(l), e_1\rangle \in G_\sigma.\mathsf{hb}$ or $\langle \mathsf{acq}(l), e_2\rangle \in G_\sigma.\mathsf{hb}$ by definition. In the former case, since $\mathsf{hb} \subseteq \mathtt{trace}$, we know that $\mathsf{acq}(l) \xrightarrow{\mathtt{trace}} e_1$. In the latter case, since $l \in \mathsf{LocksHeld}(e_1)$, we can infer that there is a hb path between e_1 and $\mathsf{acq}(l)$. Then $\mathsf{acq}(l)$ has to be trace ordered before e_1 as the alternative would imply $e_1 \xrightarrow{\mathsf{hb}} \mathsf{acq}(l) \xrightarrow{\mathsf{hb}} e_2$, i.e., a contradiction with the assumption that $e_1 \parallel_{\mathsf{hb}} e_2$. Then we can apply Lemma 5 and conclude that there exists a memory insertion scheme such that G_ρ is sequentially consistent.

► *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

Theorem 5. If $e_1 \parallel_{\mathsf{shb}} e_2$, then $\langle e_1, e_2\rangle$ is a sound race.

Proof. Let S_σ be a set defined as the following.

$$S_\sigma = \{e \in G_\sigma.\mathsf{Evt} \mid \langle e, e_1\rangle \in G_\sigma.\mathsf{shb} \ \vee \ \langle e, e_2\rangle \in G_\sigma.\mathsf{shb}\}$$
$$C_\sigma = G_\sigma.(\mathsf{Rd} \cup \mathsf{Wrt}) \cap S_\sigma$$

Note that if there is an rf edge between e_1 and e_2, it is excluded from shb by definition. Hence, $e_1, e_2 \notin S_\sigma$ is guaranteed if $\langle e_1, e_2\rangle$ is reported as a race. Because shb is transitive, S_σ is downward-closed w.r.t. shb. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evt} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$ is the identity function. The rest of the proof follows the exact same reasoning process of the soundness proof for HB because $\mathsf{hb} \subseteq \mathsf{shb}$. □

Theorem 6. If $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \emptyset$, then $\langle e_1, e_2\rangle$ is a sound race.

Proof.

▶ *Constructing a Plain Execution Graph.* Let S_σ be a set defined as the following.

$$S_\sigma = \mathsf{SRFIdeal}_\sigma(e_1, e_2)$$
$$C_\sigma = G_\sigma.(\mathsf{Rd} \cup \mathsf{Wrt}) \cap S_\sigma$$

By the second condition of the definition of $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$, we know that S_σ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})$. Let G_ρ be a plain execution graph such that $G_\rho.\mathsf{Evt} = S_\sigma$ and $G_\rho.\mathsf{po} = G_\sigma.\mathsf{po}|_{S_\sigma}$. In addition, $\delta : G_\rho.\mathsf{Evt} \to G_\sigma.\mathsf{Evt}$ is the identity function.

We first show that $G_\rho \rhd b$ where $b = e_1 e_2$. To start with, $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \emptyset$ comes from the assumption. In addition, No Skipping is satisfied by the base condition of the definition, i.e., $\{\mathsf{prev}(e_1), \mathsf{prev}(e_2)\} \subseteq \mathsf{SRFIdeal}_\sigma(e_1, e_2)$. Since δ is the identity function, for each $r \in G_\sigma.\mathsf{Rd}$ such that $\langle r, e_1\rangle \in G_\sigma.\mathtt{ctrl}$ or $\langle r, e_2\rangle \in G_\sigma.\mathtt{ctrl}$, $\delta(r) = id(r) = r \in G_\rho.\mathsf{Evt}$. Hence, Same Control Flow is also satisfied.

We now show that G_ρ is well-formed and executable. Since $G_\rho.\mathsf{Evt}$ is downward-closed w.r.t. $G_\sigma.(\mathsf{po} \cup \mathtt{rf})$, by Lemma 3, G_ρ is read feasible and executable. If there are two open critical sections co-exist in G_ρ, then there exists two acquire events $\mathsf{acq}_1(l)$ and $\mathsf{acq}_2(l)$ such that both their matching release events are not included in $G_\rho.\mathsf{Evt}$. However, this contradict with the third condition of the definition of $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$. Hence, for each lock $l \in \mathsf{Lock}$, there is at most one open critical section present in G_ρ. That is, G_ρ is lock feasible.

▶ *Inserting Memory Orders.* Lastly, we show that there exists a memory order insertion scheme such that G_ρ is sequentially consistent. Observe that for any $\mathsf{acq}(l) \in G_\rho.\mathsf{Acq}$ where $l \in \mathsf{LocksHeld}(e_i)$ and $i \in \{1, 2\}$, $\mathsf{acq}(l) \xrightarrow{\mathsf{trace}} e_i$. This is because for each $\mathsf{acq}(l)$, if $l \in \mathsf{LocksHeld}(e_i)$, then either $\mathsf{acq}(l)$ is the acquire event of the critical section of e_i, or there are acquire event $\mathsf{acq}_i(l)$ and release events $\mathsf{rel}(l)$ and $\mathsf{rel}_i(l) \in G_\sigma$ where $\mathsf{match}(\mathsf{rel}(l)) = \mathsf{acq}(l)$, $\mathsf{match}(\mathsf{rel}_i(l)) = \mathsf{acq}_i(l)$, and $\mathsf{acq}_i(l)$ is the acquire event of e_i's critical section. In the former case, it's obvious that $\mathsf{acq}(l) \xrightarrow{\mathsf{trace}} e_i$. In the latter case, either $\mathsf{rel}_i(l) \xrightarrow{\mathsf{sync}} \mathsf{acq}(l)$ or $\mathsf{rel}(l) \xrightarrow{\mathsf{sync}} \mathsf{acq}_i(l)$. We know that $\mathsf{rel}_i(l) \notin G_\rho.\mathsf{Evt}$ because G_ρ is downward-closed w.r.t. $G_\sigma.\mathsf{po}$ and $e_i \notin G_\rho.\mathsf{Evt}$. Hence, if $\mathsf{acq}(l) \in G_\rho$, it must be that $\mathsf{rel}(l) \xrightarrow{\mathsf{sync}} \mathsf{acq}_i(l)$, which implies that $\mathsf{acq}(l) \xrightarrow{\mathsf{trace}} e_i$. By Lemma 5, there exists an order insertion scheme such that G_ρ is sequentially consistent.

▶ *Mapping to Concrete Execution Graph.* Since the execution graph G_ρ is already concrete by construction, there is no further step needed. □

F Enhanced-MCR-tso

In this section, we define an enhanced version of the previous algorithm that uses a novel idea of *transformation* to predict data races under the TSO model. This enables us to catch more data races than the work of Huang and Huang [9]. We provide an example in Appendix H.

In addition to the constraints defined above, we preprocess the input execution G_σ with *read elimination*. During this phase, all removable read events are eliminated from the trace, where $\mathsf{removable}(G_\sigma)$ is a subset of read events that can be found inductively as described as the following.

For each read event $r \in G_\sigma.\mathsf{Rd}$, if there is a write event $w \in G_\sigma.\mathsf{Wrt}$ such that $\langle w, r\rangle \in G_\sigma.(\mathtt{rf} \cap \mathsf{po})$, then $r \in \mathsf{removable}(G_\sigma)$ if there is no event such that $w \xrightarrow{\mathsf{po}} e \xrightarrow{\mathsf{po}} r$, or for each event e such that $w \xrightarrow{\mathsf{po}} e \xrightarrow{\mathsf{po}} r$, either e is a write event where $e.loc \neq r.loc$, or $e \in \mathsf{removable}(G_\sigma)$.

After the read events in $\mathsf{removable}(G_\sigma)$ are removed from the execution, we use the same set of constraints, $\Phi_{\mathsf{ppo}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}$, to determine if $\langle e_1, e_2\rangle$ forms a data race.

Before we start proving its soundness, we introduce a proposition from the work of Lahav and Vafeiadis [14].

Proposition 11. If $G \rightsquigarrow_{tso} G'$ and G' is TSO-consistent, then G is TSO-consistent.

In [14], two transformations were considered, read elimination and write-read reordering. $G \rightsquigarrow_{tso} G'$ iff G' can be obtained via either of the transformations. Therefore, we can use this proposition for read elimination.

The soundness theorem is the following.

Theorem 7. If there exists a map $\rho : O \to \mathsf{Int}$ such that Φ is satisfiable for (e_1, e_2), i.e., $\exists \rho \models \Phi_{\mathsf{ppo}} \wedge \Phi_{\mathsf{lock}} \wedge \Phi_{\mathsf{race}}(e_1, e_2)$ after read elimination, then $\langle e_1, e_2 \rangle$ is a sound race.

Proof.

► *Constructing a Plain Execution Graph.* Let S_σ be a lock-feasible event set that is downward-closed w.r.t. $G_\sigma.\mathsf{po} \cup \mathtt{rf}$ from $b = e_1 e_2$.Let S_ρ be a set of data-abstract equivalent events of S_σ where $e' \in S_\rho$ iff there is $e \in S_\sigma$ such that $e' \approx e$. We now define a bijective map $\delta : S_\rho.\mathsf{Evt} \to S_\sigma.\mathsf{Evt}$ such that $\delta(e) \approx e$. Let $C = G_\sigma.\mathsf{Wrt} \cup \mathsf{Rd}$. If $\delta(e) \in C$, then we set $\delta(e) = e$. Let $\hat{G_\rho}$ be a symbolic plain execution graph such that $\hat{G_\rho}.\mathsf{Evt} = S_\rho$ and $\hat{G_\rho}.\mathsf{po} = \delta(G_\sigma.\mathsf{po}|_{S_\sigma})$.

By Lemma 3 and the fact that S_σ is lock-feasible, we get $\hat{G_\rho}$ is well-formed and executable. In addition, since S_σ is downward-closed from b, $\hat{G_\rho} \rhd b$.

► *Inserting Memory Orders.* Note that not all events in $\hat{G_\rho}$ is assigned to a number by ρ, due to the read elimination transformation. Let $\hat{G_\rho}'$ be the symbolic execution after applying read elimination on $\hat{G_\rho}$. Then every event in $\hat{G_\rho}'$ is assigned to a natural number by ρ. By the same order insertion scheme as in the proof of Theorem 3, we have that $\hat{G_\rho}'$ is TSO-consistent. By Proposition 11, $\hat{G_\rho}$ is also TSO-consistent.

► *Mapping to Concrete Execution Graph.* Lastly, by Lemma 6, there is a concrete execution graph that inherit all the properties above. □

G A TSO Race Predictable from SC trace

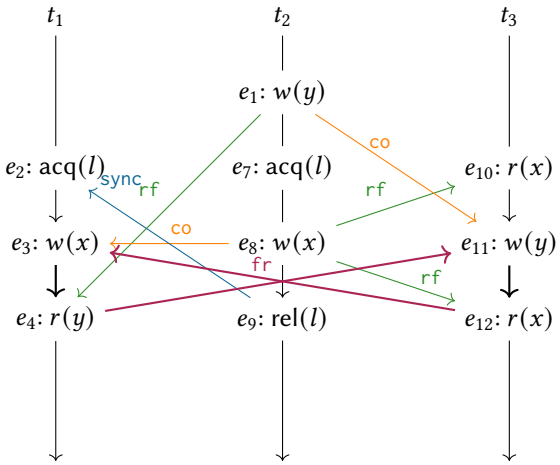
In this section, we provide an example of a TSO data race that is discoverable by MCR-TSO [9]. The example was originally from the paper by Pavlogiannis [21, Figure 8] to show a non-SC-race.

The input execution traces are represented in the form shown in Fig. 9a. Each column represents a thread and each row represents a time-stamp. At each time-stamp, there is exactly one event being executed. Events from different threads can be executed in a interleaving style while respecting the mutual exclusion property of locks. We use e_i to identify each event, where i is the time-stamp when e_i is executed. There are four types of events: read, write, lock acquire, and lock release. We write $r(x)$ and $w(x)$ for a read and a write event on a memory location x, and $\mathsf{acq}(l)$ and $\mathsf{rel}(l)$ for an acquire and a release event on a lock l. The highlighted events are reported as data races. In Fig. 9a, the two highlighted events, e_5 and e_{13} is not a data race under sequential consistency. Indeed,

- If e_5 and e_{13} is a data race, then Fig. 9b shows all the events that have to occur before e_5 and e_{13}. The order among these events has to respect the program order. Therefore, the order of events on each thread follows the same order as captured in the input trace from 9a.
- M2 requires each read event to read from the same write event as they appeared in the input trace to maintain soundness. Therefore, we have $e_8 \to e_{10}$ and $e_8 \to e_{12}$ for location x, and $e_1 \to e_4$ for location y.
- Since e_5 is in a critical section protected by lock l, its critical section has to be ordered after all other critical sections protected by the same lock. In this example, it means the critical section on t_2 has to be ordered before the critical section on t_1. Hence, we can infer $e_9 \to e_2$.

	t_1	t_2	t_3
1		$w(y)$	
2	acq(l)		
3	$w(x)$		
4	$r(y)$		
5	**w(z)**		
6	rel(l)		
7		acq(l)	
8		$w(x)$	
9		rel(l)	
10			$r(x)$
11			$w(y)$
12			$r(x)$
13			**r(z)**

(a) The Input Trace



(b) A TSO-Consistent Witness Execution

Fig. 9. Example from Fig. 8 in [21]

- By the transitivity of the partial order constructed so far, we can see that $e_8 \longrightarrow e_3$. Since there are two read events on t_3 reading from e_8, then they must be ordered before e_3. Therefore, $e_{12} \longrightarrow e_3$.
- Since $e_8 \longrightarrow e_{10}$, by the transitivity of the partial order again, we have $e_1 \longrightarrow e_{11}$. Since e_4 reads from e_1, then it must be that $e_4 \longrightarrow e_{11}$.
- But now a cycle occurs: $e_3 \longrightarrow e_4 \longrightarrow e_{11} \longrightarrow e_{12} \longrightarrow e_3$.

On the other hand, the cycle derived in Fig. 9b is allowed under TSO [20]. Indeed, Fig. 9b also shows the execution with orders among events augmented with specific semantics defined by the axiomatic memory model of TSO. The po order $e_3 \longrightarrow e_4$ and $e_{11} \longrightarrow e_{12}$ are not preserved program order. As a result, the po $\cup$ fr cycle shown in the figure is allowed by the TSO model. Therefore, we can conclude that e_5 and e_{13} form a data race under TSO.

Note that the constraint encoding of MCR can capture this data race, precisely because the constraints do not require e_3 to be ordered before e_4 and e_{11} to be ordered before e_{12}. As a result, there exists a satisfiable solution of the event orders.

H TSO Race Discovered by Read Elimination

In §6, we showed an enhanced version of a data race predictor augmented with Read Elimination. In this section, we show an example that exhibit a sound TSO data race that *cannot* be caught by the TSO analysis by Huang and Huang [9], but can be caught by our enhanced race predictor.

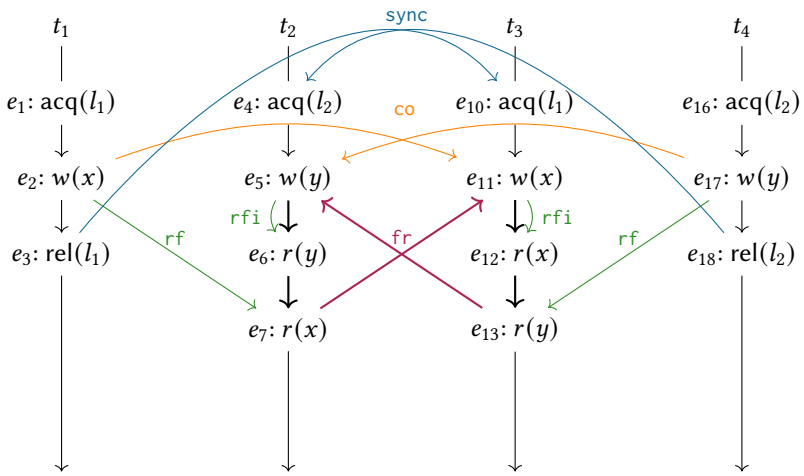
Fig. 10a shows a sequential trace of four threads. The highlighted events, e_8 and e_{17}, form a data race under the TSO [20] model. However, the constraints from [9] fail to catch this data race due to overly strong restriction imposed by Φ_{mem}. The constraints on memory events, Φ_{mem} is defined as

$$\Phi_{mem} = \Phi_{ww} \wedge \Phi_{rr} \wedge \Phi_{rw} \wedge \Phi_{addr}$$

where Φ_{ww} enforces the write-write program orders, Φ_{rr} enforces the read-read program orders, Φ_{rw} enforces the read-write program orders, and Φ_{addr} enforces the program orders between memory events accessing the same location. Note that the first three constraints enforce the preserved

	t_1	t_2	t_3	t_4
1	acq(l_1)			
2	$w(x)$			
3	rel(l_1)			
4		acq(l_2)		
5		$w(y)$		
6		$r(y)$		
7		$r(x)$		
8		**w(z)**		
9		rel(l_2)		
10			acq(l_2)	
11			$w(y)$	
12			rel(l_2)	
13				acq(l_1)
14				$w(x)$
15				$r(x)$
16				$r(y)$
17				**r(z)**
18				rel(l_1)

(a) The Input Trace



(b) Witness Execution

Fig. 10. A TSO Race Captured by Race Predictor after Read Elimination

program orders (ppo) of TSO and the last constraint enforces the program order restricted to the same location (po-loc). The constraint includes all four of them in *one* transitive partial order, while the TSO model considers them *separately* in two assertions:

$$\text{irreflexive } (\texttt{po-loc} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co})^+$$
$$\text{irreflexive } (\texttt{ppo} \cup \texttt{rfe} \cup \texttt{fr} \cup \texttt{co})^+$$

In the example, the program orders $e_5 \longrightarrow e_6$ and $e_{11} \longrightarrow e_{12}$ are po-loc orders whereas $e_6 \longrightarrow e_7$ and $e_{12} \longrightarrow e_{13}$ are ppo orders. Together with the fr edges from e_7 to e_{11} and from e_{13} to e_5, they form a cycle, as shown in Fig. 10b. The cycle makes the constraints unsatisfiable, but is allowed under the memory model of TSO.

On the other hand, augmenting the constraints with the Read Elimination transformation suffices to cover this case. Note that e_6 and e_{12} are removed after the Read Elimination trasformation. The resulting execution graph corresponds to a standard store-buffering (SB) litmus test, which can be captured by the constraint Φ_{mem}.

I WRC-race: another example

In this section, we provide another example of data race under weak memory that can be predicted from an SC trace while preserving the same observable behavior. Fig. 11a shows a sequential trace of four threads. The highlighted events, e_3 and e_{12}, form a data race under the ARMv8 [1] model, assuming there is no data or address dependencies among the events and no control dependency between e_{10} and e_{11}.

First, note that e_3 and e_{12} is not an SC race. To see this, following steps similar to M2 [21], we have:

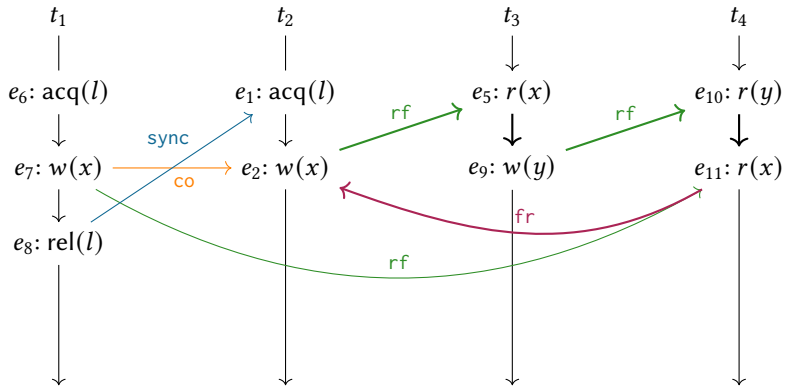
- All events except for the release in t_2 (e_4) are included in a witness execution.
- Since we have an open acquire in t_2, the critical section in t_2 should be ordered after the critical section in t_1, yielding a coherence order from $w(x)$ at line 7 to $w(x)$ at line 2.
- From the inferred coherence order, we get $r(x)$ at line 11 is ordered before $w(x)$ at line 2 (since it reads from the write at line 7)
- Now we have a cycle: $e_2 : w(x) \xrightarrow{\texttt{rf}} e_5 : r(x) \xrightarrow{\texttt{po}} e_9 : w(y) \xrightarrow{\texttt{rf}} e_{10} : r(y) \xrightarrow{\texttt{po}} e_{11} : r(x) \xrightarrow{\texttt{fr}} e_2 : w(y)$. The cycle is highlighted in Fig. 11b.

On the other hand, the cycle is allowed under the ARMv8 [1] model and the witness execution in Fig. 11b is consistent. To see this, note that the program order $e_{10} \longrightarrow e_{11}$ is not included in the *locally-ordered-before* order and hence the external visibility requirement of the ARMv8 model is satisfied.

Received 2024-10-16; accepted 2025-02-18

	t_1	t_2	t_3	t_4
1		acq(l)		
2		$w(x)$		
3		**w(z)**		
4		rel(l)		
5			$r(x)$	
6	acq(l)			
7	$w(x)$			
8	rel(l)			
9			$w(y)$	
10				$r(y)$
11				$r(x)$
12				**r(z)**

(a) The Input Trace



(b) Witness Execution

Fig. 11. A Data Race under the ARMv8 model