An MSO Framework for Weak-Memory Verification and Robustness

Giovanna Kobus Conrado
Aarhus University, Denmark

Andreas Pavlogiannis
Aarhus University, Denmark

Abstract

Memory models are formal specifications of concurrent-program executions, accounting for weak behaviors introduced by compiler and architectural optimizations. The increase of their number and complexity has spawned efforts for uniform verification across whole classes of models, by axiomatizing the models in an adequate metatheory that admits a uniform treatment. In this work, we formally study Monadic Second-Order logic (MSO) as a metatheory for weak memory, by proving results on the treewidth and MSO-expressibility of various popular weak-memory models, as this combination allows us to uniformly tackle several verification problems. In summary, our results are as follows.

First, we prove that executions under Sequential Consistency (SC) have bounded treewidth, while already those under Total Store Order (TSO) do not. Second, we prove that a broad range of models, including Release/Acquire and the full RC20, are MSO-axiomatizable, while others, such as Strong Release/Acquire and TSO, are not, unless the Orthogonal Vectors problem – which requires quadratic time under SETH – can be solved in linear time. Finally, we introduce the notion of *reads-from robustness*, as an extension to recent work on coarse robustness criteria. We show that our treewidth bounds (both upper and lower) have far-reaching algorithmic implications for any of our MSO-axiomatizable models MM: there is an algorithm that, for every program P, either verifies P under MM or reports that P is not reads-from robust against MM. Overall, our results establish a rich and versatile theoretical framework for weak-memory verification and robustness.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Formal languages and automata theory; Software and its engineering → Formal software verification

Keywords and phrases treewidth, monadic second order logic, reads-from robustness

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2026.25

Related Version *Full Version*: https://arxiv.org/abs/2606.20134 [30]

Funding This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN, and by a research grant from STIBOFONDEN.

1 Introduction

The traditional sequential view (SC) of concurrent systems, introduced by Lamport in [57], does not reflect how programs execute on modern software and hardware platforms. Compiler optimizations, cache hierarchies, instruction prefetching and speculative execution are only some of the means that a concurrent program may deviate from its sequential behaviors, introducing weak data consistency between the executing threads. *Weak memory models* are formal specifications of all subtle behavior that a program may exhibit in such settings, and are becoming a standard approach to rigorous concurrent programming, both for software [15, 74, 52, 56, 47, 39, 11, 51, 62] and hardware [72, 10, 68, 67].

Since weak behaviors can be tricky to predict, or even understand for non-experts, there has been considerable work on developing program-analysis methods to support debugging, verification and testing of concurrent programs under weak memory, along various

directions. These include automated verification [12, 3, 5, 51], various notions of program robustness (e.g., data-race freedom, execution-graph robustness) [43, 66, 23, 53], stateless model checking [2, 6, 25, 50], testing [58, 42, 73, 27, 60], and program logics [75, 54, 46, 45]. Since the behavior of a concurrent program is generally memory-model dependent, virtually each such method is specific to a memory model.

The abundance and complexity of memory models has spawned an interest for unified theories that treat whole classes of memory models in a uniform way. Recent developments enable bounded model checking and memory-model comparison for a range of memory models [76, 44, 49, 48]. Although promising, the full potential of unified theories for weak memory is still under exploration. For example, there is no unified approach to reasoning about unbounded executions, performing program differentiation, or deciding program robustness against a range of memory models.

In the setting of message-passing concurrency, a unified theory based on Monadic Second Order Logic (MSO) [32] has been highly effective. Indeed, most common message-passing protocols have been proven expressible in MSO [38], paving the road for utilizing Courcelle's celebrated metatheorem [33] for a variety of tasks, such as under-approximate verification over unbounded executions (by bounding some width parameter instead) [36, 35] and deciding synchronizability [20]. Perhaps surprisingly, however, the connections of MSO to weak memory have thus far remained unexplored. *Which memory models are axiomatizable in MSO and what common verification tasks can this be used for? What is the treewidth of weak-memory executions? Are there MSO-definable robustness criteria that are less sensitive than existing ones based on execution-graphs?* In this work, we make the first steps to utilizing MSO as a unifying theory for weak memory, proving powerful theoretical capabilities, but also limitations.

1.1 Our Contributions

Throughout this work, we represent program executions using *reads-from (*rf*-) graphs* of the form $G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle$, which are analogous to Message Sequence Charts (MSCs) in message-passing systems [38]. Here, the events in E act as vertices of G, while the *sequenced-before* relation sb and the reads-from relation rf are binary relations over E. In contrast to the more common *execution graphs* $\langle E, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}\rangle$, rf-graphs represent only program-observable behavior – in particular, how each thread executes (sb) and which write a read obtains its value from (rf) – and abstract away the low-level architectural detail captured by the *modification order* mo, i.e., the order in which writes appear in the shared memory.

We state here our main contributions. We delegate a number of proofs and discussions to the full version of this paper [30].

1. The treewidth of weak memory. First, we study the treewidth of executions under various memory models. We prove that executions under Sequential Consistency (SC), i.e., those that do not exhibit weak behavior, have bounded treewidth, but executions under any model as weak as Total Store Order (TSO) have unbounded treewidth. We show that weak executions may still enjoy bounded treewidth, and present experiments that indicate that they tend to stay within the treewidth they would have under SC.

2. The MSO of weak memory. We then ask the question: *which popular weak-memory models can be axiomatized in MSO?* We prove that this is the case for Release/Acquire (RA) [15], Relaxed [15], Weak Release/Acquire (WRA, aka Causal Consistency [22]) [51], as well as the Read-Modify-Write (RMW)-free fragment of Strong Release/Acquire (SRA, aka

Causal Convergence [22]) [52]. We further remark that our MSO axiomatization extends to the full RC20 memory model for C/C++ [61], although we defer its treatment to [30] due to space constraints. The MSO axiomatization of each of the aforementioned memory models enables us to equip Courcelle's theorem [33] to tackle a range of fundamental algorithmic questions, in a *unified way*. We also prove that SC, TSO, Partial Store Order (PSO), and the full SRA (with RMWs) are *not axiomatizable in MSO*, unless the Orthogonal Vectors problem – which requires quadratic time under SETH – can be solved in linear time.

3. Reads-from robustness. We introduce the notion of *reads-from (rf-) robustness*. This is a relaxation of standard execution-graph robustness [23, 53], and a fitting next step in the recent line of more permissive robustness criteria [53, 61, 65]. A program P is rf-robust against a memory model MM if the set of rf-graphs of P is the same under MM and SC. We show a fruitful algorithmic interplay between rf-robustness and verification: for every MSO-definable memory model MM that satisfies a natural contiguity property (satisfied by all models in (2) except Relaxed), there is an algorithm that either solves verification for P under MM, or reports that P is not rf-robust against MM. For memory models which have undecidable reachability, like RA, this result implies that the hard instances can be algorithmically classified as non-rf-robust. To our knowledge, rf-robustness is the first computationally useful robustness notion that reasons purely about program-observable behaviors, disentangling them from lower-level architectural details involving the order in which writes appear on the shared memory. We further extend the above to *observational rf-robustness*, which is a relaxation in a similar spirit to observational execution-graph robustness [61], that effectively filters out robustness violations due to read operations that are unused by the program.

2 Concurrent Programs and Memory Models

In this section we develop general notation, introduce the memory models we consider in this work, and define program semantics in an automata-theoretic way.

General notation. Given an integer i, we define $[i] = \{1, \cdots, i\}$. For a binary relation R, we denote the reflexive, transitive, reflexive-transitive closures and inverse relations of R as $R^?$, R^+, R^*, and R^{-1}, respectively. We occasionally call $(e_1, e_2) \in R$ an edge, and write it as $e_1 \xrightarrow{R} e_2$. The composition of R_1 and R_2 is denoted by $R_1; R_2$. We write $\mathsf{acy}(R)$ and $\mathsf{irr}(R)$ to denote that a relation is acyclic and irreflexive, respectively. Given a set S, we write $[S]$ for the identity relation on S. We use $x \triangleq y$ to denote that x is defined to be equal to y.

Program domains. We consider concurrent programs consisting of threads over a finite domain Tid, communicating over a finite set of shared registers Reg, which store values over a finite value domain Val.

2.1 Executions

Here we set up standard notation for representing concurrent executions.

Labels and events. A *(event) label*[1] is either a *read label* $\mathtt{r}(t, x, v)$, a *write label* $\mathtt{w}(t, x, v)$, or a *read-modify-write (RMW) label* $\mathtt{rmw}(t, x, v_\mathtt{r}, v_\mathtt{w})$, where $t \in \mathsf{Tid}$ is a thread id, $x \in \mathsf{Reg}$ is a shared register, and $v, v_\mathtt{r}, v_\mathtt{w} \in \mathsf{Val}$ are values. We let Lab be the domain of labels. An

[1] In related literature, events sometimes also carry an *access mode*, but this will be clear from the context in our setting.

event e represents an execution step of the program, and is defined as $e = \langle \mathsf{id}, \mathsf{lab} \rangle$, where $\mathsf{id} \in \mathbb{N}$ is an identifier and $\mathsf{lab} \in \mathsf{Lab}$ is a label. We write $\mathsf{lab}(e)$ for the label of e, $\mathsf{tid}(e)$ and $\mathsf{loc}(e)$ for the thread id and register of e, while $\mathsf{op}(e) \in \{\mathtt{r}, \mathtt{w}, \mathtt{rmw}\}$ returns the operation of e. If $\mathsf{op}(e) \in \{\mathtt{r}, \mathtt{rmw}\}$, we write $\mathsf{val}_{\mathtt{r}}(e)$ for the value read by e, and if $\mathsf{op}(e) \in \{\mathtt{w}, \mathtt{rmw}\}$, we write $\mathsf{val}_{\mathtt{w}}(e)$ for the value written by e. We often identify events by their label when their id is not important, e.g. we may refer to events $\mathtt{w}(t, x, v)$. We occasionally ignore the value of an event when it is clear from the context or not important, e.g., we may refer to events $\mathtt{w}(t, x)$, or even $\mathtt{w}(x)$. We let E be the domain of events, and further distinguish the sets of read events $\mathsf{R} = \{e \in \mathsf{E} \colon \mathsf{op}(e) = \mathtt{r}\}$, write events $\mathsf{W} = \{e \in \mathsf{E} \colon \mathsf{op}(e) = \mathtt{w}\}$, and RMW events $\mathsf{RMW} = \{e \in \mathsf{E} \colon \mathsf{op}(e) = \mathtt{rmw}\}$. Given a set of events X and a register $x \in \mathsf{Reg}$, we let $X_x = \{e \in X \colon \mathsf{loc}(e) = x\}$, e.g., W_x denotes all writes on register x. Similarly, given a thread $t \in \mathsf{Tid}$, we let $X^t = \{e \in X \colon \mathsf{tid}(e) = t\}$, e.g., R^t denotes all reads of thread t. We extend the sub/super script notation to binary relations R over E, i.e., $R_x = [\mathsf{E}_x]; R; [\mathsf{E}_x]$ and $R^t = [\mathsf{E}^t]; R; [\mathsf{E}^t]$.

Execution graphs. In the context of weak memory, (concrete) program executions are often represented as execution graphs [56] (aka candidate executions [15]). An *execution graph* is a tuple $G = \langle E, \mathsf{sb}, \mathsf{rf}, \mathsf{mo} \rangle$, where $E \subseteq \mathsf{E}$ is a finite set of (distinct) events, and sb, rf, and mo are binary relations over E, satisfying the following conditions.

- The *sequenced-before order* sb records the order of events executed in each thread. The sb relation is *not* transitive, but instead relates each event to its immediate successor. In particular, letting the *program order* $\mathsf{po} \triangleq \mathsf{sb}^+$, we require that (i) po^t is a total order for each $t \in \mathsf{Tid}$, and (ii) sb coincides with the (unique) transitive reduction of po.
- The *reads-from relation* $\mathsf{rf} \subseteq \bigcup_{x \in \mathsf{Reg}} (\mathsf{W}_x \cup \mathsf{RMW}_x) \times (\mathsf{R}_x \cup \mathsf{RMW}_x)$ relates a write/RMW event to a read/RMW event, and indicates that the latter reads its value from the former. The values of the related events must match, i.e., for each $e_1 \xrightarrow{\mathsf{rf}} e_2$, we have $\mathsf{val}_{\mathtt{w}}(e_1) = \mathsf{val}_{\mathtt{r}}(e_2)$, and rf^{-1} must be a function, i.e., (i) if $e_1 \xrightarrow{\mathsf{rf}} e$ and $e_2 \xrightarrow{\mathsf{rf}} e$ then $e_1 = e_2$, and (ii) for all $e \in E \cap (\mathsf{R} \cup \mathsf{RMW})$, there exists some $e' \in E$ such that $e' \xrightarrow{\mathsf{rf}} e$.
- The *modification order* $\mathsf{mo} \subseteq \bigcup_{x \in \mathsf{Reg}} (\mathsf{W}_x \cup \mathsf{RMW}_x) \times (\mathsf{W}_x \cup \mathsf{RMW}_x)$ captures the order in which the write events on each register take effect, i.e., mo_x is total for each $x \in \mathsf{Reg}$.

We often write $G.E$, $G.\mathsf{sb}$, $G.\mathsf{rf}$ and $G.\mathsf{mo}$ for the respective components of G, or simply E, sb, rf and mo when G is clear from the context. The requirement that every read event reads from some write event in E reflects the assumption that registers hold no initial value.

Reads-from and sequenced-before graphs. In our setting, we primarily represent program executions using abstractions of execution graphs. In particular, a *reads-from graph* (aka a partial execution [73]) is a tuple $G = \langle E, \mathsf{sb}, \mathsf{rf} \rangle$, i.e., it lacks the mo component of an execution graph. Going one step further, a *sequenced-before graph* (aka an abstract execution [27]) is a tuple $G = \langle E, \mathsf{sb} \rangle$, i.e., it lacks both the mo and rf components of an execution graph.

2.2 Memory Models

As per standard in literature, we define memory models using a declarative approach. Each memory model is defined as a collection of axioms that the underlying execution graph G must satisfy. These axioms are phrased in terms of the relations of G, as well as some derived relations, defined below.

Derived relations. Besides po, the various consistency axioms use the following derived relations.

Table 1 Consistency axioms.

$\mathsf{acy}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr})$	(sc)	$\mathsf{irr}((\mathsf{hb} \cup \mathsf{mo})^+)$	(strong-write-coherence)
$\mathsf{acy}(\mathsf{po}_x \cup \mathsf{rf}_x \cup \mathsf{mo}_x \cup \mathsf{fr}_x)$	(sc-per-loc)	$\mathsf{irr}(\mathsf{hb})$	(irr-hb)
$\mathsf{acy}(\mathsf{ppo} \cup \mathsf{rfe} \cup \mathsf{moe} \cup \mathsf{fre})$	(external-hb)	$\mathsf{irr}(\mathsf{mo}; \mathsf{hb})$	(write-coherence)
$\mathsf{acy}(\mathsf{pppo} \cup \mathsf{rfe} \cup \mathsf{moe} \cup \mathsf{fre})$	(pso-external-hb)	$\mathsf{irr}(\mathsf{mo}; \mathsf{hb}; \mathsf{rf}^{-1})$	(read-coherence)
$\mathsf{irr}(\mathsf{hb}_l; [\mathsf{W}]; \mathsf{hb}; \mathsf{rf}^{-1})$	(weak-read-coherence)	$\mathsf{irr}(\mathsf{mo}; \mathsf{mo}; \mathsf{rf}^{-1})$	(atomicity)
$[\mathsf{RMW}]; \mathsf{rf}^{-1}; \mathsf{rf}; [\mathsf{RMW}] = [\mathsf{RMW}]$	(weak-atomicity)		

Table 2 Weak-memory models we consider in this work. $\mathsf{MM}^{\backslash \mathtt{rmw}}$ denotes the RMW-free fragment of MM.

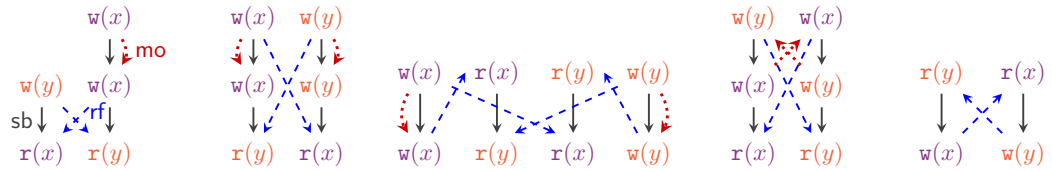
SC	(sc)	WRA	(irr-hb), (weak-read-coherence), (weak-atomicity)
$\mathsf{TSO}^{\backslash \mathtt{rmw}}$	(sc-per-loc), (external-hb)	RA	(irr-hb), (write-coherence), (read-coherence), (atomicity)
$\mathsf{PSO}^{\backslash \mathtt{rmw}}$	(sc-per-loc), (pso-external-hb)	SRA	(strong-write-coherence), (read-coherence), (atomicity)
Relaxed	(sc-per-loc)		

$$\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{mo} \qquad \mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf})^+ \qquad \mathsf{rfe} \triangleq \mathsf{rf} \setminus \mathsf{po} \qquad \mathsf{moe} \triangleq \mathsf{mo} \setminus \mathsf{po}$$
$$\mathsf{fre} \triangleq \mathsf{fr} \setminus \mathsf{po} \qquad \mathsf{ppo} \triangleq \mathsf{po} \setminus (\mathsf{W} \times \mathsf{R}) \qquad \mathsf{pppo} \triangleq \mathsf{ppo} \setminus (\mathsf{W}_x \times \mathsf{W}_y)_{x \neq y}$$

The *from-reads* relation fr orders a read/RMW before all the writes/RMWs that executed after its own writer. The *happens-before* relation hb captures causality chains in program execution. The *external relations* rfe, moe and fre exclude same-thread events from the respective relations. Finally the *preserved program order* ppo and *partially preserved program order* pppo exclude write-read and write-read/write-write orderings from po. For simplicity, we disregard RMWs in ppo and pppo, as we only reason about the RMW-free fragments of the models they appear in (TSO and PSO).

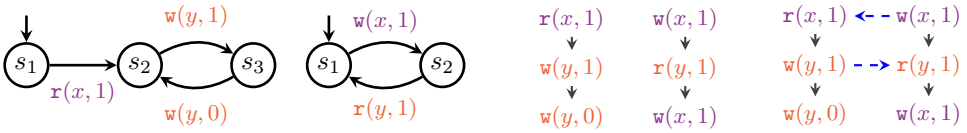
Memory models. In the scope of this work, a memory model MM is defined declaratively as a collection of consistency axioms, the full set of which is shown in Table 1. We use the notation $\mathsf{MM}^{\backslash \mathtt{rmw}}$ to refer to the RMW-free fragment of MM. At a high level, each axiom declares that a certain relation is acyclic or irreflexive, except for (weak-atomicity) which requires that no two RMWs read from the same event. Table 2 defines the memory models considered in this work. SC is the standard model of Sequential Consistency [57], while TSO is the Total Store Order model, made popular by the x86 and SPARC architectures [72], and PSO is a further weakening known as Partial Store Order. We only define their RMW-free fragments here as we only establish hardness results for them that already hold without RMWs. RA and Relaxed refer to the fragments of C11 using release/acquire and relaxed atomics, respectively [53], while SRA (Strong RA) and WRA (Weak RA) are stronger and weaker variants of RA, respectively [51], aka Causal Convergence and Causal Consistency [22].

Consistency. Given a memory model MM and an execution graph $G = \langle E, \mathsf{sb}, \mathsf{rf}, \mathsf{mo} \rangle$, we say that G is *consistent in* MM, denoted $G \models \mathsf{MM}$, if G satisfies the consistency axioms of MM. This notion naturally extends to rf-graphs, i.e., for an rf-graph $G = \langle E, \mathsf{sb}, \mathsf{rf} \rangle$, we have $G \models \mathsf{MM}$ if there exists a modification order mo such that resulting execution graph $G' = \langle E, \mathsf{sb}, \mathsf{rf}, \mathsf{mo} \rangle$ yields $G' \models \mathsf{MM}$. We let $[\![\mathsf{MM}]\!] = \{G = \langle E, \mathsf{sb}, \mathsf{rf}, \mathsf{mo} \rangle : G \text{ is an execution graph and } G \models \mathsf{MM}\}$ and $[\![\mathsf{MM}]\!]^{\mathsf{rf}} = \{G = \langle E, \mathsf{sb}, \mathsf{rf} \rangle : G \text{ is an rf-graph and } G \models \mathsf{MM}\}$ be the sets of MM-consistent execution graphs and rf-graphs, respectively. Note that, although Tid, Reg and Val are fixed, $[\![\mathsf{MM}]\!]^{\mathsf{rf}}$ is infinite in general. Given two memory models MM_1, MM_2,



(a) SC. **(b)** TSO. **(c)** SRA. **(d)** RA. **(e)** Relaxed.

Figure 1 A sequence of increasingly weak execution graphs. Each graph is consistent in its respective memory model and the models to its right, but inconsistent in the models to its left.



(a) Pseq_{t_1} and Pseq_{t_2}. **(b)** $G \in \mathcal{L}(\mathsf{P})$. **(c)** $G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{SC}}$.

Figure 2 A concurrent program $\mathsf{P} = \langle \mathsf{Pseq}_{t_1}, \mathsf{Pseq}_{t_2} \rangle$ (a), an sb-graph in $\mathcal{L}(\mathsf{P})$ (b), and an rf-graph in $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{SC}}$(c).

we say that MM_1 is *stronger than* MM_2, written $\mathsf{MM}_1 \preccurlyeq \mathsf{MM}_2$, if $[\![\mathsf{MM}_1]\!] \subseteq [\![\mathsf{MM}_2]\!]$. For example, $\mathsf{SC} \preccurlyeq \mathsf{TSO} \preccurlyeq \mathsf{SRA} \preccurlyeq \mathsf{RA} \preccurlyeq \{\mathsf{Relaxed}, \mathsf{WRA}\}$. Figure 1 shows examples of execution graphs consistent in different memory models.

2.3 A Model for Programs

Labeled Transition Systems. A *Labeled Transition System* (LTS) is a tuple $\mathcal{A} = \langle \mathcal{Q}, \Sigma, \Delta, \mathsf{init} \rangle$, where (i) $\mathcal{Q}$ is a finite set of states, (ii) Σ is a finite alphabet, (iii) Δ is a (labeled) transition relation over $\mathcal{Q} \times (\Sigma \cup \{\epsilon\}) \times \mathcal{Q}$, and (iv) $\mathsf{init} \in \mathcal{Q}$ is an initial state. We often write $p \xhookrightarrow{\alpha} q$ to denote $(p, \alpha, q) \in \Delta$. The language of $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{\alpha_1 \alpha_2 \dots \alpha_n \in \Sigma^* \colon \mathsf{init} \xhookrightarrow{\alpha_1} p_1 \xhookrightarrow{\alpha_2} \dots \xhookrightarrow{\alpha_n} p_n\}$. For clarity, we may represent a word as a sequence of comma-separated elements of Σ, e.g, $\alpha_1, \alpha_2, \dots, \alpha_n \in \mathcal{L}(\mathcal{A})$.

Sequential programs. Given some thread $t \in \mathsf{Tid}$, the *sequential program* of t is a transition system $\mathsf{Pseq}_t = \langle \mathcal{Q}_t, \Sigma_t, \Delta_t, \mathsf{init}_t \rangle$, where $\Sigma_t = \mathsf{Lab}^t$. Intuitively, each transition of Pseq_t emits the label of an event executed by that thread.

Concurrent programs and their semantics. A *concurrent program* is a collection of sequential programs $\mathsf{P} = \langle \mathsf{Pseq}_{t_1}, \dots, \mathsf{Pseq}_{t_{|\mathsf{Tid}|}} \rangle$. Given an sb-graph G and a thread $t \in \mathsf{Tid}$, let $G|^t_{\mathsf{sb}} = \mathsf{lab}(e_1), \mathsf{lab}(e_2), \dots, \mathsf{lab}(e_n)$, where $e_1, e_2, \dots, e_n$ is the linear sequence of $G.E^t$ according to sb. The *language* of P is defined as $\mathcal{L}(\mathsf{P}) = \{G = \langle E, \mathsf{sb} \rangle \colon G$ is an sb-graph and $\forall t \in \mathsf{Tid}.\ G|^t_{\mathsf{sb}} \in \mathcal{L}(\mathsf{Pseq}_t)\}$. In words, $\mathcal{L}(\mathsf{P})$ contains all sb-graphs in which each thread t executes according to its own sequential program Pseq_t. Given a memory model MM, we define the rf-*semantics* of P as $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}} = \{G = \langle E, \mathsf{sb}, \mathsf{rf} \rangle \colon \langle E, \mathsf{sb} \rangle \in \mathcal{L}(\mathsf{P})$ and $G \models \mathsf{MM}\}$, and the *execution semantics* of P as $[\![\mathsf{P}]\!]_{\mathsf{MM}} = \{G = \langle E, \mathsf{sb}, \mathsf{rf}, \mathsf{mo} \rangle \colon \langle E, \mathsf{sb} \rangle \in \mathcal{L}(\mathsf{P})$ and $G \models \mathsf{MM}\}$. See Figure 2 for an example. Note that $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ and $[\![\mathsf{P}]\!]_{\mathsf{MM}}$ are infinite in general, even though P is finite.

3 The Treewidth of Memory Models

In this section we investigate the treewidth of consistent rf-graphs under different memory models. We prove that SC enjoys bounded-treewidth rf-graphs but TSO already does not. Later, we use treewidth for the verification of concurrent programs under weak memory.

3.1 Treewidth

Treewidth is an elegant graph-theoretic notion that measures how similar a graph is to a tree [70]. Below, we introduce this notion formally.

Tree decompositions and treewidth. Given a (di)graph $G = (V, L)$, where V is a set of vertices and L is a set of edges, a *tree decomposition* of G is a tuple $\langle \{\mathcal{X}_i : i \in I\}, \mathsf{T} = (I, F)\rangle$, where T is a tree and each $\mathcal{X}_i \subseteq V$ is called a *bag*, such that the following conditions are met.

- Every vertex appears in some bag, i.e., $\bigcup_{i\in I} \mathcal{X}_i = V$.
- Every edge appears in some bag, i.e., for all $(u, v) \in L$ there exists some $i \in I$ with $u, v \in \mathcal{X}_i$.
- Every vertex appears in a connected subgraph of T, i.e., for all $i, j, k \in I$, if j is on the path from i to k in T, we have $\mathcal{X}_i \cap \mathcal{X}_k \subseteq \mathcal{X}_j$.

The *width* of a tree decomposition is $\max_{i\in I} |\mathcal{X}_i| - 1$. The *treewidth* of G, written $\mathsf{tw}(G)$, is the smallest width among the tree decompositions of G. A tree decomposition of G is *optimal* if its width equals $\mathsf{tw}(G)$. See [30] for an example.

The treewidth of rf-graphs. The notion of tree decompositions and treewidth naturally extends to rf-graphs $G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle$, where the vertex set is E and the edge set is $\mathsf{sb} \cup \mathsf{rf}$. We will thus be using the above notation for (plain) graphs and rf-graphs interchangeably, and we will write $\mathsf{tw}(G)$ for the treewidth of an rf-graph G. Given a memory model MM, we say that MM *has bounded treewidth* if there exists some $k \in \mathbb{N}^+$ such that for every $G \in [\![\mathsf{MM}]\!]^{\mathsf{rf}}$, we have $\mathsf{tw}(G) \leq k$.

3.2 On The Treewidth of Memory Models

We now study the treewidth of different memory models. Succinctly, our main question is *what is the strongest memory model that gives rise to unbounded treewidth?*

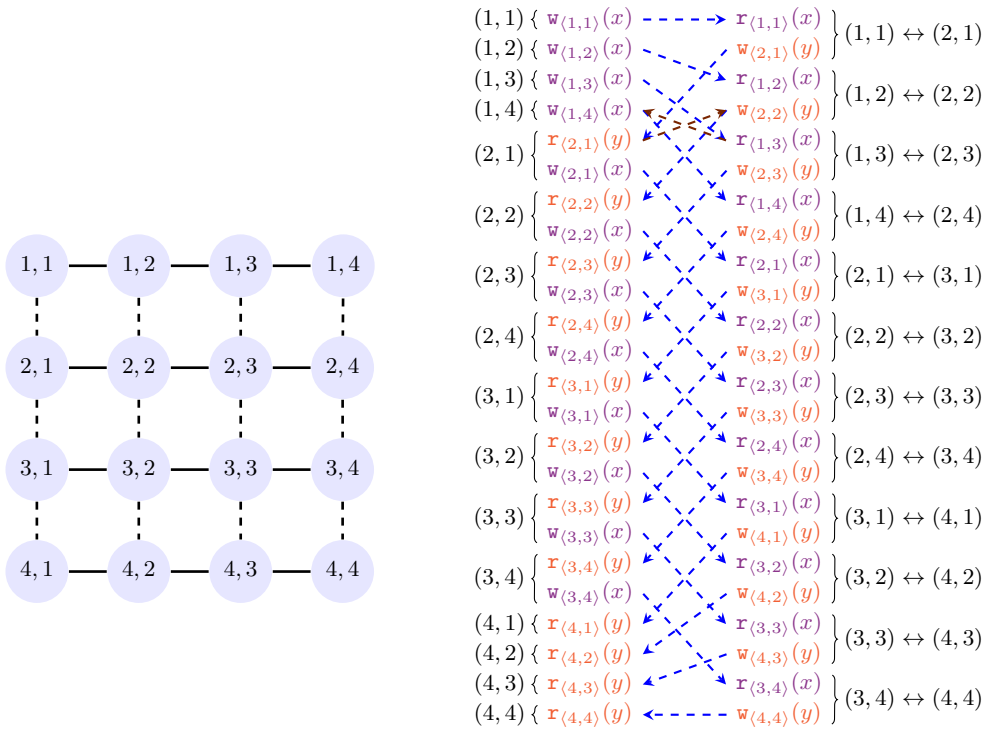
The treewidth of SC. We first show that SC has bounded treewidth.

▶ **Lemma 1.** *For every rf-graph G, if $G \models \mathsf{SC}$ then $\mathsf{tw}(G) \leq |\mathsf{Tid}| + |\mathsf{Reg}|$.*

We remark that this bound is tight up to an additive term $+1$, in the sense that for all values of $|\mathsf{Tid}|$ and $|\mathsf{Reg}|$, there exists an SC-consistent rf-graph of treewidth $\geq |\mathsf{Tid}| + |\mathsf{Reg}| - 1$ (see [30]).

The treewidth of weak-memory. The $|\mathsf{Tid}| + |\mathsf{Reg}|$ bound on the treewidth of SC makes it natural to seek extensions for weak memory models. Clearly, weaker models must admit executions of larger treewidth, that may become unbounded when the model becomes very weak. It turns out that unbounded treewidth arises already in TSO, which might be somewhat surprising given how conceptually similar TSO is to SC [55].

▶ **Lemma 2.** *For every $n > 0$, there exists an rf-graph $G_n \in [\![\mathsf{TSO}]\!]^{\mathsf{rf}}$ of $O(n^2)$ events, 2 threads and 2 registers, such that $\mathsf{tw}(G_n) \geq n$.*



(a) The 4×4 grid.

(b) The rf-graph G_4 that has the 4×4 grid as a minor.

Figure 3 The 4×4 grid is obtained from G_4 by contracting the events that share a subscript into the single vertex carrying that label, and then deleting every remaining edge except those joining vertices adjacent in Figure 3a. Horizontal (filled) edges are encoded by the sb of the left thread, and vertical (dashed) edges are encoded by the sb of the right thread. Note that G_4 is not SC-consistent, due to the cycle $\mathtt{w}_{\langle 1,4\rangle}(x) \xrightarrow{\mathsf{po}} \mathtt{r}_{\langle 2,1\rangle}(y) \xrightarrow{\mathsf{fr}} \mathtt{w}_{\langle 2,2\rangle}(y) \xrightarrow{\mathsf{po}} \mathtt{r}_{\langle 1,3\rangle}(x) \xrightarrow{\mathsf{fr}} \mathtt{w}_{\langle 1,4\rangle}(x)$, with the highlighted fr edges.

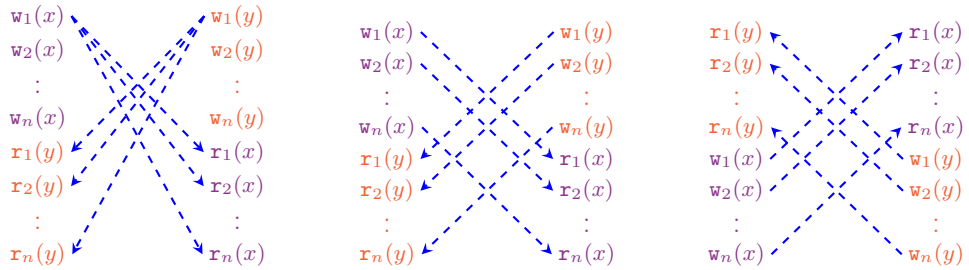
Our proof constructs such a G_n that has an $n \times n$ grid as a minor. Figure 3 illustrates the 4×4 grid and G_4. Since the treewidth of an $n \times n$ grid is n and treewidth is closed under minors [71], this gives $\mathsf{tw}(G_n) \geq n$.

▶ **Corollary 3.** *All memory models of Table 2 except* SC *have unbounded treewidth.*

Weak behaviors of bounded treewidth. In light of Lemma 2, it is natural to ask how large treewidth relates to weak behavior at a conceptual level. By generalizing our construction behind Lemma 1, it can be shown that large treewidth requires very weak behavior, in the following sense.

▶ **Lemma 4.** *For any $k \in \mathbb{N}$, if $\mathsf{tw}(G) \geq |\mathsf{Tid}| + k \cdot |\mathsf{Reg}|$, then* every *total extension σ of $(G.\mathsf{sb} \cup G.\mathsf{rf})^+$ contains an* rf*-edge $e_1 \xrightarrow{\mathsf{rf}} e_2$ such that there are $\geq k$ writes/RMWs on $\mathsf{loc}(e_1)$ between e_1 and e_2 in σ.*

It is worth highlighting, however, that the opposite is *not* true: there are executions with very weak behavior that, nevertheless, enjoy small treewidth. In Figure 4, we show 3 examples of rf-graphs that display weak behaviors but have treewidth bounded by 3. A minimum-width tree decomposition for Figure 4b is shown in Figure 5. A complete discussion of these examples appears in [30].



(a) Store Buffer. **(b)** Extended Store Buffer. **(c)** Extended Load Buffer.

Figure 4 Examples of rf-graphs with very weak behavior but low treewidth.

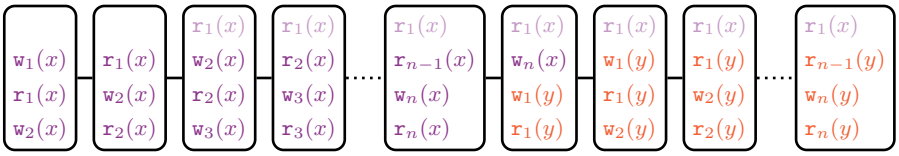


Figure 5 A tree decomposition of the rf-graph of Figure 4b.

Experiments. In order to obtain an early assessment on the applicability of MSO for weak memory in practice, we gathered statistics on the treewidth of rf-graphs of typical benchmarks in the weak-memory literature, taken from the GenMC repository [50]. We find that even under weak memory, the executions of all benchmarks analyzed never exceed their treewidth bound under SC that we prove in Lemma 1. A full description of the experimental analysis is presented in [30].

4 The MSO Axiomatization of Memory Models

In this section we recall some basics on MSO (Section 4.1), prove the MSO-definability of common weak-memory models (Section 4.2), highlight interesting algorithmic implications of this fact (Section 4.3), and show the non-MSO-definability of other common models (Section 4.4).

4.1 The Monadic Second-Order Logic of Concurrent Executions

We begin with MSO adapted to rf-graphs, and recall Courcelle's theorem.

Monadic Second Order Logic. The set of MSO formulas over rf-graphs is given by the grammar

$$\varphi ::= \mathsf{true} \mid e_1 = e_2 \mid \mathsf{lab}(e) = \alpha \mid \mathsf{sb}(e_1, e_2) \mid \mathsf{rf}(e_1, e_2) \mid e \in X \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists e.\varphi \mid \exists X.\varphi$$

where $\alpha \in \mathsf{Lab}$, e, e_1, and e_2 are first-order variables interpreted as events, and X is a second order variable interpreted as a set of events. We also use common shorthand notation such as $\neq$ and $\wedge$, as well as label predicates of the form $\mathsf{op}(x) \in \{\mathtt{w}, \mathtt{rmw}\}$. The semantics of MSO is rather intuitive – we refer to [32] for a formal definition. For instance, if R is some relation expressible in MSO (e.g., sb or rf), the transitive and reflexive-transitive closures of R can be expressed by the following formulas.

$$\varphi_{R^+}(e_1, e_2) \triangleq \neg\exists X.\, (e_2 \notin X \wedge \forall e_3, e_4.\, ((e_1 = e_3 \vee e_3 \in X) \wedge R(e_3, e_4) \implies e_4 \in X)) \quad (1)$$

$$\varphi_{R^*}(e_1, e_2) \triangleq e_1 = e_2 \vee \varphi_{R^+}(e_1, e_2) \quad (2)$$

We call an MSO formula *closed* if it has no free variables. Given a closed MSO formula φ and an rf-graph G, we write $G \models \varphi$ to denote that G satisfies φ, and write $[\![\varphi]\!] = \{G \colon G \models \varphi\}$. A class $\mathcal{C}$ of rf-graphs is *MSO-definable* if there exists an MSO formula φ such that $[\![\varphi]\!] = \mathcal{C}$. Courcelle's celebrated metatheorem [33] gives a fixed-parameter tractable procedure for MSO model checking, which in our setting has the following form.

▶ **Theorem 5** (Courcelle's Theorem). *There exists an algorithm that, given a closed MSO formula φ, a natural number $k \in \mathbb{N}^+$ and a graph $G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle$, verifies whether $\mathsf{tw}(G) \leq k$ and $G \models \varphi$ in time $O(f(|\varphi|, k) \cdot |G|)$, for some computable function f.*

The proof of Theorem 5 also yields the following corollary regarding satisfiability in MSO [34].

▶ **Corollary 6** (Courcelle's Theorem). *Given a closed MSO formula φ and some $k \in \mathbb{N}^+$, the problem of whether there exists an rf-graph $G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle$ with $\mathsf{tw}(G) \leq k$ and $G \models \varphi$ is decidable.*

The set of rf-graphs. Towards our MSO treatment of program semantics under various memory models, we start by establishing that the set of (well-formed) rf-graphs is MSO-definable.

▶ **Lemma 7.** *The set $\{G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle \colon G$ is an rf-graph$\}$ is MSO-definable.*

The free rf-graphs of a program. Given a concurrent program P, we define its language of *free rf-graphs* as $\mathcal{L}^{\mathsf{rf}}(\mathsf{P}) = \{G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle \colon G$ is an rf-graph and $\langle E, \mathsf{sb}\rangle \in \mathcal{L}(\mathsf{P})\}$, i.e., it is the set of rf-graphs which, if we ignore the rf component, are generated by P. Note that here G is not necessarily consistent in some memory model, hence the term "free". The following lemma states that this language is MSO-definable. It is established by conjuncting the MSO formula behind Lemma 7 with an MSO formula Φ_{P} which expresses that $\langle E, \mathsf{sb}\rangle \in \mathcal{L}(\mathsf{P})$, in spirit similar to the Büchi–Elgot–Trakhtenbrot theorem on the MSO definability of regular languages [26].

▶ **Lemma 8.** *For any concurrent program P, the language $\mathcal{L}^{\mathsf{rf}}(\mathsf{P})$ is MSO-definable.*

In light of Lemma 8, for any memory model MM, the rf-semantics $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ are MSO-definable as long as the set $[\![\mathsf{MM}]\!]^{\mathsf{rf}}$ (i.e., the memory model itself) is MSO-definable. Towards this, in the coming sections we study the MSO definability of the memory models of Table 2.

4.2 The MSO-Definability of RA, SRA$^{\setminus \mathtt{rmw}}$, Relaxed, and WRA

Here we prove the following theorem, by establishing the MSO-definability of the consistency axioms of the corresponding memory models.

▶ **Theorem 9.** *For any memory model $\mathsf{MM} \in \{\mathsf{RA}, \mathsf{SRA}^{\setminus \mathtt{rmw}}, \mathsf{Relaxed}, \mathsf{WRA}\}$ and concurrent program P, the set $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ is MSO-definable.*

The case of RA. The MSO formula for RA is a conjunction of three formulas: $\Phi_{(irr-hb)}$, enforcing (irr-hb); $\Phi_{(weak-atomicity)}$, enforcing (weak-atomicity); and $\Phi^{\mathsf{RA}}_{(\text{coherence})}$, enforcing (read-coherence), (write-coherence), and (atomicity) under the assumption that (weak-atomicity) already holds.

Expressing $\Phi_{(irr-hb)}$ and $\Phi_{(weak-atomicity)}$. The first two formulas are somewhat straightforward, as the corresponding axioms do not involve an mo, and can thus be phrased directly on the components of an rf-graph. To define $\Phi_{(irr-hb)}$, we first express that two events e_1, e_2 are such that $e_1 \xrightarrow{\mathsf{hb}} e_2$ in the formula $\varphi_{\mathsf{hb}} = \varphi_{R^+}$, where φ_{R^+} is the transitive-closure formula (Equation (1)) and $R(e_1, e_2) = \mathsf{rf}(e_1, e_2) \vee \mathsf{sb}(e_1, e_2)$. Then

$$\Phi_{(irr-hb)} \triangleq \neg\exists e.\varphi_{\mathsf{hb}}(e, e) \tag{3}$$

To define $\Phi_{(weak-atomicity)}$, we observe that (weak-atomicity) is even first-order definable.

$$\Phi_{(weak-atomicity)} \triangleq \neg\exists e_1, e_2, e_3.(e_2 \neq e_3 \wedge \mathtt{rmw}(e_2) \wedge \mathtt{rmw}(e_3) \wedge \mathsf{rf}(e_1, e_2) \wedge \mathsf{rf}(e_1, e_3)) \tag{4}$$

Expressing $\Phi^{\mathsf{RA}}_{(coherence)}$. We now turn our attention to the coherence and atomicity axioms. These are more challenging because they involve the mo relation, which is not present in an rf-graph, and thus has to be guessed. Our main insight behind the MSO definability of RA is that there exists an MSO-definable, sound and complete partial modification order for RA, that can be used instead of mo. Next, we outline our approach.

An *RMW chain* is a maximal sequence $\mathtt{w}_1 \xrightarrow{\mathsf{rf}} \mathtt{rmw}_2 \xrightarrow{\mathsf{rf}} \mathtt{rmw}_3 \xrightarrow{\mathsf{rf}} \cdots \xrightarrow{\mathsf{rf}} \mathtt{rmw}_m$. Any rf-graph that satisfies (weak-atomicity) (which is expressed independently by $\Phi_{(weak-atomicity)}$ in Equation (4)) guarantees that all its write and RMW events are decomposable into disjoint RMW chains (i.e., every such event appears in exactly one RMW chain). First, we observe that given any RMW event rmw, the (unique) write w heading the RMW chain of rmw can be captured in MSO. This allows us to infer mo-orderings only between write events – and not between RMWs – using MSO, in non-fixed-point style. Second, we prove that the acyclicity of these inferred orderings is nevertheless sufficient and necessary for consistency.

We start with the following (first-order) formula that identifies a conflicting triplet $\langle e_1, e_2, e_3\rangle$ for which (i) $\mathsf{op}(e_1) \in \{\mathtt{w}, \mathtt{rmw}\}$, and (ii) $e_2 \xrightarrow{\mathsf{rf}} e_3$, since this may imply the existence of an mo edge.

$$\varphi_{\text{conf}}(e_1, e_2, e_3) \triangleq e_1 \neq e_2 \wedge e_1 \neq e_3 \wedge \mathsf{loc}(e_1) = \mathsf{loc}(e_2) \wedge \mathsf{op}(e_1) \in \{\mathtt{w}, \mathtt{rmw}\} \wedge \mathsf{rf}(e_2, e_3) \tag{5}$$

The following formula infers mo-orderings between write events e_1, e_2. We infer $e_1 \xrightarrow{\mathsf{mo}} e_2$ if $e_1 \xrightarrow{\mathsf{hb}} e_2$, due to (write-coherence), or there exists a conflicting triplet $\langle e_3, e_4, e_5\rangle$ such that (i) $e_3 \xrightarrow{\mathsf{hb}} e_5$, (ii) e_3 (resp., e_4) is in the RMW chain headed by e_1 (resp., e_2), and (iii) it is not the case that e_3 and e_4 are in the same RMW chain with e_3 preceding e_4, due to (read-coherence) and (atomicity).

$$\begin{aligned}\varphi^{\mathsf{RA}}_{\mathsf{mo}}(e_1,e_2) \triangleq \mathsf{op}(e_1) = \mathtt{w} \wedge \mathsf{op}(e_2) = \mathtt{w} \wedge \mathsf{loc}(e_1) = \mathsf{loc}(e_2) \wedge \big[\varphi_{\mathsf{hb}}(e_1, e_2) \vee \exists e_3, e_4, e_5.\\ \big(\varphi_{\text{conf}}(e_3, e_4, e_5) \wedge \varphi_{\mathsf{hb}}(e_3, e_5) \wedge \neg\varphi_{\mathsf{rf}^+}(e_3, e_4) \wedge \varphi_{\mathsf{rf}^*}(e_1, e_3) \wedge \varphi_{\mathsf{rf}^*}(e_2, e_4)\big)\big]\end{aligned} \tag{6}$$

Finally, we express coherence and atomicity as the requirement that $\varphi^{\mathsf{RA}}_{\mathsf{mo}}$ is acyclic, using the transitive closure formula φ_{R^+} from Equation (1) with $R = \varphi^{\mathsf{RA}}_{\mathsf{mo}}$,

$$\Phi^{\mathsf{RA}}_{(\text{coherence})} \triangleq \neg\exists e.\varphi_{R^+}(e, e) \tag{7}$$

and define

$$\Phi_{\mathsf{RA}} \triangleq \Phi_{(irr-hb)} \wedge \Phi_{(weak-atomicity)} \wedge \Phi^{\mathsf{RA}}_{(\text{coherence})} \tag{8}$$

The following lemma, together with Lemma 7, concludes the MSO-definability of $[\![\mathsf{RA}]\!]^{\mathsf{rf}}$.

▶ **Lemma 10.** *For any rf-graph $G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle$, we have $G \models \Phi_{\mathsf{RA}}$ iff $G \models \mathsf{RA}$.*

The case of RA acts as a template for the other memory models, which admit simpler axiomatizations. We describe these here at a high level, and refer to [30] for their full lemmas and MSO formulas.

The case of Relaxed. The model Relaxed requires only (sc-per-loc). Recall that, for a single register, Relaxed and RA coincide. It thus suffices to tailor the formulas for RA to a single register, and conjoin the resulting per-register formulas, together with $\Phi_{(weak-atomicity)}$, over all $x \in \mathsf{Reg}$.

The case of SRA$^{\setminus \mathtt{rmw}}$. The RMW-free fragment of SRA requires (strong-write-coherence) and (read-coherence). As for RA, we infer mo orderings between writes via an MSO-definable partial order $\varphi^{\mathsf{SRA}}_{\mathsf{mo}}$, here capturing only the orderings due to (read-coherence). This is simpler than $\varphi^{\mathsf{RA}}_{\mathsf{mo}}$ for two reasons: we do not need to infer $\mathtt{w}_1 \xrightarrow{\mathsf{mo}} \mathtt{w}_2$ from $\mathtt{w}_1 \xrightarrow{\mathsf{hb}} \mathtt{w}_2$, as (strong-write-coherence) mixes hb and mo; and, since there are no RMW chains, an ordering $\mathtt{w}_1 \xrightarrow{\mathsf{mo}} \mathtt{w}_2$ arises only from a conflicting triplet $\langle \mathtt{w}_1, \mathtt{w}_2, \mathtt{r} \rangle$ with $\mathtt{w}_1 \xrightarrow{\mathsf{hb}} \mathtt{r}$.

The case of WRA. Recall that we have already defined $\Phi_{(irr-hb)}$ and $\Phi_{(weak-atomicity)}$, respectively expressing (irr-hb) and (weak-atomicity). It remains to express (weak-read-coherence), which is straightforward, as it is directly translatable to MSO.

4.3 Algorithmic Implications

We now state some direct algorithmic implications of Lemma 8 and Theorem 9, by equipping Courcelle's theorem to solve a number of fundamental algorithmic problems concerning the behavior of programs restricted to bounded-treewidth graphs. Given a memory model MM, a concurrent program P, and some $k \in \mathbb{N}^+$, let $[\![\mathsf{P}]\!]^{\mathsf{rf},k}_{\mathsf{MM}} = \{G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}} : \mathsf{tw}(G) \leq k\}$.

Reachability. One of the most fundamental algorithmic questions is that of state reachability: given a concurrent program $\mathsf{P} = \langle \mathsf{Pseq}_{t_1}, \ldots, \mathsf{Pseq}_{t_{|\mathsf{Tid}|}} \rangle$ and a sequence of states $\langle s_1, \ldots, s_{|\mathsf{Tid}|} \rangle$, is there an execution that, for each $i \in [|\mathsf{Tid}|]$, brings Pseq_{t_i} to state s_i? A natural generalization to this question is to decide whether P exhibits an execution that satisfies an arbitrary MSO specification. As the following lemma states, this question is decidable when restricted to graphs of bounded treewidth.

▶ **Lemma 11.** *For any memory model* $\mathsf{MM} \in \{\mathsf{RA}, \mathsf{Relaxed}, \mathsf{SRA}^{\setminus \mathtt{rmw}}, \mathsf{WRA}\}$*, concurrent program* P*, natural number* $k \in \mathbb{N}^+$ *and MSO formula* φ*, the question of whether there exists an* rf*-graph* $G \in [\![\mathsf{P}]\!]^{\mathsf{rf},k}_{\mathsf{MM}}$ *such that* $G \models \varphi$ *is decidable.*

Note that state reachability for RA is undecidable [3], even when restricted to RMW-free executions [31]; furthermore, although state reachability for SRA and WRA is decidable [51], MSO reachability is not, due to the fact that execution graphs under these models include large grid minors (Section 3.2), and MSO satisfiability over grids is undecidable [34].

Verification. Another fundamental problem is that of verification: given a concurrent program P and a specification φ, is it the case that all executions of P satisfy φ? Since MSO is closed under negations, Lemma 11 implies that this problem is decidable for $\mathsf{MM} \in \{\mathsf{RA}, \mathsf{Relaxed}, \mathsf{SRA}^{\setminus \mathtt{rmw}}, \mathsf{WRA}\}$ when φ is an MSO formula and we consider only executions of bounded treewidth.

Robustness and differentiation. Given two memory models MM_1 and MM_2 with $\mathsf{MM}_1 \preccurlyeq \mathsf{MM}_2$, the robustness question of a concurrent program P wrt $(\mathsf{MM}_1, \mathsf{MM}_2)$ asks whether P exhibits executions under MM_2 that are not possible under MM_1. On the other hand, given two concurrent programs P_1 and P_2 under a memory model MM, the differentiation question

asks whether P_1 exhibits behaviors that are not possible under P_2. The MSO definability of memory models implies that robustness and differentiation are decidable over graphs of bounded treewidth.

▶ **Lemma 12.** *For any* $\mathsf{MM}_1, \mathsf{MM}_2 \in \{\mathsf{RA}, \mathsf{Relaxed}, \mathsf{SRA}^{\setminus \mathtt{rmw}}, \mathsf{WRA}\}$*, two concurrent programs* $\mathsf{P}_1, \mathsf{P}_2$*, and natural number* $k \in \mathbb{N}^+$*, the question of whether* $[\![\mathsf{P}_1]\!]^{\mathsf{rf},k}_{\mathsf{MM}_1} \subseteq [\![\mathsf{P}_2]\!]^{\mathsf{rf},k}_{\mathsf{MM}_2}$ *is decidable.*

4.4 Non-MSO-Definability for SC, TSO, PSO and SRA

We now turn our attention to memory models which are not expressible in MSO, under standard beliefs in complexity theory.

Orthogonal vectors. The orthogonal vectors problem (OV) is stated on two sets X, Y, each containing n boolean d-dimensional vectors. A solution to OV is a pair of vectors $\langle x_1, \cdots, x_d \rangle \in X, \langle y_1, \cdots, y_d \rangle \in Y$ that is orthogonal, i.e., $\sum_{i \in [d]} x_i \cdot y_i = 0$. OV is easily solvable in $O(n^2 \cdot d)$ and $O(2^d \cdot n)$ time. The OV hypothesis states that this problem has no $O(n^{2-\epsilon} \cdot d^c)$ algorithm, for any fixed $\epsilon, c > 0$ [24]. We establish the following theorem, which states that a hypothetical MSO definability of the stated memory models would require, not only to break the OV hypothesis, but to actually achieve *nearly-linear* complexity for OV.

▶ **Theorem 13.** *For any memory model* $\mathsf{MM} \in \{\mathsf{SC}^{\setminus \mathtt{rmw}}, \mathsf{SRA}, \mathsf{TSO}^{\setminus \mathtt{rmw}}, \mathsf{PSO}^{\setminus \mathtt{rmw}}\}$*, the set* $[\![\mathsf{MM}]\!]^{\mathsf{rf}}$ *is not MSO-definable unless OV is solvable in* $O(n \cdot d)$ *time.*

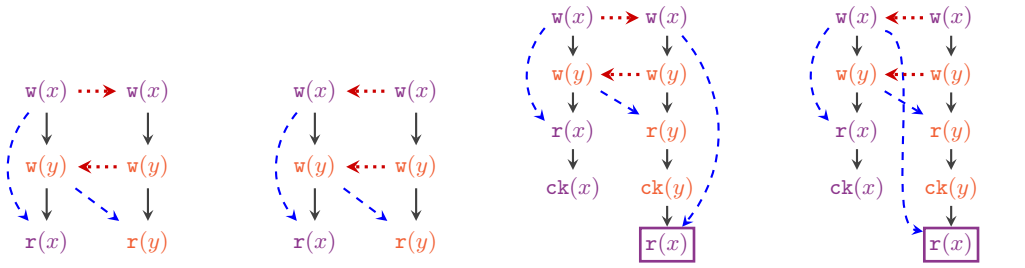
Naturally, the impossibility of Theorem 13 extends to memory models with mixed accesses that subsume any of the above, such as the C++ memory model with SC accesses [15], and the POWER architecture for programs compiled from the release/acquire fragment of C++, since those enjoy SRA semantics [52]. The proof for each case of Theorem 13 is shown in [30].

5 Reads-From Robustness

Robustness against a memory model MM is the property of a concurrent program P of not exhibiting new behaviors under MM compared to SC. A robust program enjoys the best of MM and SC: its weak-memory operations provide efficiency, while its SC-only behaviors makes it amenable to simpler reasoning and analysis methods. Robustness comes in different variants, based on what exactly constitutes program behavior in this context. In this section we show that the treewidth and MSO, as developed in the previous sections, have appealing applications to robustness.

State robustness. One of the least sensitive notions of robustness is wrt to states: P is state robust if every state of P that is reachable via an execution in $[\![\mathsf{P}]\!]_{\mathsf{MM}}$ is also reachable via an execution in $[\![\mathsf{P}]\!]_{\mathsf{SC}}$. State robustness against MM is decidable iff state reachability under MM is decidable [37]. In particular for RA, this implies the undecidability of state robustness [3].

Execution-graph robustness. A concurrent program P is execution-graph robust against a memory model MM if $[\![\mathsf{P}]\!]_{\mathsf{MM}} = [\![\mathsf{P}]\!]_{\mathsf{SC}}$. Execution graph robustness is more sensitive than state robustness, and is decidable against TSO [23] and RA [53].



(a) $G_1 \in [\![\mathsf{P}]\!]_{\mathsf{RA}} \setminus [\![\mathsf{P}]\!]_{\mathsf{SC}}$. **(b)** $G_2 \in [\![\mathsf{P}]\!]_{\mathsf{RA}} \cap [\![\mathsf{P}]\!]_{\mathsf{SC}}$. **(c)** $G_3 \in [\![\mathsf{P}^*]\!]_{\mathsf{RA}} \setminus [\![\mathsf{P}^*]\!]_{\mathsf{SC}}$. **(d)** $G_4 \in [\![\mathsf{P}^*]\!]_{\mathsf{RA}} \cap [\![\mathsf{P}^*]\!]_{\mathsf{SC}}$.

Figure 6 A program P that only produces the execution graphs shown in (a) and (b) is rf-robust, but not execution-graph robust. The augmented program P^* that only produces the execution graphs shown in (c) and (d) is observationally rf-robust, but not execution-graph robust, rf-robust, or observationally execution-graph robust.

Reads-from robustness. In this work, we introduce the notion of reads-from robustness (or rf-robustness). A concurrent program P is rf-robust against a memory model MM if $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}} = [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{SC}}$. It is easy to see that execution-graph robustness implies rf-robustness, and this relationship is strict, in the sense that there exist programs that are rf-robust but not execution-graph robust (see Figures 6a and 6b for an example). Hence, this notion is a step closer to state robustness, by being insensitive to mo.

5.1 Reads-From Robustness and State Reachability

Is rf-robust decidable for common memory models? This is a challenging question, but, perhaps surprisingly, we can establish a conceptually stronger result: for MSO-definable memory models MM that further exhibit a contiguity property, every program P either enjoys decidable state reachability, or it is "obviously" not rf-robust; and this distinction is decidable. Our key insight is that treewidth divergence is both (i) necessary for undecidable reachability/state robustness, due to Lemma 11, and (ii) sufficient for rf-non-robustness, due to Lemma 1. In the following, we make these insights formal.

(sb ∪ rf)-contiguity. We call a memory model MM $(\mathsf{sb} \cup \mathsf{rf})$-*contiguous* if (i) for every rf-graph $G \in [\![\mathsf{MM}]\!]^{\mathsf{rf}}$, $(G.\mathsf{sb} \cup G.\mathsf{rf})$ is acyclic, and (ii) for every program P and $G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$, removing an event e from G that is maximal wrt $(\mathsf{sb} \cup \mathsf{rf})^+$ (together with any adjacent sb- or rf-edges) yields a graph $G' \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$. All memory models in Table 2, except Relaxed, are $(\mathsf{sb} \cup \mathsf{rf})$-contiguous.

Treewidth contiguity. Given a natural number $\ell \in \mathbb{N}^+$, a memory model MM is ℓ-treewidth-contiguous (or ℓ-tw-contiguous) if for every concurrent program P and rf-graph $G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ with $\mathsf{tw}(G) \geq \ell$, there is an rf-graph $G' \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ with $\mathsf{tw}(G) - \ell \leq \mathsf{tw}(G') < \mathsf{tw}(G)$. In words, tw-contiguity excludes large treewidth gaps on the rf-graphs of P under MM. The following lemma follows from the facts that (i) removing a single vertex from a graph decreases its treewidth by at most 1, and (ii) the treewidth of a graph is upper-bounded by its size.

▶ **Lemma 14.** *Every* $(\mathsf{sb} \cup \mathsf{rf})$*-contiguous memory model (and thus all memory models in Table 2 except* Relaxed*) is also* 1*-tw-contiguous.*

Exact-treewidth satisfiability. Consider a concurrent program P over an MSO-definable memory model MM, and some natural number $k \in \mathbb{N}^+$. The question of whether there exists an rf-graph $G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ with $\mathsf{tw}(G) \leq k$ is decidable, as a special case of Theorem 5 where

the corresponding MSO formula is simply true. In order to be able to detect treewidth divergence in P, we need to be able to decide whether $[\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ exhibits a graph with treewidth *exactly* k. We achieve this based on the fact that optimal tree decompositions of graphs can be computed via MSO transductions [18, 19], which can be checked to have width exactly k. Formally, we obtain the following satisfiability lemma as a strengthening of Corollary 6.

▶ **Lemma 15.** *Given an MSO formula φ and some $k \in \mathbb{N}^+$, the problem of whether there exists an* rf*-graph $G = \langle E, \mathsf{sb}, \mathsf{rf}\rangle$ with $\mathsf{tw}(G) = k$ and $G \models \varphi$ is decidable.*

An algorithm for reachability and rf-robustness. We are now ready to establish the algorithmic interplay between reachability and rf-robustness. Let MM be an MSO-definable memory model that is also ℓ-tw-contiguous, for some $\ell \in \mathbb{N}^+$. Consider a concurrent program P, and let $k = |\mathsf{Tid}| + |\mathsf{Reg}|$. For each $m \in \{1, \ldots, \ell\}$, we can decide whether there exists an rf-graph $G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ with $\mathsf{tw}(G) = k + m$, using Lemma 8 and Lemma 15. We distinguish the following cases.

- If the answer is negative for all such m, since MM is ℓ-tw-contiguous, we are guaranteed that $[\![\mathsf{P}]\!]^{\mathsf{rf},k}_{\mathsf{MM}} = [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{MM}}$, i.e., all rf-graphs of P have treewidth $\leq |\mathsf{Tid}| + |\mathsf{Reg}|$. Then, Lemma 11 implies that state reachability (and thus also state robustness) is decidable.
- If the answer is positive for any such m, then the program is not rf-robust, since every rf-graph $G \in [\![\mathsf{P}]\!]^{\mathsf{rf}}_{\mathsf{SC}}$ is such that $\mathsf{tw}(G) \leq k$, due to Lemma 1.

▶ **Lemma 16.** *For any ℓ-tw-contiguous and* MSO*-definable memory model* MM*, there is an algorithm that receives a program* P *as an input and either* (i) *solves state/MSO reachability for* P *under* MM*, or* (ii) *reports that* P *is not* rf*-robust against* MM*.*

In turn, Lemma 16, Lemma 14 and Theorem 9 yield the following theorem.

▶ **Theorem 17.** *For any memory model $\mathsf{MM} \in \{\mathsf{RA}, \mathsf{SRA}^{\setminus \mathtt{rmw}}, \mathsf{WRA}\}$, there is an algorithm that receives a program* P *as an input and either* (i) *solves state/MSO reachability for* P *under* MM*, or* (ii) *reports that* P *is not* rf*-robust against* MM*.*

Theorem 17 is particularly interesting for RA, for which verification is undecidable in general, and also achieves better complexity than state-reachability algorithms for the other models, whose complexity is non-primitive recursive [51].

5.2 Observational Reads-From Robustness

The robustness notions mentioned thus far are sometimes too sensitive. One example is when a program executes speculative reads, followed by a validation step that determines whether these reads obtained obsolete data that should be ignored [17]. Such programs are not even state-robust, yet it is desirable to classify them as robust since the reads responsible for the violation are unused. In order to deal with such spurious violations, execution graph robustness has been relaxed to *observational* robustness, effectively disregarding unused reads from triggering a violation [61]. Here we show that reads-from robustness also supports this type of observational coarsening.

Augmented concurrent programs. In order to avoid introducing heavy extra notation, we make some simplifying assumptions on the structure of programs. In particular, each thread t executes an *augmented LTS* Pseq^*_t where the following hold wlog.

(1) Pseq^*_t starts with a sequence of $\mathtt{w}(t, x, \bot), \mathtt{w}(t, y, \bot), \ldots$ of writes to each shared register, where $\bot$ is a fresh value not written by any other transition.

(2) There is an additional event label $\texttt{error}(t)$, which signifies that t reached an error state.
(3) There is an additional label $\texttt{ck}(t, x)$, which signifies that t is about to use its most recent read on x. Given a sequence of labels $\sigma = \texttt{r}(t, x, v), \alpha_1, \alpha_2, \ldots, \alpha_m$, we say that $\texttt{r}(t, x, v)$ is *unused* in σ if for all $i \in [m]$, if $\alpha_i = \texttt{ck}(t, x)$, then there exists some $j < i$ such that $\mathsf{op}(\alpha_j) \in \{\texttt{r}, \texttt{rmw}\}$ and $\mathsf{loc}(\alpha_j) = x$. Then Pseq_t^* must satisfy the following *invariance under unused reads* property: for every sequence $\sigma_1 \cdot \texttt{r}(t, x, v) \cdot \sigma_2 \in \mathcal{L}(\mathsf{Pseq}_t^*)$, if $\texttt{r}(t, x, v)$ is unused in $\texttt{r}(t, x, v) \cdot \sigma_2$, then for all $v' \in \mathsf{Val}$, we have $\sigma_1 \cdot \texttt{r}(t, x, v') \cdot \sigma_2 \in \mathcal{L}(\mathsf{Pseq}_t^*)$.

At the source-code level, these three conditions can be satisfied by simple, syntactic transformations. The first two are rather trivial. The third condition can be achieved by (i) directing every load on x to a dedicated local register l_x, and (ii) executing a check instruction $\texttt{ck}(t, x)$ right before reading the contents of l_x, or right after a read-modify-write instruction on x (regardless of whether it succeeds). An *augmented concurrent program* is represented as $\mathsf{P}^* = \langle \mathsf{Pseq}_{t_1}^*, \ldots, \mathsf{Pseq}_{t_{|\mathsf{Tid}|}}^* \rangle$.

Augmented graphs. We lift the definitions of execution-/rf-graphs to augmented execution-/rf-graphs naturally, by also considering $\texttt{ck}(t, x)$ and $\texttt{error}(t)$ events. We say that an augmented rf-graph G *reaches* `error` if $\texttt{error}(t) \in G.E$, for some thread t.

Given an augmented graph G, we define the set of its unused reads $\mathsf{UnR}(G)$ in the natural way: a read $\texttt{r}(t, x, v)$ of G is unused if for every check $\texttt{ck}(t, x)$ such that $\texttt{r}(t, x, v) \xrightarrow{\mathsf{sb}^+} \texttt{ck}(t, x)$, there exists a read/RMW event e with $\mathsf{loc}(e) = x$ and $\texttt{r}(t, x, v) \xrightarrow{\mathsf{sb}^+} e \xrightarrow{\mathsf{sb}^+} \texttt{ck}(t, x)$. We say that G and G' are *equivalent*, written $G \simeq G'$, if they are component-wise equal, except for the values and rf-edges of their unused reads. Formally, there is a bijective function $f \colon G.E \to G'.E$ satisfying the following.

(1) For all $e_1, e_2 \in G.E$, we have $(e_1, e_2) \in G.\mathsf{sb}$ iff $(f(e_1), f(e_2)) \in G'.\mathsf{sb}$.
(2) For all $e \in G.E \setminus \mathsf{UnR}(G)$, we have $e = f(e)$ and $f(e) \in G'.E \setminus \mathsf{UnR}(G')$.
(3) For all $e \in \mathsf{UnR}(G)$, we have that $\mathsf{op}(f(e)) = \texttt{r}$ and $\mathsf{loc}(e) = \mathsf{loc}(f(e))$.
(4) For all $e_1, e_2 \in G.E$ with $e_2 \notin \mathsf{UnR}(G)$, we have $(e_1, e_2) \in G.\mathsf{rf}$ iff $(f(e_1), f(e_2)) \in G'.\mathsf{rf}$.

Observational rf-robustness. An augmented program P^* is *observationally rf-robust* against MM if for all $G \in [\![\mathsf{P}^*]\!]^{\mathsf{rf}}_{\mathsf{MM}}$, there exists some $G' \in [\![\mathsf{P}^*]\!]^{\mathsf{rf}}_{\mathsf{SC}}$ s.t. $G \simeq G'$. Note that rf-robustness implies observational rf-robustness, but the latter notion is strictly coarser (see Figure 6). Finally, observational rf-robustness implies robustness against `error` reachability.

▶ **Lemma 18.** *Consider a memory model* MM *and a (augmented) concurrent program* P^* *that is observationally rf-robust against* MM*. If there exists some* $G \in [\![\mathsf{P}^*]\!]^{\mathsf{rf}}_{\mathsf{MM}}$ *reaching* `error`*, then there exists some* $G' \in [\![\mathsf{P}^*]\!]^{\mathsf{rf}}_{\mathsf{SC}}$ *also reaching* `error`*.*

Observational rf-robustness vs reachability. Given Lemma 18, we would like to have a decision procedure for detecting when an augmented program violates observational rf-robustness, analogously to Theorem 17 for rf-robustness. At a first glance, this seems not possible, as observationally rf-robust programs might generate augmented rf-graphs of unbounded treewidth. Nevertheless, we show that this is possible by defining a reduced variant of rf-graphs that has bounded treewidth for all observationally rf-robust programs. The full procedure is detailed in [30]. This allows us to arrive at the following theorem.

▶ **Theorem 19.** *For any memory model* $\mathsf{MM} \in \{\mathsf{RA}, \mathsf{SRA}^{\setminus \texttt{rmw}}, \mathsf{WRA}\}$*, there is an algorithm that receives an augmented program* P^* *as an input and either* (i) *solves* `error` *reachability for* P^* *under* MM*, or* (ii) *reports that* P^* *is not observationally rf-robust against* MM*.*

6 Related Work

Here we discuss related work on the automated analysis of concurrent systems, and connect the results of this paper to existing research directions.

MSO and treewidth. The study of computational models under the lens of treewidth+MSO has been long and fruitful. In [59], it was shown to yield efficient algorithms for automata with access to various types of unbounded storage (e.g., multiple stacks/queues), for which verification is undecidable in general. The executions of multi-stack automata have also been parameterized by splitwidth [36, 35], which is similar to treewidth and also enables efficient decision procedures based on MSO. In a distributed setting, MSO was recently shown capable to capture most common message-passing communication protocols [20, 38], leading to similar decision procedures.

Our work is inspired by aforementioned literature, but our results are novel and based on distinct insights for weak memory. For example, in contrast to message passing, we show that several weak-memory models are not MSO-definable, via novel connections to fine-grained complexity (Section 4.4). Moreover, the features that allow axiomatization in MSO can be subtle (e.g., recall that RMWs break the MSO definability of SRA but not RA), and hint towards a connection to (non) multi-copy atomicity, a notion existing primarily in shared-memory concurrency.

At a technical level, our axiomatization relies on an MSO-definable *partial* modification order, i.e., a subset of mo that can be inferred directly from rf and po, and is sufficient to establish consistency (aka "saturation"). Beyond the models of Section 4.2, a partial mo is known to suffice for Causal Memory [22], as well as for certain weak database isolation levels [16, 63]. We remark, however, that saturability does not always imply MSO-definability; in fact, our proof behind Theorem 13 is over execution families where saturation is sound and complete.

Bounded analyses. The analysis of shared-memory concurrent programs has been a long and burgeoning topic. Typical algorithms perform some sort of bounding of the space of program executions, in order to obtain a tractable, or even decidable, version of the analysis. The most common bounding concerns the length of the executions, enabling stateless model checking algorithms [41, 1, 2, 28, 6, 50, 25, 9]. Another popular type of bounding is on context switches between threads, and has strong practical [69, 13, 64, 4] and complexity-theoretic [40, 29] implications. Although verification under a context bound is typically decidable, this is not the case for programs executing under RA [3], which was amended by the significantly stronger notion of view bounding.

This paper develops this direction further, by proposing treewidth as another dimension of bounding. Bounded-treewidth executions are an appealing approximation mechanism, since (i) they are of unbounded size, (ii) they capture very weak behaviors, even involving arbitrary causality cycles, which are challenging to all existing methods, and (iii) they may be complete in practice, as programs appear to naturally restrict the treewidth of their executions.

Although treewidth and context-bounding are incomparable, executions with bounded view-switching yield rf-graphs of bounded treewidth. A crude treewidth bound of $O(|\mathsf{Tid}||\mathsf{Reg}|+k|\mathsf{Reg}|)$ for an execution with k view switches can be derived from a construction similar to that of Lemma 1.

Robustness. Program robustness has been a long-standing concept in weak-memory concurrency, utilized by both researchers and programmers. The simplest and most common guarantee of robustness has been data-race freedom (DRF) [8, 7, 43], variants of which hold for TSO [66] and the C/C++ memory model [14, 56]. Although DRF implies state robustness, it is generally considered too sensitive, in the sense that simple (state-) robust programs contain races. To this avail, researchers have proposed the coarser notion of execution graph robustness, and proved its decidability against TSO [23, 21] and RA [53]. This direction of ever coarser robustness criteria was recently extended with observational robustness [61] and induced subgraph robustness [65].

The notion of reads-from robustness (together with its observational relaxation) that we introduce in this work constitutes a further step toward coarser robustness criteria. This is the first algorithmically useful robustness notion that reasons purely about program-observable behaviors, disentangling them cleanly from the lower-level architectural details captured in the modification order.

7 Conclusion

We have studied weak-memory concurrency under the lens of treewidth and MSO, giving a systematic account of whether and under what conditions various memory models are MSO-axiomatizable, and characterizing the treewidth of their reads-from semantics. MSO and Courcelle's theorem are usually the first approach to tackling a number of challenging problems, by revealing insights to be exploited by more refined methods. In similar spirit, our work suggests a number of interesting future directions, such as establishing precise complexity bounds for the studied problems (analogously to complexity refinements made for message-passing [35]), possibly exploring different width parameters of rf-graphs, and practical tooling.

References

1 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, January 2014. Association for Computing Machinery. doi:10.1145/2535838.2535845.

2 Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 353–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46681-0_28.

3 Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1117–1132, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314649.

4 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Florian Furbach, and Shashwat Garg. Verification under TSO with an infinite data domain. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 276–295, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57256-2_14.

5 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, S. Krishna, and Viktor Vafeiadis. The decidability of verification under PS 2.0. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 1–29, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-72019-3_1.

6 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal Stateless Model Checking under the Release-Acquire Semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018. doi:10.1145/3276505.

7 S. V. Adve and Mark D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, June 1993. doi:10.1109/71.242161.

8 Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/325164.325100.

9 Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless Model Checking Under a Reads-Value-From Equivalence. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, volume 12759, pages 341–366. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-81685-8_16.

10 Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, October 2012. doi:10.1007/s10703-012-0161-5.

11 Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 405–418, New York, NY, USA, March 2018. Association for Computing Machinery. doi:10.1145/3173162.3177156.

12 Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 7–18, New York, NY, USA, January 2010. Association for Computing Machinery. doi:10.1145/1706299.1706303.

13 Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting rid of store-buffers in TSO analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 99–115, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22110-1_9.

14 Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The Problem of Programming Language Concurrency Semantics. In Jan Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 283–307, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-46669-8_12.

15 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, January 2011. Association for Computing Machinery. doi:10.1145/1926385.1926394.

16 Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360591.

17 Hans-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 12–20, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2247684.2247688.

18 Mikołaj Bojańczyk and Michał Pilipczuk. Definability equals recognizability for graphs of bounded treewidth. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 407–416, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933575.2934508.

19 Mikolaj Bojanczyk and Michal Pilipczuk. Optimizing Tree Decompositions in MSO. In Heribert Vollmer and Brigitte Vallée, editors, *34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, volume 66 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2017.15.

20 Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Etienne Lozes, and Amrita Suresh. A Unifying Framework for Deciding Synchronizability. *LIPIcs, Volume 203, CONCUR 2021*, 203:14:1–14:18, 2021. doi:10.4230/LIPIcs.CONCUR.2021.14.

21 Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and Enforcing Robustness against TSO. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 533–553, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-37036-6_29.

22 Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. *SIGPLAN Not.*, 52(1):626–638, January 2017. doi:10.1145/3093333.3009888.

23 Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11, pages 428–440, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-22012-8_34.

24 Karl Bringmann. Fine-Grained Complexity Theory. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:7, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2019.4.

25 Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. The reads-from equivalence for the TSO and PSO memory models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021. doi:10.1145/3485541.

26 J. Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960. doi:10.1002/malq.19600060105.

27 Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. How Hard Is Weak-Memory Testing? *Proceedings of the ACM on Programming Languages*, 8(POPL):66:1978–66:2009, January 2024. doi:10.1145/3632908.

28 Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, January 2018. doi:10.1145/3158119.

29 Peter Chini, Jonathan Kolberg, Andreas Krebs, Roland Meyer, and Prakash Saivasan. On the Complexity of Bounded Context Switching. *LIPIcs, Volume 87, ESA 2017*, 87:27:1–27:15, 2017. doi:10.4230/LIPIcs.ESA.2017.27.

30 Giovanna Kobus Conrado and Andreas Pavlogiannis. An MSO framework for weak-memory verification and robustness, 2026. arXiv:2606.20134.

31 Giovanna Kobus Conrado and Andreas Pavlogiannis. On the decidability of verification under release/acquire, 2026. doi:10.48550/arXiv.2604.13683.

32 B. Courcelle. *The expression of graph properties and graph transformations in monadic second-order logic*, pages 313–400. World Scientific Publishing Co., Inc., USA, 1997.

33 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

34 Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2012.

35 Aiswarya Cyriac and Paul Gastin. Reasoning About Distributed Systems: WYSIWYG (Invited Talk). *LIPIcs, Volume 29, FSTTCS 2014*, 29:11–30, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.11.

36 Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO Decidability of Multi-Pushdown Systems via Split-Width. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Maciej Koutny, and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory*, volume 7454, pages 547–561. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-32940-1_38.

37 Egor Derevenetc. *Robustness against Relaxed Memory Models*. Doctoral thesis, Technische Universität Kaiserslautern, 2015. URL: `https://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-40743`.

38 Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. A Partial Order View of Message-Passing Communication Models. *Proceedings of the ACM on Programming Languages*, 7(POPL):55:1601–55:1627, January 2023. `doi:10.1145/3571248`.

39 Stephen Dolan, Kc Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–255, Philadelphia PA USA, June 2018. ACM. `doi:10.1145/3192366.3192421`.

40 Javier Esparza, Pierre Ganty, and Tomáš Poch. Pattern-Based Verification for Multithreaded Programs. *ACM Trans. Program. Lang. Syst.*, 36(3):9:1–9:29, September 2014. `doi:10.1145/2629644`.

41 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, January 2005. Association for Computing Machinery. `doi:10.1145/1040305.1040315`.

42 Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. Probabilistic Concurrency Testing for Weak Memory Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 603–616, New York, NY, USA, January 2023. Association for Computing Machinery. `doi:10.1145/3575693.3575729`.

43 Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, 1992. `doi:10.1016/0743-7315(92)90052-D`.

44 Thomas Haas, Roland Meyer, and Hernán Ponce de León. CAAT: Consistency as a theory. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):129:114–129:144, October 2022. `doi:10.1145/3563292`.

45 Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. An axiomatic basis for computer programming on the relaxed Arm-A architecture: The AxSL logic. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. `doi:10.1145/3632863`.

46 Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2017.17`.

47 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 175–189, New York, NY, USA, January 2017. Association for Computing Machinery. `doi:10.1145/3009837.3009850`.

48 Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. Kater: Automating weak memory model metatheory and consistency checking. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. `doi:10.1145/3571212`.

49 Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):49:1–49:28, January 2022. `doi:10.1145/3498711`.

50 Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*, pages 427–440, Berlin, Heidelberg, 2021. Springer-Verlag. `doi:10.1007/978-3-030-81685-8_20`.

51 Ori Lahav and Udi Boker. What's Decidable About Causally Consistent Shared Memory? *ACM Transactions on Programming Languages and Systems*, 44(2):1–55, June 2022. doi:10.1145/3505273.

52 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. *SIGPLAN Not.*, 51(1):649–662, January 2016. doi:10.1145/2914770.2837643.

53 Ori Lahav and Roy Margalit. Robustness against release/acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 126–141, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314604.

54 Ori Lahav and Viktor Vafeiadis. Owicki-Gries Reasoning for Weak Memory Models. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 311–323, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-47666-6_25.

55 Ori Lahav and Viktor Vafeiadis. Explaining Relaxed Memory Models with Program Transformations. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 479–495, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-48989-6_29.

56 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, June 2017. Association for Computing Machinery. doi:10.1145/3062341.3062352.

57 Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979. doi:10.1109/TC.1979.1675439.

58 Weiyu Luo and Brian Demsky. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 630–646, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446711.

59 P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. *ACM SIGPLAN Notices*, 46(1):283–294, January 2011. doi:10.1145/1925844.1926419.

60 Roy Margalit, Michalis Kokologiannakis, Shachar Itzhaky, and Ori Lahav. Dynamic robustness verification against weak memory. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3729277.

61 Roy Margalit and Ori Lahav. Verifying observational robustness against a c11-style memory model. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434285.

62 Evgenii Moiseenko, Matteo Meluzzi, Innokentii Meleshchenko, Ivan Kabashnyi, Anton Podkopaev, and Soham Chakraborty. Relaxed memory concurrency re-executed. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704908.

63 Lasse Møldrup and Andreas Pavlogiannis. AWDIT: An optimal weak database isolation tester. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3742465.

64 Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, June 2007. doi:10.1145/1273442.1250785.

65 Kartik Nagar, Anmol Sahoo, Romit Roy Chowdhury, and Suresh Jagannathan. Automated robustness verification of concurrent data structure libraries against relaxed memory models. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689802.

66 Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-14107-2_23.

67 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290382.

68 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158107.

69 Shaz Qadeer and Jakob Rehof. Context-Bounded Model Checking of Concurrent Software. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Nicolas Halbwachs, and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440, pages 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/978-3-540-31980-1_7.

70 Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.

71 Neil Robertson and P.D Seymour. Graph minors. V. Excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, August 1986. doi:10.1016/0095-8956(86)90030-4.

72 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for X86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. doi:10.1145/1785414.1785443.

73 Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. Optimal reads-from consistency checking for C11-style memory models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591251.

74 Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, Mumbai India, January 2015. ACM. doi:10.1145/2676726.2676995.

75 Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. *SIGPLAN Not.*, 48(10):867–884, October 2013. doi:10.1145/2544173.2509532.

76 John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. *SIGPLAN Not.*, 52(1):190–204, January 2017. doi:10.1145/3093333.3009838.