Parameterized Verification under Release Acquire is PSPACE-complete

Shankaranarayanan Krishna
krishnas@cse.iitb.ac.in
IIT Bombay
Mumbai, India

Adwait Godbole
adwait@berkeley.edu
UC Berkeley
Berkeley, USA

Roland Meyer
roland.meyer@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Soham Chakraborty
s.s.chakraborty@tudelft.nl
TU Delft
Delft, Netherlands

ABSTRACT

We study the safety verification problem for parameterized systems under the release-acquire (RA) semantics. In the non-parameterized setting, access to atomic compare-and-swap (CAS) instructions renders the safety verification problem undecidable. In the light of this result, we consider parameterized systems consisting of an unbounded number of *environment* threads executing identical but CAS-free programs combined with a fixed number of distinguished threads that are unrestricted. Our first contribution is an effective and simplified RA semantics for such systems. We leverage the simplified semantics to show that safety verification becomes PSPACE in the parameterized case, an optimistic result for algorithmic verification. Our proof uses an encoding to Datalog which, in addition to the complexity upper bound, suggests a verification algorithm based on Horn clause solvers. We also provide a matching lower bound showing that safety verification is PSPACE-hard.

CCS CONCEPTS

• Theory of computation → Program verification; **• Software and its engineering → Formal software verification**;

KEYWORDS

Model-checking, Parameterized verification, Shared memory, Weak memory models, Release-Acquire semantics

ACM Reference Format:

Shankaranarayanan Krishna, Adwait Godbole, Roland Meyer, and Soham Chakraborty. 2022. Parameterized Verification under Release Acquire is PSPACE-complete. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC '22), July 25–29, 2022, Salerno, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3519270.3538445

1 INTRODUCTION

Release-acquire (RA) is a popular fragment of C++11 [12] (in which reads are annotated by acquire and writes by release) that strikes a good balance between programmability and performance and has received considerable attention (see e.g., [7, 24, 26, 28, 32, 35, 39–41]). The model is not limited to concurrent programs, though. RA has tight links [33] with causal consistency (CC) [6], a prominent consistency guarantee in distributed databases [36]. In partcular, variants of RA, namely Weak Release-Acquire (WRA) and Strong Release-Acquire (SRA) [32] have been observed to be equivalent to the transactional models of CC and causal convergence (CCv) in the single instruction transaction setting [13, 30, 31]. Our results can be extended in a straightforward manner to these models.

We are interested in the decidability and complexity of safety verification for RA implementations. Common to RA implementations and distributed databases is that they tend to offer functionality to multi-threaded client programs, be it means of synchronization or access to shared data. Clients to such RA implementations all call and execute the same code, and their identity does not have an influence on the functionality they get, an assumption often referred as "indistinguishability". As pointed out by Attiya and Rajsbaum [11], indistinguishability is one of the pillars of computer science and has been the basis for abstraction techniques, lower bounds, and impossibility results. When verifying the RA implementation, the consequence of indistinguishability is that we can abstract the client program to the invocations of the offered functionality [16]. The result is a so-called *instance* of the RA implementation in which concurrent threads execute the code of interest. There is a subtlety. As the RA implementation should be correct for every client, we cannot fix the instance to be verified. We have to prove correctness *irrespective of the number of threads* executing the code. This is the classical formulation of a parameterized system as it has been studied over the last 35 years [16].

To explain the challenges of parameterized verification under RA, it will help to understand how to program under RA. The slogan of RA is *never read "overwritten" values* [33]. Assume we have shared variables x and y, initially 0, and a thread first stores 1 to y and then 1 to x. Assume a second thread reads the 1 from x. Under RA, that thread can no longer read the value 0 from y. Formulated axiomatically [8], the reads-from, modification order, program order, and from-read should be acyclic [33]. While less concise, there are operational formulations of RA that make explicit information about the computation which will be useful for our development [26, 27, 38]. The high-level picture is this. Program and modification order are encoded as natural numbers, called *timestamps*. Each thread stores locally a *view* object, a map from shared variables

This work is licensed under a Creative Commons Attribution International 4.0 License.

PODC '22, July 25–29, 2022, Salerno, Italy.
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9262-4/22/07…$15.00
https://doi.org/10.1145/3519270.3538445

Variables x and y initially 0

producer	consumer
λ_1 : r := y	τ_1 : y := 1
λ_2 : if(r == 1) :	τ_2 : for i in 1..z :
λ_3 : x := 1	τ_3 : s := x
⊕ . . .	τ_4 : assume s = (i %% 1)+1
⊕ x := l	τ_5 : y := 2

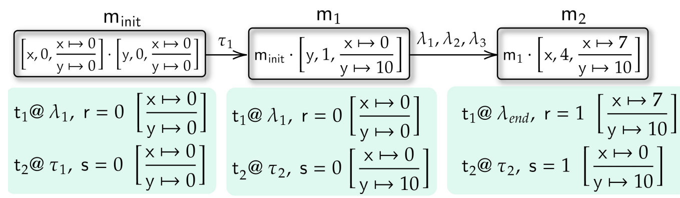

Figure 1: A producer-consumer program (left) and an execution snippet with two threads playing the roles of producer and consumer, respectively (right). We have $z \in \mathbb{N}$, x, y shared variables, r, s local registers, and ⊕ representing non-deterministic choice.

to timestamps. This map reflects the thread's progress in terms of seeing or, as above, hearing from stores to shared variables. The communication is organized in a way that achieves the desired acyclicity. Store instructions generate *messages* that decorate the variable-value pair by a view. This view is the one held by the thread except that the timestamp of the variable being written is raised to a strictly higher value. The shared memory is implemented as a pool to which the generated messages are added and in which they remain forever. When loading a message from the pool, the timestamp of the variable given by the message must be at least the timestamp in the thread. The views are then joined so that the receiver cannot load values older than what the sender has seen.

An Execution under RA. Consider the program in Figure 1. The initial shared memory $\mathsf{m}_{\mathsf{init}}$ consists of two messages, one per variable. The green box below $\mathsf{m}_{\mathsf{init}}$ represents the program counter of each thread (λ, τ), the values of their local registers r resp. s, and their local views. After the store at τ_1, the local view of the consumer is changed to $\frac{x \mapsto 0}{y \mapsto 10}$, by increasing the timestamp of y to some $t \in \mathbb{N}$ (here 10) larger than the current value for y (here 0). Then a message is added to $\mathsf{m}_{\mathsf{init}}$ extending it to m_1. When loading a message from the pool, the timestamp of the variable given by the message must be at least the timestamp of the same variable in the thread's local view. The views are then joined so that the receiver cannot load values older than what the message generator has seen. When Thread 1, the producer, executes λ_1, it loads the message $\left[y, 1, \frac{x \mapsto 0}{y \mapsto 10}\right]$ in m_1, resulting in the local view $\frac{x \mapsto 0}{y \mapsto 10}$. The shared memory is implemented as a pool of messages to which the generated messages are added and in which they remain forever. Continuing on the example, another message is added when Thread 1 executes λ_3 and makes a store for x, generating the memory m_2 (note the increase in the timestamp of x). Let Thread 2 execute the load instruction τ_3 after Thread 1 executed λ_3. The local view of Thread 2 before τ_3 is $\frac{x \mapsto 0}{y \mapsto 10}$. Thread 2 can load the message $\left[x, 0, \frac{x \mapsto 0}{y \mapsto 0}\right]$ from $\mathsf{m}_{\mathsf{init}}$ or the message $\left[x, 4, \frac{x \mapsto 7}{y \mapsto 10}\right]$ from m_2, because the timestamp associated to x in either message is at least as large as Thread 2's view on x. After loading, the local view of Thread 2 will either be $\frac{x \mapsto 0}{y \mapsto 10}$ or $\frac{x \mapsto 7}{y \mapsto 10}$, depending on the load.

The timestamps render the RA semantics infinite-state, which makes algorithmic verification difficult. Indeed, the problem of solving safety verification under RA in a complete way has recently been studied and proven to be undecidable even for programs with finite data domains [1]. Despite considerable efforts [1, 17, 31], the community is missing an expressive class of programs for which the safety verification problem under RA is tractable. We observe that all these works focus on the non-parameterized setting. As argued in the introduction, the parameterized setting is equally common for RA implementations. Yet, for parameterized systems the problem has not been studied at all. We contribute such a study and find that it brings the desired tractability to verification.

Problem Statement. We consider parameterized systems consisting of arbitrarily many *environment* (env) threads executing the same program and an apriori fixed number of *distinguished* (dis) threads executing possibly different programs. Programs are written in a simple while-language Com with the following statements:

$$\mathsf{c} ::= \mathsf{skip} \mid \mathsf{assume}\ \mathsf{e}(\bar{\mathsf{r}}) \mid \mathtt{assert\ false} \mid \mathsf{r} := \mathsf{e}(\bar{\mathsf{r}}) \mid \\ \mathsf{c};\mathsf{c} \mid \mathsf{c} \oplus \mathsf{c} \mid \mathsf{c}^* \mid \mathsf{r} := \mathsf{x} \mid \mathsf{x} := \mathsf{r} \mid \mathsf{cas}(\mathsf{x}, \mathsf{r}_1, \mathsf{r}_2)$$

We obtain an *instance* of the system by fixing the number of env threads. Programs compute on (thread-local) registers r from the finite set Reg using assume, assert, assignments, sequential composition, non-deterministic choice, and iteration. Conditionals if and iteratives while can be derived from these operators, and we use them where convenient. The shared memory is modeled through variables x which are accessed by means of load r := x, store x := r, and compare-and-swap operations $\mathsf{cas}(\mathsf{x}, \mathsf{r}_1, \mathsf{r}_2)$. A cas is a load instruction followed by a store instruction, executed atomically. We have a finite set Var of shared variables, and work with the data domain Dom. We do not insist on a shape of expressions e but require an interpretation $[\![\mathsf{e}]\!] : \mathsf{Dom}^n \to \mathsf{Dom}$ that respects the arity n of the expression. The problem considered is as follows.

Safety Verification for Parameterized Systems:

Given a parameterized system, is there a system instance such that some computation of that instance reaches an assertion violation?

The complexity of the problem depends on the system class under consideration. We denote system classes by signatures of the form env(type) ∥ dis_1(type) ∥ ⋯ ∥ dis_n(type). The "types" constrain the programs executed by the threads, and we consider two restrictions: a loop-free control flow, denoted by acyc, and the instruction set which forbids the atomic compare-and-swap (CAS) command, denoted by nocas. Thus, env(nocas, acyc) ∥ dis(acyc) represents the class of systems in which the arbitrarily many env threads neither have loops nor CAS operations and a single distinguished dis thread executes a loop-free program.

$\|$	$\mathsf{dis}_1(\mathsf{nocas}) \parallel \mathsf{dis}_2(\mathsf{nocas}) \parallel \mathsf{dis}_3 \parallel \mathsf{dis}_4$	$\mathsf{dis}(\mathsf{nocas}) \parallel \mathsf{dis}(\mathsf{nocas})$	$\mathsf{dis}_1(\mathsf{acyc}) \parallel \cdots \parallel \mathsf{dis}_n(\mathsf{acyc})$
env(nocas)	[1]	non primitive recursive [1]	PSPACE-complete (§4,5)
env(acyc)	undecidable (even without dis threads, [22])		

Table 1: Overview of the complexity results. Each entry corresponds to a system class where the type of the environment (env) resp. distinguished (dis) threads is given by the row resp. column. Safety verification is undecidable for classes in red.

We focus on the class env(nocas) $\parallel$ dis_1(acyc) $\parallel \cdots \parallel$ dis_n(acyc), where the env threads execute a CAS-free program (identical for all of them), and the dis threads execute loop-free programs that may contain CAS operations. We forbid CAS operations for the env threads due to the following result of ours: in the presence of CAS, even loop-free env threads are sufficient for undecidability.

Theorem 1.1. *Parameterized safety verification for* env(acyc) *is undecidable.*

To motivate the class of parameterized programs we consider, we look at a number of concurrency benchmarks from the literature [29, 34, 37]. The Phoenix-2.0 benchmarks from Kozyrakis [29] are shared memory concurrent programs that perform data-intensive processing tasks, the programs from Lahav and Margalit [34] are used for robustness analysis, and the benchmarks from [37] are concurrent data structures. To classify the benchmarks in our terms, the programs *peterson-ra-bratosz, rcu* [34], as well as the Phoenix benchmark programs [29] (histogram, kmeans, linear-regression, matrix_multiply, pca, string_match, word_count, sort_pthread) contain a fixed-size loop that can be unrolled and no cas accesses. This means they belong to the class env(nocas, acyc). Likewise, the benchmarks *dekker-fences* [37], *lamport-2-ra, lamport-2-3-ra, peterson-ra* [34] fall into the class env(nocas). Finally, we also have the benchmarks *barrier*, *chase-lev-deque*, and *peterson-ra-bratosz* from [37]. The program *chase-lev-dequeue* contains a loop with a fixed bound which can be unrolled completely and a CAS access which is not within any loop; *barrier* and *peterson-ra-bratosz* contain wait loops (read-till-specific-value). Wait loops can be remodeled as a load followed by an assume statement, and hence these benchmarks fall into the class env(nocas) $\parallel$ dis_1(acyc) $\parallel \cdots \parallel$ dis_n(acyc).

Our Contributions. We list the main contributions of the paper. Table 1 summarizes the landscape of complexity results.

A Simplified Semantics. Our first contribution is a simplified semantics (§3) for parameterized systems of the form env(nocas) that is equivalent with the standard RA semantics as far as safety verification is concerned, and can be seen as an extension of the RA semantics to the parameterized case. The simplified semantics uses the notion of *timestamp abstraction*, which allows us to be imprecise about the timestamps of the env threads. Our simplified semantics is not restricted to the case of having indistinguishable threads, but also works when we allow distinguished threads, without any restrictions.

PSPACE **Upper Bound** As our second contribution, we give a PSPACE-algorithm (§4) for the safety verification problem in the class env(nocas) $\parallel$ dis_1(acyc) $\parallel \cdots \parallel$ dis_n(acyc). This class captures bounded model checking [19] where the distinguished threads are explored up to an under-approximate loop-unrolling bound. Our PSPACE upper bound is obtained by encoding the safety verification problem into the query evaluation problem for linear Datalog, known to be in PSPACE [23]. The linear Datalog format is supported by Horn-clause solvers [14, 15], a state-of-the-art backend in verification.

Lower Bounds. Our third contribution is a matching lower bound for the safety verification problem in the above class. Actually, we provide a stronger lower bound, namely for env(nocas, acyc) which implies that safety verification of env(nocas) $\parallel$ dis_1(acyc) $\parallel \cdots \parallel$ dis_n(acyc) is PSPACE-complete. Additionally, to justify our choice of CAS-free env threads, we prove that safety verification for env(acyc) is undecidable (even for loop-free programs).

Related Work Atig et al. showed that safety verification is decidable for x86-TSO [3, 9]. This has been generalized to models with non-speculative writes [10] and with persistence [2]. These decision procedures rely on well-structuredness arguments [5, 21], often leading to high complexities. Verification for parameterized TSO programs has been considered in [4]. Esparza, et. al. studied the complexity of leader-contributor systems [20]. At the heart of their technique is the so-called copycat-lemma. Our simplified semantics relies on an infinite-supply property which can be thought of as a copycat variant for RA. The verification of concurrent programs under RA in the non-parameterized setting has been studied in [1], where safety verification is shown to be undecidable for programs having four distinguished threads with CAS operations and non-primitive-recursive for systems having two distinguished threads and no CAS operations.

Supplementary material. This paper is accompanied by a full version [22] which contains additional material and proofs.

2 THE RELEASE-ACQUIRE SEMANTICS

A parameterized system consists of an unknown and potentially large number of threads, all running the same program. Threads compute locally over a set of registers and interact with each other by writing to and reading from a shared memory. The interaction with the shared memory is under the Release Acquire (RA) semantics [27, 33, 38]. Below we present the operational semantics of RA [27, 38]. Recall the program syntax from the introduction and that we work with parameterized systems having unboundedly many env threads.

Local Configurations. The RA semantics enforces a total order on all stores to the same variable. We model these total orders by Time $= \mathbb{N}$ and refer to elements of Time as timestamps. Using the total orders, each thread keeps track of its progress in the computation. It maintains a *view* from View $=$ Var $\rightarrow$ Time, that maps each shared variable x to the timestamp of the most recent event the thread has observed on x. The thread keeps track of the program from Com to be executed next (which can in practice be represented as a program counter) , and the register valuation from $\mathsf{RVal} = \mathsf{Reg} \to \mathsf{Dom}$. The set of *thread-local configurations* is thus $\mathsf{LCF} = \mathsf{Com} \times \mathsf{RVal} \times \mathsf{View}$.

Unbounded Threads. The number of threads is not known a priori. Let $\mathsf{TID} = \mathbb{N}$ be the set of thread identifiers. The thread-local configuration map then assigns a local configuration to each thread: $\mathsf{LCFMap} = \mathsf{TID} \to \mathsf{LCF}$.

Views. The views maintained by the threads are used for synchronization. They determine where in the (appropriate) total order a thread can place a store and from which stores it can load a value. To this end, the shared memory holds *messages* - variable-value pairs enriched by a view - of the form $(\mathsf{x}, \mathsf{d}, \mathsf{vw})$: $\mathsf{Msgs} = \mathsf{Var} \times \mathsf{Dom} \times \mathsf{View}$.

Shared Memory. A *memory state* is a set of such messages, and we use $\mathsf{Mem} = 2^{\mathsf{Msgs}}$ for the set of all memory states. With this, the set of all *configurations* of a parameterized system under RA is: $\mathsf{CF} = \mathsf{Mem} \times \mathsf{LCFMap}$.

Transitions. To define the transition relation among configurations, we first give a *thread-local transition relation* among thread-local configurations $\rightharpoonup \subseteq \mathsf{LCF} \times \mathsf{LAB} \times \mathsf{LCF}$ in Figure 2. Thread-local transitions may be labelled with messages when representing interaction with the shared memory (load, store, and CAS): $(\{\mathsf{ld}, \mathsf{st}\} \times \mathsf{Msgs}) \cup (\{\mathsf{cas}\} \times \mathsf{Msgs} \times \mathsf{Msgs})$. Transitions that operate only on the local state of a thread are unlabeled, and referred to as *silent* transitions. The set of possible labels is $\mathsf{LAB} = \{\varepsilon\} \cup (\{\mathsf{ld}, \mathsf{st}\} \times \mathsf{Msgs}) \cup (\{\mathsf{cas}\} \times \mathsf{Msgs} \times \mathsf{Msgs})$. We elaborate on the load, store, and CAS transitions by which a thread with local view vw interacts with the shared memory.

Load. A load transition $\mathsf{r} := \mathsf{x}$ picks a message $(\mathsf{x}, \mathsf{d}, \mathsf{vw}')$ from the shared memory and updates register r with the value d. The message should not be outdated, meaning the timestamp of x in the message, $\mathsf{vw}'(\mathsf{x})$, should be at least the thread's current timestamp for x, $\mathsf{vw}(\mathsf{x})$. The timestamps of other variables do not influence the feasibility of the load transition. They are taken into account, however, when the load is performed. The thread's local view is updated by joining the current view vw and vw′ by taking the maximum timestamp per address; $(\mathsf{vw} \sqcup \mathsf{vw}') = \lambda x. \max(\mathsf{vw}(x), \mathsf{vw}'(x))$.

Store. When a thread executes a store $\mathsf{x} := \mathsf{r}$ it adds a message $(\mathsf{x}, \mathsf{d}, \mathsf{vw}')$ to the memory, where d is the value held by the register r. The new thread-local view (and the message view), vw′, is obtained from the current vw by increasing the time-stamp of x to a fresh timestamp. We use $\mathsf{vw} <_x \mathsf{vw}'$ to mean $\mathsf{vw}(\mathsf{x}) < \mathsf{vw}'(\mathsf{x})$ and $\mathsf{vw}(\mathsf{y}) = \mathsf{vw}'(\mathsf{y})$ for all $\mathsf{y} \neq \mathsf{x}$.

CAS. A CAS transition is a load and store instruction executed atomically. An instruction $\mathsf{cas}(\mathsf{x}, \mathsf{r}_1, \mathsf{r}_2)$ has the intuitive meaning $\mathsf{atomic}\{\mathsf{r} := \mathsf{x};\ \mathsf{assume}\ \mathsf{r} = \mathsf{r}_1;\ \mathsf{x} := \mathsf{r}_2\}$. The instruction loads the shared variable x, checks whether the value matches that of r_1, and, if it does, sets it to the value of r_2. The check and the assignment happen atomically which means the timestamp ts of the load and the timestamp ts′ of the store should be adjacent, $\mathsf{ts}' = \mathsf{ts} + 1$.

The transition relation among configurations $\to \subseteq \mathsf{CF} \times \mathsf{TID} \times (\mathsf{Msgs} \cup \{\varepsilon\}) \times \mathsf{CF}$ is defined in Figure 2. It is labeled by a thread identifier and possibly a message (if the transition interacts with the shared memory). In the case of loads, we require the memory to hold the message to be loaded. In the case of stores, the message to be stored should not conflict with the memory. In the case of CAS, we require both of the above, and that the two messages should have consecutive timestamps. For now, two messages are *non-conflicting* if either they are on different variables or their timestamps are different. We defer a full definition of non-conflict to later where we can give it a broader perspective.

Initial Configuration. Fix a parameterized system c of interest. The initial thread-local configuration is $\mathsf{lcf}_{\mathsf{init}} = (\mathsf{c}, \mathsf{rv}_0, \mathsf{vw}_0)$, where the register valuation assigns $\mathsf{rv}_0(\mathsf{r}) = 0$ to all registers and the view has $\mathsf{vw}_0(\mathsf{x}) = 0$ for all $\mathsf{x} \in \mathsf{Var}$. The *initial configuration* of the parameterized system is $\mathsf{cf}_0 = (\mathsf{Mem}_{\mathsf{init}}, \mathsf{lcfm}_{\mathsf{init}})$. The initial memory $\mathsf{Mem}_{\mathsf{init}}$ holds messages where all shared variables store value $\mathsf{d}_{\mathsf{init}} \in \mathsf{Dom}$ and the view that is constantly zero. The initial thread-local configuration map assigns $\mathsf{lcfm}_{\mathsf{init}}(\mathsf{th}) = \mathsf{lcf}_{\mathsf{init}}$ to all threads. A *computation* (or execution or run) is a finite sequence of consecutive transitions

$$\rho \ = \ \mathsf{cf}_0 \xrightarrow{(\mathsf{th}_1, \mathsf{msg}_1)} \mathsf{cf}_1 \xrightarrow{(\mathsf{th}_2, \mathsf{msg}_2)} \ldots \xrightarrow{(\mathsf{th}_n, \mathsf{msg}_n)} \mathsf{cf}_n.$$

It is initialized if $\mathsf{cf}_0 = \mathsf{cf}_{\mathsf{init}}$. We use $\mathsf{TS}(\rho)$ for the set of all non-zero timestamps that occur in all configurations across all variables. We use $\mathsf{TID}(\rho)$ to refer to the set of thread identifiers labeling the transitions. For a set $\mathsf{TID}' \subseteq \mathsf{TID}$ of thread identifiers, we use $\rho{\downarrow}_{\mathsf{TID}'}$ to project the computation to transitions from the given threads. With $\mathsf{first}(\rho) = \mathsf{cf}_0$ and $\mathsf{last}(\rho) = \mathsf{cf}_n$ we access the first resp. last configurations in the computation.

3 A SIMPLIFIED SEMANTICS

In this section, we propose a simplified semantics for the class of systems $\mathsf{env}(\mathsf{nocas}) \parallel \mathsf{dis}_1 \parallel \cdots \parallel \mathsf{dis}_n$. The key insight behind the simplification is Lemma 3.3 (*Infinite Supply Lemma*) which shows that if some env thread th generates a message $(\mathsf{x}, \mathsf{val}, \mathsf{vw})$ in a computation ρ, then ρ can be extended to a computation where a *clone* of th generates the message $(\mathsf{x}, \mathsf{val}, \mathsf{vw}')$ with $\mathsf{vw}' = \mathsf{vw}[\mathsf{x} \mapsto t]$ for some $t > \mathsf{vw}(\mathsf{x})$. The lemma and hence the simplification result rely on the following assumption: *arbitrarily many* env *threads execute identical, CAS-free programs.*

Making clones of env threads. Let us call a message msg an env message if it is generated in a computation ρ by an env thread, and define dis messages similarly. The fact that the number of env threads is arbitrarily high allows *clone* env threads to duplicate the computation and hence the generated messages. CAS-freeness is crucial here, as it guarantees the duplicated computation to be valid under RA. To ensure that the clone env threads can mimic the env computation in ρ, we require that dis messages can be read by the env clones whenever they can be read by the env threads in ρ. This means that we respect the relative order among timestamps between env and dis threads.

Making space for clones. To accommodate the timestamps of the clone env messages in the extended computation, we create unused timestamps along Time. Clones generate their messages in this unused region via *timestamp lifting* (§3.1). Then, we define how to combine the original computation ρ with that of the clones via an operation called *superposition* (§3.2). Finally, Lemma 3.3 shows how clones can generate messages with arbitrarily higher timestamps.

Timestamp abstraction. Since we can duplicate-at-will the env messages, we need not store the entire set of env messages produced.

$$(\text{ST-LOCAL})\ \frac{\mathsf{rv}(\mathsf{r}) = \mathsf{d} \quad \mathsf{vw} <_{\mathsf{x}} \mathsf{vw}'}{(\mathsf{x} := \mathsf{r}, \mathsf{rv}, \mathsf{vw}) \xrightarrow{\mathsf{st}, (\mathsf{x}, \mathsf{d}, \mathsf{vw}')} (\mathsf{skip}, \mathsf{rv}, \mathsf{vw}')} \qquad (\text{LD-LOCAL})\ \frac{\mathsf{vw}(\mathsf{x}) \le \mathsf{vw}'(\mathsf{x}) \quad \mathsf{rv}' = \mathsf{rv}[\mathsf{r} \mapsto \mathsf{d}]}{(\mathsf{r} := \mathsf{x}, \mathsf{rv}, \mathsf{vw}) \xrightarrow{\mathsf{ld}, (\mathsf{x}, \mathsf{d}, \mathsf{vw}')} (\mathsf{skip}, \mathsf{rv}', \mathsf{vw} \sqcup \mathsf{vw}')}$$

$$(\text{CAS-LOCAL})\ \frac{\mathsf{rv}(\mathsf{r}_1) = \mathsf{d}_1 \quad \mathsf{rv}(\mathsf{r}_2) = \mathsf{d}_2 \quad \mathsf{vw}(\mathsf{x}) \le \mathsf{vw}'(\mathsf{x}) = \mathsf{ts} \quad \widetilde{\mathsf{vw}} = \mathsf{vw}'[\mathsf{x} \mapsto \mathsf{ts} + 1] \quad \mathsf{vw}'' = \mathsf{vw} \sqcup \widetilde{\mathsf{vw}}}{(\mathsf{cas}(\mathsf{x}, \mathsf{r}_1, \mathsf{r}_2), \mathsf{rv}, \mathsf{vw}) \xrightarrow{\mathsf{ld}, (\mathsf{x}, \mathsf{d}_1, \mathsf{vw}')} \xrightarrow{\mathsf{st}, (\mathsf{x}, \mathsf{d}_2, \mathsf{vw}'')} (\mathsf{skip}, \mathsf{rv}, \mathsf{vw}'')}$$

$$(\text{LD-GLOBAL})\ \frac{\mathsf{lcfm}(\mathsf{th}) = \mathsf{lcf} \quad \mathsf{lcf} \xrightarrow{\mathsf{ld}, \mathsf{msg}} \mathsf{lcf}' \quad \mathsf{msg} \in \mathsf{m}}{(\mathsf{m}, \mathsf{lcfm}) \xrightarrow{(\mathsf{th}, \mathsf{msg})} (\mathsf{m}, \mathsf{lcfm}[\mathsf{th} \mapsto \mathsf{lcf}'])} \qquad (\text{ST-GLOBAL})\ \frac{\mathsf{lcfm}(\mathsf{th}) = \mathsf{lcf} \quad \mathsf{lcf} \xrightarrow{\mathsf{st}, \mathsf{msg}} \mathsf{lcf}' \quad \mathsf{msg} \# \mathsf{m}}{(\mathsf{m}, \mathsf{lcfm}) \xrightarrow{(\mathsf{th}, \mathsf{msg})} (\mathsf{m} \cup \{\mathsf{msg}\}, \mathsf{lcfm}[\mathsf{th} \mapsto \mathsf{lcf}'])}$$

$$(\text{CAS-GLOBAL})\ \frac{\mathsf{lcfm}(\mathsf{th}) = \mathsf{lcf} \quad \mathsf{lcf} \xrightarrow{\mathsf{ld}, \mathsf{msg}_l} \xrightarrow{\mathsf{st}, \mathsf{msg}_s} \mathsf{lcf}' \quad \mathsf{msg}_l \in \mathsf{m} \quad \mathsf{msg}_s \# \mathsf{m}}{(\mathsf{m}, \mathsf{lcfm}) \xrightarrow{(\mathsf{th}, \mathsf{msg})} (\mathsf{m} \cup \{\mathsf{msg}_s\}, \mathsf{lcfm}[\mathsf{th} \mapsto \mathsf{lcf}'])} \qquad (\text{UNLABELLED})\ \frac{\mathsf{lcfm}(\mathsf{th}) = \mathsf{lcf} \quad \mathsf{lcf} \rightharpoonup \mathsf{lcf}'}{(\mathsf{m}, \mathsf{lcfm}) \xrightarrow{\mathsf{th}} (\mathsf{m}, \mathsf{lcfm}[\mathsf{th} \mapsto \mathsf{lcf}'])}$$

Figure 2: Shared memory transitions: local transition relation (blue, silent transitions omitted) and global transition relation (green).

Those with the smallest timestamps act as sufficient representatives. Additionally, when a thread reads from an env message, we need not be bothered about timestamp comparisons since we could always generate a clone with as high a (missing) timestamp as required. We capture this notion with timestamp abstraction (§3.4).

3.1 Timestamp Lifting

Timestamp Transformations. In our development, we make use of *timestamp transformations* $\mu : \mathsf{Time} \to \mathsf{Time}$. We extend this to views vw with a collection of *per variable* timestamp transformations $\mathcal{M} = \{\mu^{\mathsf{x}}\}_{\mathsf{x} \in \mathsf{Var}}$, where μ^{x} transforms the timestamps of variable x. The transformed view $\mathcal{M}(\mathsf{vw}) : \mathsf{Var} \to \mathsf{Time}$ is $\lambda \mathsf{x}.\mu^{\mathsf{x}}(\mathsf{vw}(\mathsf{x}))$. We also extend timestamp transformations to messages, memories, configurations, and computations by transforming the view entries.

RA-valid timestamp lifting. A timestamp transformation $\mathcal{M} = \{\mu^{\mathsf{x}}\}_{\mathsf{x} \in \mathsf{Var}}$ is an *RA-valid timestamp lifting* for a computation ρ if it satisfies two properties for each $\mathsf{x} \in \mathsf{Var}$: (1) it is strictly increasing in that for all $t_1, t_2 \in \mathbb{N}$ with $t_1 < t_2$ we have $\mu^{\mathsf{x}}(t_1) < \mu^{\mathsf{x}}(t_2)$ and moreover $\mu^{\mathsf{x}}(0) = 0$, (2) CAS-timestamps remain consecutive in that a CAS operation on x with (load, store) timestamps $(t, t+1)$ leads to $\mu^{\mathsf{x}}(t+1) = \mu^{\mathsf{x}}(t) + 1$, . Note that $\mathcal{M}(\mathsf{cf}_{\mathsf{init}}) = \mathsf{cf}_{\mathsf{init}}$. The following lemma says that the run $\mathcal{M}(\rho)$ obtained by modifying the timestamps of an RA computation ρ with an RA-valid timestamp lifting $\mathcal{M}$ is also an RA computation.

Lemma 3.1 (Timestamp Lifting). *Let $\mathcal{M} = \{\mu^{\mathsf{x}}\}_{\mathsf{x} \in \mathsf{Var}}$ be an RA-valid timestamp lifting. If ρ is an RA computation, then so is $\mathcal{M}(\rho)$. If a configuration* cf *is reachable, so is* $\mathcal{M}(\mathsf{cf})$.

Lemma 3.1 tells us how to make space for clone env threads in a given computation ρ. Next we see how to obtain a new computation by embedding the clone computations in ρ.

3.2 Superposition

We define the *superposition* $\rho \triangleright \rho'$ of two computations ρ, ρ' as the computation that first executes ρ and then ρ', and such that the threads transitioning in ρ resp. ρ' are disjoint. This requires us to combine the memory in $\mathsf{last}(\rho)$ with the memory of every configuration in ρ'. The combination, in turn, requires ρ and ρ' to be *non-conflicting*, which we discuss first.

Conflict. We need a notion of conflict not only for messages as given by the RA semantics, but also for memories, configurations, and computations. Two messages $\mathsf{msg}_1 = (\mathsf{x}_1, \mathsf{d}_1, \mathsf{vw}_1)$ and $\mathsf{msg}_2 = (\mathsf{x}_2, \mathsf{d}_2, \mathsf{vw}_2)$ are *non-conflicting*, denoted by $\mathsf{msg}_1 \# \mathsf{msg}_2$, if either their variables are different, $\mathsf{x}_1 \neq \mathsf{x}_2$, the timestamps are different, $\mathsf{vw}_1(\mathsf{x}_1) \neq \mathsf{vw}_2(\mathsf{x}_2)$, or the timestamps are both zero, $\mathsf{vw}_1(\mathsf{x}_1) = 0 = \mathsf{vw}_2(\mathsf{x}_2)$. Two memory states are non-conflicting, $\mathsf{m}_1 \# \mathsf{m}_2$, if for all $\mathsf{msg}_1 \in \mathsf{m}_1$ and $\mathsf{msg}_2 \in \mathsf{m}_2$, we have $\mathsf{msg}_1 \# \mathsf{msg}_2$. Two configurations are non-conflicting, $\mathsf{cf}_1 \# \mathsf{cf}_2$, if their memory states are non-conflicting. Two computations are non-conflicting, denoted $\rho \# \rho'$, if they use different threads and non-conflicting messages, $\mathsf{TID}(\rho) \cap \mathsf{TID}(\rho') = \emptyset$ and $\mathsf{last}(\rho) \# \mathsf{last}(\rho')$.

The superposition of two non-conflicting computations is

$$\rho \triangleright \rho' \quad = \quad \rho; (\mathsf{last}(\rho) \oplus \rho').$$

We define the addition operation $\oplus$. The addition of a configuration cf to a computation $\rho = \mathsf{cf}_0 \xrightarrow{(\mathsf{th}_1, \mathsf{msg}_1)} \dots \xrightarrow{(\mathsf{th}_n, \mathsf{msg}_n)} \mathsf{cf}_n$ yields the new computation

$$\mathsf{cf} \oplus \rho \ = \ (\mathsf{cf} \oplus \mathsf{cf}_0) \xrightarrow{(\mathsf{th}_1, \mathsf{msg}_1)} \dots \xrightarrow{(\mathsf{th}_n, \mathsf{msg}_n)} (\mathsf{cf} \oplus \mathsf{cf}_n).$$

The addition of configurations $\mathsf{cf}_1 = (\mathsf{m}_1, \mathsf{lcfm}_1), \mathsf{cf}_2 = (\mathsf{m}_2, \mathsf{lcfm}_2)$ is the configuration $\mathsf{cf}_1 \oplus \mathsf{cf}_2 = (\mathsf{m}_1 \cup \mathsf{m}_2, \mathsf{lcfm})$, where $\mathsf{lcfm}(\mathsf{th}) = \mathsf{lcfm}_1(\mathsf{th})$ if $\mathsf{lcfm}_1(\mathsf{th}) \neq \mathsf{lcf}_{\mathsf{init}}$ and $\mathsf{lcfm}(\mathsf{th}) = \mathsf{lcfm}_2(\mathsf{th})$ otherwise. In particular, note that the initial configuration is neutral for addition, that is $\mathsf{cf} \oplus \mathsf{cf}_0 = \mathsf{cf}$. Consequently, when $\rho \# \rho'$ holds and ρ' is initialized, we have, $\mathsf{last}(\rho) = \mathsf{last}(\rho) \oplus \mathsf{first}(\rho') = \mathsf{first}(\mathsf{last}(\rho) \oplus \rho')$.

The concatenation $\rho_1; \rho_2$ expects computations ρ_1 and ρ_2 with $\mathsf{last}(\rho_1) = \mathsf{first}(\rho_2)$ and returns the sequence consisting of the transitions in ρ_1 followed by the transitions in ρ_2. We write $\rho\downarrow_{\mathsf{env}}$ and $\rho\downarrow_{\mathsf{dis}}$ to denote the projections of ρ to env resp. dis. Let $\mathsf{Msgs}(\rho)$ be the memory in $\mathsf{last}(\rho)$, and $\mathsf{Msgs}(\rho\downarrow_{\mathsf{dis}}) \subseteq \mathsf{Msgs}(\rho)$ the subset of messages added by dis threads during ρ. The following lemma shows when superposition leads to a valid computation under RA.

Lemma 3.2 (Superposition). *Consider RA computations ρ, ρ' with $\rho \downarrow_{env} \# \rho' \downarrow_{env}$ and $\mathsf{Msgs}(\rho \downarrow_{dis}) = \mathsf{Msgs}(\rho' \downarrow_{dis})$. Then the superposition $\rho \triangleright (\rho' \downarrow_{env})$ is an RA computation.*

3.3 Infinite Supply Lemma

Let ρ be a computation in which an env message $\mathsf{msg} = (\mathsf{x}, \mathsf{d}, \mathsf{vw})$ is generated. We will show how to duplicate the message. We space out the timestamps of $\mathsf{Msgs}(\rho)$ using timestamp lifting so that we create *holes* (unused timestamps) along Time. Then we generate clones of env threads, denoted by copy(env). The holes are made to accomodate the timestamps of copy(env) and the (higher) timestamp of the copy of msg. We preserve the order of timestamps in copy(env) threads relative to those of dis threads. This ensures that reads-from dependencies between env and dis are maintained.

Define the computation $\widetilde{\rho}$ as a clone of $\rho \downarrow_{env}$ that is executed by copy(env) threads. The write timestamps used by copy(env) threads are the unoccupied timestamps generated by the timestamp lifting operation $\mathcal{M}(\rho)$. We show an example of this via a graphic. Let eT^{i} resp. dT^{i} denote the timestamps chosen by env and dis along ρ (first row).

RA computation ρ: init dT^0 eT^0 dT^1 eT^1 eT^2

Timestamp lifted computation $\mathcal{M}(\rho)$: init dT^0 $\mathsf{eT}^0_{\mathsf{b}}$ $\mathsf{eT}^0_{\mathsf{a}}$ dT^1 $\mathsf{eT}^1_{\mathsf{b}}$ $\mathsf{eT}^1_{\mathsf{a}}$ $\mathsf{eT}^1_{\mathsf{b}}$ $\mathsf{eT}^2_{\mathsf{a}}$

Clone copy($\rho \downarrow_{env}$) computation $\widetilde{\rho}$: init dT^0 $\mathsf{eT}^0_{\mathsf{b}}$ $\mathsf{eT}^0_{\mathsf{a}}$ dT^1 $\mathsf{eT}^1_{\mathsf{b}}$ $\mathsf{eT}^1_{\mathsf{a}}$ $\mathsf{eT}^2_{\mathsf{b}}$ $\mathsf{eT}^2_{\mathsf{a}}$

The second row shows the lifted computation (lifted timestamps have subscript a) $\mathcal{M}(\rho)$ and the holes (faded). The third row shows holes being used by copy(env) for $\widetilde{\rho}$ (subscript b). The construction guarantees $\mathcal{M}(\rho) \# \widetilde{\rho}$ and superposition $\mathcal{M}(\rho) \triangleright \widetilde{\rho}$ is allowed. In this computation, $\widetilde{\rho}$ generates a clone of the message $\mathsf{msg} = (\mathsf{x}, \mathsf{d}, \mathsf{vw})$, namely $\mathsf{msg}' = (\mathsf{x}, \mathsf{d}, \mathsf{vw}')$ with higher $\mathsf{vw}'(\mathsf{x})$. Additionally, since $\mathsf{eT}^i_{\mathsf{a}}, \mathsf{eT}^i_{\mathsf{b}}$ have the same position relative to all dT^j timestamps, so do $\mathsf{vw}(\mathsf{y}), \mathsf{vw}'(\mathsf{y})$ for all variables $\mathsf{y} \neq \mathsf{x}$.

Now we state the Infinite Supply Lemma. As helper notation, for a computation ρ and each variable x, we denote the timestamps of stores of dis threads on x as $\mathsf{ts}^{\mathsf{x}}_0 < \mathsf{ts}^{\mathsf{x}}_1 < \cdots$.

Lemma 3.3 (Infinite Supply). *Let ρ be an RA computation in which an env thread generates the message $(\mathsf{x}, \mathsf{d}, \mathsf{vw})$. For each $t^* \in \mathbb{N}$, there exist timestamp lifting functions $\mathcal{M}_1 = \{\mu^{\mathsf{x}}_1\}_{\mathsf{x} \in \mathsf{Var}}$ and $\mathcal{M}_2 = \{\mu^{\mathsf{x}}_2\}_{\mathsf{x} \in \mathsf{Var}}$, and an RA computation ρ_1 so that*

$$\mathcal{M}_1(\rho) \triangleright \mathcal{M}_2(\rho \downarrow_{env}) \triangleright \rho_1$$

is an RA computation. This computation generates a message $(\mathsf{x}, \mathsf{d}, \mathsf{vw}')$ satisfying (ts comes from ρ)

(1) $\forall i$ $((t^* \leq ts^{\mathsf{x}}_i \wedge \mathsf{vw}(\mathsf{x}) \leq ts^{\mathsf{x}}_i) \implies \mathsf{vw}'(\mathsf{x}) \leq \mu^{\mathsf{x}}_1(ts^{\mathsf{x}}_i))$,
(2) $\forall i, \forall \mathsf{y} \neq \mathsf{x}, \mathsf{vw}(\mathsf{y}) \leq ts^{\mathsf{y}}_i \implies \mathsf{vw}'(\mathsf{y}) \leq \mu^{\mathsf{y}}_1(ts^{\mathsf{y}}_i)$,
(3) $\mathsf{vw}'(\mathsf{x}) \geq \mu^{\mathsf{x}}_2(t^*)$.

To see the lemma, understand $\mathcal{M}_1(\rho)$ as the timestamp lifted computation with holes. Computation $\mathcal{M}_2(\rho \downarrow_{env})$ is the copy(env) run, and ρ_1 is generated by another set of clones that produce the new message (with higher timestamp). We note that run triplication is not strictly necessary for message duplication, but makes the proof easier. Points (1) and (2) in the lemma refer to the relative ordering between env and dis timestamps, (3) refers to the new message having an arbitrarily high x timestamp.

3.4 Abstracting the Timestamps

We introduce the *timestamp abstraction*, the key building block for the simplified semantics. Considering the asymmetry between the dis and env messages, we distinguish the timestamps for the two types of threads.

Timestamp Abstraction. If an env thread has read a message $(\mathsf{x}, \mathsf{d}, \mathsf{vw})$ from a dis thread with timestamp $\mathsf{ts} = \mathsf{vw}(\mathsf{x})$ and has generated a message msg on x, then clones of msg are available with arbitrarily high timestamps at least as high as ts. To capture this in our abstraction, we assign the env message msg a timestamp ts^+ that is by definition larger than ts. We define the set of timestamps in the simplified semantics as $\mathbb{N} \uplus \mathbb{N}^+$, where $\mathbb{N}^+$ contains for each $\mathsf{ts} \in \mathbb{N}$ a timestamp ts^+. The timestamps are equipped with the order $\leq$ in which ts^+ is greater than ts and smaller than $\mathsf{ts} + 1$: $0 < 0^+ < 1 < 1^+ < \ldots$ Timestamps of the form $\mathsf{ts} \in \mathbb{N}$ are used for the stores of dis threads while those of the form ts^+ are used for env threads. We admit multiple stores with the same timestamp ts^+, but at most one store for timestamps of the form ts. This abstracts timestamps of multiple env messages between two dis messages by a single ts^+ timestamp. Initial messages have timestamp 0 as usual.

Simplified Semantics, on an Example. We illustrate the simplified semantics in Figure 3 by parameterizing the program from Figure 1. The formal definition of the simplified semantics can be found in the full version of the paper [22]. The parameterized program has a single dis thread running program consumer, and arbitrarily many env threads running producer. We consider a computation in which dis, and l (out of the unboundedly many) env threads participate. To refer to the different instances of the env threads, we decorate the instruction labels by superscripts from $\{1, \ldots, l\}$.

The consumer thread generates timestamps of the form ts, 1 in the example. The producer threads generate timestamps of the form $\mathsf{ts}^+_1, \ldots, \mathsf{ts}^+_l$. There can be several writes with timestamp ts^+, in particular some ts^+_i may be equal. Additionally, when reading from the producer generated messages, consumer does not perform any timestamp checks, but only updates its view by taking joins. As a result, the load with value 2 during the second loop iteration (i=2) is feasible even if $\mathsf{ts}^+_2 < \mathsf{ts}^+_1$, unlike in the classical RA semantics. Due to the lack of timestamp comparisons, consumer can perform the loop arbitrarily many times ($\mathsf{z} > l$), and the number of env threads needed is independent of z.

The simplified semantics captures in a precise way the reachability problem in the original semantics. Let α_{de} be the function that drops all views from messages and local configurations, and let $=_{\mathsf{de}}$ be the equality of local configurations modulo views.

Theorem 3.4 (Soundness and Completeness). *A configuration cf is reachable in RA iff there is an abstract configuration $\mathsf{cf}^{\mathsf{de}}$ reachable in the simplified semantics so that $\mathsf{cf}^{\mathsf{de}} =_{\mathsf{de}} \alpha_{\mathsf{de}}(\mathsf{cf})$.*

4 PSPACE UPPER BOUND FOR SAFETY VERIFICATION

This section discusses the safety verification problem for the class $\mathsf{env}(\mathsf{nocas}) \parallel \mathsf{dis}_1(\mathsf{acyc}) \parallel \cdots \parallel \mathsf{dis}_n(\mathsf{acyc})$. Assuming a finite data domain Dom, we show that the problem can be solved in PSPACE by leveraging the simplified semantics from Section 3. Our approach

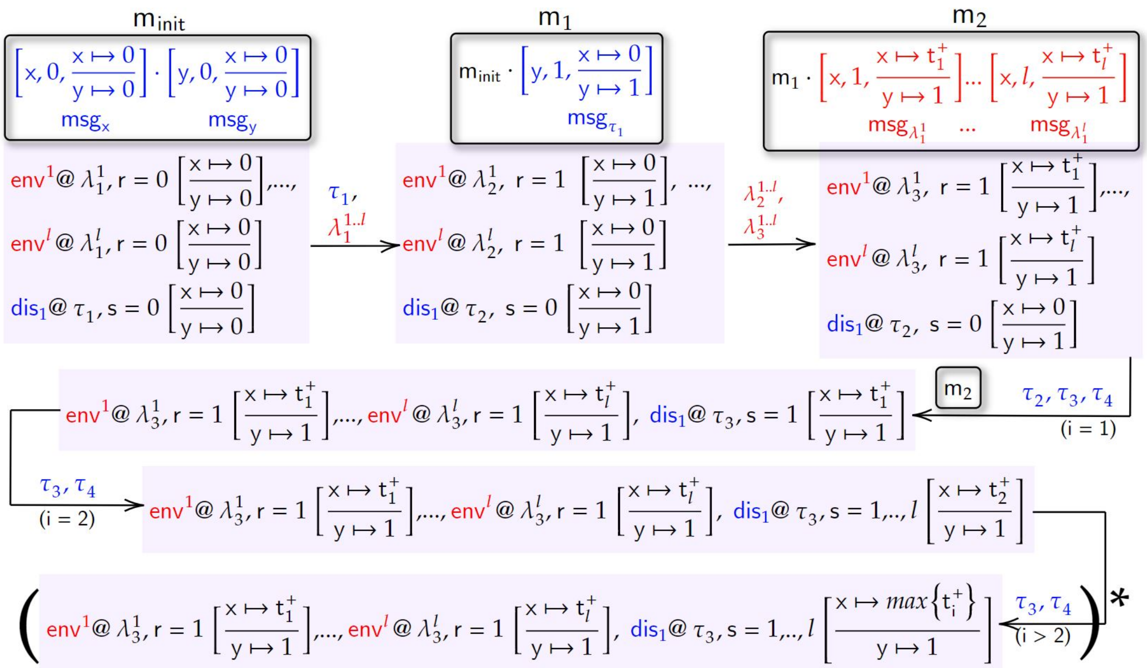

Figure 3: Execution under the simplified semantics, producer transitions and messages are given in red, consumer transitions and messages in blue. The execution begins with the consumer thread generating a message on y with value 1 and timestamp 1 leading to the memory m_1. The producer threads executing $\lambda_1^{1...l}$ read from this message and reach states $\lambda_2^{1...l}$. They generate messages on x with values $\{1, ..., l\}$ shown in memory m_2. These are then read by the consumer as it loops around τ_3, τ_4 for different iterates i, (i=1, i=2, i>2) as shown along the transition.

is to encode the safety verification problem into a Datalog program. The encoding is interesting for two reasons: (1) it yields a complexity upper bound that, given [1], came as a surprise, and (2) it provides practical automated verification opportunities, considering that Datalog-based Horn-clause solvers are state-of-the-art in program verification [14, 15].

Theorem 4.1. *The safety verification problem for* env(nocas) $\|$ dis_1(acyc) $\| \cdots \|$ dis_n(acyc) *is non-deterministic polynomial-time relative to the query evaluation problem in linear Datalog* ($\mathsf{NP}^{\mathsf{PSPACE}}$), *and hence is in* PSPACE.

Linear Datalog is a syntactically restricted variant of Datalog for which query evaluation is in PSPACE. Theorem 4.1 mentions *non-deterministic* polynomial time relative to the linear Datalog oracle. We provide a *non-deterministic* poly-time procedure makeP that converts a given verification instance to a Datalog problem P such that (1) for an unsafe instance, atleast one execution of makeP results in P with successful query evaluation, and (2) for a safe instance, no execution of makeP gives P with successful query evaluation.

The generated Datalog problem P = (Prog, g) consists of (1) a Datalog program Prog and (2) a ground atom g. A Datalog program [18] consists of a predicate set Preds, a data domain Data, and a set of inference rules Rules. An inference rule has the form head :− $\mathsf{body}_1, \ldots, \mathsf{body}_t$, where head and body_i are positive literals. A rule with one literal in the body is a linear rule, one without a body is called a fact. A linear Datalog program is one where all rules are linear or facts.

An instantiation of a rule is the result of replacing each occurrence of a variable in the rule by a constant, and a ground atom is a predicate in which all terms are constants. For every instantiation of a rule, if all ground atoms constituting the body are true then the ground atom in the head can be inferred to be true. The *query evaluation problem* for Datalog is, given a *problem instance* (Prog, g) as above, determine whether Prog $\vdash$ g, meaning we can infer the atom g from the program Prog using the given inference rules. The combined complexity (in terms of Prog and g as input) of query evaluation [23] for linear Datalog is PSPACE, while non-linear rules raise it to NEXPTIME [25, 42]. We do not directly reduce safety verification to query evaluation in linear Datalog, but instead use an intermediate notion of Cache Datalog. We proceed as follows.

(1) For ease of encoding, we introduce Cache Datalog, Datalog with an additional parameter, the Cache, that is decisive in controlling the complexity of encodings as follows: every Cache Datalog program can be turned into a linear Datalog program at a cost that is linear in the size of the program and the Cache (Lemma 4.2);
(2) makeP generates Cache Datalog programs Prog and a query instance (Prog, g) such that Prog $\vdash$ g iff the given verification instance is unsafe, thereby constituting a correct reduction (Lemma 4.3). Further, a Cache of polynomial size is sufficient for query evaluation (Lemma 4.4).

Cache **Datalog**. A Cache is a set of ground atoms that is used to control the inference process. In the presence of a Cache, the semantics of Datalog is adapted by the following two rules.

Add: For an instantiated rule, the ground atom in the head can be inferred and added to Cache only when all the ground atoms in the body are in Cache.

Drop: Atoms in Cache can be dropped non-deterministically.

The standard semantics of Datalog can be seen in Cache Datalog by monotonically adding all inferred atoms (starting with facts) to the Cache and never dropping anything. To show the PSPACE upper bound, we use a notion of inference that bounds the size of the Cache. For a Cache Datalog program Prog and $k \in \mathbb{N}$, we write Prog $\vdash_k$ g to mean that ground atom g can be inferred from Prog with a computation in which $|\text{Cache}| \leq k$.

Lemma 4.2. *Given* Cache *Datalog program* Prog*, ground atom* g*, and bound* k*, in time quadratic in* $|\text{Prog}| + |g| + k$ *we can construct a linear Datalog program* Prog′ *so that* Prog $\vdash_k$ g *iff* Prog′ $\vdash$ g.

4.1 Datalog Encoding

Theorem 3.4 tells us that safety verification under RA is equivalent to safety verification in the simplified semantics. Safety verification in the simplified semantics, in turn, can be reduced to the following *Message Generation (MG)* problem.

Message Generation (MG):

> Given system c and *goal message* $\text{msg}^\# = (\text{x}^*, \text{d}^*, \_)$, is there a reachable configuration $\text{cf}^{\text{de}} = (\text{m}^{\text{de}}, \text{lcfm}^{\text{de}})$ with $\text{msg}^\# \in \text{m}^{\text{de}}$ (for some vw^{de})?

To see the connection between MG and safety verification, note that we can replace each assert `false` statement in the program by $\text{x}^* := \text{d}^*$ for variable x^* and value d^* unused elsewhere. The system is unsafe if and only if a goal message $\text{msg}^\# = (\text{x}^*, \text{d}^*, \text{vw}^{\text{de}})$ is generated for some vw^{de}.

While encoding into Datalog, we non-deterministically guess vw^{de}. For this, we crucially show that there are only exponentially-many choices of vw^{de}. Given c, $\text{msg}^\#$, our non-deterministic poly-time procedure makeP satisfies the following.

Lemma 4.3. *Given parametrized system* c *and goal message* $\text{msg}^\#$*, Message Generation holds iff there is an execution of* makeP *that generates a query instance* (Prog, g) *with* Prog $\vdash$ g*. The construction of* Prog *and* g *is in (non-deterministic) time polynomial in* |c|.

The procedure makeP generates one query instance (Prog, g) per execution. Here, we give the intuition, the details of makeP can be found in the full version [22]. Since the parameterized system consists of n loop-free dis threads, each can execute only linearly-many instructions in their size. The total number of instructions executed (and so the number of timestamps used) by the dis threads is thus polynomial in the combined size of the dis programs c^i_{dis}. Let this bound be T. Then we have the timestamps $\{0, 0^+, \cdots, T, T^+\}$, and this number of timestamps forms the crux of the polynomial bound in Lemma 4.3. Procedure makeP guesses the dis threads' part of the computation when generating a query instance.

Program Prog uses four predicates. The environment message predicate $\text{emp}(\text{x}, \text{d}, \text{vw}^{\text{de}})$ represents an available env message on variable x with value d and view vw^{de}. The environment thread predicate $\text{etp}(\text{lc}, \text{rv}, \text{vw}^{\text{de}})$ encodes the env thread configuration, where lc is the control state, rv the register valuation, and vw^{de} the thread view. We have similar message and thread predicates for the dis threads. The distinguished message predicate $\text{dmp}(\text{x}, \text{d}, \text{vw}^{\text{de}})$ represents an available dis message. Additionally, for each dis thread i, we have a distinguished thread predicate $\text{dtp}_i(\text{lc}, \text{rv}, \text{vw}^{\text{de}})$ that encodes the configuration of the thread dis_i.

As rules, we have the fact $\text{dmp}(\text{x}, \text{d}_{\text{init}}, \text{vw}^{\text{de}}_{\text{init}})$ for each variable x with d_{init} the initial value and $\text{vw}^{\text{de}}_{\text{init}}$ the initial view. We moreover have the facts $\text{etp}(\lambda_{\text{init}}, \text{rv}_{\text{init}}, \text{vw}^{\text{de}}_{\text{init}})$ and $\text{dtp}_i(\lambda_{\text{init}}, \text{rv}_{\text{init}}, \text{vw}^{\text{de}}_{\text{init}})$ that represent the initial states of the env resp. dis threads. We also have rules corresponding to the env transitions and the guessed dis thread run fragments. Finally, the query atom g is a ground atom of the form emp or dmp capturing the goal message $\text{msg}^\#$. The instances generated in the non-deterministic branches of makeP differ only in the guessed dis run and in the atom g.

4.2 Cache Size

With the encoding at hand, the challenge is to establish a polynomial bound on the cache size for the query instances generated by makeP. Let $Q_0 = |\text{Dom}||\text{Var}| + |\text{dis}|$ where $|\text{dis}|$ is the combined size of all dis threads. A Cache of size $O(Q_0^2)$ is sufficient to infer g.

Lemma 4.4. *For each* (Prog, g) *generated by* makeP*,* Prog $\vdash$ g *if and only if* Prog $\vdash_k$ g *with* $k \in O(Q_0^2)$.

To see that the above size of Cache is sufficient, we analyze the structure of computations in the simplified semantics. The analysis will reveal a dependency relation among the generated messages. This dependency relation will give enough information to guide the Datalog computation so as to use a small Cache.

Consider computation ρ^{de} ending in configuration $\text{last}(\rho^{\text{de}}) = (\text{m}^{\text{de}}, \text{lcfm}^{\text{de}})$. For every message msg^{de} in memory m^{de}, we use $\text{genthread}(\text{msg}^{\text{de}})$ for the first thread which added msg^{de} to m^{de}. (Recall that the simplified semantics admits the repeated insertion of env messages due to the reuse of timestamps from $\mathbb{N}^+$). We define $\text{depend}(\text{msg}^{\text{de}})$ as the set of messages which $\text{genthread}(\text{msg}^{\text{de}})$ has read before generating the first instance of msg^{de}. Further below, we will also need the *read-count* $\text{rc}(\text{msg}^{\text{de}}, \text{msg}') \in \mathbb{N}$, the number of times $\text{genthread}(\text{msg}^{\text{de}})$ reads $\text{msg}' \in \text{depend}(\text{msg}^{\text{de}})$ before generating msg^{de}.

Definition 1. *The dependency graph of a computation* ρ^{de} *with* $\text{last}(\rho^{\text{de}}) = (\text{m}^{\text{de}}, \text{lcfm}^{\text{de}})$ *is the directed graph* $G_{\rho^{\text{de}}} = (V, E)$ *with* $V = \text{m}^{\text{de}}$ *and* $E = \text{depend}$*, the vertices are the messages and we have an edge* $(\text{msg}^{\text{de}}_1, \text{msg}^{\text{de}}_2) \in E$ *if* $\text{msg}^{\text{de}}_1 \in \text{depend}(\text{msg}^{\text{de}}_2)$.

As depend(−) is based on the linear order of the computation, the dependency graph is acyclic. We denote the sets of sink and source vertices of G by $\text{sink}(G)$ resp. $\text{source}(G)$. A path in G is also called a *dependency sequence*. The height of a vertex v is the length of a longest path from a source vertex to v. The maximal height over all vertices is $\text{height}(G)$. See Figure 4 for an example.

Compact Computations. Unfortunately, dependency graphs may contain exponentially many vertices (due to the views), and given

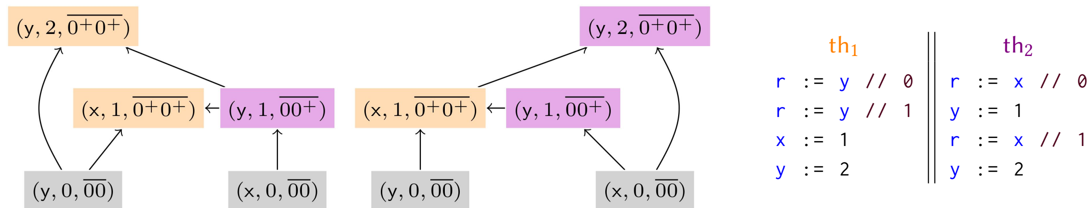

Figure 4: Two possible dependency graphs for the code snippet. Both th_1 **and** th_2 **are env threads. The color of each message** msg **gives** genthread(msg)**, with** th_1 **being orange,** th_2 **violet, and** init **gray. We denote the view as a vector** $\overline{t_x t_y}$**. Since we only consider the thread adding a message for the first time** genthread(y, 2, $\overline{0^+0^+}$) **can be either** th_1 **(left graph) or** th_2 **(right graph).**

the PSPACE-hardness there is no way to reduce this to polynomial size. Yet, there are two parameters that we can reduce, the fan-in of each vertex v, the number of messages read by genthread(v) before generating v, and and the height of the dependency graph. We call a computation ρ^{de} *compact* if its dependency graph $G_{\rho^{\mathsf{de}}}$ satisfies the following two bounds. (1) Every message v depends on a small number of other messages, $|\mathrm{depend}(v)| \leq Q_0$. (2) The dependency sequences are polynomially long, that is, $\mathrm{height}(G_{\rho^{\mathsf{de}}}) \leq Q_0$. If a vertex/message msg in the dependency graph has fan-in $> Q_0$, then, thanks to the simplified semantics, genthread(msg) can read from an earlier message with the same variable/value pair. Likewise, if the dependency sequence is longer than Q_0, then it will contain two messages with the same variable and value. The segment of the sequence between these two can be truncated without affecting the remainder of the computation. The following lemma says that compact computations are sufficient:

Lemma 4.5. *Any message that can be generated in the simplified semantics can be generated by a compact computation.*

In Cache Datalog, the inference of an atom g from a program Prog involves a sequence of applications of the Add (to Cache) and Drop (from Cache) rules that ends with g being inferred. Such a sequence for Prog $\vdash$ g corresponds to a run ρ^{de} under the simplified RA semantics. The run ρ^{de} can be compacted to $\rho^{\mathsf{de}\prime}$ with Lemma 4.5. From the dependency graph of $\rho^{\mathsf{de}\prime}$ we can read off an inference strategy that keeps the Cache size polynomial in |Var|, |Dom|, and $|\mathsf{c}_{\mathsf{dis}}|$. The following lemma formalizes the argument and concludes the proof of Lemma 4.4.

Lemma 4.6 (Datalog Inference Strategy). *Let* makeP *generate the query instance* (Prog, g)*. The inference for* Prog $\vdash$ g *implies the existence of an execution* ρ^{de} *under the simplified semantics, which can be compacted to* $\rho^{\mathsf{de}\prime}$*. The computation* $\rho^{\mathsf{de}\prime}$ *can be mapped back to a new inference sequence such that* Prog $\vdash_k$ g *for* $k \in O(Q_0^2)$.

4.3 Quantifying the number of env threads to generate msg#

While parameterization is useful to model systems with an apriori unknown number of components, for (non-parameterized) systems with a large, but *fixed* number of components, parameterization is *sound but not complete*. That is, a bug in the non-parameterized system implies that it will be detected in the parameterized version of the system, however, the converse is not necessarily true.

We now determine a concrete value at which parameterization becomes complete. That is, if a non-parameterized system has at least this number of env threads, then there is a bug in the non-parameterized system iff there is a bug in the corresponding parameterized variant. In general, the bound can be doubly exponential in the system parameters |Var|, |Dom|, |dis|. However, for certain programs, it can be much lower, reducing the gap with which parameterization over-approximates a non-parameterized system.

Attributing costs to nodes. We attribute costs to nodes in the dependency graph via the function cost : $\mathsf{m}^{\mathsf{de}} \rightarrow \mathbb{N}$. Intuitively, the cost of a message corresponds to the number of env threads required for generating the message. For an initial message, cost(msg) = 0. For an env message,

$$\mathrm{cost}(\mathsf{msg}) = 1 + \sum_{\mathsf{msg}' \in \mathsf{m}^{\mathsf{de}}\downarrow_{\mathsf{env}}} \mathrm{rc}(\mathsf{msg}, \mathsf{msg}') \cdot \mathrm{cost}(\mathsf{msg}').$$

For a dis message,

$$\mathrm{cost}(\mathsf{msg}) = \sum_{\mathsf{msg}' \in \mathsf{m}^{\mathsf{de}}\downarrow_{\mathsf{env}}} \mathrm{rc}(\mathsf{msg}, \mathsf{msg}') \cdot \mathrm{cost}(\mathsf{msg}').$$

For a dependency graph G which generates the goal message $\mathsf{msg}^\#$, the cost of the graph is defined as $\mathrm{cost}(G) = \mathrm{cost}(\mathsf{msg}^\#)$.

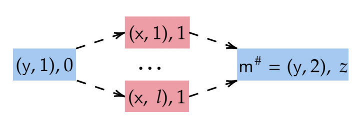

Figure 5: Cost annotated dependency graph for the producer-consumer example ($z \in \mathbb{N}$**, the cost of the** $\mathsf{msg}^\#$ **message is the loop-bound for the consumer).**

Consider the producer-consumer example in Figure 1. We are interested the reachability of τ_5. Figure 5 shows the dependency graph with the costs added to the nodes. We have $\mathrm{cost}(G) = z$, the cost of the target message $\mathsf{msg}^\# = (\mathsf{y}, 2)$ is the loop-iteration count of the consumer. Note that we have modeled the consumer as dis thread and the producers as env threads. The cost shows that z-many env threads are sufficient to generate message $\mathsf{msg}^\#$. However, in reality, l env threads suffice, and hence the cost is an over-approximate bound.

5 PSPACE-HARDNESS OF env(nocas, acyc)

We show that the semantic simplification we have given is tight, and further simplification is not possible. Having shown that safety verification of env(nocas) ∥ dis_1(acyc) ∥ ⋯ ∥ dis_n(acyc) is in PSPACE, we now give a matching lower bound. For the lower bound, it suffices to consider the variant without dis threads and with only loop-free env threads, env(nocas, acyc). Even more, the result refers to Parameterized RA in its simplest form, called PureRA, in which (1) registers are forbidden and (2) stores can only write value one to a memory that is initially zero. PureRA eliminates thread-local computations and lays bare the complexity inherent to reasoning *purely about the synchronization* possible in RA. Suprisingly, the problem is PSPACE-hard even for this restricted form. Note that PSPACE-hardness in the presence of local registers is trivial, since PSPACE-computations can be encoded with register valuations.

$$\begin{aligned}
c_{\text{env}} &= c_{\text{AG}} \oplus c_{\text{SATC}} \oplus c_{\text{FE[0]}} \oplus \cdots \oplus c_{\text{FE[n-1]}} \oplus c_{\text{assert}} \\
c_{\text{AG}} &= \text{pick}(u_0); \text{pick}(e_1); \text{pick}(u_1); \cdots; \text{pick}(u_n); s := 1 \\
&\qquad \text{where pick}(u) = (t_u := 0) \oplus (f_u := 0) \\
c_{\text{SATC}} &= \text{assume } (s = 1); \text{check}(\Phi); \\
&\quad ((\text{assume } (t_{u_n} = 0); a_{n,1} := 1;) \oplus \\
&\quad (\text{assume } (f_{u_n} = 0); a_{n,0} := 1)) \\
c_{\text{FE[i]}} &= \text{assume } (a_{i+1,0} = 1); \text{assume } (a_{i+1,1} = 1); \\
&\quad (\text{assume } (f_{e_{i+1}} = 0) \oplus \text{assume } (t_{e_{i+1}} = 0)); \\
&\quad ((\text{assume } (t_{u_i} = 0); a_{i,1} := 1) \oplus \\
&\quad (\text{assume } (f_{u_i} = 0); a_{i,0} := 1)) \\
c_{\text{assert}} &= \text{assume } (a_{0,0} = 1); \text{assume } (a_{0,1} = 1); \text{assert } \texttt{false}
\end{aligned}$$

Figure 6: Program c_{env} executed by the env threads is a non-deterministic choice between functions c_{AG}, c_{SAT}, $c_{\text{FE[i]}}$, and c_{assert}.

We prove the lower bound by a reduction from the canonical PSPACE-complete problem, TQBF, described as follows. Given a Quantified Boolean Formula

$$\Psi \quad = \quad \forall u_0 \; \exists e_1 \; \forall u_1 \; \cdots \; \exists e_n \; \forall u_n \; \Phi(u_0, e_1, \ldots, u_n)$$

over variables $Vars(\Psi) = \{u_0, \ldots, u_n, e_1, \ldots, e_n\}$, decide whether Ψ is true. Formula Ψ has $n+1$ universally and n existentially quantified variables. Given a TQBF instance Ψ, we construct an instance of the parametrized safety verification problem for PureRA consisting of the program c_{env} (only env threads), such that c_{env} is unsafe *iff* the TQBF instance is true. Assuming the TQBF instance is Ψ from above, the program c_{env} consists of functions (sub-programs), one of which may be executed non-deterministically. The task of checking whether Ψ holds is distributed over the env threads executing these functions. Each function has a particular role which we now describe.

- c_{AG}: The Assignment Guesser guesses a possible satisfying assignment for $Vars(\Psi)$.

- c_{SATC}: The Satisfiability Checker checks satisfiability of Φ w.r.t. an assignment guessed by c_{AG}.

- $c_{\text{FE[i]}}$: The ∀∃ (ForallExists) Checker at level $0 \leq i \leq n-1$ verifies that the $(i+1)$th quantifier alternation $\forall u_i \exists e_{i+1}$ is respected by the guessed assignments. This proceeds in levels, where the function $c_{\text{FE[i+1]}}$ at level $i+1$ triggers the function $c_{\text{FE[i]}}$ at level i, till we have verified that all assignments satisfying Φ confirm the truth of Ψ.

- c_{assert}: The Assertion Checker reaches assert `false` when all the previous functions act as intended, implying that the formula was true.

Due to the parameterization, an arbitrary number of threads may execute the different functions at the same time. However, there is no interference between the threads, and there is a natural order between the roles: c_{SATC} requires c_{AG} to function as intended, and $c_{\text{FE[i]}}$ requires c_{AG}, c_{SATC}, and $c_{\text{FE[j]}}$ with $n-1 \geq j > i$.

We show a novel way to encode the guessed assignments to the Boolean variables in the RA *views*: for each $b \in Vars(\Psi)$, we maintain shared variables t_b, f_b. A view vw encodes b as

$$(\text{vw}(t_b) = 0 \iff b = 1) \land (\text{vw}(f_b) = 0 \iff b = 0).$$

Then, by the RA semantics, the value of b is true if the init message on t_b is readable (recall that the init message is readable only if the thread-local view on t_b is 0). Finally, we need to check that quantifier alternation is maintained. For all $i \in [n \ldots 1]$, a set of threads checks that the alternation $\forall u_{i-1} \exists e_i$ is respected by the guessed assignments. Then they pass on their assignments to the checkers at level $i-1$. This sets up a dependency structure (similar to Section 4) so that a special message can be written iff Ψ is true.

Theorem 5.1. *Parameterized verification for* env(nocas, acyc) *is* PSPACE*-hard, even in* PureRA.

6 CONCLUSION

Atomic compare-and-swap (CAS) operations are indispensible for practical implementations of distributed protocols. At the same time, they hinder verification efforts. Undecidability of safety verification in the non-parameterized setting [1] and even in our loop-free parameterized setting env(acyc) are a testament to this. We tried to reconcile the two by studying the controlled use of CAS in parameterized systems (CAS-free env threads, loop-free dis threads). For such systems, we were able to simplify the RA semantics by abstracting from the timestamps of env threads. The simplified semantics is sound and complete for safety verification and leads to a PSPACE-upper bound. We provide a matching PSPACE-hardness result that gives an insight into the complexity inherent to the synchronization capabilities of RA.

We conclude with interesting avenues for future work. A problem arising from this work is the decidability of CAS-free parameterized systems env(nocas)||dis_1(nocas) ∥ ⋯ ∥ dis_n(nocas) which seems to be as elusive as its non-parameterized twin dis_1(nocas) ∥ ⋯ ∥ dis_n(nocas). We believe the ideas in this paper can be adapted to causally consistent shared memory models [31] and transactional programs [13] in the parameterized setting.

ACKNOWLEDGMENTS

This work was partly supported by SERB MATRICS grant MTR/2019/000095.

REFERENCES

[1] P. A. Abdulla, J. Arora, M. F. Atig, and S. N. Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI*. ACM, 1117–1132.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021. Deciding reachability under persistent x86-TSO. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434337

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2018. A Load-Buffer Semantics for Total Store Ordering. *Log. Methods Comput. Sci.* 14, 1 (2018). https://doi.org/10.23638/LMCS-14(1:9)2018

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. 2020. Parameterized verification under TSO is PSPACE-complete. *Proc. ACM Program. Lang.* 4, POPL (2020), 26:1–26:29. https://doi.org/10.1145/3371094

[5] Parosh Aziz Abdulla and Bengt Jonsson. 1993. Verifying Programs with Unreliable Channels. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*. IEEE Computer Society, 160–170. https://doi.org/10.1109/LICS.1993.287591

[6] Mustaque Ahamad, Gil Neiger, James E. Burns, Prohit Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

[8] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74.

[9] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 7–18. https://doi.org/10.1145/1706299.1706303

[10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's Decidable about Weak Memory Models?. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 26–46. https://doi.org/10.1007/978-3-642-28869-2_2

[11] Hagit Attiya and Sergio Rajsbaum. 2020. Indistinguishability. *Commun. ACM* 63, 5 (2020), 90–99. https://doi.org/10.1145/3376902

[12] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. *SIGPLAN Not.* 46, 1 (Jan. 2011), 55–66. https://doi.org/10.1145/1925844.1926394

[13] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Robustness Against Transactional Causal Consistency. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands.* 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2019.30

[14] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II*. Springer, 24–51.

[15] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On solving universally quantified Horn clauses. In *SAS (LNCS, Vol. 7935)*. Springer, Springer, 105–125.

[16] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2016. Decidability in Parameterized Verification. *SIGACT News* 47, 2 (2016), 53–64. https://doi.org/10.1145/2951860.2951873

[17] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 626–638. http://dl.acm.org/citation.cfm?id=3009888

[18] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1990. Syntax and semantics of datalog. In *Logic Programming and Databases*. Springer, 77–93.

[19] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34.

[20] Javier Esparza, Pierre Ganty, and Rupak Majumdar. 2016. Parameterized Verification of Asynchronous Shared-Memory Systems. *J. ACM* 63, 1 (2016), 10:1–10:48. https://doi.org/10.1145/2842603

[21] Alain Finkel and Philippe Schnoebelen. 2001. Well-structured transition systems everywhere! *Theor. Comput. Sci.* 256, 1-2 (2001), 63–92. https://doi.org/10.1016/S0304-3975(00)00102-X

[22] Adwait Godbole, Shankara Narayanan Krishna, and Roland Meyer. 2021. Safety Verification of Parameterized Systems under Release-Acquire. https://doi.org/10.48550/ARXIV.2101.12123

[23] Georg Gottlob and Christos Papadimitriou. 2003. On the complexity of single-rule datalog queries. *Information and Computation* 183, 1 (2003), 104–122.

[24] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. 2018. GPS $$+$$ + : Reasoning About Fences and Relaxed Atomics. *Int. J. Parallel Program.* 46, 6 (2018), 1157–1183.

[25] Neil Immerman. 2012. *Descriptive complexity*. Springer Science & Business Media.

[26] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[27] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM, 175–189.

[28] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

[29] Christos Kozyrakis. [n.d.]. Phoenix 2.0 Benchmarks. https://github.com/kozyraki/phoenix.

[30] Ori Lahav. 2019. Verification under Causally Consistent Shared Memory. *ACM SIGLOG News* 6, 2 (apr 2019), 43–56. https://doi.org/10.1145/3326938.3326942

[31] Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. https://doi.org/10.1145/3385412.3385966

[32] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 649–662. https://doi.org/10.1145/2837614.2837643

[33] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM, 649–662.

[34] Ori Lahav and Roy Margalit. 2019. Robustness against Release/Acquire Semantics. In *PLDI 2019*. 126âĂŞ141. https://doi.org/10.1145/3314221.3314604

[35] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *ICALP (LNCS, Vol. 9135)*. Springer, 311–323.

[36] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS.. In *SOSP*, Ted Wobber and Peter Druschel (Eds.). ACM, 401–416. http://dblp.uni-trier.de/db/conf/sosp/sosp2011.html#LloydFKA11

[37] Brian Norris. [n.d.]. Model Checker Benchmarks. https://github.com/computersforpeace/model-checker-benchmarks.

[38] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. 2016. Operational Aspects of C/C++ Concurrency. *CoRR* abs/1606.01400 (2016). arXiv:1606.01400 http://arxiv.org/abs/1606.01400

[39] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2018. On Parallel Snapshot Isolation and Release/Acquire Consistency. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 940–967. https://doi.org/10.1007/978-3-319-89884-1_33

[40] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, 691–707.

[41] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *OOPSLA*. ACM, 867–884.

[42] M Vardi. 1982. The complexity of relational database queries. In *Proc. STOC*. 137–146.