Verification of Programs under the Release-Acquire Semantics

Parosh Aziz Abdulla
Uppsala University
Uppsala, Sweden
parosh@it.uu.se

Jatin Arora
India Institute of Technology Bombay
Mumbai, Maharashtra, India
jatinarora@cse.iitb.ac.in

Mohamed Faouzi Atig
Uppsala University
Uppsala, Sweden
mohamed_faouzi.atig@it.uu.se

Shankaranarayanan Krishna
India Institute of Technology Bombay
Mumbai, Maharashtra, India
krishnas@cse.iitb.ac.in

Abstract

We address the verification of concurrent programs running under the release-acquire (RA) semantics. We show that the reachability problem is undecidable even in the case where the input program is finite-state. Given this undecidability, we follow the spirit of the work on context-bounded analysis for detecting bugs in programs under the classical SC model, and propose an under-approximate reachability analysis for the case of RA. To this end, we propose a novel notion, called *view-switching*, and provide a code-to-code translation from an input program under RA to a program under SC. This leads to a reduction, in polynomial time, of the bounded view-switching reachability problem under RA to the bounded context-switching problem under SC. We have implemented a prototype tool VBMC and tested it on a set of benchmarks, demonstrating that many bugs in programs can be found using a small number of view switches.

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

Keywords Model-Checking, weak memory models, RA

ACM Reference Format:
Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of Programs under the Release-Acquire Semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314649

1 Introduction

Sequential consistency (SC) [26] is the traditional memory model assumed by programmers and verification tools for the design and analysis of concurrent programs. Under SC, instructions of the different processes (threads) are interleaved and executed atomically while the program order between the instructions of the same process is preserved. To optimize performance, modern architectures and compilers implement memory models that weaken the guarantees given by SC, by allowing various processes to execute instructions out-of-order, and also by allowing access to memory stores by different processes in different orders. While single-threaded programs are unaffected by these re-orderings, the re-orderings are visible in concurrent programs, leading to unexpected and undesirable behaviours. As classic examples, the Intel x86 [14] and SPARC [40] processors use the TSO (total store order) memory model [38]. The Power [13] and ARM [5] architectures employ memory models that are even more relaxed than TSO [4, 27, 29, 31]. Moreover, compilers, such as Java [28] and C++ [9], reorder commands to prefetch values from memory and fill the processor's execution line. In all these models, different processes may observe different memory states at a given point of time. This results in programs with behaviours that are not possible in SC.

A weak memory model that has received a lot of attention recently is the Release-Acquire (RA) model (see e.g., [4, 9, 16, 18, 24, 37]). RA is a useful and well-behaved subset of the C11 memory model that provides a good balance between performance and programmability. In RA, all writes are release accesses, while all reads are acquire accesses. The operational model for RA has been independently developed in [17, 34]. Subsequently, [16] combined RA and NA (non-atomics) on top of the models developed in [17, 34]. According to the operational model in [17, 34], every read operation should be justified by a write, which has to happen *logically* before the read. More precisely, the shared memory in the RA model is represented by a pool of writes (that have been generated so far). The RA semantics enforces a total order on all the writes to the same variable. This order between the writes is kept track of through the use of timestamps (which are simply numbers from $\mathbb{N}$). The progress of a process is maintained using a view that records the timestamp of the most recent write event observed by the process for each variable. A read operation of a process is successful only if it pertains to a write which is the most recent write observed by that process at the time of the read. The view of a process is also used to determine the timestamp of a newly executed write. Furthermore, the RA semantics imposes a *causality* relation between the writes on different variables: if a process observes a write then it has also observed all the previous writes that have been observed by the process who has issued that write. Thus, the shared memory in RA can be seen as a partially ordered graph where the nodes correspond to the writes and an edge between two nodes reflects either the timestamp order or the causality relation.

The decidability and complexity of the verification problems (e.g., the control state reachability) for various memory models are not obvious a priori; and even worse, they have not been deeply investigated so far. In fact, the standard operational definitions for weak memory models often use unbounded data structures such as partially ordered graphs, trees and/or buffers, thus giving rise to an infinite-state space even in the case where the original program is finite-state. Under the SC semantics, the control state reachability problem is known to be PSPACE-complete [19]. Under TSO and PSO, this problem has been shown to be non-primitive recursive [6, 7]. The decidability of the control reachability problem for other memory models, such as the RA memory model, is an open and highly challenging problem.

Our first contribution is to show that the control state reachability for concurrent programs over a finite data-domain under the RA memory model is undecidable. This is quite a surprising result considering the simplicity and intuitiveness of the operational semantics for the RA memory model. To show undecidability, we use a non-trivial reduction from the Post's Correspondence Problem. The reduction involves four processes, two *guessing* processes that guess a solution of the problem, and two *verifying* processes that prevent loss of information due to non-deterministic updates of the process views. Observe that several undecidability results for the reachability problems for programs running under weak memory models (e.g., [6, 7]) as well as message-passing programs [10], [2] use reduction from PCP but are entirely different from our proof. Proving undecidability for RA is much more difficult than proofs in [6, 7], [10], [2]. In fact, these proofs use perfect FIFO queues for storing the writes/reads. In the RA semantics, the shared memory can be seen as a partially ordered graph (due to the use of timestamps and the causality relation). Therefore, it is not possible to use the same ideas as in the two previously cited papers. Another challenge is the possibility of loss of information in RA which is related to the fact that a process can read any value of a variable x whose timestamp is larger than the current view of that process for x (i.e., a process can skip reading some values). To address these challenges, our proof makes use of compare-and-swap (CAS) operations and of the causality property to ensure that a process does not skip reading any written value. It is still an open problem if this undecidability result holds in the absence of CAS. As a first step towards this, we prove a non-primitive recursive lower-bound for reachability in the absence of CAS.

Besides establishing the frontier of decidability for programs under RA, it is clear that there is a need of verification techniques with reasonable complexity that are applicable in practice. These techniques will necessarily be approximate considering our undecidability result for the RA model in the presence of CAS and our non-primitive recursive lower-bound in the absence of CAS. Our approach consists in defining a search policy that restricts the set of program executions considered by the verification procedure, so that we only keep those executions that will most likely lead to bugs. Such a policy is usually based on suitable bounding concepts in the spirit of context-bounding (e.g., [1, 8, 21, 25, 30, 36]).

Context-bounding has been proposed in [21, 25, 30, 36] as a framework for efficient bug detection in concurrent programs running under SC. Context-bounding provides a very good tradeoff between scalability and verification coverage. For finite-state programs, the bounded-context reachability is NP-complete (in contrast to the PSPACE-complete complexity in the general case). Furthermore, it has been shown experimentally that in many cases, concurrency bugs show up within a small number of context switches [30].

In the case of weak memory models, context-bounded analysis has been proposed to programs running under the TSO memory model in [8, 39] and under the POWER memory model [1]. Such context-bounding techniques have led to efficient *sequentialization* techniques. Sequentialization reduces the state reachability problem under a given memory model to the same problem under SC. Sequentialization usually provides a simpler encoding of the weak memory model, since it removes unbounded data structures (such as queues in TSO) used by the weak memory model. It also allows the leveraging of existing verification tools under SC, obtaining efficient solutions with reasonable cost (e.g., [1, 8]).

In our second contribution, we propose such a bounding approach in the case of RA. This is a challenging task, and requires new techniques that are radically different from the ones used for SC, TSO or POWER. Context-bounding is not suitable for RA since our undecidability result for the state reachability problem under RA still holds even if we restrict our analysis to executions that can be split to a bounded number of contexts, where in each context, only one process is active. Therefore, we introduce a new concept of bounding (called *view-bounding*) which is suitable for RA. View-bounding is based on the observation that RA is similar to a memory system where each process has its own view of the write operations. In our analysis, we restrict ourselves to executions that have a finite number of *view-switches*. A *view-switch* takes place when an external event (i.e., a write of another process) alters the view of some process on a read or an atomic-read-write operation.

Our main contribution is a polynomial time code-to-code translation that takes as input a concurrent program *Prog* running under RA, together with a given budget k of view-switches, and produces another concurrent program *Prog′* such that for every k-bounded view-switching execution of *Prog* under RA, there is a corresponding context-bounded execution of *Prog′* under the SC semantics. Hence, the bounded view-switching state reachability problem for *Prog* under RA can be reduced to the context-bounded state reachability problem for *Prog′* under SC. Furthermore, the obtained program *Prog′* has the same variable domains as the original *Prog*. As an immediate consequence of this reduction, we obtain that the bounded view-switching reachability problem for finite-state programs under RA is decidable. To show the feasibility of our approach, we have implemented our code-to-code translation in a prototype tool, VBMC. The tool is built as an extension of the Lazy Cseq tool [15]. We use CBMC version 5.1 [11] as the backend tool for solving SC reachability queries. We have applied VBMC on a set of benchmarks confirming our hypothesis that many bugs manifest themselves within a small number of view-switches.

In summary, the contributions of the paper include:
• The undecidability of the reachability problem for programs under RA over a bounded domain. We prove that this undecidability still holds even if we restrict our analysis to executions that can be split to a bounded number of contexts, where in each context, only one process is active. The decidability of the reachability problem under RA without CAS over a bounded domain is still open; we show a non-primitive recursive lower bound for this class.
• A novel concept of bounded view-switching to obtain the decidability of the reachability problem under RA. From a bounded view-switching program under RA, we have a polynomial code-to-code translation to a bounded-context program under SC. As an immediate consequence, we obtain that the bounded view-switching reachability problem for finite-state programs under RA is decidable.
• An implementation of our code-to-code translation in a tool (called VBMC). Our experimental results demonstrate the effectiveness of our approach. Akin to bounded-context model checking, we have many examples where shallow bugs were caught with a small value for the view-switching, even in large programs.

Related Work. As stated in the introduction, the RA memory model has received a lot of attention during the last years (see e.g., [4, 9, 16, 18, 24, 37]). The formal operational semantics for the RA memory model that we are using in this paper is the one proposed in [17, 34].

```
Prog  ::=  var x* (proc p   reg $r*   i*)*
   i  ::=  λ : s;
   s  ::=  $r = x | x = $r | cas(x, $r1, $r2)
           assume(exp) | $r = exp | term
           if exp then i* else i* end if | while exp do i* done
```

Figure 1. Syntax of concurrent programs.

Despite all these efforts, there is still little work on the verification of programs running under the RA memory model. Most of the existing work concerns the development of stateless model checking (SMC), coupled with (dynamic) partial order reduction techniques (see e.g., [3, 18, 33]). [16] presents an adaptation of the framework of concurrent separation logics to the RA memory model. These papers on the verification of RA programs are orthogonal to ours.

Context-bounding has been proposed in [36] for programs running under SC. This work has been extended in different directions and has led to efficient and scalable techniques for the analysis of concurrent programs (see e.g., [12, 20–23, 25, 30]). In the context of weak memory models, context-bounded analysis has been only proposed to programs running under the TSO/PSO memory model in [8, 39] and under the POWER memory model [1]. Our approach and the ones proposed in [1, 8, 39] are orthogonal since we are using different bounding concepts more suitable for the RA memory model. This implies also that our code-to-code translation is conceptually different from the ones proposed in [1, 8, 39].

2 Preliminaries

We use $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ to denote the set of booleans, and use $\mathbb{N}$ to denote the set of natural numbers. Fix a set A. If A is finite then we use $|A|$ to denote the size of A. For sets A and B, we use $f : A \rightharpoonup B$ to denote that f is a (possibly partial) function from A to B. We use $f[a \mapsto b]$ to denote the function f' such that $f'(a) = b$ and $f'(a') = f(a')$ if $a' \neq a$. We use A^* to denote the set of finite words over A, and use ϵ to denote the empty word. We use $|w|$ to denote the length of a word $w \in A^*$, use $w[i]$ to denote the i^{th} element of w, and use $w[i..j]$ to denote the word $w[i]w[i+1] \cdots w[j]$.

3 Concurrent Programs

In this section, we first define the syntax we use for concurrent programs and then we present the RA semantics including the transition system it induces (following [16, 17, 34]).

Program Syntax. We describe in Fig. 1, using an assembly-language like notation, the grammar used in defining concurrent programs. A program *Prog* first declares the set $\mathbb{X}$ of (shared) variables, followed by the code of a set $\mathbb{P}$ of processes. The code of each process p starts by declaring a set $\mathbb{R}(p)$ of (local) *registers* followed by a sequence of labeled instructions. It is assumed that the sets of registers of the different processes are disjoint, and we let $\mathbb{R} := \cup_p \mathbb{R}(p)$.

We assume that the registers and shared variables take values from a (potentially unbounded) data domain $\mathbb{D}$. All the shared variables and registers are initialized with the special value $0 \in \mathbb{D}$ (if not mentioned otherwise). A labelled instruction i has the form $\lambda : s$ where λ is a unique label and s is a statement. Let $\mathbb{L}_p$ denote the set of instruction labels for a process p, and let $\mathbb{L} = \bigcup_{p \in \mathbb{P}} \mathbb{L}_p$ be the set of all instruction labels of all processes. We assume that the execution of any process p starts with a unique initial instruction labeled by λ^p_{init}. A write instruction of the process p is of the form $\lambda : x = \$r$ where $\$r \in \mathbb{R}(p)$ and x is a shared variable. This results in assigning the value of $\$r$ to the variable x. A read instruction of the process p has the form $\lambda : \$r = x$ where x is a shared variable and $\$r \in \mathbb{R}(p)$. This results in reading the value of variable x into the local register $\$r$. A compare-and-swap (cas) instruction has the form $\lambda : \mathtt{cas}(\mathtt{x}, \$\mathtt{r}_1, \$\mathtt{r}_2)$ where x is a shared variable and $\$r_1, \$r_2 \in \mathbb{R}(p)$. This results in checking whether the value of the variable x matches the values of the register $\$r_1$ and if it is the case then the variable x is assigned the value of the register $\$r_2$. An assignment instruction $\lambda : \$r = \exp$ assigns to $\$r \in \mathbb{R}(p)$ the value of exp, where exp is an expression over a set of operators, constants as well as the contents of the registers $\mathbb{R}(p)$, but not referring to the set of variables. The Conditional, Assume and Iterative instructions (collectively called *cai* instructions) are explained in the usual way (here also exp is an expression that does not contain any shared variable). Given a label λ of an *cai* instruction, we use $\mathtt{Exp}(\lambda)$ to denote the expression appearing in the instruction. The instruction $\lambda_{\mathtt{term}} : \mathtt{term}$ appears only once in the execution of the code of process p, and causes the process p to terminate its execution.

Given an instruction label λ of a process p, let $\mathtt{next}(\lambda)$ denote the labels of the next instructions that can get executed in p. Observe that this set contains one unique element if λ is the label of a write, read, atomic-read-write, or an assignment instruction. This set contains two elements if λ is the label of an *cai* instruction (in the case of an assume instruction, we assume that if the condition evaluates to false, then the process remains at λ thereafter). We define $\mathtt{Tnext}(\lambda)$ (resp. $\mathtt{Fnext}(\lambda)$) to be the (unique) label of the instruction to which the process execution moves in case the expression appearing in the statement of the instruction labeled by λ evaluates to true (resp. false). We also define $\mathtt{next}(\lambda_{\mathtt{term}}) = \bot$.

Finally, given an expression exp and a function $R : \mathbb{R} \rightharpoonup \mathbb{D}$ that maps any register to a value in $\mathbb{D}$, we use $\mathtt{Val}(\exp, \mathtt{R})$ to denote the value of the expression exp obtained by replacing any occurrence of a register $\$r$ by $R(\$r)$.

RA Operational Semantics. In the following, we define the RA semantics following [16, 17, 34]. The RA semantics enforces a total order on all the writes to the same variable. The progress of each process is kept track of in terms of which writes are visible to it, and this determines what it can read from, and where its own writes will end up. Each variable uses a set of totally ordered timestamps which, roughly, is used to determine how "fresh" a write associated to the variable is. Each timestamp is an element of $\mathbb{N}$. We use $\mathsf{Time} \triangleq \mathbb{N}$ to denote the set of all possible timestamps. Each write of some value v to a variable x is assigned a unique timestamp. This results in a write event $e \in \mathsf{Event} \triangleq \mathbb{X} \times \mathbb{D} \times \mathsf{Time}$. The progress of a process is maintained using a view which is a function from $\mathbb{X}$ to Time that records the timestamp of the most recent write event observed by the process for each variable. We use View to denote the set of all possible views (i.e., all the functions from $\mathbb{X}$ to Time). Every write event is augmented by the view of the writing process, yielding a message $m \in \mathcal{M} \triangleq \mathsf{Event} \times \mathsf{View}$.

Configurations. A configuration of *Prog* is a tuple (M, P, J, R) where $M \subseteq \mathcal{M}$ is a message pool (called the memory), $P : \mathbb{P} \rightharpoonup \mathsf{View}$ maps each process to its view, $J : \mathbb{P} \rightharpoonup \mathbb{L}$ maps each process to its current instruction label, and $R : \mathbb{R} \rightharpoonup \mathbb{D}$ maps each register to its current value. We use $\mathbb{C}$ to denote the set of all configurations. The initial configuration c_{init} of *Prog* is defined by the tuple $(M_{init}, P_{init}, J_{init}, R_{init})$ where: (i) The initial memory M_{init} consists of tuples of the form $(x, 0, 0, V_{init})$ where all variables x are initialized to the special value $0 \in \mathbb{D}$, and have the initial timestamp $0 \in \mathsf{Time}$. Furthermore, the initial view V_{init} maps all variables to the initial timestamp $0 \in \mathsf{Time}$. (ii) P_{init} maps each process to the initial view (i.e., $P_{init}(p) = V_{init}$ for all $p \in \mathbb{P}$). (iii) J_{init} maps each process to its initial instruction (i.e., $J_{init}(p) = \lambda^p_{init}$ for all $p \in \mathbb{P}$). And (iv) R_{init} maps each register to the special value $0 \in \mathbb{D}$ (i.e., $R_{init}(\$r) = 0$ for all $\$r \in \mathbb{R}$).

Transition Relation. We define the transition relation as a relation $\rightarrow \subseteq \mathbb{C} \times \mathbb{L} \times \mathbb{C}$ between the configurations of the program *Prog*. For an instruction $\lambda : s$ of a process $p \in \mathbb{P}$ and two configurations $\mathsf{c} = (\mathsf{M}, \mathsf{P}, \mathsf{J}, \mathsf{R})$ and $\mathsf{c}' = (\mathsf{M}', \mathsf{P}', \mathsf{J}', \mathsf{R}')$ such that $J(p) = \lambda$, we write $\mathsf{c} \xrightarrow[\mathsf{p}]{\lambda:\mathsf{s}} \mathsf{c}'$ to denote that $(\mathsf{c}, \mathsf{p}, \mathsf{c}') \in \rightarrow$. The relation $\rightarrow$ is defined through a set of inference rules given in Figure 2. Below, we explain these inference rules.

The rule Read executes a read instruction $\lambda : \$r = x$ of a process p. For the read to be successful, there must be some message of the form $(x, v, t, V) \in M$ in the memory such that $P(p)(x) \leq t$ (i.e., the current view of the process p for the variable x is older than the message (x, v, t, V)). In this case, the value v is assigned to $\$r$ and the view $P(p)$ is merged with V (denoted $P(p) \sqcup V$) obtaining a new view for p (i.e., $P' = P[p \mapsto P(p) \sqcup V)$. (Observe that the merge operation $\sqcup$ is defined in the caption of Figure 2.) The memory remains the same and the label of the instruction to get executed by any process $p' \neq p$ remains unchanged while the one of the process p is updated to $\mathtt{next}(\lambda)$).

The rule $\mathtt{Write}$ executes a write instruction of the form $\lambda : x = \$r$ of a process p. To perform this instruction, there must exist an unused timestamp $t \in \mathbb{N}$ (i.e., there are no $m \in$

Read $$\frac{\exists v, \exists V[(x,v,t,V)\in M],\ P(p)(x)\leq t,\ J(p)=\lambda}{(M,P,J,R)\xrightarrow[p]{\lambda:\$r=x}(M,P[p\mapsto P(p)\sqcup V],J[p\mapsto \mathtt{next}(\lambda)],\mathrm{R}[\$\mathrm{r}\mapsto \mathrm{v}])}$$

Write $$\frac{\neg[\exists v', \exists V(x,v',t,V)\in M],\ P(p)(x)<t,\ J(p)=\lambda,\ V'=P(p)[x\rightarrow t]}{(M,P,J,R)\xrightarrow[p]{\lambda:x=\$r}(M\cup(x,R(\$r),t,V'),P[p\mapsto V'],J[p\mapsto \mathtt{next}(\lambda)],\mathrm{R})}$$

CAS $$\frac{\exists V[(x,R(\$r_1),t,V)\in M],\ P(p)(x)\leq t,\ \neg[\exists v, \exists V'(x,v,t+1,V')\in M],\ J(p)=\lambda,\ U=P(p)\sqcup V}{(M,P,J,R)\xrightarrow[p]{\lambda:\mathtt{cas}(\mathtt{x},\$\mathtt{r}_1,\$\mathtt{r}_2)}(M\cup(x,R(\$r_2),t+1,U[x\mapsto t+1]),P[p\mapsto U[x\mapsto t+1]],J[p\mapsto \mathtt{next}(\lambda)],\mathrm{R})}$$

Figure 2. Inference rules defining the transition relation $\xrightarrow[p]{\lambda:s}$ where $p \in \mathbb{P}$ and $\lambda : s$ is labelled instruction of p. The merge operation $\sqcup$ between two views V and V' is defined as follows: Let $U = V \sqcup V'$ then for any variable $y \in \mathbb{X}$, we have $U(y) = V'(y)$ if $V'(y) \geq V(y)$, and $U(y) = V(y)$ otherwise.

M such that m is of the form (x, v', t, V)). This timestamp t should be also larger than $P(p)(x)$ (i.e., the current view of p for the variable x). If a such timestamp exists then a new message $(x, R(\$r), t, V')$, with $R(\$r)$ as the new value of x, t as its timestamp, and $V' = P(p)[x \mapsto t]$ as its view (i.e., the view of the process p updated to t for the variable x), is added to the memory. Moreover, the view of any process different from p does not change while the view of the process p is updated to t for the variable x. Furthermore, the label of the instruction to get executed by any process $p' \neq p$ remains unchanged while the one for the process p is updated to $\mathtt{next}(\lambda)$. Finally, the values of registers do not change.

The rule CAS executes a compare-and-swap instruction of the form $\lambda : \mathtt{cas}(\mathtt{x}, \$\mathtt{r}_1, \$\mathtt{r}_2)$ of a process p. To execute the compare-and-swap instruction, there must be a message $(x, R(\$r_1), t, V) \in M$ in the memory pool such that $P(p)(x) \leq t$ (i.e., the current view of p for the variable x). Furthermore, we require that there is no message of the form $(x, v, t + 1, V'')$ in M. Then, the views $P(p)$ and V are merged obtaining a new view U as in the case of read. This is followed by adding the message $(x, R(\$r_2), t + 1, U[x \mapsto t + 1])$ to the memory pool. (Observe that two cas's cannot use the same message in the memory for their reads). Furthermore, the view of any process different from p does not change while the new view of the process p is set to $U[x \mapsto t + 1]$. Moreover, the label of the instruction to get executed by any process $p' \neq p$ remains unchanged while the one of the process p is updated to $\mathtt{next}(\lambda)$. Finally, the values of registers do not change.

The inference rules for assignments of the form $\lambda : \$r =$ exp and *cai* instructions are not shown in Figure 2 and they can be defined in the usual way. These are internal instructions for p that affect only the label of the instruction to get executed by p and the values of its registers while the memory while the views of all processes remain unchanged.

A run of *Prog* is a sequence of the form $\mathtt{c}_{init} \xrightarrow[\mathtt{p}_1]{\lambda_1:\mathtt{s}_1} \mathtt{c}_1 \xrightarrow[\mathtt{p}_2]{\lambda_2:\mathtt{s}_2} \mathtt{c}_2 \ldots \mathtt{c}_{\mathtt{m}-1} \xrightarrow[\mathtt{p}_\mathtt{m}]{\lambda_\mathtt{m}:\mathtt{s}_\mathtt{m}} \mathtt{c}_\mathtt{m}$. In this case the configurations $\mathtt{c}_{init}$, $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_\mathtt{m}$ are said to be reachable from the initial configuration $\mathtt{c}_{init}$.

The Reachability Problem. Given an instruction label function $J : \mathbb{P} \rightharpoonup \mathbb{L}$ that maps each process $p \in \mathbb{P}$ to an instruction label in $\mathbb{L}_p$, the *control reachability* problem asks, starting from the initial configuration $\mathtt{c}_{init}$, whether we can reach a configuration $\mathtt{c} = (M, P, J, R)$ for some M, P and R.

4 The Reachability Problem under RA

In the following, we first show that the reachability problem for concurrent programs over a finite data-domain (i.e., $\mathbb{D}$ is finite) under the RA semantics is undecidable by a reduction from the Post's Correspondence Problem (PCP) [35]. Our proof makes use of CAS and of the causality property to ensure that a process does not skip reading any written value. Furthermore, we prove that this undecidability still holds even if we restrict our analysis to executions that can be split to a bounded number of contexts, where in each context, only one process is active. The decidability of the reachability problem under RA without CAS over a bounded domain is still an open problem. However, we were able to prove a non-primitive recursive lower-bound for the reachability problem in the absence of CAS (see Theorem 4.3).

Theorem 4.1. *The reachability problem for concurrent programs over a finite data domain is undecidable under RA .*

The proof is by a reduction of the Post's Correspondence Problem (PCP), which is well-known to be undecidable. Recall that PCP consists in, given two finite sequences $\{u_1, \ldots, u_n\}$ and $\{v_1, \ldots, v_n\}$ of non-empty words over some finite alphabet Σ, checking whether there is a sequence $j_1, \ldots, j_k \in \{1, ..., n\}$ of indices such that $u_{j_1} \ldots u_{j_k} = v_{j_1} \ldots v_{j_k}$. We construct a concurrent program *Prog* (Figure 3) with 4 processes p_1, p_2, p_3 and p_4, sharing a set of variables $\mathbb{X} = \{x_i, y_i \mid 1 \leq i \leq 4\}$. The idea of the reduction is as follows:

• Process p_1 guesses a solution of PCP as a sequence of indices $i_1, \ldots, i_k$. The indices i are guessed using a register aux_1. p_1 writes (using $\mathsf{Module}^{p_1}_{i_1}, \ldots, \mathsf{Module}^{p_1}_{i_k}$) the sequence of $u_{i_1}, \ldots, u_{i_k}$, symbol by symbol in an alternating manner, to the variables x_1, x_2. Whenever p_1 finishes writing the symbols of a word u_i, it writes the index i into y_1 and y_2

Process p_1	Module$_i^{p_1}$	Process p_2	Module$_i^{p_2}$	Process p_3	Process p_4
while $(aux_1 \neq \perp)$ do	if$(turn_x^1 = 1)$	while $(aux_2 \neq \perp)$ do	if$(turn_x^2 = 1)$	while	while
if$(aux_1 = 1)$	$x_1 = u_i[1]$;	if$(aux_2 = 1)$	$x_3 = v_i[1]$;	$(aux_3 \neq \perp)$ do	$(aux_4 \neq \perp)$ do
Module$_1^{p_1}$; end if	$x_2 = u_i[2]$;	Module$_1^{p_2}$; end if	$x_4 = v_i[2]$;	if$(turn^3 = 1)$	if$(turn^4 = 1)$
if$(aux_1 = 2)$	$\cdots$	if$(aux_2 = 2)$	$\cdots$	cas(x_1, aux$_3$, 0);	cas(y_1, aux$_3$, 0);
Module$_2^{p_1}$; end if	$x_{\lvert u_i \rvert} = u_i[\lvert u_i \rvert]$;	Module$_2^{p_2}$; end if	$x_{\lvert v_i \rvert} = v_i[\lvert v_i \rvert]$;	assume(x_2 = 0);	assume(y_2 = 0);
$\cdots$	$turn_x^1 = k_i$;	$\cdots$	$turn_x^2 = \ell_i$;	cas(x_3, aux$_3$, 0);	cas(y_3, aux$_3$, 0);
if$(aux_1 = n)$	else	if$(aux_2 = n)$	else	assume(x_4 = 0);	assume(y_4 = 0);
Module$_n^{p_1}$; end if	$x_2 = u_i[1]$;	Module$_n^{p_2}$; end if	$x_4 = v_i[1]$;	$turn^3 = 2$;	$turn^4 = 2$;
done	$x_1 = u_i[2]$;	done	$x_3 = v_i[2]$;	else;	else;
if$(turn_x^1 = 1)$	$\cdots$	if$(turn_x^2 = 1)$	$\cdots$	cas(x_2, aux$_3$, 0);	cas(y_2, aux$_3$, 0);
$x_1 = \perp$;	$x'_{\lvert u_i \rvert} = u_i[\lvert u_i \rvert]$;	$x_3 = \perp$;	$x'_{\lvert v_i \rvert} = v_i[\lvert v_i \rvert]$;	assume(x_1 = 0);	assume(y_1 = 0);
else $x_2 = \perp$;	$turn_x^1 = k'_i$;	else $x_4 = \perp$;	$turn_x^2 = \ell'_i$;	cas(x_4, aux$_3$, 0);	cas(y_4, aux$_3$, 0);
end if	end if	end if	end if	assume(x_3 = 0);	assume(y_3 = 0);
if$(turn_y^1 = 1)$	if$(turn_y^1 = 1)$	if$(turn_y^2 = 1)$	if$(turn_y^2 = 1)$	$turn^3 = 1$;	$turn^4 = 1$;
$y_1 = \perp$;	$y_1 = i$; $turn_y^1 = 2$;	$y_3 = \perp$;	$y_3 = i$; $turn_y^2 = 2$;	end if	end if
else $y_2 = \perp$;	else	else $y_4 = \perp$;	else	done	done
end if	$y_2 = i$; $turn_y^1 = 1$;	end if	$y_4 = i$; $turn_y^2 = 1$;	term	term
term	end if	term	end if		

Figure 3. Instruction labels have been omitted. Depending on the parity of $|u_i|, |v_i|$, variables $x_{|u_i|}, x'_{|u_i|}$ $(x_{|v_i|}, x'_{|v_i|})$ represent x_1, x_2 or (x_3, x_4) and values k_i, k'_i (ℓ_i, ℓ'_i) stand for 1 or 2.

alternately. Each Module$_i^{p_1}$ uses the register $turn_x^1$ to ensure the alternation while writing symbols of u_i to x_1, x_2.

• Process p_2 guesses a solution of PCP as a sequence of indices $j_1, \ldots, j_\ell$. The indices j are guessed using a register aux_2. p_2 writes (using Module$_{j_1}^{p_2}, \ldots,$ Module$_{j_\ell}^{p_2}$) the symbols of $v_{j_1}, \ldots, v_{j_\ell}$, symbol by symbol in an alternating manner, to the variables x_3, x_4. Whenever p_2 finishes writing the symbols of a word v_j, it writes the index j into y_3 and y_4 alternately. Each Module$_i^{p_2}$ uses the register $turn_x^2$ to ensure the alternation while writing symbols of v_i to x_3, x_4.

• Process p_3 checks the two words $u_{i_1} \ldots u_{i_k}$ and $v_{j_1} \ldots v_{j_\ell}$ to be same, while process p_4 checks the sequence of indices $i_1, \ldots i_k$ and $j_1, \ldots, j_\ell$ to be same. p_3 uses the register aux_3 to guess a symbol $\alpha \in \Sigma$, and checks if the value stored in x_1, x_3 or (x_2, x_4) is α alternately. This alternation is ensured by register $turn^3$. Likewise, p_4 uses the register aux_4 to guess an index $i \in \{1, \ldots, n\}$, and checks if the value stored in y_1, y_3 or (y_2, y_4) is i alternately. This alternation is ensured by register $turn^4$. Ensuring these two checks are the most challenging in the proof.

We prove that PCP has a solution iff it is possible to reach the label of instruction term in all the processes. In any case, p_1 and p_2 can reach term after finishing their guesses and writing them on $\{x_1, x_2, y_1, y_2\}$ and $\{x_3, x_4, y_3, y_4\}$ respectively. p_3 can reach term iff it reads the values written by p_1 and p_2 onto the variables x_1, x_2 and x_3, x_4 exactly in the same order in which they were written. Making sure that p_3 reads all the values in the order written by the process p_1 on the variables x_1 and x_2 is not easy, since p_3 can, non-deterministically, read any value as long as its associated timestamp is larger than its current view for x_1 (x_2). This may result in p_3 omitting certain writes of p_1, and thereby reading a sub-word of what has been written rather than the whole word. The same issue can happen while p_3 reads from x_3 (x_4). Given that p_1 writes symbols of Σ into x_1, x_2 alternately, we ensure that p_3 does not omit anything by (i) guessing the value $a \in \Sigma$ stored in x_1 (x_2) using a register aux_3 (ii) performing an atomic-read write instruction on x_1 (x_2), where p_3 reads the symbol a and writes 0 to x_1 (x_2), and (iii) checks if the value stored at x_2 (x_1) is still 0. Notice that $0 \notin \Sigma$, the PCP alphabet. We know that p_1 writes into x_1, x_2 alternately. When p_3 reads the k^{th} write of x_1 in step (ii), p_3's view of x_2 is updated to be the timestamp corresponding to the $(k-1)^{th}$ write of x_2 (if it was already smaller). The check in step (iii) can succeed only if the most recent atomic read-write of p_3 with respect to x_2 used the message in memory corresponding to the $(k-1)^{th}$ write to x_2 by p_1. To see why, let us assume that this was not the case, i.e., p_3's most recent atomic-read-write with respect to x_2 used the message in memory corresponding to p_1's j^{th} write to x_2 $(j < k-1)$. Let that message have a timestamp t. When p_3 reads that message through an atomic-read-write instruction, it will update the variable x_2 to 0 and this will result in a new message in memory with a timestamp $t+1$. Since the $j < k-1$, the message in memory corresponding to p_1's $(k-1)^{th}$ write to x_2 has a timestamp strictly larger than $t+1$. Thus, when p_3 performs step (ii), its view will be updated to the timestamp corresponding to the message in memory corresponding to p_1's $(k-1)^{th}$ write to x_2 and therefore, p_3 will not be able to read the value 0 of the variable x_2 at step (iii). This argument also applies when checking the consecution of x_3, x_4. This ensures that the string formed by the sequence $u_{i_1}, \ldots, u_{i_k}$ guessed by p_1 agrees with the string formed by the sequence $v_{j_1}, \ldots, v_{j_\ell}$ guessed by p_2. To certify that this string is indeed a solution to the PCP, p_4 checks the indices written to y_1, y_2, y_3 and y_4 in exactly the same way p_3 checks x_1, x_2, x_3 and x_4. The argument that p_4 cannot jump while reading a y_i follows the same lines as given to argue why p_3 cannot jump while reading some x_j.

The Formal Reduction. In the following, we define more details about the reduction. Let $\mathcal{D} = \Sigma \uplus \{\perp, 0, 1, \ldots, n\}$, where 0,1 to n and $\perp$ are special elements not in Σ, be the set of data manipulated by the processes.

To simplify the presentation, we need to introduce some notations. We use $\lambda : x = c$ (resp. `assume(x = c)`), where $x \in \mathbb{X}$ is a variable and $c \in \mathcal{D}$ is a value, to denote the following two consecutive instructions λ : $\$r = c$; λ' : $x = \$r$ (resp. λ : $\$r = x$; λ' : `assume`($\$r$ = `c`)) where $\$r$ is an auxiliary register that is not used anywhere else. This notation is also extended in a straightforward manner to atomic-read-write instructions. For ease of reading, we will also omit instruction labels when they are irrelevant. Finally, given a subset D of $\mathcal{D}$, we use $\$r = v \in D$ to denote a statement that non-deterministically assigns a value $v \in D$ to the register $\$r$. We will also use `if` *exp* `then` i^* `end if` to denote the following statement `if` *exp* `then` i^* `else assume(true) end if` (i.e., when the else branch is irrelevant). The statement $\$r = v \in D$ is equivalent to $\$r = x$ where x is an auxiliary variable together with an auxiliary process that repeatedly writes to x the values in D in a sequential manner (i.e., the code of the process is given by `while` *true* `do` $x = d_1$; $x = d_2$; $\ldots, x = d_n$; `done` with $D = \{d_1, d_2, \ldots, d_n\}$).

Now, we are ready to give the formal description of each process. The process p_i, with $i \in \{1, 2\}$, has three local registers aux_i, $turn^i_x$, $turn^i_y$. The register aux_i is used to store the current guessed index $j \in \{1, \ldots, n\}$ or the symbol $\perp$ to signal the end of the simulation. The register $turn^i_x$ (resp. $turn^i_y$) is used to store to which variable from x_{2i-1} and x_{2i} (resp. y_{2i-1} and y_{2i}) the next symbol (resp. index) should be written by the process p_i. The process p_k, with $k \in \{3, 4\}$ has aux_i and $turn^i$ as local registers. As in the previous case, the register aux_k stores the current guessed index $j \in \{1, \ldots, n\}$ by the process p_k or the symbol $\perp$ to signal the end of the simulation. The register $turn^3$ (resp. $turn^4$) is used to store from which variable from x_1, x_2, x_3 and x_4 (resp. y_1, y_2, y_3 and y_4) the next symbol should be read by the process p_3 (resp. p_4). All variables and registers are initialized to 0 except the registers $turn^1_x$, $turn^1_y$, $turn^2_x$, $turn^2_y$, $turn^3$ and $turn^4$ which are initialized to 1. In the following, we describe the code of each process.

• The code of process p_1 is given in Figure 3. The process p_1 assigns an index to the register aux_1. If it assigns i, then one of the module $\texttt{Module}^{\texttt{p}_1}_{\texttt{i}}$ is executed. This module will write the symbols of u_i into variables x_1, x_2 alternately. Furthermore, it will write the index i to either the variable y_1 or y_2 depending on the value of the variable $turn^1_y$. The variable $turn^1_x$ (resp. $turn^1_y$) is used to ensure the alternation between the variables x_1 and x_2 (resp. y_1 and y_2). When p_1 finishes its sequence of guesses, it assigns $\perp$ to the register aux_1. At this time, the special symbol is written to either the variable x_1 or x_2 (resp. y_1 and y_2) depending on the value of the variable $turn^1_x$ (resp. $turn^1_y$) to signal to process p_3 and p_4 that the last symbols have been written.

• The code of process p_2 (given in Figure 3) follows the same approach : it guesses a word v_j and assigns j to the register aux_2. Using register $turn^2_x$, it ensures alternation between variables x_3, x_4 while writing the contents of the word v_j. It starts writing into x_3, and depending on the length of the first word v_j guessed, the writing finishes in x_3 or x_4. When p_2 finishes guessing, it assigns $\perp$ to the register aux_2; at this time, it writes $\perp$ into either the variables x_3 or x_4 depending on the value of the variable $turn^2_x$. The index j of the guessed word v_j is written into one of the variables y_3, y_4 by alternation; the register $turn^2_y$ ensures the correctness of this alternation.

• Processes p_1, p_2 are independent of each other. They make their own guesses and write the symbols of the words and corresponding indices into variables x_1, x_2, y_1, y_2 and x_3, x_4, y_3, y_4 respectively. p_3 reads these writes and verifies the sequences of symbols written into x_1, x_2 and x_3, x_4 are the same. The main challenge here is to ensure that p_3 reads all the symbols written without missing any of them. The code of process p_3 is given in Figure 3. To begin, it guesses the symbol that will be read and assigns it to register aux_3. This is followed by reading either x_1 and x_3 or x_2 and x_4 depending on the value of $turn^3$. In fact, the register $turn^3$ ensures that the read operations from x_1, x_3 and x_2, x_4 are done in an alternating fashion. If $turn^3$ is equal to 1 (resp. 2), then the process p_3: (1) performs an `cas`, updating the value of x_1 (resp. x_2) from aux_3 to 0, (2) checks that the value of x_2 (resp. x_1) is indeed 0, (3) performs an `cas`, updating the value of x_3 (resp. x_4) from aux_3 to 0, and (4) checks that the value of x_4 (resp. x_3) is indeed 0. Since $0 \notin \Sigma$, this is successful iff (i) p_3 reads the first value written to x_1 (resp. x_3) by p_1 (resp. p_2), and (ii) this first value is aux_1.

In a similar way, we can see that the first `cas` in p_3 where the value of x_2 is updated from $u_i[2]$ to 0 indeed corresponds to the first write to x_2 by p_1; finally, the first `cas` of x_4 in p_3 ensures that the first write to x_4 by p_2 has the same value $u_i[2]$. This shows that p_3 cannot "jump" while reading messages in the pool when it does the first `cas` updating variables x_1, x_2, x_3, x_4. Thus, the first `cas` on each x-variable indeed corresponds to the first write by p_1 (or p_2).

We can inductively show that the kth `cas` of variables x_1, x_2 (x_3, x_4) by p_3 indeed correspond to the kth write of x_1, x_2 (x_3, x_4) by p_1 (p_2). Further, one can also ensure that the kth value written to x_3 (x_4) by p_2 is the same as the kth value written to x_1 (x_2) by p_1. The key argument here is that p_3 cannot "jump" : during the kth cas of a variable say x_1, it necessarily must use the message in the memory that corresponds to the kth write to x_1 by p_1. The same argument applies to the other variables x_2, x_3 and x_4. Lemma 4.2 formally proves this.

Lemma 4.2. *The kth* cas *by p_3 corresponds to the kth writes by p_1 and p_2, for all $k \geq 1$.*

Given *Prog* as described above, starting from the initial configuration c_{init}, we ask if it is possible to reach a configuration where all processes reached the term statement. As discussed, processes p_3, p_4 cannot jump while reading variables x_i, y_j. Since the contents of x_1, x_3 as well as x_2, x_4 are checked to be equal by p_3, the guesses made by processes p_1, p_2 must agree; in particular, aux_1 and aux_2 must be assigned $\perp$ after the same number of steps by p_1, p_2 respectively. If p_1 writes $\perp$ to the variable x_1 (x_2) then p_2 will write $\perp$ to the variable x_3 (x_4). The process p_3 reaches term iff it updates this variable $x_1(x_3)$ or $x_2(x_4)$ from $\perp$ to 0 and checks if the other one $x_2(x_4)$ or $x_1(x_3)$ is 0. Similarly, p_1, p_2 write $\perp$ to y_1, y_3 respectively or to y_2, y_4 at the end of the guesses. This leads to p_4 updating y_1, y_3 (y_2, y_4) from $\perp$ to 0 and checking y_2, y_4 (y_1, y_3) are 0. p_4 reaches term at the end of this. In short, all processes reach term iff (i) Processes p_1, p_2 write the same sequence of symbols to variables x_1, x_3 and x_2, x_4 in alternation, (ii) Processes p_1, p_2 write the same indices to variables y_1, y_3 and y_2, y_4 in alternation, (iii) Process p_3 reads x_1, x_3 and x_2, x_4 in the same order in which they were written and verifies that the first two and latter two variables have the same content, and (iv) Process p_4 reads y_1, y_3 and y_2, y_4 in the same order in which they were written and verifies that the first two and latter two variables have the same content. This is possible iff the instance of PCP we start with has a solution.

Remark. Theorem 4.1 holds even if we restrict our analysis to 4-*context* executions where, following [36], a context is a contiguous sequence of operations performed by only one process and a *k-context* execution, for a given $k \in \mathbb{N}$, is an execution that can be partitioned into k-contexts.

In the following, we establish that verification of RA programs is highly non-trivial (i.e., non-primitive recusive) even without cas instructions. The proof is done by reduction from the reachability problem for lossy channel systems, similar to the case of TSO [6].

Theorem 4.3. *Reachability of RA programs over a bounded domain is non-primitive recursive in the absence of CAS.*

5 Reachability Under View-Bounding

Given the undecidability of the reachability problem even under a bounded number of contexts (see the above remark), we introduce a notion of "bounded view-switching" which is relevant for RA programs (as confirmed by our experimental results described in Section 7). We then propose an algorithm that reduces, for a given $\mathrm{K} \in \mathbb{N}$, the K-bounded view reachability under RA to the $(\mathrm{K}+n)$-bounded-context reachability problem under SC where n is the number of processes in the input program. The latter problem is known to be decidable and has been addressed in [21, 25, 30, 36]. More precisely, given an input program *Prog*, the algorithm constructs an output program *Prog'*, whose size is polynomial in the size of *Prog* and $\mathbb{K}$ such that for every K-bounded view run for *Prog* under RA, there is an $(\mathrm{K}+n)$-bounded-context run of *Prog'* under SC that reaches the same set of process labels, and vice-versa. Furthermore, the constructed program *Prog'* has the same variable domains as the input program *Prog*. As an immediate consequence of this reduction, we obtain that the bounded view-switching problem for finite-state programs under RA is decidable.

The rest of this section is structured as follows: First, we define bounded-view runs and briefly recall the definition of context under SC. Then, we give a high-level overview of the translation of *Prog* and mention some of the encountered challenges. Finally, we give more details about the used data structures and the process codes in *Prog'*.

The Bounded-View Reachability Problem. Let us consider a run ρ of the form $\mathsf{c}_{init} \xrightarrow[\mathsf{p}_1]{\lambda_1:\mathsf{e}_1} \mathsf{c}_1 \xrightarrow[\mathsf{p}_2]{\lambda_2:\mathsf{e}_2} \mathsf{c}_2 \ldots \mathsf{c}_{m-1} \xrightarrow[\mathsf{p}_m]{\lambda_m:\mathsf{e}_m} \mathsf{c}_m$. A statement e_j (called here event) in process p_j is *view-altering* in ρ if it involves reading some message from the memory which changes the view of p_j. The event e_j is said to cause a view-switch in ρ. We say that the run ρ in RA is *k-bounded* if the number of view-switches in ρ is $\leq k$. In the reachability problem, for a given label $\overline{\lambda} = (\lambda_{p_i})_{p_i \in \mathbb{P}}$, we have to find a run ρ from the initial configuration c_{init} to some configuration $\mathsf{c} = (M, P, J, R)$ where $J(p) = \lambda_p$ for all $p \in \mathbb{P}$. For $K \in \mathbb{N}$, the K-*bounded* control reachability problem under RA is defined by requiring the run ρ to be K-*bounded*. Note that a change in view because of a write event is not considered a view-switch. This definition can be extended in the straightforward manner to CAS.

Given a program under SC, recall that a run τ (under SC) can be defined, in the usual way, as a sequence of transitions $\gamma_0 \overset{p_1}{\Longrightarrow}_{SC} \gamma_1 \ldots \overset{p_n}{\Longrightarrow}_{SC} \gamma_n$. A context-switch in τ is a configuration γ_j, for some $j > 1$ such that $p_{j-1} \neq p_j$. The run τ is called *k-context bounded* if the number of context switch points is $\leq k$. The K-*bounded* control reachability problem under SC is defined by requiring that τ is K-*context bounded*. For more details about bounded reachability problem under SC, we refer the reader to [21, 25, 30, 36].

For the rest of the paper, we use ρ_{ra} to represent a run in RA and τ_{sc} to denote a run in SC.

Translation Overview. Our reduction is based on code-to-code translation that transforms the RA program *Prog* into an SC program *Prog'*, that operates on the same data domain as *Prog*, using the map $[\![.]\!]_{\mathrm{K}}$ in figure 4. Let $\mathbb{P}$ and $\mathbb{X}$ respectively

$$[\![Prog]\!]_{\mathbb{K}} \overset{\text{def}}{=} \texttt{var}\ x^* \ ;\ \langle add\_gvars\rangle_{\mathbb{K}}\ ;\ \langle \textsc{main}\rangle_{\mathbb{K}};\ ([\![\texttt{proc}\ p\quad \texttt{reg}\ \$r^*\quad i^*]\!]_{\mathbb{K}})^*$$
$$[\![\texttt{proc}\ p\ \ \texttt{reg}\ \$r^*\ \ i^*]\!]_{\mathbb{K}} \overset{\text{def}}{=} \texttt{proc}\ p\ \ \texttt{reg}\ \$r^*;\ \langle add\_lvars\rangle_{\mathbb{K}}\ ;\ \textsc{init\_proc}()\ ;([\![i]\!]_{\mathbb{K}})^*$$
$$[\![i]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} \lambda\ :\ \textsc{is\_init\_round}()\ ;\ [\![s]\!]^p_{\mathbb{K}}\ ;\ \textsc{is\_end\_round}()$$
$$[\![\texttt{if}\ exp\ \texttt{then}\ i^*\ \texttt{else}\ i^*\ ]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} \texttt{if}\ exp\ \texttt{then}\ ([\![i]\!]^p_{\mathbb{K}})^*\ \texttt{else}\ ([\![i]\!]^p_{\mathbb{K}})^*$$
$$[\![\ \texttt{while}\ exp\ \texttt{do}\ i^*\ \texttt{done}\ ]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} \ \texttt{while}\ exp\ \texttt{do}\ ([\![i]\!]^p_{\mathbb{K}})^*\texttt{done}$$
$$[\![\texttt{assume(exp)}]\!]^{\texttt{p}}_{\mathbb{K}} \overset{\text{def}}{=} \texttt{assume(exp)}$$
$$[\![\$r = \texttt{exp}]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} \$r = \texttt{exp}$$
$$[\![\$r = x\ ]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} [\![\$r = x\ ]\!]^{p,\mathbf{read}}_{\mathbb{K}}$$
$$[\![x = \$r\ ]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} [\![x = \$r\ ]\!]^{p,\mathbf{write}}_{\mathbb{K}}$$
$$[\![\texttt{cas}(\texttt{x}, \$\texttt{r}_1, \$\texttt{r}_2)]\!]^{\texttt{p}}_{\mathbb{K}} \overset{\text{def}}{=} [\![\texttt{cas}(\texttt{x}, \$\texttt{r}_1, \$\texttt{r}_2)]\!]^{\texttt{p},\mathbf{cas}}_{\mathbb{K}}$$
$$[\![\texttt{term}]\!]^p_{\mathbb{K}} \overset{\text{def}}{=} \texttt{term}$$

Figure 4. Translation map $[\![.]\!]_{\mathbb{K}}$. Labels of intermediate instructions are omitted when irrelevant.

denote the set of processes and shared variables in *Prog*. The map $[\![.]\!]_{\mathbb{K}}$ keeps the existing set of processes $\mathbb{P}$ but adds a process named **Main** which is used to initialize the global variables used in *Prog*$'$. The map $[\![.]\!]_{\mathbb{K}}$ transforms the code of each process $p \in \mathbb{P}$ to a corresponding process $p' = [\![p]\!]_{\mathbb{K}}$ which will simulate the moves of p. The processes p, p' have the same set of registers. A finite set of data structures is added by $\langle add\_gvars\rangle_{\mathbb{K}}$ to the set of shared variables $\mathbb{X}$ in *Prog*. The finiteness of the domain of these data structures is solely dependent on the domain of the shared variables. Each process p' has a data structure $\mathsf{View}_{p'}$ that stores for p', a finite number of values and time-stamps of shared variables from $\mathbb{X}$ in *Prog*. $\mathsf{View}_{p'}$ is shared with other processes using the data structure Message. We will formally define them in the paragraph **Data Structures**. The map $[\![.]\!]_{\mathbb{K}}$ adds to each p', using $\langle add\_lvars\rangle_{\mathbb{K}}$, some local variables of type $\mathsf{View}_{p'}$ at the start of p', and the function $\textsc{init\_proc}()$ initializes these variables. For each instruction i of p, the map $[\![i]\!]_{\mathbb{K}}$ transforms it to a sequence of instructions as follows: first, it adds the code defined by $\textsc{is\_init\_round}()$ to check if p is active in the current context, then it transforms statement s of instruction i into a sequence of instructions following the map $[\![s]\!]^p_{\mathbb{K}}$, and finally it adds the sequence of instructions defined by $\textsc{is\_end\_round}()$ to guess a context switch. Translation of `aci` and `term` statements keep the same statement. Translations of `read` and `write` statements are described later. The description of `cas` is omitted for ease of presentation.

Challenges. There are different aspects of the RA semantics and the bounded-view reachability problem under RA that make the reduction to the bounded-context reachability problem under SC semantics difficult. The first challenging aspect lies in the definition of the bounded-view reachability problem where we do not put any bound on the number of allowed context-switches. To address this challenge, we classify the pending messages based on the roles they play, in a given $\mathbb{K}$-bounded RA run ρ_{ra}, into three categories. A message m is *redundant* if it is not read by any process. A message m is *useful* if it does not alter the view of any process but is read by some process. An *essential* message is one which alters the view of some process. To explain these different categories, let us consider a process p that first writes to a variable x followed by a write to a variable y and a write to a variable z. Consider another process that reads the value written by p to y followed by a read of the value written by p to x. Then, the write of p to y is *essential*, the write of p to x is *useful*, while the write of p to z is *redundant*.

When two different processes execute two different instructions that do not change the view of the other, these two instructions can be commuted. This roughly means that the only constraint on the order of operation executions that we need to preserve in ρ_{ra} is between a read and a write such that the read gets its value from the write and this results in the changes of the view of the reading process. Thus, from ρ_{ra}, we can construct another $\mathbb{K}$-bounded RA run ρ'_{ra} where processes are run in the order of generation of *essential* messages. This means that, in ρ'_{ra}, we execute first the process that generates the first *essential* message till that message is generated, then we execute the process that generates the second *essential* message till the point that this message is generated, and so on. Observe that ρ'_{ra} has at most $(\mathbb{K} + n)$-contexts where n is the number of processes (here these additional n-contexts are needed to ensure that all processes are run till completion).

The second challenge is related to the unboundedness of the number of possible pending messages in a RA configuration. Observe that this unboundedness results in the fact that the domain of the used timestamps is not finite. To address these two difficulties, consider a k-*bounded* RA run ρ_{ra}, and let $\mathcal{M}$ be the memory pool when ρ_{ra} finishes its execution. Let $M_x \subseteq \mathcal{M}$ be the messages which pertain to variable x

and $T_x \subseteq \mathbb{N}$ be the time-stamps of x that appear in M_x. For each time-stamp $t \in T_x$, we define $Val_x(t) = v$ such that $(x, v, t, V) \in M_x$ for some view V. The size of the memory pool $\mathcal{M}$ can be unbounded; however, not all of them are used in a view-altering event. We address this challenge by showing that we only need a bounded number of messages from $\mathcal{M}$ to correctly simulate the run in SC. The second difficulty is to show that we do not require all time stamps from ρ_{ra} to enforce the order in RA.

When a process p reads a useful message, the value of some register $\$r$ in p is changed. Since this is the only effect of useful messages, it is sufficient to maintain the values of variables contained in useful messages, rather than retain the messages themselves in the memory pool. Each useful message $(x, t, v, V) \in \mathcal{M}$ can be removed from $\mathcal{M}$, and we simply retain $Val_x(t) = v$. The memory pool $\mathcal{M}$ can be pruned to retain only the essential messages, $\mathsf{Es}(\mathcal{M})$. For a K-bounded run ρ_{ra}, $|\mathsf{Es}(\mathcal{M})| \leq K$. It remains to show that we can bound the number of time stamps t for which $Val_x(t)$ has to be maintained. Consider a *read* instruction $\ell : \$r = x$ by a process p and let $P(p)(x) = t$ (recall that P is a mapping from processes to views) after execution of ℓ. If ℓ is view-altering, then, there is some $m = (x, t, v, V) \in \mathsf{Es}(\mathcal{M})$ that will be read by p. If ℓ is not view-altering, then $P(p)$ is unchanged after the execution of ℓ (in particular, $P(p)(x)$ was already t before execution of ℓ). The view $P(p)(x) = t$ can either be from a *write* to x by p itself or can be due to a view-altering event which happened earlier. Therefore, when the view of process p does not change on a read, it either requires a value written by itself, or it needs a value $Val_x(t)$ such that $x \mapsto t$ is part of the view in some $m \in \mathsf{Es}(\mathcal{M})$. Thus, to simulate reads of x which are not view-altering, we do the following

• For each process p, maintain the value of the latest write to x by p in a local variable.

• Guess the time-stamps $x \mapsto t$ appearing in the view of $m \in \mathsf{Es}(\mathcal{M})$ and maintain $Val_x(t)$ for them. Since only K time-stamps of x can appear in $\mathsf{Es}(\mathcal{M})$, we have a bound on the number of time-stamps t for which the values $Val_x(t)$ have to be maintained.

It remains to argue why boundedly many time stamps suffice to enforce the ordering in RA. Each time a view is altered, for each variable, two different time stamps are compared : one from the essential message, and one from the view $P(p)$ which is going to be altered. Note that essential messages give rise to K time stamps and are exactly the ones for which Val_x is maintained. The other time-stamp used in the comparison, can come from any message. Thus, we need to maintain $2K$ time-stamps. For each variable, we guess the $2K$ time stamps that are involved in these comparisons. Instead of using the actual values of the $2K$ time-stamps, we use values from $\{1, \ldots 2K\}$ to represent their order.

Data Structures. Let Time = $\{0, \ldots 2K\}$ represent the set of time-stamps which we use in the translation. The choice of the bound $2K$ is explained in the above paragraph.

• **View**. This is a data-structure that stores a tuple from Time × Value × $\{true, false\}$ for each shared variable in $\mathbb{X}$ where Value is the domain of the corresponding shared variable. Let *view* be a variable of type View. For a shared variable $x \in \mathbb{X}$, we will refer to the entries of x in *view* as *view_x_t*, *view_x_v* and *view_x_l* respectively. We will explain the use of *view_x_l* in the *Write Statements* paragraph.

• **Message**. This is a data-structure which is used to store the messages generated by the *write* events of a run in RA. This data-structure has a variable of type View and also holds the unique address of one of the shared variables which signifies the variable that was written to. For a variable m of type Message, we will refer to its address holding variable as *m_var* and its variable of type View as *m_view*. Each variable m of type Message is a tuple consisting of *m_var*, *m_view_var_t*, *m_view_var_v* and *m_view_var_l*.

• **Local and Global Variables**. Using the data structures above, we describe the local and global variables in *Prog′*.

• $\langle$**ADD_LVARS**$\rangle_{\mathrm{K}}$. In addition to local variables defined in *Prog*, we keep a variable *view* of type View which is initialised by INIT_PROC for each process. We also use a variable *sim* to detect and limit context switches.

• $\langle$**ADD_GVARS**$\rangle_{\mathrm{K}}$. To store the messages and make them available to all processes, we have an array message_store of size K with entries of type Message. message_store is used to simulate the memory pool of RA while translating RA to SC. The bound K on the size of message_store is explained in paragraph **Challenges**. We have a variable messages_used which tracks the number of messages that have been "published" to message_store so far. Publishing a message of type Message to the message_store simulates the addition of a new message to the memory pool in RA. The *write* event of any process involves updating its local variable of type View and non-deterministically deciding to publish the write to the array message_store. Since message_store has a bounded size, not all the writes will be published. Details about this non-determinism can be found in the *Write Statements* paragraph. We maintain a boolean array, avail_x for each shared variable $x \in \mathbb{X}$, of size |Time| which keeps track of the time-stamps that have been used. It is initialized to *true* for all variables and all indices except 0. We store the number of context switches and the number of view-altering events so far in variables *s_SC* and *s_RA* respectively.

The Code-to-Code Translation. The map $[\![s]\!]_{\mathrm{K}}$ replaces each occurrence of a shared variable in s by its corresponding local copy in View and appropriately uses *Read Statements* and *Write Statements* which modify View non-deterministically.

Algorithm 1 : Main

```
1: atomic_begin;
2: $s\_SC \leftarrow 0$
3: $s\_RA \leftarrow 0$
4: for all $x \in \mathbb{X}$ do
5:     $avail\_x[0] \leftarrow false$ ▹ 0 is the initial time-stamp of all the variables
6:     for all $i \in \{1, \ldots 2K\}$ do
7:         $avail\_x[i] \leftarrow true$
8:     end for
9: end for
10: $messages\_used \leftarrow 0$ ▹ tracks the number of messages added to the array
11: atomic_end;
```

Write Statements. To translate a write ($x = \$r$) to variable x in process p, one of the following is done.

• We update the local variable *view_x_v* to be the value of the current write, and set *view_x_l* to false (Algorithm 2, lines 12-13). In this case, we do not update the time stamp *view_x_t*. The fact that *view_x_l* is set to false tells us that the time stamp corresponding to this write is not among the $2K$, and hence cannot be used in any comparisons.

• We update *view_x_v* to be the value of the current write, update *view_x_t* as a number in $[1 + view\_x\_t, 2K]$, and set *view_x_l* to true. In this case, we guess that this time stamp will be used later in a comparison. This is done in Algorithm 2, lines 2-7). If we further guess that this time stamp is that of an essential message, then we publish this view (Algorithm 2, lines 8-10). To publish, we add a new message in the array message_store as follows. If *view_var_l* is true for all variables *var*, then we copy *view_var_t* to *m_view_var_t*, *view_var_v* to *m_view_var_v* and *view_var_l* to *m_view_var_l*. This publishes the current view of p to the memory pool, by adding an entry to the array message_store (Algorithm 3). Note that checking *view_var_l* is true for all variables ensures that all time stamps *view_var_t* are exact, i.e., the time stamp was chosen when the value *view_var_v* was written to variable *var*. This check ensures that the view is publishable.

Read Statements. To translate a read ($\$r = x$) in process p, we first guess if this read is view-altering (Algorithm 4, lines 1-6). If we guess the read as view-altering, we first check if *view_x_l* is true (Algorithm 5). This ensures that the time stamp of x in the view of p, *view_x_t* is legitimate, and can be used for a comparison with the time stamp in an essential message. Then we look at the array message_store and choose an entry to update *view_x_t* and *view_x_v*. If the chosen entry is s.t. $view\_x\_t \leq m\_view\_x\_t$, then we check if *view_var_l* is true for all variables. This ensures that all time stamps *view_var_t* are legitimate and can be compared with the corresponding entry *m_view_var_t*. We then update *view_var_t*, *view_var_v* whenever $view\_var\_t \leq m\_view\_var\_t$. On guessing that the read is not view-altering, we simply use the value of x stored in p, *view_x_v* to update the value of $\$r$ (Algorithm 4, line 7).

Algorithm 2 : Translating $[\![x = \$r\ ]\!]^{\text{write}}_{\text{K}}$

```
1: if (*) then    ▹ guess if the time-stamp of this write is one of the $2K$
2:     $new\_stamp \leftarrow nondet\_int(1 + view\_x\_t, 2K)$
3:     assume(avail_x[$new\_stamp$])
4:     avail_x[$new\_stamp$] $\leftarrow false$
5:     $view\_x\_t \leftarrow new\_stamp$
6:     $view\_x\_l \leftarrow true$
7:     $view\_x\_v \leftarrow val(\$r)$
8:     if (*) then    ▹ guess if this message is in Es($\mathcal{M}$)
9:         PUBLISH($x$, View)
10:    end if
11: else
12:     $view\_x\_v \leftarrow val(\$r)$
13:     $view\_x\_l \leftarrow false$ ▹ mark that $view\_x\_t$ no longer corresponds to the latest write
14: end if
```

Algorithm 3 : Publish

```
1: procedure PUBLISH($x$, View)
2:     for $y \in \mathbb{X}$ do
3:         assume($view\_y\_l$)
4:     end for
5:     assume(messages_used $< K$)
6:     message_store[messages_used] $\leftarrow Message(x, \text{View})$
7:     messages_used $\leftarrow$ 1 + messages_used
8: end procedure
```

Algorithm 4 : Translating $[\![\$r = x\ ]\!]^{\text{read}}_{\text{K}}$

```
1: if (*) then
2:     assume(s_RA $< K$)    ▹ read from another process's write
3:     $message\_num \leftarrow nondet\_int(0, messages\_used - 1)$
4:     UPDATE_VIEW($x$, $message\_num$)
5:     s_RA $\leftarrow$ 1 + s_RA
6: end if
7: $val(\$r) = view\_x\_v$
```

Algorithm 5 : Update_View

```
1: procedure UPDATE_VIEW($x$, $message\_num$)
2:     $m \leftarrow$ message_store[$message\_num$]
3:     assume($m\_var == \&x$)    ▹ this checks that this message was a write to x
4:     assume($view\_x\_l$)
5:     assume($view\_x\_t \leq m\_view\_x\_t$)
6:     for all $y \in \mathbb{X}$ do
7:         assume($view\_y\_l$)
8:         if $view\_y\_t \leq m\_view\_y\_t$ then
9:             $view\_y\_v \leftarrow m\_view\_y\_v$    ▹ update the time-stamps and the values
10:            $view\_y\_t \leftarrow m\_view\_y\_t$
11:        end if
12:    end for
13: end procedure
```

6 Implementation

In this section, we discuss an implementation that materializes the ideas described in the previous section. The reachability problem is encoded as the problem of detecting assertion failures in a C program. Our tool, VBMC, takes as input the C program to be analysed and translates it to an SC program under a supplied view bound (K). The tool is built as an extension of the Lazy Cseq framework [15] and we use CBMC version 5.10 as a sequential verification backend [11]. CBMC is suitable for us since it can handle non-determinism. Other tools, like the ones based on Stateless Model-Checking techniques, offer limited support and do not handle non-determinism to the full extent. Fences in the input programs are treated as CAS operations to a special variable ([24]). As an optimisation, we do not allow a process in the translated program to context switch until it writes to a shared variable.

CBMC, and hence VBMC, does not offer an option to generate all possible executions that lead to assertion failures but it allows us to find all the assertions that can fail. It also provides the option of stopping the analysis at the first detected assertion failure and print that trace. As CBMC performs bounded model checking, it requires that all loops have a finite upper run-time bound. This is handled by unrolling each loop L times, where L is provided as an input like K. The restriction of bounded loops is imposed because of the choice of the SC-verifier and not because of the translation procedure. At a high level, VBMC considers a meaningful subset of possible executions defined by K and checks them for assertion violations. If it returns that the program is SAFE, then the program does not violate any assertions for that subset of the executions. If it returns UNSAFE, then the program has a trace which uses only K view-switches and violates one of the assertions. This subset can be increased iteratively, by increasing K, to find bugs in real world programs.

7 Evaluation

Table 1. Comparison on the original unfenced versions of mutual exclusion protocols. The loop unrolling parameter used here is $L = 2$.

Program	VBMC	Tracer	Cdsc	Rcmc
bakery	0.5	0.01	0.01	0.08
burns	0.15	0.01	0.01	0.06
dekker	0.85	0.01	0.02	0.09
lamport	2.8	0.05	0.01	0.05
peterson_0	0.26	0.01	0.01	0.03
peterson_0(3)	0.95	0.01	0.01	0.22
sim_dekker	0.16	0.01	0.01	0.08
szymanski_0	0.4	0.03	0.01	0.06

In this section, we report experimental results which illustrate the effectiveness of our technique and also show that bugs can be found with small K. We compare the performance of VBMC with three other state-of-the-art tools, Tracer [3] Cdschecker [32] and Rcmc [18] for programs under RA. Cdschecker , Tracer and Rcmc implement Stateless Model Checking techniques that support RA semantics, and we will collectively refer to them as SMC tools based techniques. These tools essentially consider the set of all executions and then quotient this set with some equivalence relations, which helps in reducing the number of executions that need to be checked. For space reasons, we will use the acronym Cdsc for Cdschecker in tables. Rcmc has two options: Rc11 and Wrc11. The Rc11 option generates only consistent executions by maintaining total coherence orders. The Wrc11 may generate inconsistent executions, which are not validated. We use the Rc11 option for the results in this section. Like VBMC, these tools also require that all loops have a finite upper run-time bound. We engineer the benchmarks so that all the tools consider L iterations as the upper bound for the loops.

Table 2. Comparison of time to find a bug on Peterson_1(i) and Szymanski_1(i). These versions are obtained by fencing $i - 1$ threads of Peterson_0(i) and Szymanski_0(i) respectively.

Program	VBMC	Tracer	Cdsc	Rcmc
peterson_1(4)	4.75	0.08	0.01	0.04
peterson_1(6)	26.73	0.10	0.02	0.1
peterson_1(8)	203	T.O	T.O	307
peterson_1(10)	2400	T.O	T.O	T.O
szymanski_1(4)	2.51	0.01	0.01	69
szymanski_1(6)	8.16	28.14	144	T.O
szymanski_1(8)	32.67	T.O	T.O	T.O
szymanski_1(10)	83	T.O	T.O	T.O

Table 3. **Peterson_2(N)**: Fenced version of Peterson protocol with N threads and a one line change to a fixed thread.

Program	VBMC	Tracer	Cdsc	Rcmc
peterson_2(3)	1.19	0.01	0.05	0.02
peterson_2(4)	3.43	0.04	0.25	0.03
peterson_2(5)	9.13	28.88	27.76	0.03
peterson_2(6)	22.83	T.O	T.O	0.04
peterson_2(7)	47.94	T.O	T.O	0.06

Table 4. **Peterson_3(N)**: Fenced version of Peterson protocol with N threads and a one line change to the last thread .

Program	VBMC	Tracer	Cdsc	Rcmc
peterson_3(3)	1.19	0.01	0.01	0.07
peterson_3(4)	3.17	0.02	0.01	0.45
peterson_3(5)	8.26	0.36	1.73	0.53
peterson_3(6)	19.31	0.1	0.65	T.O
peterson_3(7)	38.62	T.O	T.O	T.O

We conduct all experiments on a Debian 4.9.30-2+deb9u5 machine with a Intel Core i5-5257U CPU(2.7GHz) and 2GB of RAM. All the tools are run with options such that they stop executing as soon as a bug is detected. Rcmc does this by default and for Tracer, we used the argument -s. Cdschecker does not have such an option, so we use a modified version of the tool that stops at the first bug [3]. While reporting the time for bug detection, we ignore the translation time for all the tools. For VBMC, we report the time taken by CBMC to analyse the translated program. The translation time is negligible for reasonably complex programs and scales linearly with the size. For all the benchmarks, we set the time-out value to 3600 seconds and use T.O in tables to signify that the tool timed out on that benchmark. We use seconds as the unit of measurement of time throughout the paper and will tacitly assume it. Our benchmarks are based on mutual exclusion protocols and since we use various versions of the same protocols in our benchmarks, we will use the suffix "_i" to define the ith version of that protocol. For protocols where we do not change anything, this suffix is omitted. There is no change in the version when we change the number of threads in a protocol, and the number of threads (say t) is specified by adding (t) at the end, the default being two threads. For instance, peterson_2(5) means version 2 of the Peterson protocol on 5 threads.

We first applied VBMC to a set of litmus benchmarks, taken from [4]. Litmus tests are standard benchmarks that are used for sanity checks of the correctness of the tools. We were able to successfully run all 4004 of them, with $K \leq 5$, excluding the ones with address calculations. The output result returned by VBMC matches the ones returned by the Herd tool together with the RA-axioms provided in [24].

UNSAFE Cases. Let us first compare the performance of the tools in the UNSAFE cases. Table 1 shows the results for the unfenced versions of 8 mutual exclusion protocols taken from SV-COMP [2018] (the TACAS Software Verification competition 2018). The fenced versions of these protocols are known to be SAFE .

Table 5. **Szymanski_2(N)**: Fenced version of Szymanski protocol with N threads and a one line change to a fixed thread.

Program	VBMC	Tracer	Cdsc	Rcmc
szymanski_2(3)	1.02	0.01	0.03	0.08
szymanski_2(4)	2.33	4.28	20.2	0.13
szymanski_2(5)	4.1	T.O	T.O	8.2
szymanski_2(6)	7.54	T.O	T.O	T.O
szymanski_2(7)	10.3	T.O	T.O	T.O

VBMC was able to find bugs in all of them with $K = 2$. Other tools are able to find bugs faster than VBMC because of the fact that the ratio of buggy executions to total executions is in the range of $0.1 - 0.5$ in all of the above examples. Comparing performance on these cases is not meaningful because

Table 6. Comparison of the performance in fenced versions of mutual exclusion protocols. For this table $K = 2$, $L = 1$

Program	VBMC	Tracer	Cdsc	Rcmc
bakery	1.32	0.04	0.1	0.15
lamport	2.5	0.03	0.05	0.12
tbar	0.46	0.01	0.01	0.07
tbar(3)	1.63	0.08	0.35	0.3
peterson_4(2)	0.61	0.01	0.01	0.06
peterson_4(3)	2.72	122.7	651	200.3

Table 7. Comparison of the performance in fenced versions of mutual exclusion protocols. For this table $K = 2$, $L = 2$

Program	VBMC	Tracer	Cdsc	Rcmc
bakery	3.2	131	443	85
lamport	17.9	2910	T.O	1340
tbar(2)	0.93	0.03	0.09	0.07
tbar(3)	2.9	T.O	T.O	T.O
peterson_4(2)	1.36	0.56	2.23	0.67
peterson_4(3)	5.57	T.O	T.O	T.O

Table 8. Comparison of the performance in fenced versions of mutual exclusion protocols. For this table $K = 2$, $L = 4$

Program	VBMC	Tracer	Cdsc	Rcmc
bakery	8.1	T.O	T.O	T.O
lamport	268	T.O	T.O	T.O
tbar(2)	1.9	32	209	39
tbar(3)	6.1	T.O	T.O	T.O
peterson_4(2)	3.36	T.O	T.O	T.O
peterson_4(3)	12.6	T.O	T.O	T.O

even a stochastic simulation of the RA model that explores executions randomly will be able to find bugs quickly since the probability is high. Also, our backend CBMC takes some time to process the input program and then translate it into a formula. Though this time is negligible in more complex programs, it does affect the results here.

Robustness Checks for UNSAFE Cases. To test the robustness of tools, we use the loop unrolling parameter(L), number of threads(N) and some other modifications that control the above probability. Comparing the performances of the considered tools with different L, N is useful in analysing the effect of this probability and also in demonstrating the orthogonal way in which the tools consider executions. The performance trend increasing L or N will also illustrate the scalability of the tools. Besides showing the scalability of our method, we use L,N and some modifications to check the robustness of the tools and to control the probability of finding the bug. The first modification is to only leave one thread unfenced. Since it is known that the fully fenced protocols are SAFE , having just one unfenced thread would enforce all buggy executions to go through that thread. This will decrease the number of buggy runs. This probability will decrease further if we increase the total number of threads. We refer to these new benchmarks as peterson_1 and szymanski_1. The results for this modification on Peterson and Szymanski protocols are shown in Table 2.

In the modified Peterson protocol, VBMC required $K = 4$ and it starts doing better than others after the number of threads are increased to 8. In the modified Szymanski protocol, VBMC could find the bug with $K = 2$. In this case, VBMC scales much better than others as the number of threads is increased and manages to outperform other tools from szymanski_1(6) onwards. Tracer suffers a blow up of 10^3 from szymanski_1(4) to szymanski_1(6) and in fact times out on szymanski_1(8). Cdschecker also suffers from this and times out on szymanski_1(8). Though not shown in the table, VBMC was able to find a bug in szymanski_1(20) in about 1200 seconds. Rcmc performs poorly on this benchmark as compared to other SMC tools and times out for szymanski_1(6) itself. The next set of benchmarks unveil the reason behind this observation.

We now consider another modification of the SAFE (fenced) version of the Peterson protocol. This time, we make a one line change in a fixed thread to introduce a bug in the SAFE versions. Since, all buggy executions need to go through that thread, the probability of a random execution being buggy is low. The effect is further amplified by increasing the number of threads.

The results for this set are shown in Table 3. On these benchmarks, we see that Tracer and Cdschecker do not perform well as the number of threads is increased. VBMC scales very well in this case too. Rcmc on the other hand appears to be unaffected by the increase in number of threads and finds the bug very quickly. The difference is because of different search strategies used by Rcmc, Cdschecker and Tracer . Rcmc coincidentally finds one of the buggy traces early. To further investigate, we changed the position of the same bug to the last thread. The results for this case are shown in Table 4. Here, we see that Tracer and Cdschecker perform better than in the previous case. Rcmc is not resilient to positional change and does not scale well with the increase of the number of threads. VBMC, however, is unaffected by this and still finds the bug in reasonable time.

We also considered a similar modification to the Szymanski protocol. szymanski_2 is the version in which all the threads are fenced but with a one line change done in a fixed thread. The results are shown in Table 5. Unlike for peterson_2 where Rcmc was able to find bugs quickly, Rcmc is not able to find bugs in szymanski_2(6) within the time limit. Tracer and Cdschecker do not scale up in this case as well and time out on szymanski_2(5). VBMC found bugs in szymanski_2(7) in 10 seconds with $K = 2$ and is not affected severely by small changes in number of threads.

SAFE Cases. We will now analyse the performance and scalability of our tool on SAFE cases. These cases are important because they indicate the efficiency of covering the search space of executions. We consider the fenced versions of some of the mutual exclusion protocols. Unless mentioned, the number of threads in these benchmarks is 2. In order to increase the search space, we increase the loop unrolling parameters and then compare the performance. The results for $L = 1$, $L = 2$ and $L = 4$ are shown in Tables 6, 7 and 8 respectively. While the performances of SMC tools maybe better than VBMC for $L = 1$, the blow up they incur on just doubling the size of the program is huge. This effect is especially prominent when the number of threads is 3. All SMC tools were able to declare tbar(3) as SAFE in under a second for $L = 1$, while none of them managed to solve for $L = 2$ under 3600 seconds. The performance for peterson(3) was worse than VBMC at $L = 1$ itself. VBMC scales much better with code size and increase in number of threads. Although our technique does not guarantee full safety, it does guarantee safety for a meaningful subset of traces.

VBMC versus SMC Tools. From the analysis of the UNSAFE cases, we conclude that the way SMC tools and VBMC consider executions is quite different. This orthogonality of search strategy can be very useful for complex programs in which termination of SMC tools, though guaranteed in theory, is often infeasible. Although SMC techniques are highly optimised in terms of validating the number of executions in a given time frame, they sometimes fail to find simple bugs due the time they spend in exploring a bug-free region. Note that on a different set of benchmarks, this may be the case with VBMC as well. This depends on the input instance, the scheduling heuristic of the algorithms and the percentage of executions that are buggy. We also conclude that our technique is robust to changes in the number of threads and positioning of the bug and that VBMC and SMC based tools use two orthogonal techniques.

8 Conclusion

We developed an effective model checking approach for RA semantics using a *view-bounding* criterion, under which we reduce the reachability under RA to the context bounded reachability under SC. This reduction allows us to leverage existing tools for discovery of bugs under RA. We developed a tool VBMC, which uses this bounding criterion and we apply it on various benchmarks to affirm that bugs can be found with small number of view-switches. We showed that our approach to finding bugs is orthogonal to the DPOR algorithms. Another benefit of our technique is that it can localize the essential messages that contributed to the erroneous trace and separates them from redundant ones, which will make it easier to fix the bug.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2017. Context-Bounded Analysis for POWER. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10206. Springer, 56–74.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Jonathan Cederberg. 2013. Analysis of Message Passing Programs Using SMT-Solvers. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings (Lecture Notes in Computer Science)*, Dang Van Hung and Mizuhito Ogawa (Eds.), Vol. 8172. Springer, 272–286.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *PACMPL* 2, OOPSLA, 135:1–135:29.

[4] 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.

[5] ARM. 2012. ARM architecture reference manual, ARMv7-A and ARMv7-R edition.

[6] 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.

[7] 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)*, Helmut Seidl (Ed.), Vol. 7211. Springer, 26–46.

[8] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. 2011. Getting Rid of Store-Buffers in TSO Analysis. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 99–115.

[9] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66.

[10] Pierre Chambart and Philippe Schnoebelen. 2008. Mixing Lossy and Perfect Fifo Channels. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings (Lecture Notes in Computer Science)*, Franck van Breugel and Marsha Chechik (Eds.), Vol. 5201. Springer, 340–355.

[11] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science)*, Kurt Jensen and Andreas Podelski (Eds.), Vol. 2988. Springer, 168–176.

[12] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422.

[13] IBM. May 2013. Power ISATM version 2.07.

[14] Intel. 2014. Intel 64 and IA-32 architectures software developer's manual.

[15] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 807–812.

[16] 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)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 17:1–17:29.

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

[18] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *PACMPL* 2, POPL (2018), 17:1–17:32.

[19] Dexter Kozen. 1977. Lower Bounds for Natural Proof Systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 254–266.

[20] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2008. Context-Bounded Analysis of Concurrent Queue Systems. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 299–314.

[21] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 477–492.

[22] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2010. The Language Theory of Bounded Context-Switching. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings (Lecture Notes in Computer Science)*, Alejandro López-Ortiz (Ed.), Vol. 6034. Springer, 96–107.

[23] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2010. Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 629–644.

[24] 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.

[25] Akash Lal and Thomas W. Reps. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design* 35, 1 (2009), 73–97.

[26] L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.* C-28, 9 (1979).

[27] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 495–512.

[28] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 378–391.

[29] P. E. McKenney. September 2005. Memory ordering in modern microprocessors, part II. *Linux Journal* 137 (September 2005).

[30] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 446–455.

[31] Jarek Nieplocha and Bryan Carpenter. 1999. ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems. In *Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico, USA, April 12-16, 1999, Proceedings (Lecture Notes in Computer Science)*, Vol. 1586. Springer, 533–546.

[32] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 131–150.

[33] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51.

[34] 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

[35] Emil L. Post. 1946. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.* 52 (1946), 264–268.

[36] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.), Vol. 3440. Springer, 93–107.

[37] 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 (Lecture Notes in Computer Science)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 940–967.

[38] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[39] Ermenegildo Tomasco, Truc Lam Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models. In *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings (Lecture Notes in Computer Science)*, Alessandro Cimatti and Marjan Sirjani (Eds.), Vol. 10469. Springer, 185–202.

[40] D. Weaver and T. Germond. PTR Prentice Hall, 1994. The SPARC Architecture Manual Version 9.