Verification of C11 Programs with Relaxed Accesses

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

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

Adwait Godbole
IIT Bombay, India
adwait@cse.iitb.ac.in

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

Viktor Vafeiadis
MPI-SWS
viktor@mpi-sws.org

Abstract

In POPL'17, Kang et al. introduced the *promising* semantics for relaxed-memory concurrency (PS-RLX), the first memory model supporting many features of the relaxed fragment of the C++ concurrency model while satisfying the DRF guarantee. PS-RLX uses a consistency check that prevents *semantical* deadlocks. However, this check comes at the price of making the verification of even simple programs practically infeasible. This is due to the unbounded number of runs that need to be checked in order to validate the promises. In this paper, we propose a new consistency definition called *strong consistency* semantics which (1) captures most of the common program transformations performed by the relaxed fragment of C++, (2) is deadlock free (i.e., all promises will eventually be fulfilled), and (3) does not require the analysis of an unbounded number of runs. Then, we show that the reachability problem under the promising semantics with the (strong) consistency definition is highly complex. Given this high complexity, we consider a bounded version of the reachability problem. To this end, we bound both the number of promises and the "view-switches", i.e, the number of times the processes may switch their local views of the global memory. We provide a code-to-code translation from an input program under PS-RLX to a program under SC. This leads to a reduction of the bounded reachability problem under PS-RLX to the bounded context-switching problem under SC. We have implemented a prototype tool and tested it on a set of benchmarks, demonstrating that many bugs in programs can be found using a small bound.

Keywords Model-Checking, weak memory models, Relaxed Semantics

1 Introduction

An important long-standing open problem in PL research was to define a 'good' weak memory model for capturing the semantics of concurrent 'relaxed' memory accesses in languages like Java and C/C++. A model is considered 'good' if it can be implemented efficiently (i.e., if it supports all usual compiler optimizations and its accesses are compiled to plain x86/ARM/Power/RISCV accesses), and is "easy" to reason about. The latter is not formally defined. Instead, the literature uses various proxies such as supporting basic invariant reasoning or the DRF guarantee [21], which states that programs without races exhibit only SC-behavior.

After many attempts at solving this problem (e.g., [6, 8, 12, 19, 21, 25, 30]), a breakthrough was achieved by Kang et al. [13], who introduced the *promising semantics* (PS). PS was the first model that supported basic invariant reasoning, the DRF guarantee, and even a non-trivial program logic [28]. In PS, the memory is modeled as a set of timestamped messages, each corresponding to a write made by the program. Each process/thread records its own view of the memory—i.e., the latest timestamp for each memory location that it is aware of. When reading from memory, it can either return the value stored at the timestamp in its view or advance its view to some larger timestamp and read from that message. When a process t writes to memory location x, PS creates a new message with a timestamp larger than t's view of x, and t's view is advanced to include the new message. In addition, in order to allow load-store reorderings, PS allows a process to *promise* to produce a certain write in the future. PS uses a *consistency* check to ensure that every promised message can be *certified* (i.e., made fulfillable) by executing that process on its own. Furthermore, this should hold from any future memory (i.e., from any extension of the memory with additional messages). The quantification prevents deadlocks (i.e., processes from making promises they are not able to fulfil). PS generally allows program executions to contain unboundedly many concurrent promised messages, provided that all of them can be certified. As one can immediately see, PS is a fairly complex model, and beyond its support for some basic reasoning patterns, it is not at all obvious whether it is easy to reason about concurrent programs running under PS. Furthermore, the unbounded number of future memories, that need to be checked, makes the verification of even simple programs practically infeasible. However, as mentioned above, the quantification over all future memories is necessary to ensure the absence of deadlocks. A challenging problem is then to find a consistency definition that (1) captures most of the common program transformations performed by the relaxed fragment of C++, (2) is deadlock-free (i.e., all promises will eventually be fulfilled) and (3) does not quantify over all future memories.

, ,
2019.

Towards this goal, we propose a new consistency definition, called *strong consistency* semantics, for the relaxed fragment of the promising semantics (PS-RLX), which satisfies all the three requirements listed above. Roughly speaking, the new (strong) consistency check requires that promises can be fulfilled only from the current memory (i.e., no need for quantification over all possible future memories) by a run that does not (1) add new messages with non-maximal timestamp and (2) execute atomic Compare-And-Swap instructions. We show that strong consistency implies the standard consistency (as defined in [13]). Furthermore, in the case where the program *Prog* does not contain any atomic Compare-And-Swap instructions, we show that the two semantics coincide. As an immediate consequence, we have that any behavior under PS-RLX with the strong consistency definition is also a behavior under PS-RLX with the (standard) consistency definition. This implies that PS-RLX with the strong consistency definition is deadlock-free.

Then, we consider the reachability problem for programs running under PS-RLX. This is a challenging problem since even if each process is a finite state system, the program's state space is unbounded because the memory can contain unboundedly many messages and each message has a timestamp whose size is also not bounded. Furthermore, a program under PS-RLX can make an unbounded number of promise steps, whose certification can further take an unbounded number of steps. All these aspects make the reachability problem very difficult. In fact, we show the reachability problem under PS-RLX using anyone of the two consistency definitions is highly complex: it is non-primitive recursive.

Given this high complexity, we next consider a bounded version of the reachability problem for PS-RLX. We bound both the number of promises and, following [1], the number of "view switches" (i.e., the number of times that a process reads from a message it has not previously seen). We develop a practical verification algorithm for this bounded reachability problem via a reduction to SC reachability under bounded context-switching [27].

This reduction is implemented in a tool, called SwInG. Our experimental results in §6 demonstrate the effectiveness of our approach. We exhibit cases where hard-to-find bugs are detectable using a small view-bound K. Our tool displays resilience to trivial changes in the position of bugs and the order of processes. Moreover, our experimental results confirm our hypothesis that the standard definition of consistency (as defined in [13]) would not scale while strong consistency performs much better.

Related Work As stated in the introduction, the promising semantics is the first model to support DRF guarantees and invariant reasoning. Given this, the verification of programs running under the promising semantics is a fundamental question, which has not been considered before. To the best of our knowledge, SwInG is the first tool for automated verification of programs under the promising semantics [13] and the strong semantics. Most of the existing work concerns the development of stateless model checking (SMC), coupled with (dynamic) partial order reduction techniques (e.g., [3, 14, 15, 23, 24]) and do not handle promises as defined in [13].

Context-bounding has been proposed in [27] 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., [9, 16–18, 20, 22]). In the context of weak memory models, context-bounded analysis has been only proposed to programs running under TSO/PSO in [5, 29] and under POWER in [2].

In our bounded reachability verification procedure, we adapt the view-bounding approach proposed in [1] for programs under release-acquire semantics to the promising semantics. Our code to code translation to bounded context SC is much more complex than the one in [1] because in addition to executing instructions, a process can perform various other roles like making and certifying promises as well as checking consistency. The main challenge in the code-to-code translation of [1] was to keep track of the causality between different variables. In our case, the challenge is fundamentally different and is to provide a procedure that (i) guesses the promises non-deterministically in a manner that guarantees consistency after each step, and (ii) verify that each promise so guessed is fulfilled.

As future work, a practical verification in RC11 in the presence of both relaxed and release-acquire semantics is definitely possible, albeit technically challenging because of the differences in the two view-switch notions we have versus [1]. We hope to address this in future by finding a uniform view switch concept that is compatible with the two semantics as well as with the semantics of SC accesses.

2 Preliminaries

In this section, we introduce the simple programming language and the notation that will be used throughout.

Notations. Given two natural numbers $i, j \in \mathbb{N}$ s.t. $i \leq j$, we use $[i, j]$ to denote the set $\{k \mid i \leq k \leq j\}$. Let A and B be two sets. We use $f : A \rightarrow B$ to denote that f is a function from A to B. We define $f[a \mapsto b]$ to be the function f' such that $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$. Given a set $A' \subseteq A$, we use $f|_{A'}$ to denote the function from A' to B such that $f|_{A'}(a) = f(a)$ for all $a \in A'$. For a binary relation R, we use $[R]^*$ to denote its reflexive and transitive closure. Given an alphabet Σ, we use Σ^* (resp. Σ^+) to denote the set of possibly empty (resp. non-empty) finite words over Σ. Let $w = a_1 a_2 \cdots a_n$ be a word over Σ, we use $|w|$ to denote the length of w. Given an index i in $[1, |w|]$, we use $w[i]$ to denote the i^{th} letter of w. Given two indices i and j s.t. $1 \leq i \leq j \leq |w|$, we use $w[i, j]$ to denote the word $a_i a_{i+1} \cdots a_j$. Sometimes, we consider a word as a function from $[1, |w|]$ to Σ.

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

Figure 1. Syntax of concurrent programs.

Program Syntax. A program *Prog* (see Fig. 1) consists of a set $\mathcal{X}$ of (global) variables, followed by the definition of a set $\mathcal{P}$ of processes. Each process p declares a set $\mathcal{R}(p)$ of (local) *registers* followed by a sequence of labeled instructions. We assume that these sets of registers are disjoint and we use $\mathcal{R} := \cup_p \mathcal{R}(p)$ to denote their union. We assume also a (potentially unbounded) data domain $\mathbb{D}$ from which the registers and global variables take values. All global variables and registers are assumed to be initialized with the special value $0 \in \mathbb{D}$ (if not mentioned otherwise).

An instruction $\mathfrak{i}$ is of the form $\lambda : \mathfrak{s}$ where λ is a unique label and $\mathfrak{s}$ is a statement. We use $\mathbb{L}_p$ to denote the set of all labels of the process p, and $\mathbb{L} = \bigcup_{p \in \mathcal{P}} \mathbb{L}_p$ the set of all labels. We assume that the execution of the process p starts always with a unique initial instruction labeled by λ^p_{init}. A write instruction is of the form $x = \$r$, and assigns the value of register $\$r$ to the global variable x. A read instruction $\$r = x$ conversely reads the value of the global variable x into the local register $\$r$. A blocking compare-and-swap (bcas) instruction takes the form $\texttt{bcas}(x, \$r_1, \$r_2)$ and waits until the value of the global variable x matches that of register $\$r_1$ and when it is the case, it atomically assigns the value of register $\$r_2$ to x. A *local* assignment instruction $\$r = exp$ assigns to the register $\$r$ the value of *exp*, where *exp* is an expression over a set of operators, constants as well as the contents of the registers of the current process, but not referring to the set of global variables. The fence instruction SC-fence is used to enforce sequential consistency if it is placed between two memory access operations. Finally, the *conditional*, *assume* and *iterative* instructions (collectively called *cai* instructions) have the standard semantics. We define $\mathbb{L}^W_p$ (resp. $\mathbb{L}^W$), $\mathbb{L}^R_p$ (resp. $\mathbb{L}^R$), $\mathbb{L}^{\texttt{bcas}}_p$ (resp. $\mathbb{L}^{\texttt{bcas}}$) and $\mathbb{L}^{\text{SC-fence}}_p$ (resp. $\mathbb{L}^{\text{SC-fence}}$) as the subsets of $\mathbb{L}_p$ (resp. $\mathbb{L}$) corresponding to write, read, bcas and SC fence instructions, receptively.

Given a label λ of a process p, let $\texttt{next}(\lambda)$ denote the labels of the next instructions that can be executed by p. With the exception of *cai* instructions, $\texttt{next}(\lambda)$ contains at most one element: it contains no elements for the last instruction(s) of the process, in which case we write $\texttt{next}(\lambda) = \bot$. In the case of *cai* instructions, $\texttt{next}(\lambda)$ contains at most two elements (assume can be thought of as a while loop). We define $\texttt{Tnext}(\lambda)$ (resp. $\texttt{Fnext}(\lambda)$) 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 use $\texttt{Tnext}(\lambda) = \bot$ and $\texttt{Fnext}(\lambda) = \bot$ to denote the termination of the process execution. For simplicity, we sometimes write assume(x = exp) instead of $\$r = x$; assume($\$r$ = exp) (for a register $\$r$ that is not otherwise used in the program). This notation is extended in the straightforward manner to conditional statements.

3 Promising Semantics(PS-RLX)

In the following, we present the PS-RLX memory model, which defines the semantics of global variable accesses. PS-RLX is obtained from the promising semantics [13], by restricting attention to relaxed accesses and SC fences.

In order to correctly model relaxed accesses, PS-RLX dispenses with the standard SC understanding of memory as a function from global variables to values. Instead, it represents memory as a set of messages, each denoting the effect of a single write or compare-and-swap instruction. Although the memory is shared, each process has its own view of the memory, since it is aware only of a subset of the messages it contains. In the absence of SC fences, these views can be radically different: the only constraint enforced is that messages to the same variable are totally ordered, so that processes cannot disagree on the order in which they perceive them. Finally, messages can be added to the memory either by executing the next instruction of a process or by *promising* a future write—that is, immediately adding to memory a message that could otherwise only be added after executing a bunch of instructions. As we will shortly see, promises hold the key to PS-RLX because they allow load-store reordering, and pose significant challenges to verification.

Timestamps. PS-RLX uses timestamps to maintain a total order over all the writes to the same variable. We assume an infinite set of timestamps Time, densely totally ordered by $\leq$, with 0 being the minimum element. A *view* is a function $V : \mathcal{X} \rightarrow \text{Time}$ that maps each variable to a timestamp. We use $\mathcal{T}$ to denote the set of all view functions. Let V_{init} represent the initial view where all variables are mapped to 0. Let $\mathcal{I}$ denote the set of intervals over Time. The intervals in $\mathcal{I}$ have the form $(f, t]$ where either $f = t = 0$ or $f < t$, with $f, t \in \text{Time}$. Given an interval $I = (f, t] \in \mathcal{I}$, $I.\texttt{frm}$ and $I.\texttt{to}$ denote f, t respectively.

Memory. In PS-RLX, the memory is modelled as a set of messages, where each message represents the effect of one write or compare-and-swap instruction. In more detail, a message m is a tuple $(x, v, (f, t])$ where $x \in \mathcal{X}$, $v \in \mathbb{D}$ and $(f, t] \in \mathcal{I}$. We use $m.var$, $m.val$, $m.\texttt{to}$ and $m.\texttt{frm}$ to denote respectively x, v, t and f. Two messages are said to be *disjoint* ($m_1 \perp m_2$) if they concern different variables ($m_1.var \neq m_2.var$) or their intervals do not overlap ($m_1.\texttt{to} \leq m_2.\texttt{frm}$ or $m_2.\texttt{to} \leq m_1.\texttt{frm}$). Two sets of messages M, M' are disjoint, denoted $M \perp M'$, if $m \perp m'$ for every

$m \in M, m' \in M'$. Two messages m_1, m_2 are *adjacent* denoted $\mathsf{Adj}(m_1, m_2)$ if $m_1.var = m_2.var$ and $m_1.\mathtt{to} = m_2.\mathtt{frm}$.

A memory M is a set of pairwise disjoint messages. A memory M can be extended with a message $m = (x, v, (f, t])$ in a number of ways:

Additive insertion $M \overset{A}{\hookleftarrow} m$ is defined if $M \perp \{m\}$ and returns $M \cup \{m\}$.

Maximal additive insertion $M \overset{Am}{\hookleftarrow} m$ is defined if $M \perp \{m\}$ and $m.\mathtt{to} > m'.\mathtt{to}$ for all $m' \in M$, and returns $M \cup \{m\}$. The maximal additive insertion is a special case of the additive insertion that we will need to check the consistency of the promises.

Splitting insertion $M \overset{S}{\hookleftarrow} m$ is defined if there exists $m' = (x, v', (f, t'])$ with $t < t'$ in M, in which case it results in M being updated to $M \overset{S}{\hookleftarrow} m = (M \backslash \{m'\} \cup \{m, (x, v', (t, t'])\})$.

Fulfilment insertion $M \overset{F}{\hookleftarrow} m$ is defined if $m \in M$, in which case it returns M unchanged.

Machine States. A machine state $\mathcal{MS}$ is a tuple $(J, R, \mathsf{View}, PS, M, G)$, where $J : \mathcal{P} \mapsto \mathbb{L}$ maps each process p to the label of the next instruction to be executed, $R : \mathcal{R} \to \mathbb{D}$ maps each register to its current value, $\mathsf{View} : \mathcal{P} \mapsto \mathcal{T}$ maps each process to its view of the memory, M is a memory, $PS : \mathcal{P} \mapsto 2^{M}$ maps each process to a set of messages (called promise set), and $G \in \mathcal{T}$ is the global view (that will be used by SC fences). Let C denote the set of all machine states.

Given a machine state $\mathcal{MS} = (J, R, \mathsf{View}, PS, M, G)$ and a process p, we use $\mathcal{MS}{\downarrow}p$ to denote $(J(p), R|_{\mathcal{R}(p)}, \mathsf{View}(p), PS(p), M, G)$, the projection of $\mathcal{MS}$ to the process p. The first four entries in $\mathcal{MS}{\downarrow}p$ constitute the process state. We call $\mathcal{MS}{\downarrow}p$ the process configuration. Let C_p denote the set of all process configurations.

The initial machine state $\mathcal{MS}_{\text{init}}$ is one where: (1) each process p is in its initial instruction; (2) all registers have value 0; (3) each process has the initial process view (that maps each variable to 0); (4) the set of promises is empty; (5) the initial memory M_{init} contains exactly one initial message $(x, 0, (0, 0])$ for each variable x; and (6) the initial global view maps each variable to 0.

Transition Relation. We next explain the transition relation between process configurations, from which we will induce the transition relation between machine states.

Process Relation. We define the transition relation induced by the process p as a relation $\xrightarrow[p]{} \subseteq C_p \times (\mathbb{L}_p \cup (\mathbb{L}_p \times \{A, Am, S, F\}) \cup \{\mathtt{prm}\}) \times C_p$ between the configurations of a given process p. For an instruction $\lambda : \mathfrak{s}$ of a process p and two process configurations $\mathfrak{c} = (\lambda, R, V, P, M, G)$ and $\mathfrak{c}' = (\lambda', R', V', P', M', G')$, we write $\mathfrak{c} \xrightarrow[p]{\lambda:\mathfrak{s}} \mathfrak{c}'$ to denote that $(\mathfrak{c}, \lambda, \mathfrak{c}') \in \xrightarrow[p]{}$. For a write or bcas instruction $\lambda : \mathfrak{s}$ of a process p and $a \in \{A, Am, S, F\}$, we write $\mathfrak{c} \xrightarrow[p]{(\lambda:\mathfrak{s}, a)} \mathfrak{c}'$ to denote that $(\mathfrak{c}, (\lambda, a), \mathfrak{c}') \in \xrightarrow[p]{}$. The letter $a \in \{A, Am, S, F\}$ is used to distinguish the different ways a write/bcas instruction is executed where A, Am, S, and F stand for *Additive*, *Maximal Additive*, *Splitting* and *Fulfilment*. Similarly, we write $\mathfrak{c} \xrightarrow[p]{\mathtt{prm}} \mathfrak{c}'$ to denote that $(\mathfrak{c}, \mathtt{prm}, \mathfrak{c}') \in \xrightarrow[p]{}$. The relation $\xrightarrow[p]{}$ is defined through a set of inference rules given in Figure 2. Below, we explain these inference rules.

• The Read rule handles the case when process p executes a read instruction $\lambda : \$r = x$. For the read to be successful, there must be some message of the form $(x, v, (f, t])$ in the global memory such that $V(x) \leq t$ (i.e., process p must not be aware of a later message for x). In this case, the value v is assigned to $\$r$ and the timestamp of the read message is incorporated into p's view. The current instruction of process p gets updated to $\mathtt{next}(\lambda)$. The global memory M, the set of promises P, and the global view G remain the same.

• The Write rule handles the case when a write instruction $\lambda : x = \$r$ is executed. Let v be the value of $\$r$ (i.e., $v = R(\$r)$). To perform this instruction, there must exist an unused interval $(f, t]$ s.t. $V(x) \leq f$. Then, there are three cases, depending on the set of promises P of p.

- (Maximal) Additive Insertion: If the new message $(x, v, (f, t])$ is disjoint from the memory M (i.e., $\{(x, v, (f, t])\} \perp M$), then we add $m = (x, v, (f, t])$ to M to obtain the new global memory $M \overset{A}{\hookleftarrow} m$ (or $M \overset{Am}{\hookleftarrow} m$ if we are using the maximal additive insertion operation). The view of p is updated to $V[x \mapsto t]$. Notice that $(P \overset{a}{\hookleftarrow} m) \backslash \{m\}$ leaves P unchanged.
- Splitting Insertion: Let $m = (x, v, (f, t])$. To use splitting insertion, there should exist a message $m' = (x, v', (f, t''])$ in $P \subseteq M$ with $t < t''$. Then $M \overset{S}{\hookleftarrow} m$ results in $M \backslash \{m'\} \cup \{m, (x, v', (t, t''])\}$ while $(P \overset{S}{\hookleftarrow} m) \backslash \{m\}$ results in $P' = (P \backslash \{m'\}) \cup \{(x, v', (t, t''])\}$. To add m to the memory, we modify m' in the promise set and the memory, and extend the memory with m.
- Fulfilment Insertion: Let $m = (x, v, (f, t])$. To use fulfilment insertion of m, the message m should be in $P \subseteq M$. Then $M \overset{F}{\hookleftarrow} m$ results in M while $(P \overset{F}{\hookleftarrow} m) \backslash \{m\}$ results in $P' = (P \backslash \{m\})$. Essentially, we keep the memory the same and we remove m from the set of promises.

The current instruction and view of p are respectively updated to $\mathtt{next}(\lambda)$, and $V[x \mapsto t]$.

• The CAS rule executes a compare-and-swap instruction of the form $\lambda : \mathtt{bcas}(x, \$r_1, \$r_2)$. To perform the bcas instruction, there must be a message $m = (x, R(\$r_1), (f, t]) \in M$ such that $V(x) \leq t$. Let $m' = (x, R(\$r_2), (t, t'])$. Then we have

$$\frac{(x, v, (f, t]) \in M,\ V(x) \le t}{(\lambda, R, V, P, M, G) \xrightarrow[p]{\lambda:\$r=x} (\mathtt{next}(\lambda), R[\$r \mapsto v], V[x \mapsto t], P, M, G)}\ \texttt{Read}$$

$$\frac{m = (x, R(\$r), (f, t]),\ V(x) \le f,\ P' = (P \overset{a}{\hookleftarrow} m)\backslash\{m\},\ M' = M \overset{a}{\hookleftarrow} m}{(\lambda, R, V, P, M, G) \xrightarrow[p]{(\lambda:x=\$r, a)} (\mathtt{next}(\lambda), R, V[x \to t], P', M', G)}\ \begin{array}{c}\texttt{Write}\\ a \in \{A, Am, S, F\}\end{array}$$

$$\frac{\begin{array}{l}(x, R(\$r_1), (f, t]) \in M,\ V(x) \le t,\\ m = (x, R(\$r_2), (t, t']),\ P' = (P \overset{a}{\hookleftarrow} m)\backslash\{m\},\ M' = M \overset{a}{\hookleftarrow} m\end{array}}{(\lambda, R, V, P, M, G) \xrightarrow[p]{(\lambda:\mathsf{bcas}(x,\$r_1,\$r_2), a)} (\mathtt{next}(\lambda), R, V[x \to t'], P', M', G)}\ \begin{array}{c}\texttt{CAS}\\ a \in \{A, Am, S, F\}\end{array}$$

$$\frac{P' = P \overset{a}{\hookleftarrow} m,\ M' = M \overset{a}{\hookleftarrow} m}{(\lambda, R, V, P, M, G) \xrightarrow[p]{\mathtt{prm}} (\lambda, R, V, P', M', G)}\ \begin{array}{c}\texttt{Promise}\\ a \in \{A, S\}\end{array}$$

$$\frac{}{(\lambda, R, V, P, M, G) \xrightarrow[p]{\lambda:\text{SC-fence}} (\mathtt{next}(\lambda), R, V \sqcup G, P, M, V \sqcup G)}\ \texttt{SC fence}$$

Figure 2. PS-RLX inference rules at the process level, defining the transition $(\lambda, R, V, P, M, G) \xrightarrow[p]{\alpha} (\lambda', R', V', P', M', G')$ where $p \in \mathcal{P}$ and α is one of the labels used above. The merge operation $\sqcup$ returns the pointwise maximum of the two views, i.e., $(V \sqcup V')(y)$ is the maximum of $V(y)$ and $V'(y)$.

three cases obtained by using m' in place of $(x, v, (f, t])$ in the explanation of the write operation for $a \in \{A, Am, S, F\}$.

- The `SC-fence` rule concerns the execution of an SC fence. In such cases, the process view $V(p)$ is compared to global view G and they both get updated to the maximum of the two using the merge operation $\sqcup$. Formally, the merge operation $\sqcup$ between two views V and V' is defined as follows: for any variable $y \in \mathcal{X}$, $(V \sqcup V')(y) = V'(y)$ if $V'(y) \ge V(y)$, and $V(y)$ otherwise.
- The `Promise` rule enables process p to promise any message m that can be added to both P and M by an additive or a splitting insertion.

Besides these rules shown in Figure 2, there are inference rules for the other instructions (assignments, assumes, conditionals, and iterations). These are defined in the usual way and affect only the label of the instruction to get executed and the values of its registers.

Machine Relation. Now we are ready to define the induced transition relation between machine states using the process transition relations defined in the previous paragraph. For that, let $\mathsf{INFR} = (\mathbb{L}_p \cup (\mathbb{L}_p \times \{A, Am, S, F\}) \cup \{\mathtt{prm}\})$ and

$$\overset{p}{\Rightarrow} \overset{\text{def}}{=} \bigcup_{\alpha \in \mathsf{INFR}} \xrightarrow[p]{\alpha}, \text{ and } \Rightarrow \overset{\text{def}}{=} \bigcup_{p \in \mathcal{P}} \overset{p}{\Rightarrow}$$

This induces a relation between machine states as follows. For machine states $\mathcal{MS} = (J, R, \mathsf{View}, PS, M, G)$ and $\mathcal{MS}' = (J', R', \mathsf{View}', PS', M', G')$, we write $\mathcal{MS} \overset{p}{\Rightarrow} \mathcal{MS}'$ iff (1) $\mathcal{MS}{\downarrow}p \overset{p}{\Rightarrow} \mathcal{MS}{\downarrow}p$ and $(J(p'), R|_{\mathcal{R}(p')}, \mathsf{View}(p'), PS(p')) = (J'(p'), R'|_{\mathcal{R}(p')}, \mathsf{View}'(p'), PS'(p'))$ for all $p' \ne p$.

Consistency. There is one final requirement on machine states called *consistency*, which roughly states that in every machine state encountered in a program execution, all the messages promised by a process p can be *certified* (i.e., made fulfillable) by executing p on its own from any future memory, i.e., any extension of the memory with additional messages. The quantification over all the future memory ensures that the current execution will not *deadlock* due to the impossibility of the fulfilment of a promise. In other words, a process cannot make any promises that it is not able to fulfil.

According to Kang et al. [13, §4], during the certification of promises, a process cannot make any further promises, execute any SC fences. We call such steps *consistent steps*, $\xrightarrow[p]{\text{cons}} \overset{\text{def}}{=} \bigcup_{\alpha \in \mathsf{INFR} \backslash \{\mathtt{prm}, \mathbb{L}_p^{\text{SC-fence}}\}} \xrightarrow[p]{\alpha}$.

A machine state $\mathcal{MS} = (J, R, \mathsf{View}, PS, M, G)$ is *consistent* if, from any future memory M' such that $M \subseteq M'$, every process $p \in \mathcal{P}$ can certify/fulfil all its promises by performing consistent steps, i.e., $(J(p), R, \mathsf{View}(p), PS(p), M', G) [\xrightarrow[p]{\text{cons}}]^* (\lambda, R', V', \emptyset, M'', G')$.

3.1 Quantification over all Future Memories

The purpose of the introduction of the quantification over future memories in Kang et al. [13, §4] is to prevent deadlocks (i.e., all promises will eventually be fulfilled). However, this comes at the price of making the verification of even simple programs practically infeasible. This is due to the unbounded number of future memories that need to be checked.

As mentioned in the introduction, the challenge that we consider in this paper is to find a consistency definition that (1) captures common program transformations performed by C++, (2) is deadlock free, and (3) does not quantify over future memories.

We can achieve (3) by simply dropping the quantification over future memories and instead only requiring that the set of promises can be certified from the current memory. However, this will introduce deadlocks. To see why, consider the following example:

bcas(x,0,1); y:=1;	assume(y = 1) bcas(x,0,1);	(Deadlock-c)

In the above example, the first process can promise to set y to 1 (if we do not consider all possible future memories during the certification phase). Now the second process can atomically update the value of the variable x from 0 to 1 which results in forbidding the first process to execute its `bcas` instruction and so the promise can be never fulfilled.

The deadlock that we face in this example is caused by the use of `bcas` during the certification phase. Thus, a potential fix is to disallow `bcas`. Unfortunately, this is not sufficient to prevent deadlocks; as illustrated by the following example:

x=2;
assume(x=1);
y:=1;
	x:=1
	assume(y = 1)
		x:=3	(Deadlock-w)

In the above example, let us assume that the second process executes its write instruction which results in a new message in the memory of the form $(x, 1, (1, 2])$. Then, the first process can promise $(y, 1, (1, 3])$. This is possible since this promise can be certified when we allow additive insertion of the `write` operation $x := 2$ in the certification phase. Next, the assume instruction assume$(y = 1)$ of the second process can be executed. After that, the third process performs its write instruction which results in a new message in the memory of the form $(x, 3, (0, 1])$. Now, the first process cannot fulfil its promise anymore, since the timestamp associated to its write instruction $x = 2$ should be smaller than the one of the write instruction $x = 1$. However, there is no such available timestamp due to the message $(x, 3, (0, 1])$ of the third process. The previous example suggests that we also need to disallow the additive insertion of `write` operations with non-maximal timestamp. Interestingly, this is all what we need to achieve (2), i.e., preventing deadlocks. In Section 3.2, we show (1) is also achieved.

In the following, we formally define this new semantics (called here *strong consistency*). In this model, during the certification of promises, we allow only to add writes with maximal timestamps; while `bcas` operations, promises and SC-fences are disallowed. We call these steps *strong consistent steps*, $\rightarrow_p^{\mathsf{scons}} \stackrel{\text{def}}{=} \bigcup_{\alpha \in \mathsf{INFR} \setminus \{\mathsf{prm}, (\mathbb{L}_p^W, A), (\mathbb{L}_p^{\mathsf{bcas}}, Am), (\mathbb{L}_p^{\mathsf{bcas}}, A), \mathbb{L}_p^{\mathsf{SC\text{-}fence}}\}} \xrightarrow[p]{\alpha}$. Then, a machine state $\mathcal{MS} = (J, R, \mathsf{View}, PS, M, G)$ is *strongly consistent* if $\mathcal{MS}{\downarrow}p\ [\rightarrow_p^{\mathsf{scons}}]^*\ (\lambda, R', V', \emptyset, M', G')$.

Theorem 3.1. *If a machine state is* strongly consistent *then it is also consistent. Furthermore, in the case where the program Prog does not contain any* `bcas` *instruction, we have that if a machine state is consistent then it is also strongly consistent.*

A proof of Theorem 3.1 is in the supplement. As an immediate consequence of Theorem 3.1, the strong consistency definition is deadlock-free since the (standard) consistency is deadlock-free.

3.2 Comparison of the two notions of consistency

In the following, we describe how strong consistency captures the common program transformations performed by C++ (as in Kang et al. [13, §4]).

Consider the following two variants of the "load buffer" litmus test:

a:=x; y:=1;	b:=y x:=b	(LB)	a:=x; y:=a;	b:=y x:=b	(LBd)

In the LB litmus test, C++ allows to assign 1 to the register a. Such behavior can also be observed in our semantics with the strong consistency definition. To see why, consider a run where the first process (whose code on the left side) promises to write 1 to y. Such a promise can be certified by that process. Then, the second process can read from the promise that the value of y is 1 and set the variable x to 1. Finally, the first process can fulfil its promise by setting y to 1. In the LBd litmus test, it is desirable to not observe that the value of the register a is 1. It is indeed the case in our semantics (with the strong consistency definition) since the first process cannot promise that the value of y is 1.

Let us now consider the following variant of LBd :

a:=x; y:=a+1-a;	b:=y x:=b	(LBfd)

In the LBfd litmus test, C++ allows to assign 1 to the register a. Such behavior is also allowed by our semantics with the strong consistency definition by exactly proceeding in the same way as in the case of the LB litmus test.

As an immediate consequence of Theorem 3.1, any observed behavior under PS-RLX with the strong consistency definition is also a behavior under PS-RLX with the (standard) consistency definition. Furthermore, any forbidden behavior under PS-RLX with the (standard) consistency definition is also a forbidden behavior under PS-RLX with the strong consistency definition. However, PS-RLX with the (standard) consistency definition allows strictly more behaviors than PS-RLX with the strong consistency definition as we will see in the next paragraph. This can be observed when we use `bcas` operations during the certification phase where the values read by these operations are somehow irrelevant.

To see the difference between the two consistency definitions, let us consider another variant of the LB litmus test where we add a `bcas` operation in the code of the first process between its read and write operations.

a:=x;
bcas(x,a,a);
y:=1;
	b:=y
	x:=b
		(LBcu)

The `bcas` operation can succeed for any value of x. This allows the first process to promise that the value of y is 1 under PS-RLX with the (standard) consistency definition since for any future memory, the first process sets the variable y to 1. Then, the execution continues exactly in the same way as in the case of the LB litmus test to observe that the value of a is 1. Such behavior is not possible under PS-RLX with the strong consistency definition since the first process cannot promise that the value of y is 1 (because we disallow the use of `bcas` operations during certification).

Now, let us consider a variant of LBcu where the `bcas` operation can only succeed for some particular values.

a:=x;
bcas(x,0,1);
y:=1;
	b:=y
	x:=b
		(LBcd)

In LBcd litmus test the `bcas` needs to read a particular value of the variable x and therefore the first process cannot promise to set the value of y to 1 under PS-RLX with the (strong) consistency definition for any future memory (i.e., any value of the variable x).

4 The (Strong) Reachability Problem

In this section, we discuss the question of reachability in the (strong) consistency semantics. First, we give the formal definition of the reachability problem under both semantics. Then, we show that the reachability problem under the strong consistency semantics is non-primitive recursive. Given this high complexity, we propose a bounded version of the (strong) reachability problem where we bound both the number of promises and the number of "view switches" (i.e., the number of times that a process reads from a message it has not previously seen).

Formal definition. A strongly consistent run of *Prog* is a sequence of the form: $\mathcal{MS}_0 \ [\stackrel{p_{i_1}}{\Rightarrow}]^* \ \mathcal{MS}_1 \ [\stackrel{p_{i_2}}{\Rightarrow}]^* \ \mathcal{MS}_2 \ [\stackrel{p_{i_3}}{\Rightarrow}]^* \ldots [\stackrel{p_{i_n}}{\Rightarrow}]^* \ \mathcal{MS}_n$ where $\mathcal{MS}_0 = \mathcal{MS}_{\text{init}}$ is the initial machine state and $\mathcal{MS}_1, \ldots, \mathcal{MS}_n$ are (strongly) consistent machine states. In this case, the machine states $\mathcal{MS}_0, \ldots, \mathcal{MS}_n$ are said to be (strongly) reachable from $\mathcal{MS}_{\text{init}}$.

Given an instruction label function $J : \mathcal{P} \rightharpoonup \mathbb{L}$ that maps each process $p \in \mathcal{P}$ to a label in $\mathbb{L}_p$, the *(strong) reachability* problem asks whether there exists a machine state of the form $(J, R, \text{View}, PS, M, G)$ that is (strongly) reachable from $\mathcal{MS}_{\text{init}}$. In the case of a positive answer to this problem, we say that J is (strongly) reachable in *Prog*.

Lower-bound time complexity. As mentioned in Section 3.1, checking reachability is not tractable in practice due to the unbounded number of future memories that need to be considered. In the following, we show that the (strong) reachability problem for concurrent programs under PS-RLX is highly non-trivial (i.e., non-primitive recursive). The proof is done by reduction from the reachability problem for lossy channel systems, in a similar to the case of TSO [4] where we insert SC-fence instructions everywhere in the process that simulates the lossy channel process (in order to ensure that no promises can be made by that process). A detailed proof can be found in the supplement.

Theorem 4.1. *The (strong) reachability problem for concurrent programs under PS-RLX over a finite data domain is non-primitive recursive.*

Bounded (strong) reachability problem. Given the high-complexity of the (strong) reachability problem, we restrict our attention to runs which have bounded number of promises and view-switches. The latter notion was introduced in Abdulla et al. [1] for the release-acquire model. Let us formally define such runs for PS-RLX with the strong consistency definition. The problem can be defined in a similar manner for PS-RLX with the standard consistency definition.

Consider a strongly consistent run ρ of the form $\mathcal{MS}_0 \xrightarrow[p_1]{\alpha_1} \mathcal{MS}_1 \xrightarrow[p_2]{\alpha_2} \mathfrak{c}_2 \ldots \xrightarrow[p_n]{\alpha_n} \mathcal{MS}_n$. A step labeled by α_j is view-altering in ρ if it involves reading a message from the memory which changes the view of p_j w.r.t. some variable. Let Sw be the set $\{i \mid p_i \neq p_{i+1}\}$ recording the points of context switches in ρ. Also, let *Cons* be the set of strong consistency check runs for ρ, i.e., runs of the form $\mathfrak{c}_i{\downarrow}p_i \ [\rightarrow_{p_i}^{\text{scons}}]^* \ \mathfrak{c}'_i$ for $i \in Sw$ where the promise set of p_i is empty in $\mathfrak{c}'_i$.

Let K' be the number of view-switches and promises along ρ, and let K'' by the total number of view-switches in *Cons*. The run ρ is called K-bounded under the relaxed semantics (denoted K-Bd(PS, Vw)−RLX) if $K'' + K' \leq K$. Observe that the messages read during strong consistency checks are not considered as view-switches in the traditional sense (they do not change the view permanently, but are only used locally within that strong consistency check phase).

Finally, given $K \in \mathbb{N}$, the K-(promise, view) bounded strong reachability under PS-RLX can be defined in similar manner to the strong reachability problem by replacing strong runs with the K-bounded ones.

K-Bounded-Context Reachability in SC. Given a program, a run τ under SC is a sequence $\gamma_0 \stackrel{p_1}{\Rightarrow} \gamma_1 \stackrel{p_2}{\Rightarrow} \gamma_2 \cdots \stackrel{p_n}{\Rightarrow} \gamma_n$. A context switch in τ is a machine state γ_j, s.t. $p_{j-1} \neq p_j$. A run τ is K-context-bounded if it contains at most K context switches. The K-bounded reachability under SC is defined by requiring that τ is K-context bounded.

$$
\begin{aligned}
[\![Prog]\!] &:= (\langle\text{global vars}\rangle; \langle\textsc{Main}\rangle; ([\![\texttt{proc}\ p\ \texttt{reg}\ \$r^* i^*]\!])^* \\
[\![\texttt{proc}\ p\ \texttt{reg}\ \$r^*\ i^*]\!] &:= \texttt{proc}\ p\ \texttt{reg}\ \$r^* \langle\text{local vars}\rangle\langle\textsc{InitProc}\rangle\langle\text{CSO}\rangle^{p,\lambda_0}([\![i]\!]^p)^* \\
[\![\lambda : i]\!]^p &:= \lambda : \langle\text{CSI}\rangle; [\![s]\!]^p; \langle\text{CSO}\rangle^{p,\lambda} \\
[\![\texttt{if}\ exp\ \texttt{then}\ i^*\ \texttt{else}\ i^*]\!]^p &:= \texttt{if}\ exp\ \texttt{then}\ ([\![i]\!]^p)^*\ \texttt{else}([\![i]\!]^p)^* \\
[\![\texttt{while}\ exp\ \texttt{do}\ i^*]\!]^p &:= \texttt{while}\ exp\ \texttt{do}\ ([\![i]\!]^p)^* \\
[\![\text{assume}(exp)]\!]^p &:= \text{assume}(exp) \\
[\![\$r = exp]\!]^p &:= \$r = exp \\
[\![x = \$r]\!]^p &:= \text{ see Algorithm 3} \\
[\![\$r = x]\!]^p &:= \text{ see Algorithm 4}
\end{aligned}
$$

Figure 3. Translation map $[\![.]\!]$.

5 Solving the Strong Reachability Under Bounded Promises and View-Switches

Let $\mathbb{K} \in \mathbb{N}$ be a bound on the promises and view-switches. In this section, we propose an algorithm that reduces the $\mathbb{K}$-(promise, view) bounded strong consistent reachability under PS-RLX to a $\mathbb{K} + n$ bounded context reachability problem under SC, where n is the number of processes in the concurrent program. The bounded-context reachability problem under SC for finite-state programs is decidable [27]. In concrete terms, given a concurrent program *Prog* as input, our algorithm constructs a program *Prog*′ having the same variable domain as *Prog* and size polynomial in *Prog* and $\mathbb{K}$ s.t. for every $\mathbb{K}$-(promise, view) bounded strongly consistent run of *Prog* under PS-RLX, there is a $\mathbb{K} + n$ bounded context run of *Prog*′ under SC reaching the same set of instruction labels, and vice-versa.

For the rest of the section, we use ρ_{rel} (resp. τ_{sc}) to denote a run under PS-RLX (resp. SC).

Translation Overview. Let *Prog* be a program under PS-RLX and let $\mathcal{P}$ and $\mathcal{X}$ be its sets of processes and shared variables respectively. Our reduction relies on the translation of *Prog* under the bounded strong consistency semantics to a context-bounded SC program $[\![Prog]\!]$, as shown in Figure 3. The translation keeps the same data domain for local variables, but adds a finite amount of additional global and local states, which we will describe shortly. Besides the new global variables, $[\![Prog]\!]$ also adds a new process (Main) that initializes these variables, and then translates each process in turn. The translation of a process $p \in \mathcal{P}$ adds some local variables, such as the *view* array that records the most recent value and timestamp seen by p for each shared variable $x \in \mathcal{X}$. The function $\langle$InitProc$\rangle$ initializes these local variables. Each instruction i in process p is translated to a sequence of instructions: $\langle CSI \rangle$ that checks if the process is active in the current context; the translation $[\![s]\!]^p$ of the statement s in i; and $\langle CSO \rangle^{p,\lambda}$ that checks switching out of context. $\langle CSO \rangle^{p,\lambda}$ facilitates two things: (i) it allows p to make promises after each λ (possibly in different contexts), s.t. the control is back at λ after the promises; (ii) it helps in certification of promises when p switches out of context from λ. The translation of bcas and SC-fence is discussed in the supplement, to keep the presentation simple. We will elaborate on read, write later.

One of the key ingredients in the translation is to bound the size of the memory. This is done via the notion of essential messages (these messages are either promises or alter the view of processes which read them) detailed below. A bound on the number of time stamps (details below) is achieved from the number of essential messages. Then we describe our data structures, local and global variables, subroutines, and then eventually the translation of each statement.

Essential Messages. Messages in the memory can be classified into three categories: (i) *view-switching messages* (that alter the view of some process when they are read), (ii) *promise messages* (that are generated as a promise by some process and may or may not alter the view of another process), and (iii) *redundant messages* (that are never read by any process). When a new message is created, we can guess the type of the message as one of the above. We need not allocate fresh timestamps for redundant messages. Only essential messages (either view-switching or promise) require fresh timestamps. The bound $\mathbb{K}$ on the number of promises and view switches gives the bound $\mathbb{K}$ on the number of essential messages and their timestamps. For the translation we maintain $2\mathbb{K}$ distinct timestamps. The reason is as follows: for each view-switch of a process, its existing timestamp is compared with that of an essential message. Hence we need 2 timestamps for a view-switch (a promise requires only one timestamp). Since we have at most $\mathbb{K}$ view-switches and promises, $2\mathbb{K}$ timestamps suffice. We choose Time $= \{0, 1, 2, \ldots, 2\mathbb{K}\}$ as the set of timestamps. This bound on the number of timestamps is crucial in the translation.

Data Structures. We use auxilary data structures to represent messages and process views.

The Message data structure represents a message generated by a write or a promise. It is a record with four fields: (i) *var*, the address of the shared variable that was written to; (ii) t, the timestamp in Time associated with the message; (iii) v, the value written; and (iv) *flag*, a number in $\{-1, 0, 1, \ldots, n\}$, where n is the number of processes. Flag 0 represents a non-promised message or a promise that has been fulfilled; flag -1 represents a certified promise; while a positive number *flag* > 0 denotes a (not yet certified) promise by thread *flag*.

The View data structure stores for each shared variable x, (i) a timestamp $t \in$ Time, (ii) a value v written to x, (iii) a boolean $l \in \{\texttt{true}, \texttt{false}\}$ representing whether t is a legitimate timestamp which can be used for comparisons (since we have messages which are not essential, t could represent a timestamp which is not used for comparisons), (iv) a boolean $f \in \{\texttt{true}, \texttt{false}\}$ which represents whether the value v may be used by the same process for a local read, and (v) a boolean $u \in \{\texttt{true}, \texttt{false}\}$ which is true if the process has most recently executed a continuous sequence of bcas instructions. The entries in View for a variable x are referred to as $view[x].t$, $view[x].v$, etc.

Global Variables. We introduce the following global variables: (1) *messageStore*, an array of messages of size $\mathbb{K}$ that will be populated with the essential messages generated by the program; (2) *messagesUsed*, the current number of messages in *messageStore*; (3) *numContexts*, the number of context switches that have occurred; (4) *numEE*, the number of promises and view switches that have occurred; and (5) *avail*, a boolean array of size $2\mathbb{K}|\mathcal{X}|$, that, for each variable $x \in \mathcal{X}$, records the available timestamps in Time. The Main process initializes the global counters to 0 and all entries in the *avail* array to contain true.

Local Variables. In addition to its local registers, each process has the following local variables: (i) *view*: a local instance of View, (ii) *active*: a boolean variable which is set when the process is running in the current context, (iii) *checkMode*: a boolean checking if the process is in the certification mode, (iv) *liveChain*: a boolean array indexed by global variables $x \in \mathcal{X}$, used to ensure no additive insertions of x are allowed during strong consistency checking (however maximal additive insertions are allowed), and (v) *retAddr*: a variable storing the instruction label corresponding to the most recent instruction before entering the certification phase.
Since strong consistency disallows additive insertions, we check that only splitting insertions are used during the certification phase. *liveChain*$[x]$ is true only in certification mode (i.e., when *checkMode* is true) when the most recent write to x during the current certification phase was not promised.

Algorithm 1: Main, CSI, Publish

```
Algorithm Main
    atomic_begin
    messagesUsed, numContexts, numEE ← 0
    for x ∈ X, ts ∈ {1, 2, …, 2K} do
        avail[x][ts] ← true
    end
    atomic_end
Algorithm CSI
    if ¬active then
        atomic_begin
        active ← true
        numContexts ← numContexts + 1
        assume(numContexts ≤ K + n)
    end
Algorithm Publish(message)
    assume(messagesUsed < K)
    messageStore[messagesUsed] ← message
    messagesUsed ← messagesUsed + 1
```

Algorithm 2: $CSO^{p,\lambda}$

```
σ_sw:
if * then
    if ¬checkMode then
        if ¬active then
            atomic_begin
            active ← true
            numContexts ← numContexts + 1
            assume(numContexts ≤ K + n)
        end
        checkMode ← true
        retAddr ← λ, saveState(p)
    else
        for m ∈ messageStore do
            assume(m.flag ≠ p)
            if m.flag == −1 then m.flag ← p
        end
        for x ∈ X do assume(¬liveChain[x])
        loadState(p), gotoLabel(retAddr)
        checkMode ← false
        active ← false
        atomic_end
    end
    goto σ_sw
end
```

When $liveChain[x]$ is true, the process must make the succeeding writes with consecutive timestamps ending with a promise (which will set $liveChain[x]$ to false) before it makes a global read. This precisely forbids additive insertion. $liveChain[x]$ may only be true when *checkMode* is true.

Subroutines.
• genMessage(·, ·, ·, ·) is a subroutine which generates a message with the four fields as specified above in the data structure Message. In case some fields are not specified, these are chosen non-deterministically from the relevant domain.
• saveState(p) is a subroutine which saves the state of global variables (defined above) and the local state of only the process p passed as argument. We however do not store *numEE* and the contents of *messageStore*. (details in the supplement)
• loadState(p) is a subroutine which loads the global state and process p's local state saved using saveState(p).

We use the gotoLabel(*retAddr*) statement which switches to the instruction label indexed by *retAddr*. We note that there are only finitely many instruction labels.

The Code-to-Code Translation. In what follows we illustrate how the translation simulates a run under Bd(PS, Vw)−RLX. At the outset we note that each process interleaves in its execution between two phases: a *normal* phase that runs at the beginning of each context and the *certification* phase at the end of the context, where it may make new promises and certify all the promises before switching out of context. In this way we incorporate the witness for the consistency check in the run of the program itself.

By certification of a promise, we mean an event that shows that the promise can be fulfilled as part of the witness run proving the machine state to be consistent. By fulfilment of a promise we mean making a write that permanently removes the promise message from the promise set. Fulfilment (resp. *Certification*) is only done during the *normal* (resp. *certification*) phase of the run.

Context Switch Out ($CSO^{p,\lambda}$). $CSO^{p,\lambda}$ is placed after each instruction in the original program and serves an entry and exit point for the consistency check phase of the process.

If the process is currently in normal mode, *CSO* non-deterministically switches to certification mode, and vice versa. When switching from normal to certification mode, if the process is not active, first a new context is created and the process is made active. Then, the mode is recorded, the current instruction λ and the local state of the process are recorded so that they can be reinstated at the end of the certification run.

To switch from certification mode back to normal mode, we first check that there are no outstanding promises of p (i.e., all messages in the memory have a flag different from p). For messages with a flag of -1 (denoting a certified promise by p), we set their flag back to p so that they get certified again in subsequent certification rounds.

Then, to preserve the *liveChain* invariant, we enforce that all its entries are false which ensures that there were no additive insertions during the certification phase. Now using the loadState routine, we load back the state that was stored on entering the certification phase. The process then returns to the instruction label from where it entered the certification phase, and *checkMode* is set to false, and it exits the context.

Write Statements. The translation of a write instruction $x = \$r$ of process p is shown in Algorithm 3. Let us first consider execution in the normal phase (i.e., when *checkMode* is false). First, the value of $val(\$r)$ is recorded in the local view, and $view[x].f$ is set meaning that later instructions in p can read from the write. Then, we non-deterministically

Algorithm 3: $[\![x = \$r]\!]^p$ write

```
view[x].v ← val($r), view[x].f ← true
if * then                                /* (i) no fresh timestamp */
    view[x].l ← false
    if checkMode then liveChain[x] ← true
else if * then                                   /* (ii) and (iii) */
    view[x].l ← true
    if liveChain[x] then
        newStamp ← view[x].t + 1
    else
        newStamp ← nondetInt(view[x].t + 1, 2K)
    end
    view[x].t ← newStamp
    assume(avail[x][newStamp])
    avail[x][newStamp] ← false
    if * then                             /* (ii) essential message */
        if checkMode then
            message ← genMessage(x, newStamp, val($r), −1)
            liveChain[x] ← false, numEE ← numEE + 1
        else
            message ← genMessage(x, newStamp, val($r), 0)
        end
        Publish(message)
    else                                                 /* (iii) */
        if checkMode then liveChain[x] ← true
    end
else                                  /* (iv) fulfilling a promise */
    view[x].l ← true
    messageNum ← nondetInt(0, messagesUsed − 1)
    message ← messageStore[messageNum]
    assume(message.var == &x ∧ message.t > view[x].t)
    assume(message.v == val($r) ∧ message.flag == p)
    view[x].t ← message.t
    if checkMode then
        message.flag ← −1, liveChain[x] ← false
    else
        message.flag ← 0
    end
    messageStore[messageNum] ← message
```

choose one of four possibilities for the write: it either (i) is not assigned a fresh timestamp, (ii) is assigned a fresh timestamp and published, (iii) is assigned a fresh timestamp but not published (that is, the message is not added to the memory), or (iv) fulfils some outstanding promise.

In case (i), no message is created, and $view[x].l$ is set to false, signifying that the timestamp recorded in the view does not correspond to the most recent write to x and should therefore not be used in the comparisons.

In cases (ii) and (iii), we allocate a new timestamp and store it into $view[x].t$. We use the *avail* array to ensure that allocated timestamps are unique: we check that the selected timestamp is available (i.e., not allocated), and remove it from the array of available stamps. If the message is to be published (case ii), the appropriate message is constructed and published; otherwise (case iii), this step is skipped.

Finally, if the process decides to fulfill a promise (case (iv)), a message is fetched from *messageStore* and checked to be an unfulfilled promise by the current process (checking $flag = p$), and the flag is set to 0.

Let us now consider a write executing in the certification phase (i.e., when *checkMode* is true).

We will only highlight differences between the normal and certification phase writes. Most importantly, we maintain and use the *liveChain* invariant whenever a fresh timestamp is assigned. Indeed, if *liveChain* is true, the process must assign consecutive timestamps (line 8). Also, when it does not publish the current write as a promise message, or fulfill an older promise (cases (iii) and (iv)), it sets *liveChain* to true (lines 4, 24). In cases (iii) and (iv), the message flag is set to −1 rather than 0, indicating that the promise has been certified, but not yet fulfilled.

Algorithm 4: $[\![\$r = x]\!]^p$ read

```
if * then                                 /* View-switching read */
    assume(numEE < K)
    msgNum ← nondetInt(0, messagesUsed − 1)
    msg ← messageStore[msgNum]
    assume(msg.var == &x)
    assume(view[x].l ∧ view[x].t ≤ msg.t)
    view[x].t ← msg.t, view[x].v ← msg.v
    view[x].f ← true, numEE ← numEE + 1
    assume(¬liveChain[x])
else                                  /* Non-view-switching read */
    assume(view[x].f)
end
val($r) = view[x].v
```

Read Statements. Algorithm 4 is used to translate read statements of the form $\$r = x$. At line 1, the process guesses and takes the **then** branch if the read is view-switching.

In the case of a view-switching load, we check that we have not reached the context-/view-switching bound, we fetch a new message from *messageStore* with a larger timestamp that the one in the current view, update the process view to include that new message, and increment the number of context and view switches. We finally ensure that $liveChain[x]$ is false before the read in order to forbid additive insertions when checking consistency of promises. Recall from the *liveChain* invariant that $liveChain[x]$ is true only when the process is in certification mode and the last write on x was not published as a promise message.

Reading a message from the memory when $liveChain[x]$ is true implies additive insertion during certification, as illustrated by the adjacent code fragment. Assume the process is in the promise certification mode, with $view[x].t$ set to t_1, and let the first write use a timestamp $t_2 > t_1$ with the message not published as promise, with $liveChain[x]$ as true. Now the instruction a:=x uses a message in the memory with a timestamp $t_3 \geq t_2$. If the next write certifies a promise

x:=1;	// t_2
a:=x;	// t_3
x:=2;	// $t_3 + 1$

message, the interval in the message will be $t_3 + 1$, since *liveChain*$[x]$ is true. This results in two writes during the certification, with non-adjacent timestamps $t_2, t_3 + 1$, with *only* the latter being promised. The choice of the timestamps clearly shows additive insertion. Notice that if the earlier write also resulted in a promise message then we do not have additive insertion (since both are promised) and the read with timestamp t_2 is allowed since *liveChain*$[x]$ is false.

If the read is not view-switching, the process checks that the local value is usable (line 13) and loads its local value *view*$[x].v$ into $\$r$. The local value may become unusable if the process crosses an SC-fence which updates its *view*$[x].t$.

6 Implementation and Evaluation

To evaluate the efficiency of the technique presented in the previous section, we have implemented it as a tool called SwInG. SwInG takes as input a C program and a bound, K, and translates it to an SC program. We use CBMC version 5.10 as the backend tool, which takes as input L, the loop unrolling parameter, specifying the number of iterations for which loops are unrolled. SwInG then considers the subset of executions respecting the bounds K and L provided. If it returns *unsafe*, then the program has an unsafe execution in this subset. Conversely, if it returns *safe*, then none of these executions violate any assertion.

In the promise free mode, we compare SwInG with three state-of-the-art stateless model checking (SMC) tools, CDSChecker [23], GenMC [15] and Rcmc [14] that support the relaxed semantics without promises (as defined in [13]). We use a version of CDSChecker that halts on the first bug discovered while GenMC and Rcmc do this by default. In the tables that follow, we specify the used values of L (for all tools) and K (only for SwInG).

The main takeaways of our experiments are: (1) SwInG can uncover hard-to-find bugs faster than the others with relatively small values of K; (2) our approach is more resilient to trivial changes in the position of bugs as compared to the SMC tools; (3) in many instances, our technique fares better at capturing relevant behaviours instead of exploring all possible traces as done by some SMC tools.

We note that the tools we are comparing with do not require as input the bound, K. Hence, the comparison may not be fair for some safe examples, since SwInG only considers the subset of executions which K enforces. However, in certain instances we have set the parameter K such that all executions are considered (modulo the loop unwinding bound). In such cases, we note that SwInG is comparable to the others. We highlight such cases (only for *safe* examples) with a green checkmark (✓) accompanying the value of K used. Additionally, we have put forth cases where we can iteratively increment K to prove correctness. This difference in comparison has no bearing on the reliability of the results.

Considering the above observations, we realise that the SMC tools and SwInG have orthogonal approaches to finding bugs, and can be used to complement each other. SMC tools are limited by how they explore all executions, which might be sub-optimal in cases where we have a shallow counterexample but which is explored only after several executions, while SwInG is limited by the bound K.

We do not consider compilation time for any tool while reporting the results. For SwInG, the time reported is the time taken by the CBMC backend for analysis. The timeout used is 1 hour for all benchmarks. All experiments are conducted on a machine equipped with a 2.80 GHz Intel Core i7-860 and 4GB RAM running a Debian 9 (stretch) 64-bit operating system. We denote timeout by 'TO'. We mark a hyphen '-' in the table for when the process is killed with a maximum resident set size (RAM used) of 3.7 GB or higher.

In the main paper we provide indicative examples of the experiments conducted. The complete set of benchmarks are in the supplement. We first compare strong and standard consistency on some examples. For the remaining benchmarks, to enable comparison with other tools which do not support promises (as defined in [13]), we run the SwInG in the promise-free mode. Then, we show the ability of SwInG: (1) to detect hard-to-find bugs, (2) to adapt to concurrent data-structure benchmarks and (2) resilience to location of bugs and number of executions.

testcase	K	SwInG[strong]	D	SwInG[standard]
splitCAS	5	1.378s	20	12.284s
			40	37.166s
			60	2m15s
			80	4m26s
LBcu	7	4.434s	100	1m13s
			200	2m39s
LB2cu	7	5.331s	10	1m16s
			20	15m40s
LBcd	7	1.003s	100	10.984s
			200	25.010s
fibonacci_2_safe	5	17.244s	10	3m11s
fibonacci_3_safe	5	14m14s	10	TO

Table 1. Comparing the two notions of consistency

Comparing the notions of consistency. In order to empirically confirm our hypothesis that the standard definition of consistency (as defined in [13]) would not scale, we run SwInG, on similar small examples under the strong and standard consistency, while varying the size of the data domain, specified by D. Observe that we need to vary D for the standard consistency definition since it is required during the quantification over all future memories (which implicitly includes all possible data values). We run SwInG on a variety of safe and unsafe test cases from [7, 13]. The first three examples are unsafe while the other ones are safe. In all these cases, we observe, the dependence of run-time on the size of the data domain when the standard consistency definition is used. Strong consistency, on the other hand performs much better without any restriction on the size of the data domain.

Evaluation using parametrized benchmarks. We compare SwInG with CDSChecker, GenMC and Rcmc in Table 2 on three parametrized benchmarks: `ExponentialBug` (from Fig. 2 of [11]), `Fibonacci` and safe and unsafe versions of `Triangular` taken from SV-COMP 2018. In `ExponentialBug`(N) and `Triangular(N)`, the processes compete to write to a shared variable and N represents the number of times a process may write. In `ExponentialBug`(N), the number of executions grows as $O(N!)$, while the fraction of interleavings that expose the bug reduce exponentially with N. In the unsafe version of `Triangular`(N), there is exactly one interleaving that exposes the bug, while the total number of interleavings increases exponentially with N. In `Fibonacci`(N), two processes compute the value of the n^{th} Fibonacci number. In the safe examples, we note that we use a conservative upper bound on the value of K. Hence this table demonstrates the ability of SwInG in exposing hard-to-find bugs as well as its adaptability for safe cases.

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
exponential_25_unsafe	25	10	3.433s	14.737s	4.697s	TO
exponential_50_unsafe	50	10	9.021s	1m6s	1m2s	TO
exponential_70_unsafe	70	10	14.136s	2m52s	4m3s	TO
fibonacci_2_safe	2	✓20	4.045s	8.811s	0.104s	0.133s
fibonacci_3_safe	3	✓20	10.899s	TO	0.984s	4.443s
fibonacci_4_safe	4	✓20	30.475s	TO	41.576s	3m2s
triangular_3_safe	3	✓6	1m3s	18.737s	0.152s	0.290s
triangular_4_safe	4	✓8	4m58s	20m20s.	1.602s	2.282s
triangular_5_safe	5	✓10	8m16s	TO	28.883s	34.819s
triangular_3_unsafe	3	10	9.422s	2.903s	0.126s	0.244s
triangular_4_unsafe	4	10	2m54s	3m25s	1.254s	1.531s
triangular_5_unsafe	5	10	12m23s	TO	21.619s	26.730s

Table 2. Evaluation using parametrized benchmarks

Evaluation using concurrent data structures. We compare the tools in Tables 3 on benchmarks based on concurrent data structures. The first of these is a concurrent locking algorithm from Hehner and Shyamasundar [10]. The second, `LinuxLocks(N)` is a benchmark extracted from the Linux kernel. If not completely fenced, this benchmark is unsafe under the relaxed semantics and we fence all but one lock accesses. The other two are *safe* benchmarks adapted from SVCOMP-2018. The queue benchmark is parameterized by the number of processes and the stack benchmark is parameterized by the size of the stack. The processes operate on these data structures and we check whether certain invariants are maintained. These benchmarks illustrate the ability of our tool to handle concurrent data-structures similar to those seen in real-world examples.

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
hehner2_unsafe	4	5	6.130s	0.028s	0.042s	0.072s
hehner3_unsafe	4	5	26.729s	0.026s	4m4s	1m26s
linuxlocks2_unsafe	2	4	0.748s	0.010s	0.036s	0.081s
linuxlocks3_unsafe	2	4	1.113s	0.013s	0.037s	0.084s
queue_2_safe	4	4	2.141s	0.020s	0.039s	0.079s
queue_3_safe	4	4	9.417s	0.024s	0.053s	0.086s
stack_4_safe	4	4	2.127s	8.313s	0.819s	1.287s
stack_5_safe	5	4	6.467s	5m2s	14.132s	43.903s
stack_6_safe	6	4	24.185s	TO	7m14s	25m44s

Table 3. Evaluation using concurrent data structures

Evaluation using two synthetic safe benchmarks. We compare the tools in Table 4 on adaptations of two synthetic safe benchmarks: `ReaderWriter(N)` (from Norris and Demsky [24]) and `RedundantCo(N)` (from Abdulla et al. [3]). Both these examples involve N processes writing distinct values to a shared variable and one process reading from it. The number of traces in these examples grow as $O(N!)$. The number of possible values for the reads however is just $O(N)$ in the first example and $O(1)$ in the second. The performance of the SMC tools depends on how efficiently they explore the executions. SwInG on the other hand depends on the reads observed, illustrating the point mentioned earlier. We again note that K is chosen conservatively and our tool declares the benchmarks to be safe considering all executions.

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
readerwriter_9	0	✓5	1.068s	0.007s	0.053s	1m17s
readerwriter_10	0	✓5	1.393s	0.007s	0.056s	14m49s
redundant_co_50	50	✓5	3.219s	8.965s	4.143s	TO
redundant_co_70	70	✓5	6.093s	13.843s	18.185s	TO

Table 4. Evaluation using two synthetic safe benchmarks

Evaluation using mutual exclusion protocols. In this section, we consider mutual exclusion protocols from the SV-COMP 2018 benchmarks. The unfenced versions of the protocols are *unsafe*. All the tools considered report a bug for these examples within two seconds. We now consider variations of these benchmarks.

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
peterson1U(4)	1	4	1.868s	0.005s	TO	0.113s
peterson1U(6)	1	4	9.408s	0.005s	TO	0.179s
peterson1U(8)	1	4	43.680s	TO	TO	5.432s
peterson1U(10)	1	4	4m12s	TO	TO	TO

Table 5. Evaluation using mutual exclusion protocols with a single unfenced process

In Table 5, we evaluate the Peterson protocols for N processes and keep all but one process fenced. This leads to a lower fraction of buggy executions. The values of K taken for these benchmarks assert that the bugs can be found (even for non-trivial examples) with small K. We call this example peterson1U and it is parameterized by N.

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
peterson1C(3)	1	2	0.743s	0.012s	0.085s	0.786s
peterson1C(4)	1	2	1.827s	5.032s	TO	4.157s
peterson1C(5)	1	2	4.185s	59m42s	TO	TO
peterson1C(6)	1	2	8.483s	TO	TO	TO
peterson2C(3)	1	2	0.758s	0.005s	0.068s	0.061
peterson2C(4)	1	2	1.848s	0.015s	TO	12.308s
peterson2C(5)	1	2	4.041s	1m36s	TO	TO
peterson2C(6)	1	2	7.562s	TO	TO	TO

Table 6. Evaluation using mutual exclusion protocols with a bug introduced in the critical section of a single process

Table 6 exhibits a pair of benchmarks that exhibit the sensitivity of DPOR-based algorithms to the location of bugs. We consider the completely fenced version of the Peterson protocol. However, we introduce a bug (write a value to a shared variable and read a different value from it) in the critical section of one of the processes. Between the two examples, the only difference is the process in which this bug has been introduced. We call these examples peterson1C and peterson2C and they are parameterized by the number of processes. We can see the difference in the performance of the DPOR-based tools (especially CDSChecker) on the two examples. On the other hand, our tool is resilient to such superficial changes. We note again that K is small.

7 Undecidability

In this section, we show that both the normal and the strong reachability problem for concurrent programs under the relaxed semantics are undecidable even for finite-state programs. The proof is by a reduction from Post's Correspondence Problem (PCP) [26]. Our proof crucially uses promises to ensure that a process cannot skip any writes made by another process. Unlike the undecidability proof in [1] about RA, our proof does not make use of any bcas operations, and so it works even with just plain read and write instructions. It also works even when we restrict our analysis to executions that can be split into a bounded number of contexts, where within each context, only one process is active. Our undecidability result is also *tight* in the sense that the reachability problem becomes decidable when we restrict ourselves to machine states where the number of promises is bounded.

Theorem 7.1. *The (weakly) consistent reachability problem for concurrent programs over a finite data domain is undecidable under the promising semantics with relaxed accesses.*

Undecidability is obtained by a reduction from Post's Correspondence Problem (PCP) [26].

We construct a concurrent program with two processes p_1 and p_2, six shared variables $\mathcal{X} = \{x, y, validate, index, index', term\}$, and two registers $\{\$r, \$r'\}$. The finite data domain of *Prog* is defined as $\mathbb{D} = \Sigma \cup \{0, 1, \ldots, n\} \cup \{\perp, \#\}$, where $\perp$ and # are two special symbols (not in $\Sigma \cup \{0, 1, \ldots, n\}$). All the variables and registers are initialized to zero.

The code of the two processes is given in Figure 4. Depending on the value of the *validate* flag read, process p_1 can run in generation mode (top-level `then` branch) or validation mode (top-level `else` branch). In generation mode, process p_1 writes in sequential manner the sequence of indices (alternated with the special symbol #) to the variable *index* and at the same time writes, letter by letter, the sequence of letters of the word u_i to the variable x each time p_1 sets the variable *index* to i (using the $\mathtt{Module}^{\mathtt{p_1}}_{\mathtt{u_i}}$ procedure). In validation mode, p_1 reads from the variables *index'* and y and writes back what it has read to the variables *index* and x, respectively. The second process proceeds in a similar manner as the `else` branch of the first process: It reads from the variables *index* and x and writes the values reads to *index'* and y, respectively.

Let λ (resp. λ') be the label of the assume(*true*) instruction of p_1 (resp. p_2). We will show that a solution of the PCP problem exists iff we can reach the pair of labels (λ, λ') in the program *Prog*.

Assume that we can (weakly) reach the pair of labels (λ, λ'). The idea behind the reduction is as follows. In order for p_1 to reach label λ, it must execute the `else` branch of its conditional statement. Let us assume it does so. Then, p_1 will read the sequence of indices $i_1, i_2, \ldots, i_k$ written by the process p_2 on the variable *index'*. Let us assume that the process p_2 writes the sequence of indices $j_1, j_2, \ldots, j_m$ on the variable *index'*. Each time that the process p_1 reads an index from the variable *index'*, it writes it back on the variable *index*. The process p_1 (resp. p_2) alternates between writing/reading an index in $\{1, \ldots, n\}$ and the special symbol # in order to make sure that each written index is at most read once. In similar manner, the process p_2 reads the sequence of indices $j_1, j_2, \ldots, j_m$ written by the process p_1 on the variable *index* and it writes it back on the variables *index'*. This implies that the sequence $j_1, j_2, \ldots, j_m$ is a subsequence of $i_1, i_2, \ldots, i_k$ (since the process p_2 can miss reading some written indices by the process p_1) and also that the sequence $i_1, i_2, \ldots, i_k$ is also a subsequence of $j_1, j_2, \ldots, j_m$ (since p_1 can miss reading some written index by the process p_2). Thus, we have that the sequences $i_1, i_2, \ldots, i_k$ and $j_1, j_2, \ldots, j_m$ are the same. Every time the process p_1 (resp. p_2) reads an index i from the variable *index'* (resp. *index*), it (1) tries to read in sequential manner the sequence of letters appearing in v_i (resp. u_i) (alternated with the special symbol #) from the variable y (resp. x), and (2) writes the same sequence of letters to the variable x (resp. y). Using a similar argument as in the case of indices, we can deduce that if p_1 (resp. p_2) writes the words $v_{i_1} v_{i_2} \cdots v_{i_k}$ (resp. $u_{j_1} u_{j_2} \cdots u_{j_m}$), letter by letter (with an alternation with the symbol#), to the variable x (resp. y), then $v_{i_1} v_{i_2} \cdots v_{i_k}$ (resp. $u_{j_1} u_{j_2} \cdots u_{j_m}$) is a subsequence of $u_{j_1} u_{j_2} \cdots u_{j_m}$ (resp. $v_{i_1} v_{i_2} \cdots v_{i_k}$). Thus, if the pair of labels (λ, λ') is reachable then there exist two sequences $i_1, i_2, \ldots, i_k$ and $j_1, j_2, \ldots, j_m$, written, respectively, by p_1 and p_2 such that $i_1, i_2, \ldots, i_k$ is equal to $j_1, j_2, \ldots, j_m$, and $v_{i_1} v_{i_2} \cdots v_{i_k}$ is equal to $u_{j_1} u_{j_2} \cdots u_{j_m}$. Observe that sequence of indices $i_1, i_2, \ldots, i_k$ is non-empty due to the assume statement assume($\$r' \in [1, n]$).

Let us now show the other direction. Let us assume that a solution of the PCP problem exists. This means that there is a sequence of indices $i_1, i_2, \ldots, i_k$ such that $v_{i_1} v_{i_2} \cdots v_{i_k} = u_{i_1} u_{i_2} \cdots u_{i_k}$. Let $w = u_{i_1} u_{i_2} \cdots u_{i_k}$. Let us show that the pair of labels (λ, λ') can be (weakly) reachable in *Prog*. For that aim, consider the following (weakly consistent) run of the program *Prog*: p_2 starts first by setting the variable *term* to 1. Then, p_1 will use the `then` branch of its conditional statement to promise the two following sequence of promises $(index, i_1, (1, 2]), (index, i_2, (2, 3]), \ldots, (index, i_k, (k, k + 1])$ and $(x, w[1], (1, 2]), (x, w[2], (2, 3]), \ldots, (x, w[|w|], (|w|, |w|+$

Process p_1

```
if validate = 0 then
  while term = 0 do
    index = 1
    Module^{p1}_{u1}
    index = #
    . . .
    index = n
    Module^{p1}_{un}
    index = #
  done
  index = ⊥
else
  $r′ = index′
  assume($r′ ∈ [1, n])
  while $r′ ≠ ⊥ do
    if $r′ = 1 then
      Module^{p1}_{v1}
    else if $r′ = 2 then
      Module^{p1}_{v2}
    . . .
    else if $r′ = n then
      Module^{p1}_{vn}
    end if
    assume(index′ = #)
    $r′ = index′
    assume(index′ ≠ #)
  done
  index = ⊥
  assume(true)
end if
```

Process p_2

```
term = 1;
$r = index;
assume($r ∈ [1, n])
while $r ≠ ⊥ do
  if $r = 1 then
    Module^{p2}_{u1}
  else if $r = 2 then
    Module^{p2}_{u2}
  . . .
  else if $r = n then
    Module^{p2}_{un}
  end if
  assume(index = #)
  $r = index
  assume(index ≠ #)
done
validate = 1
index′ = ⊥
assume(true);
```

$\text{Module}^{p_1}_{v_i}$

```
assume(y = v_i[1])
assume(y = #)
assume(y = v_i[2])
. . .
assume(y = v_i[|v_i|])
assume(y = #)
x = v_i[1]
x = #
x = v_i[2]
. . .
x = v_i[|v_i|]
index = i
index = #
```

$\text{Module}^{p_1}_{u_i}$

```
x = u_i[1]
x = #
x = u_i[2]
. . .
x = u_i[|u_i|]
x = #
```

$\text{Module}^{p_2}_{u_i}$

```
assume(x = u_i[1])
assume(x = #)
assume(x = u_i[2])
. . .
assume(x = u_i[|u_i|])
assume(x = #)
y = u_i[1]
y = #
y = u_i[2]
. . .
y = u_i[|u_i|]
index′ = i
index′ = #
```

Figure 4. The code of processes p_1 and p_2.

1]). Observe that p_1 can certify such sequences of promises under the two semantics for relaxed accesses by iterating its iterative statement in the `then` branch of its alternative statements. Once these promises are performed, p_2 reads these two sequences and writes them back to the variables *index′* and y, respectively. p_2 then sets the variable z to 2. Now p_1 can resume its execution by reading the variable z written by the second process and enter its `else` branch of its alternative statement. Then, p_1 will iteratively read the values written by p_2 on the variable *index′* and y and write them back to the variables *index* and x, respectively. By doing this p_1 fulfils also the sequence of promises that has been issued.

Notice that the number of promises made by p_1 is unbounded. Also, the proof uses only 3-context executions, where, following Qadeer and Rehof [27], a context is a contiguous sequence of operations performed by only one process and a k-*context* run, for a given $k \in \mathbb{N}$, is a run that can be partitioned into k contexts.

References

[1] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI 2019*. ACM, 1117–1132.

[2] 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.

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

[4] 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*. ACM, 7–18.

[5] 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.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[7] Soham Sundar Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *PACMPL* 3 (2019), 70:1–70:28.

[8] Karl Crary and Michael J. Sullivan. 2015. A Calculus for Relaxed Memory. In *POPL 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 623–636. https://doi.org/10.1145/2676726.2676984

[9] 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.

[10] Eric C.R. Hehner and R.K. Shyamasundar. 1981. An implementation of P and V. *Inform. Process. Lett.* 12, 4 (1981), 196 – 198. https://doi.org/10.1016/0020-0190(81)90100-9

[11] Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 165–174.

[12] Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Logical Methods in Computer Science* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:33)2019

[13] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189.

[14] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

[15] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *PLDI*. https://doi.org/10.1145/3314221.3314649

[16] 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.

[17] 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.

[18] 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.

[19] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352

[20] 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.

[21] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL 2015*, Jens Palsberg and Martín Abadi (Eds.). ACM, 378–391. https://doi.org/10.1145/1040305.1040336

[22] 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.

[23] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *OOPSLA 2013*. ACM, New York, NY, USA, 131–150. https://doi.org/10.1145/2509136.2509514

[24] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 10 (May 2016), 51 pages. https://doi.org/10.1145/2806886

[25] Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 622–633. https://doi.org/10.1145/2837614.2837616

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

[27] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *TACAS 2005 (LNCS)*, Vol. 3440. Springer, 93–107.

[28] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *27th European Symposium on Programming, ESOP 2018 (LNCS)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[29] 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.

[30] Yang Zhang and Xinyu Feng. 2013. An Operational Approach to Happens-Before Memory Model. In *Seventh International Symposium on Theoretical Aspects of Software Engineering, TASE 2013, 1-3 July 2013, Birmingham, UK*. IEEE Computer Society, 121–128. https://doi.org/10.1109/TASE.2013.24

A Proof of Theorem 3.1

Let us prove that strong consistency implies consistency. Assume that a machine state $\mathcal{MS} = (J, R, \mathsf{View}, PS, M, G)$ is strongly consistent. Then, we have $(\lambda_0, R_0, V_0, P_0, M_0, G_0) \rightarrow_p^{\mathsf{scons}} (\lambda_1, R_1, V_1, P_1, M_1, G_1) \rightarrow_p^{\mathsf{scons}} \cdots \rightarrow_p^{\mathsf{scons}} (\lambda_n, R_n, V_n, P_n, M_n, G_n)$ with $\mathcal{MS}{\downarrow}p \rightarrow_p^{\mathsf{scons}} = (\lambda_0, R_0, V_0, P_0, M_0, G_0)$ and $P_n = \emptyset$. Since $\rightarrow_p^{\mathsf{scons}} \subseteq \rightarrow_p^{\mathsf{cons}}$, we can show that, for any future memory M' such that $M \subseteq M'$, we have $(\lambda_0, R_0, V_0, P_0, M_0', G_0) \rightarrow_p^{\mathsf{cons}} (\lambda_1, R_1, V_1, P_1, M_1'), G_1) \rightarrow_p^{\mathsf{cons}} \cdots \rightarrow_p^{\mathsf{cons}} (\lambda_n, R_n, V_n, P_n, M_n, G_n)$ with $M_0' = M'$. Intuitively, the second consistency run will proceed in the same way as the strong consistency run by reading from the same sequence of messages, performing the same write instructions with splitting, fulfilment or maximal insertions. and `bcas` instructions with splitting or fulfillement insertions.

Now let us assume that the program *Prog* does not contain any `bcas` and that the machine state $\mathcal{MS} = (J, R, \mathsf{View}, PS, M, G)$ is consistent. This means that, for any future memory M' such that $M \subseteq M'$, we have $(\lambda_0, R_0, V_0, P_0, M_0', G_0) \rightarrow_p^{\mathsf{cons}} (\lambda_1, R_1, V_1, P_1, M_1'), G_1) \rightarrow_p^{\mathsf{cons}} \cdots \rightarrow_p^{\mathsf{cons}} (\lambda_n, R_n, V_n, P_n, M_n, G_n)$ with $M_0' = M'$ and $P_n = \emptyset$. This is in particular true for the future memory M' where all the the intermediate holes in M are filled up. This means that in the following consistent run $(\lambda_0, R_0, V_0, P_0, M_0', G_0) \rightarrow_p^{\mathsf{cons}} (\lambda_1, R_1, V_1, P_1, M_1'), G_1) \rightarrow_p^{\mathsf{cons}} \cdots \rightarrow_p^{\mathsf{cons}} (\lambda_n, R_n, V_n, P_n, M_n, G_n)$ no insertion of write operations with non-maximal timestamp has been performed. Thus, we have $(\lambda_0, R_0, V_0, P_0, M_0' \setminus (M_0' \setminus M), G_0) \rightarrow_p^{\mathsf{scons}} (\lambda_1, R_1, V_1, P_1, M_1' \setminus (M_0' \setminus M), G_1) \rightarrow_p^{\mathsf{scons}} \cdots \rightarrow_p^{\mathsf{scons}} (\lambda_n, R_n, V_n, P_n, M_n' \setminus (M_0' \setminus M), G_n)$ and $\mathcal{MS}$ is strongly consistent.

B Proof of Theorem 4.1

In this section, we show the $\mathcal{F}_{\omega^\omega}$-hardness of reachability of PFS-RLX over a finite domain with only `read`, `write` and SC-fence instructions. $\mathcal{F}_{\omega^\omega}$ is a level in the fast-growing hierarchy of recursive functions. The fast growing hierarchy is a class $(F_\alpha)_\alpha)$ of number-theoretic functions indexed by ordinals. Chambart and Schnoebelen (LICS 2008) established the $\mathcal{F}_{\omega^\omega}$ lower bound for the reachability and termination of lossy channel systems.

B.1 The non-primitive recursive lower bound of PFS-RLX without `bcas`

Our proof follows by a reduction from the reachability problem of lossy channel systems.
Lossy Channel Systems. A lossy channel system (LCS) is a tuple $S = (Q, M, C, \Delta)$ where Q is a finite set of states, M is a finite message alphabet, C is a finite set of lossy channels, and $\Delta \subseteq Q \times C \times \{!, ?\} \times M \times Q$ is a finite set of transition rules. A rule of the form $(q, c, !, a, q')$ (respectively $(q, c, ?, a, q')$) is a write (respectively read) transition.

Assume $S = (Q, M, C, \Delta)$ is a LCS with ℓ channels. A configuration of S is a pair $(q, (u_1, \ldots, u_\ell))$ where $q \in Q$ and $u_i \in M^*$ for all $1 \le i \le \ell$. u_i is the sequence of messages contained in channel c_i (reading a message happens at the head of the channel, and writing from the tail of the channel). Two configurations are compared using the subword ordering : $((q, u_1, \ldots, u_\ell) \sqsubseteq (q', u_1', \ldots, u_\ell')) \Leftrightarrow (q = q') \wedge \bigwedge_{i=1}^{\ell} (u_i \sqsubseteq u_i')$

Let $Conf$ represent the set of all configurations. The operational semantics of S is given as a transition system $T_S = (Conf, \rightarrow)$. Let $\sigma = (q, (u_1, \ldots, u_\ell)$ and $\sigma' = (q', (u_1', \ldots, u_\ell')$ be two configurations. Then a perfect step is one of the following.

1. Let $\delta = (q, c_i, a, ?, q')$. Then $\sigma \xrightarrow{\delta} \sigma'$, with $u_i = a u_i'$, and $u_j = u_j'$ for $j \neq i$, or
2. Let $\delta = (q, c_i, a, !, q')$. Then $\sigma \xrightarrow{\delta} \sigma'$, with $u_i' = u_i a$, and $u_j = u_j'$ for $j \neq i$.

Since the channels are lossy, we can have lossy steps too. A lossy step can happen after a perfect read step, and we lose messages arbitrarily from any of the channels. A run is a perfect run if there are no losses in between two perfect steps. Otherwise, the run is lossy. Notice that we have chosen to lose messages after a read and also after a write. The choice of losing a message after a read or after a write or after either (like in our case) are all equivalent and does not impact the complexity result of Chambart and Schnoebelen.
Reachability in LCS. Given states q_1, q_2 in the LCS, the reachability problem asks whether, starting from state q_1 with all channels empty, one can reach state q_2 with arbitrary contents in the channels.
Reduction from LCS to PFS-RLX with only reads and writes. We now present our reduction from an LCS $S = (Q, M, C, \Delta)$ to a concurrent program using only read and write operations, over PFS-RLX semantics. Assume there are ℓ lossy channels in S, and let $Q = \{q_1, \ldots, q_n\}$. Assume that all transitions going out of each state q_i are numbered. Thus, if q_i has k outgoing transitions, then we refer to them as $tran_{i,1}, \ldots, tran_{i,k}$.

We construct a concurrent program with $\ell + 2$ processes. Each channel c_i is modeled using shared variables x_i, y_i. A shared variable $tran$ holds the values of the possible transitions $tran_{11}, \ldots, tran_{nj}$. Finally, a shared variable $reach$ (initialized to false) keeps track of whether we have reached the desired state in LCS. The number of shared variables needed in the construction of the RA program is hence $2|C| + 2$. The domain of the constructed program is the set of states and transitions of the LCS, along with the set of messages M. The processes are as follows.

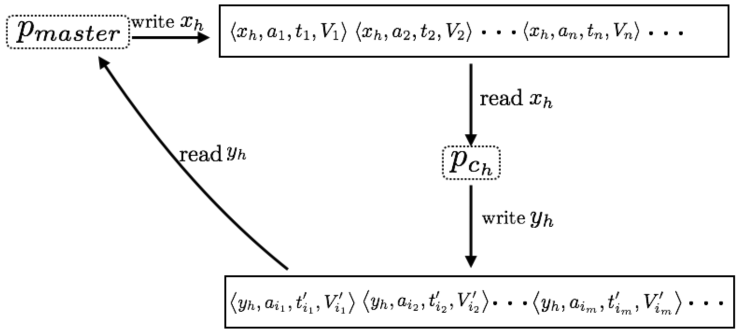

Figure 5. Processes p_{master} and p_{c_h} simulating writes and reads in channel c_h. p_{master} writes to variable x_h simulating a write to channel c_h; p_{c_h} reads x_h and copies it to y_h. p_{master} reads y_h simulating a read from channel c_h.

The Processes

Process p_{tran} : There is a process p_{tran} which repeatedly writes to a shared variable $tran$ (as long as $reach$ is false), the names $tran_{11}, \ldots, tran_{n,j}$ of the transitions in Δ.

Processes p_{master} **and** p_{c_h} :

Given the reachability problem from state q_i to state q_j, the process p_{master} starts by initializing a local register to q_i. It keeps track of the states in the LCS, and the control flow while simulating a run in the LCS starting from q_i. This process simulates the transitions of the LCS depending on the current state. In doing so, p_{master} simulates the read and write transitions and ensures that control moves to the correct next state depending on the choice of the transition. p_{master} does the following repeatedly.

- To begin, p_{master} initializes a local register $\$r$ with the value q_i, if we are interested in reaching a state q_j in the LCS starting from state q_i. At any point of time, $\$r$ holds the name of the state in the LCS where the control flow resides currently. Assume $\$r$ stores the state q_1, and let there be k outgoing transitions from q_1. p_{master} has blocks of code corresponding to each state in the LCS. Each such block has the form `while`($\$r$ == q) `do` ...`done` and simulates an outgoing transition from the current state, and either remains in the same block if the state remains the same, or goes to another block depending on the transition chosen.
  - p_{master} reads the shared variable $tran$. The value which is read must be one of the transitions $tran_{11}, \ldots, tran_{1k}$ since the control resides in the block corresponding to state q_1. Let the value of $tran$ be $tran_{1,j}$,
  - Assume the $1, j$th transition is $(q_1, c_h, a, !, q_i)$. Then, p_{master} writes the value a to the shared variable x_h, and writes the state name q_i into $\$r$. It then exits the block corresponding to q_1 and enters the one corresponding to q_i.
  - Assume the $1, j$th transition is $(q_1, c_h, a, ?, q_i)$. Notice that if p_{master} reads from the variable x_h, it can only read its latest write following the relaxed semantics, since it is the only process which writes to variables $x_1, \ldots, x_n$. This does not simulate the (lossy) channel discipline. To facilitate the proper simulation of the lossy channel c_h, p_{master} must be able to jump to any message in the channel c_h and read it as if that was the head of the channel. To enable p_{master} in doing so, we have a process p_{c_h} which repeatedly reads values of x_h and writes the into y_h. Indeed, p_{c_h} may omit certain values of x_h, copying a proper subset of the values into y_h. p_{c_h} is the only process which reads from x_h, and is the only process which writes to y_h. Likewise, p_{master} is the only process which writes to x_h and reads from y_h. See figure 5.
    To simulate $(q, c_h, a, ?, q')$, p_{master} reads the variable y_h and checks if its value is a. If so, it writes the state name q_i into $\$r$, and then exits the block corresponding to q_1 and enters the one corresponding to q_i. Notice that if p_{c_h} copies x_h to y_h every time p_{master} has written to y_h, then p_{master} has the possibility to read the first value it wrote to x_h (simulating a lossless read). However, p_{master} can choose to read any y_h from the memory pool, and being the sole reader of y_h, ensures the channel discipline, along with the lossiness.
- Once the state q_j is reached in p_{master}, (this is true when p_{master} sets the register $\$r$ to q_j from the current state (say q_k)). Once this is done, p_{master} sets a boolean shared variable $reach$ to true, and reaches $term$. The other processes (p_{tran}, p_{c_h}) check if $reach$ is true, and if so, also reach $term$.

Inserting SC-fence **instructions**. To ensure no promises can be made, each of the above read, write in p_{master} and each p_{c_h} are followed by SC-fence instructions.

Theorem B.1. *The constructed program under PFS-RLX semantics faithfully simulates the LCS : starting from state q_i, we reach state q_j in the LCS iff the instruction term is reached in all processes.*

Example B.2. We illustrate the reduction on an example. Consider the LCS in Figure 6. The constructed program can be seen in Table 7.

p_{tran}	p_{master}	p_{c_1}	p_{c_2}
while $(reach \neq \top)$ do	$\$r = q_1$	while $(reach \neq \top)$ do	while $(reach \neq \top)$ do
$tran = tran_{11}$	while $(reach \neq \top)$ do	$\$r_1 = x_1$	$\$r_2 = x_2$
$tran = tran_{12}$	while$(\$r == q_1)$ do	$y_1 = \$r_1$	$y_2 = \$r_2$
$tran = tran_{13}$	assume$(tran = \bigvee_{i=1}^{3} trans_{13})$	if$(reach == \top)$	if$(reach == \top)$
$tran = tran_{21}$	if$(tran == tran_{11})$	$break$	$break$
$tran = tran_{22}$	$x_1 = a$	end if	end if
$tran = tran_{31}$	else if$(tran == tran_{12})$		
if$(reach == \top)$	$\$r' = y_2$		
$break$	assume$(\$r' = b)$		
end if	$\$r = q_2$ $break$		
	else if$(tran = tran_{13})$		
	$\$r' = y_1$		
	assume$(\$r' = a)$		
	$\$r = q_3$; $reach = \top$; $break$		
	end if		
	done		
	while$(\$r == q_2)$ do		
	…		
	done		
	while$(\$r == q_3)$ do		
	assume$(tran = tran_{31})$		
	$x_1 = b$; $\$r = q_2$; $break$		
	done		
done	done	done	done
$term$	$term$	$term$	$term$

Table 7. Instruction labels have been omitted. To avoid clutter, we have also not written the SC-fence instruction that follows each instruction in p_{master}, p_{c_1} and p_{c_2}. The PFS-RLX program simulating the LCS.

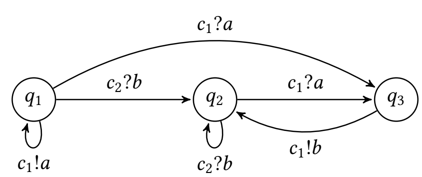

Figure 6. A LCS with 2 lossy channels c_1, c_2 and states q_1, q_2, q_3. The message alphabet is $\{a, b\}$.

As mentioned above, we number the transitions in the LCS depending on their source state. In the LCS given, we have $tran_{11}$ representing the self-loop at q_1, $tran_{12}$ representing the transition from q_1 to q_2 and $tran_{13}$ representing the transition from q_1 to q_3. Likewise, $tran_{31}$ represents the transition from q_3 to q_2, and so on. The domain of the constructed program *Prog* is $\mathcal{D}$=$\{a, b, q_1, q_2, q_3, tran_{11}, tran_{12}, tran_{13}, tran_{21}, tran_{22}, tran_{31}\}$. The shared variables are $\{x_1, y_1, x_2, y_2, tran, reach\}$, of which *reach* is a boolean variable which is initialized to false. We reduce the reachability of LCS to the control reachability problem in *Prog*, and show that starting from q_1, q_3 is reachable in the LCS iff we reach the instruction *term* in all processes.

C Details for Section 5

We first give a glossary of all the variables used in the code. The list contains variables global to all processes or local to a process. A small description of their role is also mentioned, which serve as invariants.

1. *numEE* : global variable, initialized to 0, keeps track of the number pf promises and view switches so far. Each time a promise or a view altering read happens, *numEE* is incremented.
2. *numContexts* : global variable, initialized to 0, keeps track of the number of context switches so far. This is used in the translation to SC.
3. $view[x].v$: local variable, stores the value of $x \in \mathcal{X}$ in the local view of the process
4. $view[x].t$: local variable, stores the time stamp $\in$ Time of $x \in \mathcal{X}$ in the local view of the process.
5. $view[x].l$: local variable, boolean, which is set to true when $view[x].t$ is a valid timestamp, used in comparisons with timestamps of other messages.
6. $view[x].f$: local variable, boolean. A true value indicates that $view[x].v$ is recent, and can be used for reading locally.
7. $view[x].u$: local variable, boolean. A true value indicates that the sequence of events starting from the one that resulted in the timestamp $view[x].t$ till the most recent, form a chain of bcas operations on x. Whenever a write is published, $view[x].u$ is set to true. $view[x].u$ is set to false on an unpublished write. On a sequence of bcas operations, $view[x].u$ is left unchanged.
8. *checkMode* : local variable, boolean. Set to true when the process is in certification phase, which means the process is making and certifying promises.
9. $liveChain[x]$: local variable, for each $x \in \mathcal{X}$, boolean. Can be true only when *checkMode* is true. A true value represents that the last write done while the process is in certification phase is not a published promise message.
10. $extView[x]$: local variable, for each $x \in \mathcal{X}$, boolean. A true value represents that the local value $view[x].v$ of the process comes from a message generated external to the certification phase.
11. $avail[x][]$: for each $x \in \mathcal{X}$, a global boolean array of length $2K + 1$ corresponding to the $2K + 1$ time stamps, checks availability of a time stamp on a fresh write.
12. $upd[x][]$: for each $x \in \mathcal{X}$, a global boolean array of length $2K + 1$ corresponding to the $2K + 1$ time stamps, checks whether a certain timestamp has been used to read in a bcas.
13. $globalTimeMap[x]$: global variable, for each $x \in \mathcal{X}$, stores a time stamp $\in$ Time. Maintains the globally maximal time stamp of each variable.
14. *messageStore* : This is an array of messages, where each message is of type Message as described in the main paper. The length of the array is K, the bound on the number of promises + view switches.
15. *messagesUsed* : a number from 0 to K which keeps track of the number of populated messages in *messageStore*.
16. *messageNum* : a number from 0 to K which chooses a number from the available free cells in *messageStore*.

We will denote the $\mathbb{K}$-(promise, view) bounded strong consistency as $\mathsf{Bd}(\mathsf{PS}, \mathsf{Vw}){-}\mathsf{RLX}$.

Translating $\mathsf{Bd}(\mathsf{PS}, \mathsf{Vw}){-}\mathsf{RLX}$ to bounded-context SC

Now we describe all the missing algorithms, and provide details of the codes. To start, we note that we are representing interval timestamps by integers in the translation. For each interval we only maintain its rightmost endpoint in our translation. Note that we can make discrete the dense points used in the intervals due to boundedness of the number of essential messages.

C.1 Main

Main. Algorithm 5 is the process that initializes all the global variables. This process executes atomically before all the other processes. $avail[x]$ for each shared variable x in *Prog* is an array of size $2K + 1$ which keeps track of time stamps which have not yet been assigned. Since all variables have a time stamp 0 initially, the first entry of this array is false for all variables. All entries of $upd[x][view[x].t]$ are initialized to true.

C.2 InitProc

Initialize Process. Before the simulation of each process, we initialize its variables of type View. The values and time stamps of all variables are 0, hence the initial view coincides with the view in the initial machine state of all runs. The variables $liveChain[x]$ is set to false for all shared variables x. Not that this sets up the invariant mentioned on the previous page. $extView[x]$ is initialized to true, since to begin, we are not in the certification phase and the initial value 0 comes from the initial message (which is generated outside any certification phase). Algorithm 6 details the function which is called at the beginning of each process.

Algorithm 5: Main

```
atomic_begin
messagesUsed ← 0
numContexts ← 0
numEE ← 0
for x ∈ X do
    upd[x][0] ← true
    globalTimeMap[x] ← 0
    for ts ∈ {1, 2, ..., 2K} do
        avail[x][ts] ← true
        upd[x][ts] ← true
    end
end
atomic_end
```

Algorithm 6: InitProc

```
atomic_begin
for x ∈ X do
    view[x].t ← 0
    view[x].v ← 0
    view[x].l ← true
    view[x].u ← true
    liveChain[x] ← false
    extView[x] ← true
end
```

C.3 ContextSwitchIn (CSI)

Algorithm 7: ContextSwitchIn

```
if ¬active then
    atomic_begin
    active ← true
    numContexts ← numContexts + 1
    assume(numContexts ≤ K + n)
end
```

Switch Into Context. This is called before each instruction $\lambda : i$ in a process p, to check if the process is active in the current context, which is kept track of by the boolean variable *active*. The counter *numContexts* is incremented signalling that one more context has been consumed. Since we translate into SC under $K + n$-bounded contexts, we check whether the context switching bound has already exceeded $K + n$. Algorithm 7 describes the context switching in.

C.4 Publish

Algorithm 8: Publish(*message*)

```
assume(messagesUsed < K)
messageStore[messagesUsed] ← message
messagesUsed ← messagesUsed + 1
```

Publish Subroutine. This is used to add messages to the *messageStore*. Each time a write or a `bcas` happens, depending on whether it results in an essential message or not, Publish(*message*) is called. Promise messages are also added using

Publish(*message*). Each time a new message is published, the size of the *messageStore* is increased. Since we have the bound on the number of essential messages, we check if the bound K on the number of view switches and promises has been exceeded.

C.5 loadState and saveState

Load and Save State while changing modes. The saveState subroutine copies the local state of the calling process and the global state into a what we refer to as 'copy' variables. We note that it does not however copy *numEE* and contents of *messageStore*. The reason for this being, the promises the process makes with *checkMode* true are retained even after *checkMode* is made false. Hence the increments made to *numEE* and the messages added to *messageStore* should be maintained even beyond after *checkMode* is false. Analogously in loadState, we load the contents of the (saved) 'copy variables' into their original counterparts.

Another subtle point to be noted is that when the process publishes a message (as a promise) when *checkMode* is true, we also update the 'copy' variables corresponding to *avail*$[x]$. This is done so that when the process returns to normal mode, the changes are reflected in their original counterparts (which is essential since promise messages are maintained beyond the time *checkMode* is false and hence their timestamps must be unavailable).

C.6 ScFence

Algorithm 9: ScFence

```
assume(¬checkMode)
for x ∈ X do
    if globalTimeMap[x] > view[x].t then
        view[x].t ← globalTimeMap[x]
        view[x].f ← false
        view[x].l ← true
    else
        if (view[x].l) then
            globalTimeMap[x] ← view[x].t
        else
            globalTimeMap[x] ← view[x].t + 1
        end
    end
end
```

SC fences. An SC-fence, in Algorithm 9, essentially takes the join of the *globalTimeMap*$[x]$ and the local timemap (*view*$[x].t$ for all $x \in \mathcal{X}$) of the process. First we ensure we are not in *checkMode* phase of the run, otherwise the run will not be consistent [13]. For each variable x the following is done.

- Lines 2-5 handle the case, where the former is greater. Then *view*$[x].t$ updated to match it; *view*$[x].l$ is set to true since the timestamp is now valid (can be used in comparisons). Also, *view*$[x].f$ is set to false, since the timestamp of the message corresponding to the current local value, *view*$[x].v$, is lower than *view*$[x].t$, and hence *view*$[x].v$ is no longer usable.
- Lines 7-11 handle the other case where the process timestamp is greater. If *view*$[x].l$ is valid (line 7-8) then, we can set *view*$[x].t$ to *globalTimeMap*$[x]$. If it is not valid (line 9-10), the process timestamp has actually proceeded beyond *view*$[x].t$. Note crucially that *view*$[x].t$ was the latest timestamp from Time that the process had. In this case, we set *globalTimeMap*$[x]$ to *view*$[x].t + 1$, the next 'useful' timestamp following *view*$[x].t$.

C.7 ContextSwitchOut (CSO)

Context Switch Out. We have described the full algorithm in the main paper. $\text{CSO}^{p,\lambda}$ allows the process allows the process to enter and exit context and it also serves to check the consistency of the process. When the process enters the certification phase, its local state (and program counter) are saved. When it returns back from the certification phase, *liveChain* being false is assumed which enforces that the process did not perform additive insertion. Then, the state is loaded and the program counter is reset to the same value it had before entering the certification phase.

Algorithm 10: $\text{CSO}^{p,\lambda}$

```
σ_sw:
if * then
    if ¬checkMode then
        if ¬active then
            atomic_begin
            active ← true
            numContexts ← numContexts + 1
            assume(numContexts ≤ K + n)
        end
        checkMode ← true
        retAddr ← λ, saveState(p)
    else
        for m ∈ messageStore do
            assume(m.flag ≠ p)
            if m.flag == −1 then m.flag ← p
        end
        for x ∈ X do  assume(¬liveChain[x])
        loadState(p), gotoLabel(retAddr)
        checkMode ← false
        active ← false
        atomic_end
    end
    goto σ_sw
end
```

C.8 Read

Algorithm 11: Translating $[\![\$r = x]\!]^p$ read

```
if * then
    assume(¬liveChain[x])
    assume(numEE < K)
    messageNum ← nondetInt(0, messagesUsed − 1)
    message ← messageStore[messageNum]
    assume(message.var == &x)
    assume(view[x].l)
    assume(view[x].t ≤ message.t)
    view[x].t ← message.t
    view[x].v ← message.v
    extView[x] ← true
    numEE ← numEE + 1
end
val($r) = view[x].v
```

Read and Write. We have already described in good detail, the algorithms for read and write. However, we commented out a few lines which deal with the variable *extView*$[x]$ ('external view') from the code, which is used in bcas. Here, we produce the complete codes (Algorithms 11, 12) for the read and write instructions. In Algorithm 11, line 11), during a global read, the variable *extView*$[x]$ is set to true, indicating that the value *view*$[x].v$ read is generated by a message external to the current certification phase. Indeed, whenever a process makes a global read while *checkMode* is true, it obviously reads from a message which has been created outside its current certification phase. Hence, *extView*$[x]$ will be set to true.

In the case of Algorithm 12, if the process has *checkMode* false, then after the write, the value of *view*$[x].v$ comes from the current write (whether or not it resulted in a published message), and hence *extView*$[x]$ is set to true, since the value in *view*$[x].v$ is generated outside any certification phase. Likewise, if the process has *checkMode* true, then after the write, the value of *view*$[x].v$ comes from the current write (whether or not it resulted in a published message), but since it does arise from the current certification phase, it is not external, and hence *extView*$[x]$ is set to false (lines 44-48). Finally, *view*$[x].u$ is set to true (line 49) iff $view[x].l$ is true. Indeed, if $view[x].l$ is false after the write, then the time stamp $view[x].t$ is not legitimate for comparisons, and hence starting from $view[x].t$, there cannot be sequence of bcass.

C.9 Write

Algorithm 12: $[\![x = \$r]\!]^p$ write

```
if * then
    view[x].v ← val($r), view[x].l ← true
    if * then
        if liveChain[x] then
            newStamp ← view[x].t + 1
        else
            newStamp ← nondetInt(view[x].t + 1, 2K)
        end
        view[x].t ← newStamp
        assume(avail[x][newStamp])
        avail[x][newStamp] ← false
        if * then
            if checkMode then
                message ← genMessage(x, newStamp, val($r), −1)
                liveChain[x] ← false, numEE ← numEE + 1
            else
                message ← genMessage(x, newStamp, val($r), 0)
            end
            Publish(message)
        else
            if checkMode then
                liveChain[x] ← true
            end
        end
    else
        messageNum ← nondetInt(0, messagesUsed − 1)
        assume(message.var == &x, message.t > view[x].t)
        assume(message.v == view[x].v, message.flag == p)
        view[x].t ← message.t
        if ¬checkMode then
            message.flag ← 0
        else
            message.flag ← −1, liveChain[x] ← false
        end
        messageStore[messageNum] ← message
    end
else
    view[x].v ← val($r), view[x].l ← false
    if checkMode then
        liveChain[x] ← true
    end
end
view[x].f ← true
if ¬checkMode then
    extView[x] ← true
else
    extView[x] ← false
end
view[x].u ← view[x].l
```

C.10 bcas(x, $\$r_1$, $\$r_2$)

Algorithm 13: Translating $\llbracket$bcas(x, $\$r_1$, $\$r_2$)$\rrbracket^p$ update

```
if * then
    assume(¬liveChain[x] ∧ numEE < K)
    messageNum ← nondetInt(0, messagesUsed − 1)
    message ← messageStore[messageNum]
    assume(message.var == &x ∧ view[x].l ∧ view[x].t ≤ message.t)
    view[x].t ← message.t, view[x].v ← message.v
    extView[x] ← true, numEE ← numEE + 1
else
    assume(view[x].f)
end
assume(view[x].v == val($r1))
if view[x].l then
    assume(upd[x][view[x].t]), upd[x][view[x].t] ← false
end
view[x].v ← val($r2)
if * then
    if checkMode then
        assume(¬extView[x]), liveChain[x] ← true
    end
    view[x].l ← false
else
    if * then
        if view[x].u ∨ liveChain[x] then
            newStamp ← view[x].t + 1
        else
            newStamp ← nondetInt(view[x].t + 1, 2K)
        end
        view[x].t ← newStamp, assume(avail[x][newStamp]), avail[x][newStamp] ← false
        if * then
            if ¬checkMode then
                message ← genMessage(x, newStamp, val($r2), 0)
            else
                message ← genMessage(x, newStamp, val($r2), −1), liveChain[x] ← false, numEE ← numEE + 1
            end
            Publish(message)
        else
            if checkMode then
                assume(¬extView[x]),liveChain[x] ← true
            end
        end
    else
        messageNum ← nondetInt(0, messagesUsed − 1), message ← messageStore[messageNum]
        assume(message.var == &x ∧ message.t > view[x].t)
        assume(message.v == val($r2) ∧ message.flag == p)
        view[x].t ← message.t
        if ¬checkMode then
            message.flag ← 0
        else
            message.flag ← −1, liveChain[x] ← false
        end
        messageStore[messageNum] ← message
    end
    view[x].l, view[x].u ← true
end
view[x].f ← true
if ¬checkMode then
    extView[x] ← true
else
    extView[x] ← false
end
```

Compare and swap bcas(x, $\$r_1$, $\$r_2$). This module (Algorithm 13) combines the read and write modules.

In lines 2-7, the process reads a message from the *messageStore*, and updates the local view setting *extView*[x] to true, and incrementing *numEE*. *extView*[x] is set to true since the value of *view*[x].v is taken from a message in the *messageStore*: irrespective of whether *checkMode* is true or not, the value comes from a message generated outside this phase. Notice that *liveChain*[x] must be false, as explained in the case of the read instruction in the main paper, to ensure no additive insertions. If the local view is already in sync with the global view, then line 9 is executed, and there is no need to read from the *messageStore*.

Lines 11-15 checks if the value in $view[x].v$ is equal to $R(\$r_1)$, and in case the time stamp $view[x].l$ is legitimate (allowing for comparisons), then whether the message with this time stamp has not been read/used already for a bcas. Then the new value $view[x].v$ is set to $R(\$r_2)$. Now comes the part where this value has to be written to a new message.

There are two possibilities, depending on whether the write is assigned a timestamp or not. If not, the first part (lines 16-20) sets $view[x].l$ to false, and if the process in the certification phase, sets $liveChain[x]$ to true (this follows from the $liveChain$ invariant explained in the main paper), and sets $extView[x]$ to false (the value $view[x].v$ comes from this certification phase). Note that when $view[x].l$ is set to false, we do not set $view[x].u$ also to false, unlike the case of the write instruction (Algorithm 12, line 49). The reason is, if $view[x].u$ is true (the process executes a consecutive chain of bcas instructions, each reading from the previous) and does not assign a timestamp to all of them, for those that it does, the timestamps chosen must be immediate successors of one another (reflecting the fact that this indeed is a sequence of adjacent intervals). Thus, the invariant related to $view[x].u$ holds.

Otherwise, $view[x].l$ is set to true (line 53). Assume $view[x].l$ is set to true; ($view[x].u$ is set to true as well). Then, there are four possibilities.

1. Lines 22-40 deal with two possibilities (i) not publishing the message (lines 36-40), (ii) publishing a promise message (immediate certification if *checkMode* is true, lines 32-35) or publishing a message in normal phase (lines 30, 31, 35). In both these cases, lines 23-27 deal with the choice of the fresh time stamp. If $liveChain[x]$ is true, then the new timestamp is an immediate successor of the existing one (this has been explained in the main paper, as part of the invariant for $liveChain[x]$). If $view[x].u$ is true, then starting from this timestamp $view[x].t$, there is a chain of bcas, to the most recent message, and hence, we need to choose the next immediate time stamp. When both $liveChain[x]$ and $view[x].u$ are false, then the new time stamp can be chosen as any available higher value (line 26). As usual, we check the availability of this position in the array $avail[x]$.
2. Lines 41-53 deal with the other two cases. (iii) Either *checkMode* is true and the process is certifying promises made before (lines 42-45, line 49) or (iv) *checkMode* is false and the process is fulfilling a promise (lines 42-47).

Finally, $view[x].f$ is set to true in any case, since the value $view[x].v$ is recent. The updates to $extView[x]$ are exactly as in Algorithm 12.

Once again, we recall that $\mathbb{K}$-(promise, view) bounded strong consistency is denoted as $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$.

D Correctness of Translation

The proof is in two parts. In the first part, we show that that every $K+n$ context bounded run of $Prog'$ in SC corresponds to a K-bounded run of $Prog$ under $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$, and in the second part, we show that for every K-bounded run in $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$, there is a $K+n$ context bounded run in SC.

At the outset we review a high level description of the translation. We denote by 'normal' (*checkMode* is false) and *checkMode* (true), the two phases in which a process functions. Each process executes instructions in the normal phase by skipping over the *CSO* blocks of code. When a process needs to switch out, it enters the *CSO* block following the most recent instruction and sets *checkMode* to true. Now, it makes a 'ghost' run in *checkMode*, a terminology to indicate that this phase of the run does not change the the global state and local state of the process permanently (this is facilitated by the *saveState* and *loadState* functions). One exception to this is the writes that the process makes as published promises which are maintained permanently. Hence, this part of the run is equivalent to the process making fresh promises after normal execution; providing a witness for consistency and then switching out of context. The run then is a sequence of interleaved normal and *checkMode* phases. Moreover the local states of the process is identical at the start and end of any given *checkMode* phase.

We request the reader to refer to the glossary [C] of the variables used which will aid in better understanding of the translation.

D.1 SC to $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$

Intuition We note that non-essential messages (which are not view-switching or promises), need to be accommodated along the time-line for each variable (while they were not in the SC-run). We account for these by separating the essential messages by sufficiently large intervals, so that, the non-essential ones can be inserted in between, respecting their order.

Details We start from SC to $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$. We show that every $K+n$ context bounded run of $Prog'$ corresponds to a K-bounded run of $Prog$. Keeping in mind the description above, we split this proof into two parts. First we consider only the normal run and prove that it has an analog in $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$. Then we prove that any *checkMode* phase is indeed an analog of a process making fresh promises and certifying them along with previous unfulfilled promises. Combining these two, indeed, we will have a run under $\mathsf{Bd}(\mathsf{PS},\mathsf{Vw})\mathsf{-RLX}$.

We begin by defining some terminology. Consider a run τ of program $Prog'$. Each event of the run τ is an execution of either a read, write, bcas or SC-fence. A read in this run is called *global* (and otherwise *local*) if the process decides to read from the global array *messageStore*. Only global reads can be view-switching in the corresponding run under Bd(PS, Vw)−RLX. A write can be of four types - pubSim, pubFul, stamped and local. These represent, 'simple published', 'fulfilling published', 'timestamp assigned but unpublished' and 'timestamp not assigned writes' respectively (published implies that timestamp is assigned too). Note that each of these types can be performed in normal as well *checkMode*. A bcas can therefore be of 8 types since it involves a read and write.

Let w_1 be the number of write events in the normal part of the run, w_2 be the maximum number of write events, maximum being taken over all *checkMode* phases of the run, $u - 1$ be the number of bcas events in the run, and let $l = w_1 + w_2 + u$. Let $\mathsf{M_x}$, for each shared variable x, be an increasing function from $[2K]$ to N representing a mapping from the notion of time-stamps in SC to time-stamps in Bd(PS, Vw)−RLX. For each variable x, and each process p, let $\mathsf{View_{SC}(x)} = view[x].t$ (defined above) and $\mathsf{View_{Bd(PS,Vw)-RLX}(x)}$ be the time stamp of x in the view of p in ρ. Given a run τ, we will construct a K bounded run ρ of *Prog* which reaches the same set of labels after i events, for any i.

We will first treat the normal (non-*checkMode*) part of the run. While going through the steps, we will also construct the increasing functions $\mathsf{M_x}$. In addition to the invariants in C, we maintain the following timestamp-based invariants for all processes p and variables x.

1. If $view[x].l$ is true for a process in τ, then $\mathsf{M_x(View_{SC}(x))} = \mathsf{View_{Bd(PS,Vw)-RLX}(x)}$.
2. If $view[x].l$ is true and the time-stamp $view[x].t$ corresponds to a write message instead of a message added due to an bcas, then $\mathsf{M_x}(view[x].t) = view[x].t \cdot l \cdot u$.
3. If $view[x].l$ is false, then $\mathsf{M_x}(view[x].t) < \mathsf{View_{Bd(PS,Vw)-RLX}(x)} < (view[x].t + 1) \cdot l \cdot u$. Moreover, if the last event to assign false to $view[x].l$ was a write, then $\mathsf{View_{Bd(PS,Vw)-RLX}(x)}$ is a multiple of u.
4. If a message is of type bcas, then its time-stamp t in ρ satisfies $t \not\equiv 0 \mod u$.
5. The sum of view-switch points and promises is $\leq K$ in ρ.
6. The time-stamps of an essential messages in τ and the corresponding message in ρ are related by $\mathsf{M_x}$. That is, $\mathsf{M_x(View_{SC}(x))} = \mathsf{View_{Bd(PS,Vw)-RLX}(x)}$.

The base case, that is, after 0 events ($i = 0$) is trivial since the configurations are semantically equivalent and we define $\mathsf{M_x}(0) = 0$ for all variables, which satisfies the invariants. We make the following three cases depending on the i^{th} event of τ.

- Case 1. e_i is an execution of a write for process p, variable x and value v.
  - If the write is of pubSim, pubFul or stamped type, then $view[x].t$ is updated from t to a new time-stamp t' (which in the case of pubFul is the timestamp of the retrieved message) and $view[x].l$ is assigned true. In ρ, if we can make $\mathsf{View_{Bd(PS,Vw)-RLX}(x)} = t'' = t' \cdot l \cdot u$ then the invariants are satisfied. It is not possible for t'' to have been assigned already to some write message in ρ since t' was not assigned to some message in τ (checked using $avail[x][t']$). A bcas message could not have been assigned t'' either, by the fourth invariant. Since $t < t'$, $\mathsf{View_{Bd(PS,Vw)-RLX}(x)} < t''$ (by invariants 2 and 3). Hence, $\mathsf{View_{Bd(PS,Vw)-RLX}(x)}$ can be updated to t'' since it is available and is greater than the current view. If the write is published, then the message is added to *messageStore*. This is done to maintain invariant (6). Note how, if the write is of pubFul type, the message flag is set to 0, effectively removing it from the promise bag and maintaining the $flag$ invariant [5].
  - If the write is local, then we pick the smallest available multiple of u between $\mathsf{M_x}(view[x].t)$ and $(view[x].t + 1) \cdot l \cdot u$. This can always be done since there are $l - 1$ multiples of u between $view[x].t \cdot l \cdot u$ and $(view[x].t + 1) \cdot l \cdot u$ and there are $\leq (l - 1)$ messages (even considering those produced in *checkMode*) in total. Notice that multiples of u have been reserved for writes by invariant 4.
- Case 2. e_i is an execution of a read for process p, variable x.
  - If the read is local in τ, then the process is either reading a local message written by itself or a useful message. In either case, this read can be performed in ρ without any change in time-stamps. Note that this cannot be a view-switching event. Moreover note that the local value in $view[x].v$ has been ascertained to be usable.
  - If the read is global, then $numEE < k$ before the read and therefore $numEE \leq k$ afterwards. In this case, a message is fetched from *messageStore* and the process view is updated according to this message. Since $\mathsf{M_x}$ is an increasing function, the results of comparisons in SC will be the same as in Bd(PS, Vw)−RLX and the read operation has the same effect on values and time-stamps of the variables. Moreover $view[x].f$ is set to true maintaining the $view[x].f$ invariant [C].
- Case 3. e_i is an execution of an bcas for process p, variable x and values v, v'.

- If the read here is local, and $view[x].u$ is true then we need to ensure that the timestamp chosen for the write immediately follows $\mathsf{M_x}(view[x].t)$. It is first checked if $view[x].t$ has been used for an update earlier or not. If it has not been, then the time-stamp $\mathsf{M_x}(view[x].t) + 1$ is available in Bd(PS, Vw)−RLX since all messages that come from writes have time-stamps in multiples of u and $\mathsf{M_x}(view[x].t)$ is a multiple of u. Note, that we also ensure that $view[x].f$ is true in this case, which implies that the local value is usable.
- If the read here is local and $view[x].u$ is false (and hence so is $view[x].l$), then it definitely has not been used for an update (bcas) in τ since the process reading the message is the only one that knows of its existence. Now, if this message was a result of a local write, then its time-stamp t in Bd(PS, Vw)−RLX is a multiple of u and $t + 1$ is available for the update message. Otherwise, this message was a result of a bcas whose write was local and has a time-stamp of the form $a \cdot u + b$ where $b < u$. Note that this implies $b - 1$ consecutive bcass were made to get here since all the messages that are a result of (non-bcas) write operations get time-stamps that are multiples of u. Since $u - 1$ is the total number of bcass in τ, $b < u - 1$ (at most $u - 2$ bcass have taken place before this one). This implies $a \cdot u + b + 1$ is available and can be used for the write.
- If the read is global, then it is done correctly as explained in Case 2. The write part of the bcas goes through as explained above.

- Case 4: e_i is an SC-fence
  - We iterate over the variables, updating $globalTimeMap[x]$ and $view[x].t$ to the maximum of the two.
  - In case, the former was greater, we set $view[x].l$ to true, signifying that $view[x].t$ is valid and maintaining invariant (1) above. Moreover we set $view[x].f$ to false. This is necessary since, the timestamp of the message corresponding to $view[x].v$ is now less than $view[x].t$ and hence the locally stored value is unusable.
  - If the latter is greater, we check whether $view[x].l$ is true (which signifies that $view[x].t$ is valid). If it is we can set $globalTimeMap[x]$ to it. If not, then the $\mathsf{M_x}(view[x].t) < \mathsf{View}_{\mathsf{Bd(PS,Vw)-RLX}}(\mathsf{x})$ (by invariant (6)), and hence we set it to $view[x].t + 1$. Finally we note that $\mathsf{View}_{\mathsf{Bd(PS,Vw)-RLX}}(\mathsf{x}) < (view[x].t + 1) \cdot l \cdot u$ and hence $\mathsf{M_x}(globalTimeMap[x])$ now matches the essential event immediately following the event with timestamp $view[x].t$.

We now briefly justify the *checkMode* phase of the run. For any such phase, we need to ascertain that the run has analogous run in Bd(PS, Vw)−RLX which respects the notion of consistency. The management of timestamps is identical to the normal phase explained above so we only highlight the special aspects. First we recall some invariants:

1. $liveChain[x]$ is true only when the most recent write made in the *current checkMode* phase was unpublished (was not a promise).
2. $extView[x]$ is true if $view[x].v$ corresponds to a message from outside *checkMode*.
3. For the process p currently in *checkMode*, $message\_flag$ is -1 for temporarily (only within current *checkMode* phase) certified promises and p for as yet uncertified promises. If it is $p' \neq p$, then the message is in the promise bag of some other process. Additionally if it is 0, it is not in the promise bag of any process. Note how this is maintained in the write, bcas sections above.

We'll review how these invariants are maintained and used throughout the code. When entering *checkMode*, $liveChain[x]$ is false. For any write happening in normal phase we set $extView[x]$ to true. Otherwise we set it to false. Once again we consider cases for a particular event e_i:

- Case 1. e_i is a write event.
  - In the case, the process performs a local or stamped write, $liveChain[x]$ is set to true, maintaining the invariant.
  - In the case the process decides to publish a write it must publish it as a promise, incrementing $numEE$ (after checking that the bound of K has not been crossed), setting the promise flag to -1, maintaining invariant (3) above. Also, if it decides to certify a previous promise, it does so, similar to the normal phase, though it now sets the timestamp to -1, indicating that the certification is local to the current phase and must be reset when normal phase resumes. Moreover note that $liveChain[x]$ is set to false maintaining invariant (1).
  - Also, note that $extView[x]$ is set to true maintaining invariant (2).
- Case 2. e_i is a read event.
  - The main highlight of read events in *checkMode*, is that we ascertain that $liveChain[x]$ is false while making a global read. This is to ensure that we forbid additive insertion. Indeed, following invariant (1) above, if $liveChain[x]$ were true during a global read, it would mean that the interval corresponding to the previous message (which caused $liveChain[x]$ to be true) is additively.
- Case 3. e_i is a bcas event.
  - Once again similar to normal phase we guess whether we make a local or a global read. Crucially however, we note that we forbid making a local or stamped write for a bcas when $extView[x]$ is true. Considering the invariant (2)

above, this is done precisely to forbid bcas where, the promised interval containing the write is non-adjacent to the message being read from. The remainder bookkeeping of is identical to previous cases.

- Case 4. e_i is an SC-fence event. This case does not arise since a process in *checkMode* may not execute an SC-fence instruction else the run will not be consistent [13].

To conclude, note due to loadState and saveState functions, only promises are retained after the *checkMode* phase. Moreover due to the check of message flags after a *checkMode* phase terminates, it is ensured that the process is in a consistent state while switching contexts. Noting that we keep track of promises as well as view-switches using *numEE* we may only generate a run in which the sum of the two is bounded by K.

D.2 Bd(PS, Vw)−RLX **to SC**

We now prove the second part, from Bd(PS, Vw)−RLX to SC. We prove that for every K-bounded run ρ in Bd(PS, Vw)−RLX, there is a $K + n$ context bounded run τ in SC. We will show this in two steps.

- Given the K-bounded ρ, first we will construct a run ρ'' which is K-bounded and $K + n$ context bounded that reaches the same configuration as ρ.
- We will then construct a run τ of SC using ρ''.

Intuition We ensure that each process only switches out of context when it is awaiting a message for a (global) read from another process. Note that in each such case the process waiting will undergo a view-switch. Since the total number of view-switches along a 'normal' phase + additional messages in all *checkMode* phases is bounded above by K, we need atmost $K + n$ context switches. We add n for the concluding contexts required to reach the *term* configurations.

Let rf (called $reads\text{-}from$) be a binary relation on events such that $(e_a, e_b) \in rf$ iff e_b reads from a message *published* by e_a. Note that every run under Bd(PS, Vw)−RLX semantics defines a rf relation as the reads are executed. For construction of ρ'', the intuition is that a context switch is required only when the current process has reached *term* or it needs a message that is yet to be published by some other process. At a configuration $\mathfrak{c}_i$ of ρ, we say that an event of ρ is a requesting event if it is a view-altering event in ρ and it reads a message that is not in the message pool at $\mathfrak{c}_i$. Also, we call the events that publish messages for these events as servicing events (write or bcas, either simple or promises). Note that the set of servicing and requesting events is dependent on the configuration $\mathfrak{c}_i$. The two sets change along the run ρ. Specifically, an event is removed from the requesting event set as soon as the servicing event corresponding to it is executed. Let the size of the set of requesting events be r. At $\mathfrak{c}_{init}$, $r = K$. We will prove by induction that given a set of processes (n), the rf relation, and a run ρ in Bd(PS, Vw)−RLX that maintains the rf relation, there is a run which uses at most $r + n$ context switches and defines the same rf relation.

The Base Case. For $r + n = 1$, there is only one process so the number of context switches is 0 and the ρ itself uses 0 context switches.

The Inductive Step. Assume the hypothesis for $r + n = l$ and we prove the claim for $r + n = l + 1$. Clearly at $\mathfrak{c}_{init}$, there is at least one process which either has no requesting events, or has a servicing event before any requesting events in its instruction sequence. Otherwise, the run ρ will not be able to execute all the events since no process will be able to move past its requesting event. If we have a process that can reach termination directly, then in ρ'', we run that process and reduce $r + n$. Otherwise, consider the instructions of the process (p_j) that has a servicing event before any of its requesting events. The instructions of p_j, till the first requesting event, can be executed since all the messages they need are already in the pool and hence we can create a new run ρ_t in which these instructions are executed first and the remaining ones follow the same order as ρ. Note that ρ_t reduces r by at least 1 while executing the instructions of p_j. By applying the hypothesis on the remaining sequence of instructions, we have a run that uses $r - 1 + n$ context switches and that maintains rf of the remaining instructions. This can now be combined by the instructions of p_j that have already been executed to give ρ''.

We now construct the run τ from ρ''. As explained in the text above, at most $2K$ time-stamps are needed to simulate the ρ''. Let the set of such time-stamps be U_x for each variable x. Let $\mathsf{M_x}$ be an increasing (mapping) function for each variable from $U\_x \cup \{0\}$ to $\{0, \ldots 2K\}$ such that $\mathsf{M_x}(0) = 0$.

We will construct the run τ in SC from ρ'', event by event, while maintaining the following invariants

1. All the time-stamps, in a particular message in *messageStore*, are related to the time-stamps in the corresponding essential message in Bd(PS, Vw)−RLX by $\mathsf{M_x}$.
2. For a process p, $\mathsf{View}_{\mathsf{Bd(PS,Vw)-RLX}}(\mathsf{x}) \in U\_x$ iff $view[x].l$ is true at that point in SC and $view[x].t = \mathsf{M_x}(\mathsf{View}_{\mathsf{Bd(PS,Vw)-RLX}}(\mathsf{x})))$

The i^{th} event of ρ'' can be one of the following:

- Case 1. e_i is a write to variable x with value v.

  - If the time-stamp t of this write belongs to U_x, then we first allocate $\mathsf{M_x}(t)$ in SC to this write and make $view[x].l$ true. This maintains invariant (2).
  - If the event is a servicing event since , we have that the time-stamp of this message satisfy the requirements of invariant (1) and hence it can be added to *messageStore*. Otherwise, we do not update the $\mathsf{View_{SC}}(\mathsf{x})$ of the process and make $view[x].l$ false.
- Case 2. e_i is a read of variable x If this event is a view-altering event, then the current timestamp in the $\mathsf{View_{Bd(PS,Vw)-RLX}}$ will be used for comparison. The effect of the read in SC will be same as in Bd(PS, Vw)−RLX since V_x is an increasing function. All the invariants will still hold after this, since all the messages in *messageStore* satisfy the invariants.
- Case 3. e_i is an `bcas` to variable x with values v, v'. If this event is not view-altering, then the process either reads some other process's message again or reads its own. If it reads its own message, then no change to the $\mathsf{View_{SC}}(\mathsf{x})$ has to be done for the read part and the new message is added to *messageStore* if $e_i' s$ message is essential. If it reads some other processes' message again, then $view[x].l$ is true, and since this message has not been used for an `bcas` yet, the check of $upd\_x[view[x].t]$ will go through in $Prog'$. Now, it needs to be decided if the new message is essential. If the read is view-altering, then it is similar to Case 2 followed by the decision of adding the new message to *messageStore*.
- Case 4. e_i is an SC-fence If $globalTimeMap[x]$ is greater than $view[x].t$, we maintain invariants (2) by setting $view[x].l$ to true and the $view[x].f$ invariant [C] by setting it to false. On the other hand if $view[x].t$ is greater, we set $globalTimeMap[x]$ to the smallest member $t \in \mathsf{Time}$, which satisfies $t \geq \mathsf{M_x}(\mathsf{View_{Bd(PS,Vw)-RLX}}(x))$. In case $view[x].l$ is true, t is $view[x].t$ itself by invariant (2). If not then we set it to $view[x].t + 1$, since we note, $view[x].t$ is the largest member of Time, that p has had as $\mathsf{View_{Bd(PS,Vw)-RLX}}(x)$, and currently the former is lower than $\mathsf{M_x}(\mathsf{View_{Bd(PS,Vw)-RLX}}(x))$.

E Details for Section 6 - Implementation and Experimental Results

In the promise free mode, we compare SwInG with three state-of-the-art stateless model checking (SMC) tools, CDSChecker [23], GenMC [15] and Rcmc [14] that support the relaxed semantics without promises. We use a version of CDSChecker that halts on the first bug discovered while GenMC and Rcmc do this by default. In the tables that follow, we specify the used values of L (for all tools) and K (only for our tool).

Here we state the results of all our experiments in full. The main takeaways of our experiments are: (1) our tool can uncover hard-to-find bugs faster than the others with relatively small values of K; (2) our approach is more resilient to trivial changes in the position of bugs as compared to the SMC tools; (3) in some instances, our technique fares better at capturing relevant behaviours instead of exploring all possible traces as done by some SMC tools.

We note that the tools we are comparing with do not require as input the bound, K. Hence, the comparison may not be fair for some safe examples, since SwInG only considers the subset of executions which K enforces. However, in particular instances we have set the parameter K such that all executions are considered (modulo the loop unwinding bound). In such cases, we note the tool is comparable to the others. We highlight such cases (only for *safe* examples) with a green checkmark (✓) accompanying the value of K used. Additionally, we have put forth cases where we can iteratively increment K to prove correctness.

Considering the above observations, we realise that the SMC tools and our tool have orthogonal approaches to finding bugs. SMC tools are limited by how they explore the space of all executions, which might be sub-optimal in cases where we have a shallow counterexample but which is explored only after several executions. Our tool is limited by the bound K.

We do not consider compilation time for any tool while reporting the results. For our tool, the time reported is the time taken by the CBMC backend for analysis. The timeout used is 1 hour for all benchmarks. All experiments are conducted on a machine equipped with a 2.80 GHz Intel Core i7-860 and 4GB RAM running a Debian 9 (stretch) 64-bit operating system. We denote timeout by 'TO'. In the tables that follow, we specify the values of L (for all tools) and K (only for our tool) used. We mark a hyphen '-' in the table for when the process is killed with a maximum resident set size (RAM used) of 3.7 GB or higher.

We first compare strong and standard consistency on some examples. For the remaining benchmarks, to enable comparison with other tools (which do not support promises), we run the tool in promise-free mode. Then, we show the ability of our tool: (1) to detect hard-to-find bugs, (2) to adapt to concurrent data-structure benchmarks and (2) resilience to location of bugs and number of executions.

E.1 Comparing the notions of consistency

We run SwInG, in promise-mode on a variety of testcases from Kang et al. [13] and Chakraborty and Vafeiadis [7]. In the upper part of Table 8 are the interesting ones amongst these. The `split` testcase exhibits the difference in the semantics presented in sections 2 and 4 of Kang et al. [13]. The `ARMweak` example suggests how a process may read its own promise via a helper

testcase	K	SwInG[strong]	D	SwInG[standard]
split	3	43.717s	×	×
ARMweak	2	1.560s	×	×
LBfd	3	0.692s	×	×
Coh-CYC	4	17.367s	×	×
splitCAS	5	1.378s	20	12.284s
			40	37.166s
			60	2m15s
			80	4m26s
LBcd	7	1.003s	100	10.984s
			200	25.010s
LBcu	7	4.434s	100	1m13s
			200	2m39s
LB2cu	7	5.331s	10	1m16s
			20	15m40s
fibonacci_2_safe	5	17.244s	10	3m11s
fibonacci_3_safe	5	14m14s	10	TO

Table 8. Comparing the two notions of consistency

thread. `LBfd` is an example exhibiting load buffering with a false (syntactic) dependency. We note that small values of K are sufficient to uncover the bug in these cases.

In order to empirically confirm our hypothesis that the standard definition of consistency (as defined in [13]) would not scale, we run SwInG, on similar small examples under the strong and standard consistency, while varying the size of the data domain, specified by D. Observe that we need to vary D for the standard consistency definition since it is required during the quantification over all future memories (which implicitly includes all possible data values). We run SwInG on a variety of safe and unsafe test cases from [7, 13]. The first three examples are unsafe while the other ones are safe. In all these cases, we observe, the dependence of run-time on the size of the data domain when the standard consistency definition is used. Strong consistency, on the other hand performs much better without any restriction on the size of the data domain. This is presented in the lower part of the table.

E.2 Evaluation using parametrized benchmarks

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
exponential_5_unsafe	5	10	1.195s	1.795s	0.189s	8.282s
exponential_10_unsafe	10	10	1.786s	4.167s	0.736s	3m50s
exponential_25_unsafe	25	10	3.433s	14.737s	4.697s	TO
exponential_50_unsafe	50	10	9.021s	1m6s	1m2s	TO
exponential_70_unsafe	70	10	14.136s	2m52s	4m3s	TO
fibonacci_2_safe	2	✓20	4.045s	8.811s	0.104s	0.133s
fibonacci_3_safe	3	✓20	10.899s	TO	0.984s	4.443s
fibonacci_4_safe	4	✓20	30.475s	TO	41.576s	3m2s
triangular_2_safe	2	✓4	5.683s	0.403s	0.069s	0.063s
triangular_3_safe	3	✓6	1m3s	18.737s	0.152s	0.290s
triangular_4_safe	4	✓8	4m58s	20m20s.	1.602s	2.282s
triangular_5_safe	5	✓10	8m16s	TO	28.883s	34.819s
triangular_2_unsafe	2	10	1.711s	0.070s	0.071s	0.102s
triangular_3_unsafe	3	10	9.422s	2.903s	0.126s	0.244s
triangular_4_unsafe	4	10	2m54s	3m25s	1.254s	1.531s
triangular_5_unsafe	5	10	12m23s	TO	21.619s	26.730s

Table 9. Evaluation using parametrized benchmarks

We now compare SwInG with CDSChecker, GenMC and Rcmc in Table 9 on three parametrized benchmarks: `ExponentialBug` (from Fig. 2 of [11]), `Fibonacci` and safe and unsafe versions of `Triangular` taken from SV-COMP 2018. In `ExponentialBug`(N) and `Triangular(N)`, the processes compete to write to a shared variable and N represents the number of times a process may write. In `ExponentialBug`(N), the number of executions grows as $O(N!)$, while the fraction of buggy interleavings decrease exponentially with N. In the unsafe version of `Triangular`(N), there is exactly one interleaving that exposes the bug, while the total number of interleavings increases exponentially with N. In `Fibonacci`(N), two processes compute the value of the n^{th} Fibonacci number. In the safe version of `Triangular`(N) as well as `Fibonacci`(N), we note that we use a conservative upper bound on the value of K. Hence this table demonstrates the ability of SwInG in exposing hard-to-find bugs as well as adaptability for safe cases.

E.3 Evaluation using concurrent data structures based benchmarks

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
hehner2_unsafe	4	5	6.130s	0.028s	0.042s	0.072s
hehner3_unsafe	4	5	26.729s	0.026s	4m4s	1m26s
linuxlocks2_unsafe	2	4	0.748s	0.010s	0.036s	0.081s
linuxlocks3_unsafe	2	4	1.113s	0.013s	0.037s	0.084s
queue_2_safe	4	4	2.141s	0.020s	0.039s	0.079s
queue_3_safe	4	4	9.417s	0.024s	0.053s	0.086s

Table 10. Evaluation using concurrent data structures - I

benchmark	L	SwInG[$K=4$]	SwInG[$K=6$]	CDSChecker	GenMC	RCMC
stack_2_safe	2	0.354s	1.467s	0.009s	0.067s	0.063s
stack_3_safe	3	0.879s	4.755s	0.229s	0.073s	0.108s
stack_4_safe	4	2.127s	14.426s	8.313s	0.819s	1.287s
stack_5_safe	5	6.467s	44.993s	5m2s	14.132s	43.903s
stack_6_safe	6	24.185s	5m8s	TO	7m14s	25m44s

Table 11. Evaluation using concurrent data structures - II

We compare the tools in Tables 10 and 11 on benchmarks based on concurrent data structures. The first of these is a concurrent locking algorithm from Hehner and Shyamasundar [10]. The second, `LinuxLocks(N)` is a benchmark extracted from the Linux kernel. If not completely fenced, this benchmark is unsafe under relaxed semantics and we fence all but one lock accesses. The other two are *safe* benchmarks adapted from SVCOMP-2018. The queue benchmark is parameterized by the number of processes and the stack benchmark is parameterized by the size of the stack. The processes operate on these data structures and we check whether certain invariants are maintained. These benchmarks illustrate the ability of our tool to handle concurrent data-structures similar to those seen in real-world examples.

E.4 Evaluation using two synthetic safe benchmarks

We compare the tools in Table 12 on adaptations of two synthetic safe benchmarks: `ReaderWriter(N)` (from Norris and Demsky [24]) and `RedundantCo(N)` (from Abdulla et al. [3]). Both these examples involve N processes writing distinct values to a shared variable and one process reading from it. The number of traces in these examples grow as $O(N!)$. The number of possible values for the reads however is just $O(N)$ in the first example and $O(1)$ in the second one. The performance of the SMC tools depends on how efficiently they explore the executions. SwInG on the other hand depends on the reads observed, illustrating the point mentioned earlier. We again note that K is chosen conservatively and our tool declares the benchmarks to be safe considering all executions.

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
readerwriter_7	0	✓5	0.719s	0.005s	0.057s	0.690s
readerwriter_8	0	✓5	0.839s	0.006s	0.056s	7.425s
readerwriter_9	0	✓5	1.068s	0.007s	0.053s	1m17s
readerwriter_10	0	✓5	1.393s	0.007s	0.056s	14m49s
redundant_co_10	10	✓5	0.470s	0.114s	0.087s	38m12s
redundant_co_20	20	✓5	1.031s	0.548s	0.218s	TO
redundant_co_50	50	✓5	3.219s	8.965s	4.143s	TO
redundant_co_70	70	✓5	6.093s	13.843s	18.185s	TO

Table 12. Evaluation using two synthetic safe benchmarks

E.5 Evaluation using variations of mutual exclusion protocols

In this section, we consider mutual exclusion protocols from the SV-COMP 2018 benchmarks. The unfenced versions of the protocols are *unsafe*. All the tools considered report a bug for these examples within two seconds. We now consider variations of these benchmarks.

In Table 13, we evaluate the Peterson and Szymanski protocols for N processes and keep all but one process fenced. This leads to a lower fraction of buggy executions. The values of K taken for these benchmarks assert the fact that there are bugs to be found (even for non-trivial examples) with small K. We call these examples peterson1U and szymanski1U, parameterized by

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
peterson1U(4)	1	4	1.868s	0.005s	TO	0.113s
peterson1U(6)	1	4	9.408s	0.005s	TO	0.179s
peterson1U(8)	1	4	43.680s	TO	TO	5.432s
peterson1U(10)	1	4	4m12s	TO	TO	TO
szymanski1U(4)	1	2	1.280s	0.008s	-	0.130s
szymanski1U(6)	1	2	3.519s	TO	-	TO
szymanski1U(8)	1	2	7.574s	TO	TO	TO
szymanski1U(10)	1	2	15.437s	TO	TO	TO

Table 13. Evaluation using mutual exclusion protocols with a single unfenced process

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
peterson1C(3)	1	2	0.743s	0.012s	0.085s	0.786s
peterson1C(4)	1	2	1.827s	5.032s	TO	4.157s
peterson1C(5)	1	2	4.185s	59m42s	TO	TO
peterson1C(6)	1	2	8.483s	TO	TO	TO
peterson1C(7)	1	2	15.678s	TO	TO	TO
peterson2C(3)	1	2	0.758s	0.005s	0.068s	0.061
peterson2C(4)	1	2	1.848s	0.015s	TO	12.308s
peterson2C(5)	1	2	4.041s	1m36s	TO	TO
peterson2C(6)	1	2	7.562s	TO	TO	TO
peterson2C(7)	1	2	14.729s	TO	TO	TO

Table 14. Evaluation using completely fenced peterson mutual exclusion protocol with a bug introduced in the critical section of a single process

the number of processes. Table 14 exhibits a pair of benchmarks that exhibit the sensitivity of DPOR-based algorithms to the location of bugs. We consider the completely fenced version of the Peterson protocol. However, we introduce a bug (write a value to a shared variable and read a different value from it) in the critical section of one of the processes. Between the two examples, the only difference is the process in which this bug has been introduced. We call these examples peterson1C and peterson2C, parameterized by the number of processes. We can see the difference in the performance of the DPOR-based tools (especially CDSChecker) on the two examples. On the other hand, our tool is resilient to such superficial changes. We note again that the value of K is small (2).

benchmark	L	K	SwInG	CDSChecker	GenMC	RCMC
szymanski(3)	1	2	0.690s	0.047s	28.886s	2m35s
szymanski(4)	1	2	1.121s	5m25s	-	TO
szymanski(5)	1	2	1.795	TO	-	TO
szymanski(6)	1	2	2.671s	TO	-	TO
szymanski(7)	1	2	3.751s	TO	-	TO

Table 15. Evaluation using completely fenced szymanski mutual exclusion protocol with a bug introduced in the critical section of a single process

We repeat in Table 15 the above experiment with the Szymanski mutual exclusion protocol.

We consider in Table 16 completely fenced versions of the mutual exclusion protocols. We note that these versions are safe due to the introduction of SC-fences. In this experiment, we sequentially increase the loop unwinding bound. These examples exhibit the practicality of iterative increments in K. Following convention, the figure in the parenthesis represents the number of processes.

benchmark	L	K	SwInG	**CDSChecker**	**GenMC**	**RCMC**
bakery(2)	1	2	0.463s	6.249s	0.056s	0.067s
lamport(2)	1	2	0.777s	5.451s	0.070s	0.089s
peterson(3)	1	2	0.878s	TO	9.665s	26.208s
peterson(2)	1	2	0.321s	0.325s	0.087s	0.068s
tbar(2)	1	2	0.240s	0.007s	0.080s	0.081s
tbar(3)	1	2	0.514s	2.077s	0.087s	0.074s
bakery(2)	2	2	0.872s	TO	0.709s	0.884s
lamport(2)	2	2	3.798s	TO	1m31s	5m5s
peterson(3)	2	2	1.695s	TO	-	TO
peterson(2)	2	2	0.539s	15m22s	0.039s	0.428s
tbar(2)	2	2	0.375s	0.504s	0.044s	0.061s
tbar(3)	2	2	0.918s	TO	0.080s	0.094s
bakery(2)	4	2	5.827s	TO	TO	TO
lamport(2)	4	2	5m31s	TO	TO	TO
peterson(3)	4	2	15.900s	TO	-	TO
peterson(2)	4	2	3.412s	TO	TO	TO
tbar(2)	4	2	1.578s	41m25s	0.262s	0.071s
tbar(3)	4	2	4.741s	TO	6.460s	15.489s

Table 16. Evaluation using safe mutual exclusion protocols