On the Verification Problem for Weak Memory Models

Mohamed Faouzi Atig Ahmed Bouajjani
LIAFA, University Paris Diderot, Paris, France
{atig,abou}@liafa.jussieu.fr

Sebastian Burckhardt Madanlal Musuvathi
Microsoft Research, Redmond, WA, USA
{sburckha,mandanm}@microsoft.com

Abstract

We address the verification problem of finite-state concurrent programs running under weak memory models. These models capture the reordering of program (read and write) operations done by modern multi-processor architectures for performance. The verification problem we study is crucial for the correctness of concurrency libraries and other performance-critical system services employing lock-free synchronization, as well as for the correctness of compiler backends that generate code targeted to run on such architectures.

We consider in this paper combinations of three well-known program order relaxations. We consider first the "write to read" relaxation, which corresponds to the TSO (Total Store Ordering) model. This relaxation is used in most hardware architectures available today. Then, we consider models obtained by adding either (1) the "write to write" relaxation, leading to a model which is essentially PSO (Partial Store Ordering), or (2) the "read to read/write" relaxation, or (3) both of them, as it is done in the RMO (Relaxed Memory Ordering) model for instance.

We define abstract operational models for these weak memory models based on state machines with (potentially unbounded) FIFO buffers, and we investigate the decidability of their reachability and their repeated reachability problems.

We prove that the reachability problem is decidable for the TSO model, as well as for its extension with "write to write" relaxation (PSO). Furthermore, we prove that the reachability problem becomes undecidable when the "read to read/write" relaxation is added to either of these two memory models, and we give a condition under which this addition preserves the decidability of the reachability problem. We show also that the repeated reachability problem is undecidable for all the considered memory models.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification.

General Terms Verification, Theory, Reliability.

Keywords Program verification, Relaxed memory models, Infinite state systems, Lossy channel systems.

1. Introduction

Shared-memory multiprocessor architectures are now ubiquitous. For performance reasons, most contemporary multiprocessors implement relaxed memory consistency models [Adve and Gharachorloo 1996]. Such memory models relax the ordering guarantees of memory accesses. For example, the most common relaxation is that writes to shared memory may be delayed past subsequent reads from memory. This write-to-read relaxation is commonly attributed to *store buffers* between each processor and the main memory. The corresponding memory model is historically called TSO, for total-store-order. Similarly, many models relax under certain conditions read-to-read order, read-to-write order, and write-to-write order.

Programmers usually assume that all accesses to the shared memory are performed instantaneously and atomically, which is guaranteed only by the strongest memory model, sequential consistency (SC) [Lamport 1979]. Nevertheless, this assumption is in fact safe for most programs. The reason is that the recommended methodology for programming shared memory (namely, to use threads and locks in such a manner as to avoid data races) is usually sufficient to hide the effect of memory ordering relaxations. This effect is known as the DRF guarantee, because it applies to data-race-free programs.

However, while very useful for mainstream programs, the DRF guarantee does not apply in all situations. For one, the implementors of the synchronization operations need to be fully aware of the hardware relaxations to ensure sufficient ordering guarantees (it is their responsibility to uphold the DRF guarantee). For example, Dekker's mutual exclusion protocol does not function correctly on TSO architectures (Fig. 1). Secondly, many concurrency libraries and other performance-critical system services (such as garbage collectors) bypass conventional locking protocol and employ lock-free synchronization techniques instead. Such algorithms need to be aware of the memory model. They may either be immune to the relaxations by design, or contain explicit memory ordering fences to prevent them. Most algorithms choose the latter option; however, two recent implementations of a work-stealing queue [Michael et al. 2009, Leijen et al. 2009] are using algorithms that are specifically written to perform well on TSO architectures without requiring fences.

Reasoning about the behavior of such algorithms on relaxed memory models is much more difficult than for sequentially consistent memory, and it is not clear how to apply standard reasoning techniques or finite-state abstractions. This highlights the need for more research on automatic verification techniques for programs on relaxed memory models [Burckhardt et al. 2007, Huynh and Roychoudhury 2006, Park and Dill 1995, Yang et al. 2004].

Classic results show that for finite-state programs under SC, the reachability problem, as well as the repeated reachability problem (relevant for checking liveness properties), are both PSPACE-complete [Sistla and Clarke 1985]. To our knowledge, no analogous decidability/complexity results are known for relaxed memory models. We thus investigate in this paper the verification problem for several variations of shared-memory systems with different relaxations.

We start by building a formal model of concurrent finite-state programs executing on a TSO system that is, a shared-memory sys-

POPL'10, January 17–23, 2010, Madrid, Spain.

Initially: X = Y = 0

Proc1	Proc2
a: X = 1	p: Y = 1
b: r1 = Y	q: r2 = X
c: if(r1 == 0) enter crit	r: if(r2 == 0) enter crit

Eventually: r1 = r2 = 0 and X = Y = 1

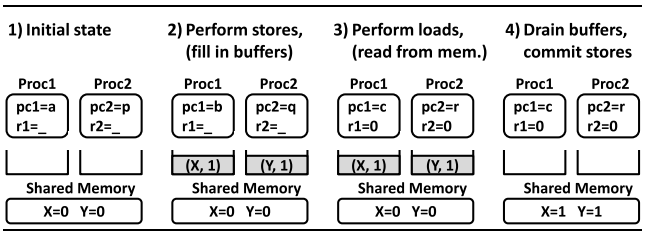

Figure 1. Snippet of Dekker's mutual exclusion protocol (top). The sequence (bottom) is possible on most contemporary multiprocessors and shows how the protocol fails to achieve mutual exclusion on TSO.

tem with the $\mathsf{w} \to \mathsf{r}$ relaxation. This model consists of a finite-state control representing the local program states of each thread along with unbounded FIFO queues representing the contents of the store buffers. Note that although the store buffers in actual machines are necessarily finite, we may not assume any fixed bound, so a finite-state model is not sufficient to verify a general algorithm. At this point, it is not clear yet whether the reachability/repeated reachability problems is decidable at all, because general FIFO queue automata are Turing powerful.

To solve this problem, we establish a close connection between TSO systems and *lossy* FIFO channel machines. Specifically, we first prove that TSO systems can be simulated by lossy FIFO channel machines, which implies that their control point reachability problem is decidable [Abdulla and Jonsson 1993]. The translation to lossy channel machines is not trivial since turning simply (TSO) store buffers to lossy channels is obviously unsound. Conversely, we prove that TSO systems can simulate lossy FIFO channel machines, which implies that the complexity of their reachability problem is non-primitive recursive [Schnoebelen 2002] and that their repeated reachability problem (and therefore their verification problem for liveness properties) is undecidable [Abdulla and Jonsson 1994].

Next, we consider models obtained by adding either (1) the $\mathsf{w} \to \mathsf{w}$ relaxation (leading to a model which is essentially PSO), or (2) the $\mathsf{r} \to \mathsf{r}/\mathsf{w}$ relaxation, or (3) both of them (as it is done in the RMO model for instance).

In fact, it is not difficult to show that all the results for TSO hold also for its write-to-write relaxation (PSO). Moreover, we prove that adding the $\mathsf{r} \to \mathsf{r}/\mathsf{w}$ relaxation to either TSO or PSO makes the reachability problem undecidable. This is due to the fact that allowing writes to overtake reads amounts to guess for each delayed read operation the value it will have later when it will be executed. Then, undecidability comes from the fact that there might be an unbounded number of such guessed read values. Intuitively, allowing writes to overtake an unbounded number of reads requires memorizing unbounded sequences of values (the guessed read values) in order to validate *all* of them later, and this requires in some sense the use of *perfect* unbounded FIFO queues (instead of lossy ones). However, when we impose that at each point in time the number of guessed read values is bounded, then the state reachability problem becomes decidable (again by reduction to the reachability in lossy channel machines) since the amount of information that needs to be stored in a "perfect" manner is always bounded (and can be encoded using strong symbols in the channels).

1.1 Relation to Commercial Multiprocessor Architectures

In general, it is quite difficult to establish a precise relationship between abstract memory models (described as a collection of relaxation rules in the style of [Adve and Gharachorloo 1996]) and actual memory models of commercially available multiprocessors. Many of those models are not fully formalized. The official specifications are often quite complicated, incomplete, and sometimes simply incorrect [Owens et al. 2009]. Since the goal of this paper is foundational research, we focus on building precise and simple models that contain just a few of the most common relaxations.

Our main model (a shared-memory system with the $\mathsf{w} \to \mathsf{r}$ relaxation) corresponds to the SPARC TSO weak memory model [Weaver and Germond 1994] and the latest formalization of the x86 memory model, x86-TSO [Owens et al. 2009]. Traditionally, TSO is specified using both formal axiomatic and informal operational descriptions [Burckhardt and Musuvathi 2008. Extended Version as Tech Report MSR-TR-2008-12, Microsoft Research, Park and Dill 1995, Sindhu et al. 1991, Weaver and Germond 1994]. Equivalence proofs of the two appear in [Owens et al. 2009] and the appendix of [Burckhardt and Musuvathi 2008. Extended Version as Tech Report MSR-TR-2008-12, Microsoft Research]. Our operational model follows the general structure of those models (but adds a finite-state control to represent the program state). Note that our operations are sufficient to model most synchronization operations on SPARC TSO and x86-TSO: the atomic read-write can represent the atomic SWAP, LDSTUB, and CAS instruction on SPARC TSO and locked operations on x86-TSO, and it can simulate a full fence (because a fence is equivalent to executing an atomic-read-write to an irrelevant location).

1.2 Related Work

Previous work on verifying programs on relaxed memory models has used underapproximation to keep the problem finite-state, and falls into two categories: (1) explicit-state model checking with operational finite-state models [Huynh and Roychoudhury 2006, Park and Dill 1995], and (2) bounded model checking with axiomatic memory models [Burckhardt et al. 2007, Yang et al. 2004]. Our work, in contrast, precisely handles the infinite state introduced by unbounded store buffers.

Much previous work has also addressed the loosely related, but qualitatively quite different problem of verifying whether a hardware implementation properly implements a given memory model. For instance, it is known to be undecidable whether a finite-state memory system implementation guarantees sequential consistency [Alur et al. 1996]. Compared to our work, the latter work considers a tougher correctness condition (sequential consistency instead of reachability) but a simpler model of the hardware (finite state instead of FIFO queues). For practical purposes, approximative algorithms [Roy et al. 2005, Baswana et al. 2008] and the "TSOTool" [et al. 2004] are useful to check whether a given individual execution trace satisfies the desired memory model, a problem which is known to be NP-complete for SC [Gibbons and Korach 1992].

2. Preliminary definitions and notations

Let $k \in \mathbb{N}$ such that $k \geq 1$. Then, we denote by $[k]$ the set of integers $\{1, \ldots, k\}$.

Let be a finite alphabet. We denote by * (resp. $^+$) the set of all *words* (resp. non empty words) over , and by the empty word. We denote by the set $\cup\{\ \}$.

The length of a word w is denoted by $\mathit{length}(w)$. (We assume that $\mathit{length}(\ ) = 0$.) For every $i \in [\mathit{length}(w)]$, let $w(i)$ denote the

symbol at position i in w. For $a \in$ and $w \in {}^*$. We write $a \in w$ if a appears in w, i.e., $\exists i \in [length(w)]$ s.t. $a = w(i)$.

Given a sub-alphabet $\subseteq$ and a word $u \in {}^*$, we denote by $u|$ the *projection* of u over , i.e., the word obtained from u by erasing all the symbols that are not in .

A *substitution* over is a mapping from to . We write $w[\ ]$ to denote the word such that for every $i \in [length(w)]$, $w[\ ](i) = (w(i))$.This definition is generalized to sets of words as follows: For every $L \subseteq {}^*$, $L[\ ] = \{w[\ ] \mid w \in L\}$.

Assume that $= \{a_1, \ldots, a_n\}$. Then, to define the substitution applied to a word w, we use the notation $w[\ (a_1)/a_1, \ldots, \ (a_n)/a_n]$ where, for the sake of conciseness, we omit $(a_i)/a_i$ when $(a_i) = a_i$, for any $i \in [n]$. This notation is extended straightforwardly to sets of words.

Let $k \geq 1$ be an integer and E be a set. Let $\mathbf{e} = (e_1, \ldots, e_k) \in E^k$ be a k-dim vector over E. For every $i \in [k]$, we use $\mathbf{e}[i]$ to denote the i-th component of $\mathbf{e}$ (i.e., $\mathbf{e}[i] = e_i$). For every $j \in [k]$ and $e' \in E$, we denote by $\mathbf{e}[j \hookleftarrow e']$ the k-dim vector $\mathbf{e}'$ over E defined as follows: $\mathbf{e}'[j] = e'$ and $\mathbf{e}'[l] = \mathbf{e}[l]$ for all $l \neq j$.

Let E and F be two sets. We denote by $[E \to F]$ the set of all mappings from E to F. Assume that E is finite and that $E = \{e_1, \ldots, e_k\}$ for some integer $k \geq 1$. Then, we sometimes identify a mapping $\mathbf{g} \in [E \to F]$ with a k-dim vector over F (i.e., we consider that $\mathbf{g} \in F^k$ with $\mathbf{g}[i] = e_i$ for all $i \in [k]$).

3. TSO/PSO concurrent systems

We introduce in this section an operational semantics for concurrent systems under the TSO memory model (corresponding to the $\mathsf{w} \to \mathsf{r}$ program order relaxation). A similar operational model can be defined in order to take into account both $\mathsf{w} \to \mathsf{r}$ and $\mathsf{w} \to \mathsf{w}$ relaxations, leading to the PSO memory model.

3.1 Shared memory concurrent systems

Let D be a finite data domain, and let $X = \{x_1, \ldots, x_m\}$ be a finite set of variables valued in D. From now on, we denote by M the set D^m, i.e., the set of all possible valuations of the variables in X.

Then, for a given finite set of process identities I, let (I, D, X) be the smallest set of operations which contains (1) the *"no operation"* nop, (2) the *read operations* $\mathsf{r}(i,x,d)$, (3) the *write operations* $\mathsf{w}(i,x,d)$, and (4) the *atomic read-write operations* $\mathsf{arw}(i,x,d,d')$, where $i \in I$, $x \in X$, and $d, d' \in D$.

A *concurrent system* over D and X is a tuple $\mathcal{N} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ where for every $i \in [n]$, $\mathcal{P}_i = (P_i, \ _i)$ is a finite-state process where (1) P_i is a finite set of control states, and (2) $\ _i \subseteq P_i \times (\{i\}, D, X) \times P_i$ is a finite set of labeled transition rules. For convenience, we write $p \xrightarrow{op}_i p'$ instead of $(p, op, p') \in \ _i$, for any $p, p' \in P_i$ and $op \in (\{i\}, D, X)$.

The behaviors of concurrent systems are usually defined according to the interleaving semantics. The weak semantics corresponding to the $\mathsf{w} \to \mathsf{r}$ program order relaxation is obtained by allowing that read operations "overtake" (along a computation) write operations performed by the same process when these operations concern different variables, i.e., a sequence of operations $\mathsf{w}(i,y,d')\mathsf{r}(i,x,d)$ (executable from left to right), where $x \neq y$, can always be replaced by the sequence $\mathsf{r}(i,x,d)\mathsf{w}(i,y,d')$. However, in this semantics the order between write operations performed by the same process must be maintained, and each atomic read-write is considered as an operation that cannot permute with no other operation of any kind. Moreover, a "cancellation rule" is applied which consists in considering that any sequence $\mathsf{w}(i,x,d)\mathsf{r}(i,x,d)$ of a write followed by a read of the same value to the same variable by the same process is equivalent to the operation $\mathsf{w}(i,x,d)$. In the next section, we define an operational model that captures this weak memory model.

3.2 An operational model for TSO

Our operational model consists in associating with each process a FIFO buffer. This buffer is used to store the write operations performed by the process (Write to store). Memory updates are then performed by choosing nondeterministically a process and by executing the first write operation in its buffer (Update). A read operation by the process $\mathcal{P}_i$ to the variable x_j can overtake the write operations stored in its own buffer if all these operations concern variables that are different from x_j (Read memory). When the buffer contains some write operations to x_j, then the read value must correspond to the value of the last of such a write operation (Read own write). Finally, atomic read-write operation can be executed only when the buffer is empty (ARW).

Let us now define formally the model. Let $\mathbf{P} = P_1 \times \cdots \times P_n$ and for every $i \in [n]$, let $B_i = \{i\} \times [m] \times D$ be the alphabet of the store buffer associated with $\mathcal{P}_i$. A *configuration* of $\mathcal{N}$ is a tuple $\langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle$ where $\mathbf{p} \in \mathbf{P}$, $\mathbf{d} \in M$, and $\mathbf{u} \in B_1^* \times \cdots \times B_n^*$ is a valuation of the store buffers.

We define the transition relation $\Rightarrow_{\mathcal{N}}$ on configurations of $\mathcal{N}$ to be the smallest relation such that, for every $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, for every $\mathbf{u}, \mathbf{u}' \in B_1^* \times \cdots \times B_n^*$, and for every $\mathbf{d}, \mathbf{d}' \in M$, we have $\langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle \Rightarrow_{\mathcal{N}} \langle \mathbf{p}', \mathbf{d}', \mathbf{u}' \rangle$ if there is an $i \in [n]$, and there are $p, p' \in P_i$, such that $\mathbf{p}[i] = p$, $\mathbf{p}' = \mathbf{p}[i \hookleftarrow p']$, and one of the following cases hold:

1. Nop: $p \xrightarrow{\mathsf{nop}}_i p'$, $\mathbf{d} = \mathbf{d}'$, and $\mathbf{u} = \mathbf{u}'$.
2. Write to store: $p \xrightarrow{\mathsf{w}(i,x_j,d)}_i p'$, $\mathbf{u}' = \mathbf{u}[i \hookleftarrow (i,j,d)\mathbf{u}[i]]$, and $\mathbf{d} = \mathbf{d}'$.
3. Update: $p = p'$, and $\exists j \in [m].\ \exists d \in D.\ \mathbf{u} = \mathbf{u}'[i \hookleftarrow \mathbf{u}'[i](i,j,d)]$ and $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d]$
4. Read: $p \xrightarrow{\mathsf{r}(i,x_j,d)}_i p'$, $\mathbf{d} = \mathbf{d}'$, $\mathbf{u} = \mathbf{u}'$, and
   - Read own write: $\exists u_1, u_2 \in B_i^*.\ \big(\mathbf{u}[i] = u_1(i,j,d)u_2$ and $\forall (i,k,d') \in u_1.\ k \neq j\big)$.
   - Read memory: $\forall (i,k,d') \in \mathbf{u}[i].\ k \neq j$, and $\mathbf{d}[j] = d$.
5. ARW: $p \xrightarrow{\mathsf{arw}(i,x_j,d,d')}_i p'$, $\mathbf{u}[i] = \ $, $\mathbf{u} = \mathbf{u}'$, and $\mathbf{d}[j] = d$, and $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d']$.

Let $\Rightarrow_{\mathcal{N}}^*$ denote the reflexive-transitive closure of $\Rightarrow_{\mathcal{N}}$.

In the rest of the paper, we call $(\mathsf{w} \to \mathsf{r})$-*relaxed memory system* a concurrent system $\mathcal{N}$ with the operational semantics induced by the transition relation $\Rightarrow_{\mathcal{N}}$ defined above.

3.3 Adding the $\mathsf{w} \to \mathsf{w}$ relaxation

The $\mathsf{w} \to \mathsf{w}$ relaxation consists in allowing that write operations overtake other write operations by the same process if they concern different variables. The consideration of both $\mathsf{w} \to \mathsf{r}$ and $\mathsf{w} \to \mathsf{w}$ relaxations leads to a memory model which is essentially the PSO model.

An operational model for $(\mathsf{w} \to \mathsf{r}/\mathsf{w})$-relaxed memory systems can be defined by modifying the model in the subsection 3.2 so that each process has m store buffers instead of a single one (one per variable). Then, two consecutive write operations $\mathsf{w}(i,x_j,d)$ and $\mathsf{w}(i,x_k,d')$ by the same process $\mathcal{P}_i$ are stored in two different buffers, which allows to reorder these write operations. We omit here the formal description of the model since it should be quite obvious to the reader.

3.4 Reachability problems

We assume that we are given a $(\mathsf{w} \to \mathsf{r})$-relaxed memory system $\mathcal{N}$. Then, the *state reachability problem* is to determine, for two given vectors of control states $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, and two given memory states $\mathbf{d}, \mathbf{d}' \in M$, whether $\langle \mathbf{p}, \mathbf{d}, \ ^n \rangle \Rightarrow_{\mathcal{N}}^* \langle \mathbf{p}', \mathbf{d}', \ ^n \rangle$.

The *repeated state reachability problem* is to determine, for given $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$ and $\mathbf{d}, \mathbf{d}' \in M$, whether there is an infinite sequence $\langle \mathbf{p_0}, \mathbf{d_0}, \mathbf{u_0} \rangle \langle \mathbf{p_1}, \mathbf{d_1}, \mathbf{u_1} \rangle \cdots$ such that: (1) $\langle \mathbf{p}, \mathbf{d}, \ ^n \rangle \Rightarrow^*_{\mathcal{N}} \langle \mathbf{p_0}, \mathbf{d_0}, \mathbf{u_0} \rangle$, and (2) $\langle \mathbf{p_i}, \mathbf{d_i}, \mathbf{u_i} \rangle \Rightarrow^*_{\mathcal{N}} \langle \mathbf{p_{i+1}}, \mathbf{d_{i+1}}, \mathbf{u_{i+1}} \rangle$, $\mathbf{p_i} = \mathbf{p}'$, and $\mathbf{d_i} = \mathbf{d}'$ for all $i \geq 0$.

The definitions of the two problems above can be extended straightforwardly to the case of $(\mathsf{w} \rightarrow \mathsf{r}/\mathsf{w})$-relaxed memory system. We address in the next sections the decidability of these problems.

Notice that we consider in the definition of the state reachability problem that the buffers at the targeted configuration must be empty instead of being arbitrary. This is only for the sake of simplicity and does not constitute at all a restriction. Indeed, we can show that the "arbitrary buffer" state reachability problem is reducible to the "empty buffer" state reachability problem. The idea of the reduction is to add to the system a special process, let us call it o (for observer), that guesses nondeterministically the moment when the system reaches the targeted control and memory states. An additional shared variable is used as a flag. This flag is (1) checked by all the original processes in the system before each of their operations, and (2) it is switched (using an atomic read-write operation) by o when it guesses that the target state is reached in order to signal to the other processes that they must stop their computations (and stay at their control location). After switching the flag, the process o checks that the memory state is indeed the targeted one. Then, since the buffers can always be flushed by executing all pending write operations (which potentially modifies the memory state but keeps unchanged the control states of all processes), it suffices to check (as an "empty buffer" state reachability problem) that each process is at its targeted control state (and some memory state among the finitely many possible states).

4. Perfect/Lossy Channel Machines

FIFO channel machines are finite control machines supplied with unbounded FIFO queues on which they can perform send and receive operations. When the channels are *lossy*, symbols in the channels can be lost at any position and at any time. We give in this section the formal definition of these machines, and we recall results concerning the decidability and the complexity of their reachability and repeated reachability problems.

In fact, we need to use in this paper a version of these machines which extends (syntactically) the basic version usually considered in the literature (e.g., in [Abdulla and Jonsson 1993, 1994, Schnoebelen 2002]). In our version, machines can also check regular constrains on the channels, and apply a substitution on symbols occurring in the channels. Moreover, we admit that there is a bounded number of special symbols (called strong symbols) that cannot be lost by the channels. The extensions that we consider do not increase the power of the lossy channel machines, but they are useful for describing our results in the next sections.

4.1 Channel constraints and operations

Let C be a set of channels. We consider that the content of each channel is a word over (i.e., an element of *).

Guards: For a given channel $c \in C$, a *regular guard* on c is a constraint of the form "$c \in L$", where $L \subseteq \ ^*$ is a regular set of words. Given a guard "$c \in L$", and a word $u \in \ ^*$, we write $u \models$ "$c \in L$" if and only if $u \in L$.

For notational convenience, we write (1) "$a \in c$" instead of "$c \in \ ^* a \ ^*$", (2) "$c = \ $" instead of "$c \in \{ \ \}$", and (3) "$c : A$" instead of "$c \in A^*$", for any $A \subseteq \ $.

A regular guard over C is a mapping $\mathbf{g}$ that associates a regular guard with each channel $c \in C$. Let $Guard(C)$ be the set of regular guards over C. The definition of $\models$ is extended to regular guards over C and to mappings from C to * as follows: For every $\mathbf{g} \in Guard(C)$ and $\mathbf{u} \in [C \rightarrow \ ^*]$, we write $\mathbf{u} \models \mathbf{g}$ iff $\mathbf{u}(c) \models \mathbf{g}(c)$ for all $c \in C$.

Operations: For a given channel $c \in C$, a *channel operation* on c is either a nop (no operation), or an operation of the form $c?a$ (receive) for some $a \in \ $, or of the form $c[\ ]!a$ (send), where $a \in$ and is a substitution over . We write simply $c!a$ instead of $c[\ ]!a$ when is the identity substitution.

The operation $c?a$ checks if the first element of c is equal to a and then erases it, whereas $c[\ ]!a$ applies the substitution to c and then adds a to the end of the channel (as the last element). Formally, we associate with each operation a relation over words (channel contents) as follows: For every $u, u' \in \ ^*$, we have (1) $[\![\mathsf{nop}]\!](u,u')$ iff $u = u'$, (2) $[\![c[\ ]!a]\!](u,u')$ iff $u' = a \cdot u[\ ]$, and (3) $[\![c?a]\!](u,u')$ iff $u = u' \cdot a$.

A channel operation over C is a mapping $\mathbf{op}$ that associates with each channel $c \in C$ a channel operation on c. Let $Op(C)$ be the set of channel operations over C. The definition of $[\![\cdot]\!]$ is extended to channel operations over C and to pairs of mappings from C to * as follows: For every $\mathbf{g} \in Guard(C)$ and $\mathbf{u}, \mathbf{u}' \in [C \rightarrow \ ^*]$, we have $[\![\mathbf{op}]\!](\mathbf{u}, \mathbf{u}')$ if and only if $[\![\mathbf{op}(c)]\!](\mathbf{u}(c), \mathbf{u}'(c))$ holds for all $c \in C$.

4.2 FIFO channel machines

A *channel machine* is a tuple $\mathcal{M} = (Q, C, \ , \ , \ )$ where Q is a finite set of control states, C is a finite set of channels, is a finite channel alphabet, is a finite set of transition labels, and $\subseteq Q \times \ \times Guard(C) \times Op(C) \times Q$ is a set of transitions. We write $q \xrightarrow{\ell, \mathbf{g} \mid \mathbf{op}} \ q'$ instead of $(q, \ell, \mathbf{g}, \mathbf{op}, q') \in \ $.

A *configuration* of $\mathcal{M}$ is a pair $\langle q, \mathbf{u} \rangle$ where $q \in Q$ and $\mathbf{u} \in [C \rightarrow \ ^*]$. Let $Conf(\mathcal{M})$ denote the set of configuration of $\mathcal{M}$. Given a configuration $= \langle q, \mathbf{u} \rangle$, let $State(\ ) = q$. This definition is generalized to sequences of configurations as follows: For every $_1, \ldots, \ _m \in Conf(\mathcal{M})$, $State(\ _1 \cdots \ _m) = State(\ _1) \cdots State(\ _m)$.

For every $\ell \in \ $, the transition relation $\stackrel{\ell}{\Longrightarrow}_{\mathcal{M}}$, between configurations of $\mathcal{M}$, is defined as follows: For every $q, q' \in Q$ and $\mathbf{u}, \mathbf{u}' \in [C \rightarrow \ ^*]$, $\langle q, \mathbf{u} \rangle \stackrel{\ell}{\Longrightarrow}_{\mathcal{M}} \langle q', \mathbf{u}' \rangle$ if and only if there is $q \xrightarrow{\ell, \mathbf{g} \mid \mathbf{op}} \ q'$ such that $\mathbf{u} \models \mathbf{g}$ and $[\![\mathbf{op}]\!](\mathbf{u}, \mathbf{u}')$. Let $\Rightarrow_{\mathcal{M}} = \bigcup_{\ell \in} \stackrel{\ell}{\Longrightarrow}_{\mathcal{M}}$ and let $\Rightarrow^*_{\mathcal{M}}$ be the reflexive-transitive closure of $\Rightarrow_{\mathcal{M}}$.

Given $\langle q, \mathbf{u} \rangle, \langle q', \mathbf{u}' \rangle \in Conf(\mathcal{M})$, a *finite run* of $\mathcal{M}$ from $\langle q, \mathbf{u} \rangle$ to $\langle q', \mathbf{u}' \rangle$ is a finite sequence $\langle q_0, \mathbf{u}_0 \rangle \ell_1 \langle q_1, \mathbf{u}_1 \rangle \ell_2 \cdots \ell_m \langle q_m, \mathbf{u}_m \rangle$ such that the following conditions are satisfied: (1) $q_0 = q$ and $q_m = q'$, (2) $\mathbf{u}_0 = \mathbf{u}$ and $\mathbf{u}_m = \mathbf{u}'$, and (3) $\langle q_{i-1}, \mathbf{u}_{i-1} \rangle \stackrel{\ell_i}{\Longrightarrow}_{\mathcal{M}} \langle q_i, \mathbf{u}_i \rangle$ for all $i \in [m]$. The *trace* of a finite run is the sequence of labels $\mid$. We write $\langle q, \mathbf{u} \rangle \stackrel{*}{\Longrightarrow}_{\mathcal{M}} \langle q', \mathbf{u}' \rangle$ when there is a run from $\langle q, \mathbf{u} \rangle$ to $\langle q', \mathbf{u}' \rangle$ with a trace (i.e., $\mid = \ $).

Let $q, q' \in Q$ be two control states. We denote by $T_{(q,q')}(\mathcal{M})$ the set of traces of all finite runs of $\mathcal{M}$ from the configuration $\langle q, \ ^{|C|} \rangle$ to the configuration $\langle q', \ ^{|C|} \rangle$.

Given a control state $q \in Q$, an *infinite run* of $\mathcal{M}$ starting from q is an infinite sequence $\langle q_0, \mathbf{u}_0 \rangle \ell_1 \langle q_1, \mathbf{u}_1 \rangle \ell_2 \cdots$ such that $q_0 = q$, $\mathbf{u}_0 = \ ^{|C|}$, and $\langle q_{i-1}, \mathbf{u}_{i-1} \rangle \stackrel{\ell_i}{\Longrightarrow}_{\mathcal{M}} \langle q_i, \mathbf{u}_i \rangle$ for all $i \geq 1$.

4.3 Lossy Channel Machines

In this section, we define the semantics of channel machines when the channels may loose some of their contents.

Subword relations: Let $\preceq \subseteq \ ^* \times \ ^*$ be the *subword relation* defined as follows: For every $u = a_1 \cdots a_n \in \ ^*$, and every $v = b_1 \cdots b_m \in \ ^*$, $u \preceq v$ if and only if there are $i_1, \ldots, i_n \in [m]$ such that $i_1 < i_2 < \ldots < i_n$ and $a_j = b_{i_j}$ for all $j \in [n]$.

Let $S \subseteq$ be a set of "*strong symbols*" and let $k \in \mathbb{N}$. Then, for every $u, v \in {}^*$, let $u \preceq_S^k v$ hold if and only if $u \preceq v$, $u|_S = v|_S$, and $length(u|_S) \leq k$. (Notice that $\preceq_\emptyset^0 = \preceq$.)

The subword relations defined above are extended to mappings from C to * as follows: For every $\mathbf{u}, \mathbf{v} \in [C \to {}^*]$, $\mathbf{u} \preceq_S^k \mathbf{v}$ holds if and only if $\mathbf{u}(c) \preceq_S^k \mathbf{v}(c)$ holds for all $c \in C$.

Lossy transitions: Now, we define a transition relation $\overset{\ell}{\Longrightarrow}_{(\mathcal{M},S,k)}$ between configurations of $\mathcal{M}$ by allowing that the channels can loose some of their symbols, provided that these symbols are not in S, and under the restriction that the number of S symbols in each channel is bounded by k. Formally, for every $\ell \in$, $q, q' \in Q$, and $\mathbf{u}, \mathbf{u}' \in [C \to {}^*]$, $\langle q, \mathbf{u} \rangle \overset{\ell}{\Longrightarrow}_{(\mathcal{M},S,k)} \langle q', \mathbf{u}' \rangle$ if and only if there are $\mathbf{v}, \mathbf{v}' \in [C \to {}^*]$ such that $\mathbf{v} \preceq_S^k \mathbf{u}$, $\mathbf{v} \overset{\ell}{\Longrightarrow}_{\mathcal{M}} \mathbf{v}'$, and $\mathbf{u}' \preceq_S^k \mathbf{v}'$.

The notions of a finite/infinite run and of a finite trace are defined as in the non-lossy case by replacing $\Rightarrow_{\mathcal{M}}$ with $\Rightarrow_{(\mathcal{M},S,k)}$. Given two control states $q, q' \in Q$, we denote by $LT^{(S,k)}_{(q,q')}(\mathcal{M})$ the set of traces of all finite runs of $\mathcal{M}$ from the configuration $\langle q, \ ^{|C|} \rangle$ to the configuration $\langle q', \ ^{|C|} \rangle$ according to the semantics defined by $\Rightarrow_{(\mathcal{M},S,k)}$. Notice that, by definition of $\Rightarrow_{(\mathcal{M},S,k)}$, in all reachable configurations along runs of $\mathcal{M}$, the channels contain less than k symbols in S.

In the rest of the paper, we say that $\mathcal{M}$ is an (S,k)-LCM (or simply a LCM if no confusion is possible) when its operational semantics is defined by $\Rightarrow_{(\mathcal{M},S,k)}$.

Basic Lossy Channel Machines: A basic LCM is a $(\emptyset, 0)$-LCM where all the guards are trivial (i.e., of the form $c \in {}^*$) and all the substitutions in the send operations $c[\ ]!a$ are equal to the identity substitution. We can prove the following fact.

PROPOSITION 1. *Let $\mathcal{M} = (Q, C, \ , \ , \ )$ be a (S,k)-LCM for some $S \subseteq$ and $k \in \mathbb{N}$. Then, it is possible to construct a basic LCM $\mathcal{M}' = (Q', C, \ ', \ , \ ')$, with $Q \subseteq Q'$ and $\subseteq \ '$, such that $LT^{(S,k)}_{(q,q')}(\mathcal{M}) = LT^{(\emptyset,0)}_{(q,q')}(\mathcal{M}')$ for all $q, q' \in Q$.*

Applying substitutions can be easily simulated using channel rotations. A channel rotation over $c \in C$ corresponds to send a special marker $\sharp$ to c to delimit the current tail position, and then iterate, using some extra control states, a sequence of receive, check/substitute, and send operations, until $\sharp$ is found. Channel rotations are also used to check regular guards. Given a guard $c \in L$, the machine $\mathcal{M}'$ uses during the rotation of c the content of this channel as input to simulate the runs of some given finite state automaton that recognizes the regular language L. Then, if the marker $\sharp$ is encountered in a non accepting state of this automaton, $\mathcal{M}'$ goes to a special blocking control state. To guarantee that all the strong symbols in the channels of $\mathcal{M}'$ are not lost, the machine $\mathcal{M}'$ stores in its control state (in addition to the control state of $\mathcal{M}$) their sequences corresponding to each of the channels. (Remember that these sequences are of bounded sizes by assumption). We consider that the control state of $\mathcal{M}'$ corresponding to a control state $q \in Q$ of $\mathcal{M}$ coupled with an empty sequence of strong symbols is identified with q. Then, after each simulation of an operation of $\mathcal{M}$, the machine updates the sequences of strong symbols in its control state, and also checks, using channel rotations over all its channels, that all strong symbols that are supposed to be in the channels are indeed present. If for some channel the sequence of strong symbols is different from the one stored in its control state, the machine $\mathcal{M}'$ goes to a blocking control state.

4.4 Product of channel machines:

We define the synchronous product between channel machines in the usual manner: Given two machines $\mathcal{M}_1 = (Q_1, C_1, \ , \ , \ _1)$ and $\mathcal{M}_2 = (Q_2, C_2, \ , \ , \ _2)$ such that $C_1 \cap C_2 = \emptyset$, let $\mathcal{M}_1 \otimes \mathcal{M}_2 = (Q_1 \times Q_2, C_1 \cup C_2, \ , \ , \ _{12})$ denote the product of $\mathcal{M}_1$ and $\mathcal{M}_2$ where $_{12}$ is defined by synchronizing transitions having the same label in and gathering their guards and operations (notice that they concern disjoint sets of channels), and letting -transitions asynchronous. The following fact is easy to show:

PROPOSITION 2. *Let $q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$, and let $\mathbf{q} = (q_1, q_2)$ and $\mathbf{q}' = (q_1', q_2')$. Then,*

- $T_{(\mathbf{q},\mathbf{q}')}(\mathcal{M}_1 \otimes \mathcal{M}_2) = T_{(q_1,q_1')}(\mathcal{M}_1) \cap T_{(q_2,q_2')}(\mathcal{M}_2)$, *and*
- $LT^{(S,k)}_{(\mathbf{q},\mathbf{q}')}(\mathcal{M}_1 \otimes \mathcal{M}_2) = LT^{(S,k)}_{(q_1,q_1')}(\mathcal{M}_1) \cap LT^{(S,k)}_{(q_2,q_2')}(\mathcal{M}_2)$, *for every $S \subseteq$ and $k \in \mathbb{N}$.*

The product operation $\otimes$ can be extended straightforwardly to any finite number of channel machines.

4.5 Decision problems on LCM

Let $\mathcal{M} = (Q, C, \ , \ , \ )$ be an (S,k)-LCM for some $S \subseteq$ and $k \in \mathbb{N}$.

The *control state reachability problem* is to determine whether, for two given control states q and q', there is a finite run of $\mathcal{M}$ from $\langle q, \ ^{|C|} \rangle$ to $\langle q', \ ^{|C|} \rangle$. Clearly, this is equivalent to check whether $LT^{(S,k)}_{(q,q')}(\mathcal{M}) \neq \emptyset$.

The *repeated control state reachability problem* is to determine whether, for two given control states q and q', there is an infinite run of $\mathcal{M}$ starting from $\langle q, \ ^{|C|} \rangle$ such that q' occurs infinitely often in $State(\ |_{Q \times ([C \to {}^*])})$.

PROPOSITION 3. *For every $S \subseteq$ and $k \in \mathbb{N}$, the control state reachability problem for (S,k)-LCM's is decidable, whereas their repeated control state reachability problem is undecidable.*

The proposition above follows immediately from Proposition 1 and from well-known results on the reachability and the repeated reachability problems in basic lossy channel machines [Abdulla and Jonsson 1993, 1994]. The decidability of the control state reachability problem of basic lossy channel machines is based on the theory of well-structured systems [Abdulla et al. 1996, Finkel and Schnoebelen 2001].

Actually, it is also possible to prove that this problem is decidable for (nonbasic) (S,k)-LCM directly using the same theory, without going through the simulation of Proposition 1. For that, it suffices to see that (1) $\preceq_S^k$ is a well-quasi ordering on the set of words with less than k symbols in S (which follows from the fact that $\preceq$ is well-known to be a WQO (by Higman's lemma), and that WQO's are closed under product and disjoint union), and that (2) $\preceq_S^k$ defines a simulation relation on the configurations of (S,k)-LCM's (if a configuration can perform a transition, a greater configuration w.r.t. this ordering can also perform the same transition to reach the same target). Then, by standard results in [Abdulla et al. 1996, Finkel and Schnoebelen 2001], a simple iterative backward reachability analysis procedure for (S,k)-LCM's (using finite representations of $\preceq_S^k$-upward closed sets of configurations by their minimals) is guaranteed to always terminate. It has been shown in [Schnoebelen 2002] that this procedure may take in general a non-primitive recursive time to converge. Nevertheless, efficient algorithms and tools for the analysis of well-structured systems such as vector addition systems and lossy channel machines have been developed, based on *complete* abstract reachability analysis techniques [Geeraerts et al. 2006, Ganty et al. 2006].

5. Simulating TSO/PSO by lossy channels

We show that the state reachability problem for $(\mathsf{w} \to \mathsf{r})$-relaxed memory systems can be reduced to the control state reachability problem for lossy channel machines. From this reduction and Proposition 3, we obtain the following fact.

THEOREM 1. *The state reachability problem for* $(\mathsf{w} \to \mathsf{r})$*-relaxed memory systems is decidable.*

The rest of the section is mainly devoted to the description of the reduction. We address in a last subsection the extension of Theorem 1 to the case of PSO models.

Given a concurrent system $\mathcal{N} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ over D and $X = \{x_1, \ldots, x_m\}$, we construct n lossy channel machines $\mathcal{M}_1, \ldots, \mathcal{M}_n$, one per process in $\mathcal{N}$, such that the reachability problem in $\mathcal{N}$ can be reduced to the reachability problem in the product of $\mathcal{M}_1, \ldots, \mathcal{M}_n$.

Turning simply the store buffers to lossy channels (i.e., skipping some write operations) leads to unsound memory states (w.r.t. the TSO semantics). For example, consider two processes $\mathcal{P}$ and $\mathcal{P}'$ sharing two variables x and y, and assume that the transitions of $\mathcal{P}$ are $p_0 \xrightarrow{\mathsf{w}(x,1)} p_1 \xrightarrow{\mathsf{w}(y,1)} p_2$ and that the transitions of $\mathcal{P}'$ are $p'_0 \xrightarrow{\mathsf{r}(x,1)} p'_1 \xrightarrow{\mathsf{r}(y,0)} p'_2$. (We omit here the identities of the processes in the description of the actions.) Then, assuming that the starting state is $(p_0, p'_0, x = 0, y = 0)$, it can be checked that, according the TSO semantics, the state $(p_2, p'_2, x = 1, y = 0)$ is not reachable. However, if the operation $\mathsf{w}(y,1)$ of process $\mathcal{P}$ is lost by its store buffer (if we consider it as a lossy channel), then this state becomes reachable.

In fact, lossyness can be tolerated only if the information in the channels is always sufficient to obtain sound memory state when read operations must be performed. Then, the idea is that channels should contain sequences of sound memory states. This means that for the simulation, instead of sending in the channel a write operation, a process must send the state that the memory (in the simulated system) will have right after the execution of this write operation (i.e., at the moment when in the simulated system this operation will be taken from the store buffer and used for updating the memory state). For instance, in the example above, the sequence of sound memory states that must be considered is $(x = 0, y = 1)(x = 1, y = 1)$. Assume now that process $\mathcal{P}$ has sent this sequence (from left to right) to its channel, and that the memory is updated successively by copying these states to the global store, but some state, say $(x = 0, y = 1)$, has been lost. In this case the state of the memory goes directly from $(x = 0, y = 0)$ to $(x = 1, y = 1)$. But this is perfectly sound since several memory updates are possible before any process can observe the changes in the memory state.

Now, let us see how a process can send a sequence of sound states to its channel. Obviously, if the process is the only one in the system to perform write operations (as in the example above), knowing what is the memory state after executing a write operation is easy to determine: The process can maintain in its control state the last memory state sent to the channel, and then, for the next write operation, the process can compute a new state that is (memorized in its control state and) sent to the channel. The difficulty comes of course from the fact that in presence of concurrency, the memory state that the process should send to the channel must take into account the interferences of the other processes. Therefore, each process must *guess* the sequence of memory update operations (along a computation) resulting from the write operations performed by all the processes in the system. In other words, the process has to guess the write operations by all processes as well as the order (after their interleaving) in which they will be executed. Given such a sequence of write operations, the simulation of the behavior of a process by a lossy channel machine can be done. In fact, since the buffer contains a sequence of sound memory states, loosing some of the channel content can be seen as skipping unobservable states. Indeed a process observes the memory only at the moment read operations are performed. Between these moments several changes to the memory (due to the writes sent by different processes) may occur, but even if the intermediary memory states resulting from these changes are not observed (and can be considered as lost by the channel), each observed state is sound since it accumulates the effect of all the operations performed so far.

For this simulation, a control state of $\mathcal{M}_i$ is composed by a control state of $\mathcal{P}_i$, a vector of data $\mathbf{d}_c \in M$ corresponding to the current memory state, and a vector of data $\mathbf{d}_g$ representing the (guessed) memory state that should be obtained after executing all the operations in the buffer. Then, an element of the channel alphabet of $\mathcal{M}_i$ is of the form $(k, j, \mathbf{d})$ where $k \in [n]$, $j \in [m]$ and $\mathbf{d} \in D^m$. (We will see shortly that we need also some other kind of elements.) The vector of data $\mathbf{d}$ in such an element represents the memory state supposed to be reached after executing the operation $\mathsf{w}(k, x_j, \mathbf{d}[j])$. There is however a technical issue which requires some care: In order to simulate correctly "Read own write" operations, it is necessary to forbid the loss of the states stored in the channel that correspond to the last write operation on each of the variables. Fortunately, the number of such states is bounded (since X is finite), and therefore we can consider them as strong symbols (see definition of sub-word relations). Technically, these special states in the channel are marked and have the form $((k, j, \mathbf{d}), \sharp)$; let Γ_2 be the set of these marked states. Then, the alphabet of the channel includes also Γ_2, and after each write operation, the marking in the channel must of course be maintained coherent. In order to impose that marked states are not lost, we consider $\mathcal{M}_i$ to be a (Γ_2, m)-LCM.

Now, the last step is to check that all processes have guessed the same sequence of write operations and the same ordering of their execution. For that, each machine makes visible the transitions corresponding to its own write operations, as well as to the guessed write operations (concerning the other processes), and then, the product of these machines is taken with synchronization on the alphabet of write operations. In fact, if the machines agree on the sequence of write operations, they have necessarily stored the same sequence of states in their channels. Although each channel is lossy (and the different channels may loose different elements), the sequence of observations made by each process (using the informations in its own channel and control state) is guaranteed to be sound. Therefore, the reachability problem in the original system is reducible to a reachability problem in a lossy channel machine. In order to be able to present the correctness proof (which will be given in the next section), we need to label also update transitions (and not only transitions corresponding to write operations), although this is not necessary for the decision procedure itself. (We will use this labeling to relate sequences of updates and writes along computations.)

Let us now give the formal description of the reduction.

5.1 Constructing the machines $\mathcal{M}_i$

Let $i \in [n]$. Then, $\mathcal{M}_i = (Q_i, \{c_i\}, \Gamma, \ldots, \Gamma_i)$ where:

- $Q_i = P_i \times M \times M$ where $M = D^m$.
- c_i is the single channel of $\mathcal{M}_i$.
- $\Gamma = \Gamma_1 \cup \Gamma_2$ where $\Gamma_1 = [n] \times [m] \times D^m$ and $\Gamma_2 = \Gamma_1 \times \{\sharp\}$.
- $\Gamma_i = \Gamma_{\mathsf{w}} \cup \Gamma_{\mathsf{upd}} \cup \Gamma_{\mathsf{arw}}$ where:
  - $\Gamma_{\mathsf{w}} = \{\mathsf{w}(k, x_j, d) : k \in [n], j \in [m], d \in D\}$
  - $\Gamma_{\mathsf{arw}} = \{\mathsf{arw}(k, x_j, d, d') : k \in [n], j \in [m], d, d' \in D\}$
  - $\Gamma_{\mathsf{upd}} = \{\mathsf{upd}(k, x_j, d) : k \in [n], j \in [m], d \in D\}$

- Δ_i is the smallest set of transitions such that $\forall p, p' \in Q_i$, and $\forall \mathbf{d}_c, \mathbf{d}_g \in M$,
  - Nop: If $p \xrightarrow{\mathsf{nop}}_i p'$ then $(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{\Gamma, c_i: \ |\mathsf{nop}}_{\Delta_i} (p, \mathbf{d_c}, \mathbf{d_g})$
  - Write to store: If $p \xrightarrow{op}_i p'$, where $op = \mathsf{w}(i, x_j, d)$ for some $j \in [m]$ and $d \in D$, then for every $\mathbf{d} \in M$,
    $$(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{op, (a,\sharp) \in c_i | c_i[a/(a,\sharp)]!a'}_{\Delta_i} (p', \mathbf{d_c}, \mathbf{d'_g})$$
    $$(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{op, c_i: \ |c_i!a'}_{\Delta_i} (p', \mathbf{d_c}, \mathbf{d'_g})$$
    where $a = (i, j, \mathbf{d})$, $\mathbf{d'_g} = \mathbf{d_g}[j \hookleftarrow d]$, $a' = ((i, j, \mathbf{d'_g}), \sharp)$, and $\Gamma = \Gamma \setminus (\{i\} \times \{j\} \times M)$.
  - Guess write: $\forall k \in [n].\ k \neq i, \forall j \in [m], \forall d \in D$,
    $$(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{\mathsf{w}(k, x_j, d), c_i: \ |c_i!a}_{\Delta_i} (p, \mathbf{d_c}, \mathbf{d'_g})$$
    where $\mathbf{d'_g} = \mathbf{d_g}[j \hookleftarrow d]$ and $a = (k, j, \mathbf{d'_g})$.
  - Update: $\forall k \in [n], \forall j \in [m], \forall \mathbf{d} \in M$,
    $$(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{op, c_i: \ |c_i?(k,j,\mathbf{d})}_{\Delta_i} (p, \mathbf{d'_c}, \mathbf{d_g})$$
    $$(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{op, c_i: \ |c_i?((k,j,\mathbf{d}),\sharp)}_{\Delta_i} (p, \mathbf{d'_c}, \mathbf{d_g})$$
    where $op = \mathsf{upd}(k, x_j, \mathbf{d}[j])$ and $\mathbf{d'_c} = \mathbf{d_c}[j \hookleftarrow d]$.
  - Read: If $p \xrightarrow{\mathsf{r}(i, x_j, d)}_i p'$ for some $j \in [m]$ and $d \in D$, then $\forall \mathbf{d} \in M.\ \mathbf{d}[j] = d$,
    $$(p, \mathbf{d_c}, \mathbf{d_g}) \xrightarrow{\Gamma, (a,\sharp) \in c_i | \mathsf{nop}}_{\Delta_i} (p', \mathbf{d_c}, \mathbf{d_g}) \in \Delta_i$$
    $$(p, \mathbf{d}, \mathbf{d_g}) \xrightarrow{\Gamma, c_i: \ |\mathsf{nop}}_{\Delta_i} (p', \mathbf{d}, \mathbf{d_g})$$
    where $a = (i, j, \mathbf{d})$ and $\Gamma = \Gamma \setminus (\{i\} \times \{j\} \times M)$.
  - ARW: If $p \xrightarrow{\mathsf{arw}(i, x_j, d, d')}_i p'$ for some $j \in [m]$ and $d, d' \in D$, then $\forall \mathbf{d} \in M.\ \mathbf{d}[j] = d$,
    $$(p, \mathbf{d}, \mathbf{d}) \xrightarrow{\mathsf{arw}(i, x_j, d, d'), c_i = \ |\mathsf{nop}}_{\Delta_i} (p', \mathbf{d}', \mathbf{d}')$$
    where $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d']$.

5.2 Composing the machines $\mathcal{M}_i$

To simulate the system $\mathcal{N}$, we consider for each $i \in [n]$ the (Δ_2, m)-LCM $\mathcal{M}'_i$ obtained from $\mathcal{M}_i$ by substituting each transition label $\mathsf{arw}(k, x_j, d, d')$ by a label $\mathsf{w}(k, x_j, d')$, and each label $\mathsf{upd}(k, x_j, d)$ by ε, and then we take simply the $\otimes$ product of the machines $\mathcal{M}'_i$. This ensures that the machines agree on the sequences of write operations performed in the simulated system. (Here atomic read-write operations are also considered as write operations.)

The precise link between $\mathcal{N}$ and the so obtained (Δ_2, m)-LCM is given by the following proposition:

PROPOSITION 4. *Let $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, and let $\mathbf{d}, \mathbf{d}' \in M$. Then,*

$$\langle \mathbf{p}, \mathbf{d}, \ ^n \rangle \Rightarrow^*_{\mathcal{N}} \langle \mathbf{p}', \mathbf{d}', \ ^n \rangle \text{ iff } \bigcap_{i=1}^{n} (LT^{(\Delta_2, m)}_{(q_i, q'_i)}(\mathcal{M}_i)[\ \Delta_w]) |_{\ \Delta_w} \neq \emptyset$$

where, for every $i \in [n]$, $q_i = (\mathbf{p}[i], \mathbf{d}, \mathbf{d})$ and $q'_i = (\mathbf{p}'[i], \mathbf{d}', \mathbf{d}')$, and $\Delta_w = (\mathsf{w}(k, x_j, d') / \mathsf{arw}(k, x_j, d, d'))_{k \in [n]; j \in [m]; d, d' \in D}$.

Theorem 1 follows immediately from Proposition 4. The proof of this proposition is presented in the Section 6.

5.3 The case of PSO

We prove the same result as Theorem 1 for $(\mathsf{w} \to \mathsf{r}/\mathsf{w})$-relaxed memory systems. It is indeed again possible for these systems to reduce the state reachability problem to the control state of lossy channel machines. The reduction is even simpler in this case. In fact, while turning store buffer to lossy channels is unsound for TSO systems, it can be shown that this is actually possible for PSO.

Consider again the example given in the beginning of the proof of Theorem 1. This time, since we are considering the PSO semantics, the operations $\mathsf{w}(x, 1)$ and $\mathsf{w}(y, 1)$ are stored in different buffers. Then, it is possible to reach the state $(p_2, p'_2, x = 1, y = 0)$ since it is possible to update the variable x before the variable y. Therefore, loosing the operation $\mathsf{w}(y, 1)$ does not lead in this case to an unsound state.

In general, since all the write operations in a same channel concern a same variable, skipping some operations can be seen as equivalent to executing a sequence of updates to a same variable, and therefore, the reached state corresponds to the last update to this variable (and it is necessarily sound). Some care has to be taken, however, concerning the last operation in the buffer. If the process using this buffer does not read on its corresponding variable (as in the example above), then loosing this operation is not a problem. However, to simulate in general the read operations, the last write operation in each buffer must be kept since the process must read its value if this operation has not been executed yet. Then, by marking the last symbol in each channel and considering it as a strong symbol, the translation to lossy channel machines is straightforward.

THEOREM 2. *The state reachability problem is decidable for $(\mathsf{w} \to \mathsf{r}/\mathsf{w})$-relaxed memory systems.*

6. Correctness proof

We present the proof of Proposition 4 in three steps.

First we relate the reachability problem in $(\mathsf{w} \to \mathsf{r})$-relaxed memory system to the reachability problem in perfect channel machines. We show that checking state reachability in $\mathcal{N}$ is equivalent to check the control state reachability in the product of the $\mathcal{M}_i$'s, seen as *perfect* channel machines, when they are synchronized over the udpate transitions (see Proposition 5). Showing that the product of the channel machines simulates $\mathcal{N}$ is rather straightforward. The reverse direction is proved by establishing a kind of weak simulation relation between the configurations of the two systems.

Then, we observe that when the $\mathcal{M}_i$'s are considered as perfect channel machines, the reachability problem in their synchronous product over *update* transitions, and the same problem considered in the synchronous product over *write* transitions, are mutually reducible to each other (see Proposition 6). (We consider that atomic read-writes are considered both as writes and updates.) This is simply due to the fact that for a perfect channel the sequence of inputs is always equal to the sequence of outputs.

Finally, we prove that checking the reachability problem in the synchronous product of the *perfect* machines $\mathcal{M}_i$ over the write transitions is equivalent to checking the same problem in the synchronous product of the *lossy* channel machines $\mathcal{M}_i$ over the write transitions (see Proposition 7). The proof is based on the fact that our encoding of the memory states stored in the channels is robust w.r.t. lossyness.

Let us state these fact more formally. We need first to introduce a notion of *consistent configuration*. A configuration $\langle (p, \mathbf{d_c}, \mathbf{d_g}), u \rangle \in Conf(\mathcal{M}_i)$ is consistent if, either $u = \varepsilon$ and $\mathbf{d_c} = \mathbf{d_g}$, or $u \in \{(k, j, \mathbf{d_g})w : k \neq i, j \in [m], w \in \Delta^*\}$, or $u \in \{(i, j, \mathbf{d_g}), \sharp)w : j \in [m], w \in \Delta^*\}$. Consistency means simply that the value of the memory state obtained after executing the operations in the buffer coincides with the one encoded in the control state of the configuration. In all the relations between systems described above, the configuration consistency of the channel machines is assumed. This is not a restriction since initially we consider configurations of the form $\langle (p, \mathbf{d}, \mathbf{d}), \ ^n \rangle$ that are clearly consistent, and it can easily be checked that the transitions in each $\mathcal{M}_i$ preserve consistency.

Now, let us consider the following notation. For $\in \{\mathsf{w}, \mathsf{upd}\}$, let f be the mapping from * to * such that, for every $u \in {}^*$, $f(u) = (u[\ ])|$ where

$$= (\ (k, x_j, d')/\mathsf{arw}(k, x_j, d, d'))_{k\in[n];j\in[m];d,d'\in D}$$

This definition is generalized to sets of words.

We state hereafter the relation between the reachability problems in $\mathcal{N}$ and in the product of the perfect channel machines $\mathcal{M}_i$.

PROPOSITION 5. *Let $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$ and $\mathbf{d}, \mathbf{d}' \in M$. Then, $\langle \mathbf{p}, \mathbf{d}, {}^n\rangle \Rightarrow^*_{\mathcal{N}} \langle \mathbf{p}', \mathbf{d}', {}^n\rangle$ iff $\bigcap_{i=1}^n f_{\mathsf{upd}}(T_{(q_i,q_i')}(\mathcal{M}_i)) \neq \emptyset$ where, for every $i \in [n]$, $q_i = (\mathbf{p}[i], \mathbf{d}, \mathbf{d})$ and $q_i' = (\mathbf{p}'[i], \mathbf{d}', \mathbf{d}')$.*

The "only if direction" is easy and omitted here. To prove the "if direction" we need to define a mapping which converts consistent configurations in the product of the $\mathcal{M}_i$'s to configurations in $\mathcal{N}$.

Let $\mathbf{d} \in M$, and for each $i \in [n]$, let $_i = \langle q_i, u_i\rangle$ be a consistent configuration of $\mathcal{M}_i$ such that $q_i[2] = \mathbf{d}$. Then, we define $\mu(\ _1, \ldots, \ _n)$ to be the configuration $\langle \mathbf{p}, \mathbf{d}, \mathbf{u}\rangle$ of $\mathcal{N}$ such that, for every $k \in [n]$, $\mathbf{p}[k] = q_k[1]$ and $\mathbf{u}[k] = (u_i[\ _1][\ _2])|_{B_i}$, where $_1 = (a/(a,\sharp))_{a\in\ _1}$, and $_2 = ((l, j, \mathbf{d}[j])/(l, j, \mathbf{d}))_{l\in[n];j\in[m];\mathbf{d}\in M}$. (Recall that $B_i = \{i\} \times [m] \times D$.)

Then, Proposition 5 follows from the two next lemmas.

LEMMA 1. *For every $i \in [n]$, let $_i = \langle q_i, u_i\rangle$ be a consistent configuration of $\mathcal{M}_i$. Assume that $\forall i, l \in [n].\ q_i[2] = q_l[2]$. Then, $\forall i \in [n]$, $\forall\ '_i \in Conf(\mathcal{M}_i)$, if $_i \overset{i}{\Longrightarrow}^*_{\mathcal{M}_i}\ '_i$ for some $_i \in {}^*_{\mathsf{w}}$, then $\mu(\ _1, \ldots, \ _n) \Rightarrow^*_{\mathcal{N}} \mu(\ '_1, \ldots, \ '_n)$.*

LEMMA 2. *For every $i \in [n]$, let $_i = \langle q_i, u_i\rangle$ and $'_i = \langle q_i', u_i'\rangle$ be consistent configurations of $\mathcal{M}_i$, and let $\ell_i \in {}_{\mathsf{upd}} \cup {}_{\mathsf{arw}}$. Assume that $\forall i, j \in [n].\ q_i[2] = q_j[2]$, $q_i'[2] = q_j'[2]$, and $f_{\mathsf{upd}}(\ell_i) = f_{\mathsf{upd}}(\ell_j)$. Then, $\forall i \in [n]$, if $_i \overset{\ell_i}{\Longrightarrow}_{\mathcal{M}_i}\ '_i$, then $\mu(\ _1, \ldots, \ _n) \Rightarrow_{\mathcal{N}} \mu(\ '_1, \ldots, \ '_n)$.*

Lemma 1 and 2 state that μ defines a simulation relation between $\mathcal{N}$ and the product of the $\mathcal{M}_i$ synchronized on the update transitions. Lemma 1 concerns the case of sequences of transitions without updates, whereas Lemma 2 concerns the case where the $\mathcal{M}_i$'s must synchronize on some update operation. The proofs of these lemmas are not difficult and are omitted here.

The following proposition follows, as said in the introduction of this section, from the definition of perfect fifo channels (i.e., the sequence of inputs is equal to the sequence of outputs).

PROPOSITION 6. *$\forall i \in [n], \forall q, q' \in Q_i$, $f_{\mathsf{upd}}(T_{(q,q')}(\mathcal{M}_i)) \neq \emptyset$ iff $f_{\mathsf{w}}(T_{(q,q')}(\mathcal{M}_i)) \neq \emptyset$.*

Finally, we establish the link between computations in perfect channel and lossy channel machines.

PROPOSITION 7. *For every $i \in [n]$, and for every $q, q' \in Q_i$, $f_{\mathsf{w}}(T_{(q,q')}(\mathcal{M}_i)) = f_{\mathsf{w}}(LT^{(\ _2,m)}_{(q,q')}(\mathcal{M}_i))$.*

The proof of the left-to-right inclusion is straightforward. For the other direction, we establish the following fact which states that there is a simulation relation between the lossy and the perfect channel systems.

LEMMA 3. *For every consistent configurations $\langle q, u\rangle, \langle q', u'\rangle$ of $\mathcal{M}_i$, and for every $\ell \in$, if $\langle q, u\rangle \overset{\ell}{\Longrightarrow}_{(\mathcal{M}_i,\ _2,m)} \langle q', u'\rangle$, then $\forall v \in {}^*$ s.t. $u \preceq^m_{\ _2} v$ and $\langle q, v\rangle$ is consistent, $\exists v' \in {}^*, \exists \in {}^*$ s.t. (1) $\langle q', v'\rangle$ is a consistent, (2) $u' \preceq^m_{\ _2} v'$, (3) $f_{\mathsf{w}}(\ ) = f_{\mathsf{w}}(\ell)$, and (4) $\langle q, v\rangle \Longrightarrow^*_{\mathcal{M}_i} \langle q', v'\rangle$.*

Proof. For every $j \in [m]$, let $_j = \ \setminus (\{i\} \times \{j\} \times M)$ be a set of labels and $G = g \in \{c_i : \ , c_i : \ _j, (a, \sharp) \in c_i \mid j \in [m], a \in \ _1\}$ be a set of guards. It is easy to see that, for any guard $g \in G$, if $u \models g$ then $v \models g$ since $u \preceq^m_{\ _2} v$.

Now, let us suppose that $\langle q, u\rangle \overset{\ell}{\Longrightarrow}_{\mathcal{M}_i} \langle q', u'\rangle$. This implies that there is a transition $= q \xrightarrow{\ell, g \mid op}_{\ _i} q'$ such that $u \models g$, and $[\![op]\!](u, u')$. Then, we consider the four cases depending on the type of the label ℓ:

- If $\ell =$, then $op = \mathsf{nop}$, $u = u'$, and $g \in G$. This implies that $\langle q, v\rangle \overset{\ell}{\Longrightarrow}_{\mathcal{M}_i} \langle q', v\rangle$ because $v \models g$.
- If $\ell \in {}_{\mathsf{w}}$, then $op = c_i[\ ]!a'$, for some $a' \in$ and substitution , $u' = a' \cdot u[\ ]$, and $g \in G$. This implies that $\langle q, v\rangle \overset{\ell}{\Longrightarrow}_{\mathcal{M}_i} \langle q', v'\rangle$ with $v' = a' \cdot v[\ ]$ since $v \models g$.
- If $\ell \in {}_{\mathsf{arw}}$, then $op = \mathsf{nop}$, $g \in \{c_i = \ \}$, $q[2] = q[3]$, and $u' = u =$. This implies that v can be any sequence in $^*_1$ since $u \preceq^m_{\ _2} v$. Moreover, the perfect channel machine $\mathcal{M}_i$ can apply a sequence of update operations in order to empty its buffer. This means that $\mathcal{M}_i$ has a run $\langle q, v\rangle \overset{'}{\Longrightarrow}_{\mathcal{M}_i} \langle q, \ \rangle$ where $' \in {}^*_{\mathsf{upd}}$, since $\langle q, v\rangle$ is consistent and $q[2] = q[3]$. From the configuration $\langle q, \ \rangle$, the perfect channel machine $\mathcal{M}_i$ can apply the transition to reach the configuration $\langle q', \ \rangle$. Thus, $\langle q, v\rangle \Longrightarrow_{\mathcal{M}_i} \langle q', \ \rangle$ is a finite run of $\mathcal{M}_i$ where $= \ '\ell$. Notice that we have indeed $f_{\mathsf{w}}(\ell) = f_{\mathsf{w}}(\ )$.
- If $\ell \in {}_{\mathsf{upd}}$, then $op = c_i?a'$ for some $a' \in$, $g \in \{c_i : \ \}$, and $u = u' \cdot a'$. Since $u \preceq^m_{\ _2} v$, there are $w \in {}^*_1$ and $v' \in {}^*$ such that $v = v' \cdot a' \cdot w$ and $u \preceq^m_{\ _2} v' \cdot a'$. Moreover, the perfect channel machine $\mathcal{M}_i$ can apply starting from $\langle q, v\rangle$ a sequence of update operations corresponding to the sequence w. This means that $\mathcal{M}_i$ has the following run $\langle q, v\rangle \overset{'}{\Longrightarrow}_{\mathcal{M}_i} \langle q'', v' \cdot a'\rangle$ where $' \in {}^*_{\mathsf{upd}}$ and $q''[2] = q[2 \hookleftarrow \mathbf{d}]$ if $w(0) = (k, j, \mathbf{d})$. Now, from the configuration $\langle q'', v' \cdot a'\rangle$, the machinne $\mathcal{M}_i$ can apply the transition since $v' \cdot a' \models g$. Therefore, $\langle q, v\rangle \overset{*}{\Longrightarrow}_{\mathcal{M}_i} \langle q', v'\rangle$ is a finite run of $\mathcal{M}_i$, where $= \ '\ell$. Notice that we have again $f_{\mathsf{w}}(\ell) = f_{\mathsf{w}}(\ )$.

□

Let $q_i, q_i' \in Q_i$, and let be a $\Rightarrow_{(\mathcal{M}_i,\ _2,m)}$-run of the lossy channel machine $\mathcal{M}_i$ between q_i and q_i'. Then, using Lemma 3 it is possible to construct a $\Rightarrow_{\mathcal{M}_i}$-run $'$ of the perfect channel machine $\mathcal{M}_i$ from q_i to q_i' such that and $'$ have the same sequence of write transitions, i.e., $f_{\mathsf{w}}(\ |\ ) = f_{\mathsf{w}}(\ '|\ )$. This terminates the proof of Proposition 7.

7. Simulating lossy channels by TSO/PSO

We show hereafter that basic lossy channel machines can be simulated by $(\mathsf{w} \to \mathsf{r})$-relaxed memory systems.

THEOREM 3. *The control state reachability problem as well as the repeated control state reachability problem for basic lossy channel machines are reducible to their corresponding problems for $(\mathsf{w} \to \mathsf{r})$ relaxed memory systems.*

Proof. Let $\mathcal{M} = (Q, C, \ , \ , \ )$ be a basic LCM. We assume w.l.o.g. that $\mathcal{M}$ has a single channel c (since every basic LCM can be simulated by a single-channel basic LCM [Abdulla and Jonsson 1993]), and that $= \emptyset$. We simulate $\mathcal{M}$ using a $(\mathsf{w} \to \mathsf{r})$-relaxed memory system with two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ and two variables x_1

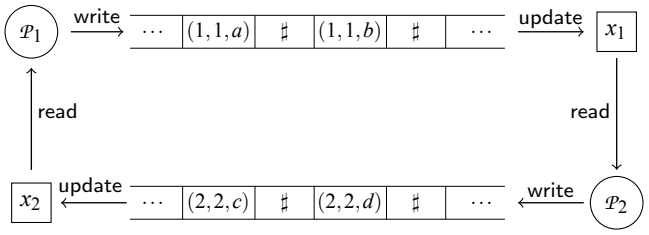

Figure 2. The communication graph of a $(\mathsf{w} \to \mathsf{r})$-relaxed memory system simulating a basic lossy channel machine

and x_2. As shown in Figure 2, the process $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) writes to the variable x_1 (resp. x_2) and reads from the variable x_2 (resp. x_1). A send operation $c!a$ of $\mathcal{M}$ is simulated by a write operation of the value a to the variable x_1 by the process $\mathcal{P}_1$. A receive operation $c?a$ of $\mathcal{M}$ is simulated by a read operation of the value a from x_2 by $\mathcal{P}_1$. (A nop operation of $\mathcal{M}$ is simulated by a nop operation of $\mathcal{P}_1$.) The role of $\mathcal{P}_2$ is to transfer the successive values of x_1 to x_2 (so that they can be read by $\mathcal{P}_1$ in the FIFO order).

To avoid multiple reads of the same value of x_2 by $\mathcal{P}_1$ (which would correspond to multiple receptions in $\mathcal{M}$ of a same message), we introduce a marker $\sharp$ such that every read (resp. write) of a value $a \in$ (by any of the processes) is followed by a read (resp. write) of $\sharp$. Then, the sequence of values written to x_1 and x_2 alternate values from with $\sharp$. This ensures that a write operation of $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) can validate at most one read operation of $\mathcal{P}_2$ (resp. $\mathcal{P}_1$).

Observe that, however, the read operations performed by $\mathcal{P}_2$ can miss some of the values written by $\mathcal{P}_1$, and conversely. This is due to the asynchrony of the two processes (i.e., $\mathcal{N}$ can execute pending writes more often than the reads). Therefore, the sequence of values transfered to x_2 by $\mathcal{P}_2$ (and that can be read by $\mathcal{P}_1$) can be any subsequence of the sequence of values written by $\mathcal{P}_1$ to x_1. Moreover, the sequence of values read by $\mathcal{P}_1$ can also be any subsequence of the sequence of values written by $\mathcal{P}_2$ to x_2. Therefore, $\mathcal{P}_1$ can read from x_2 any subsequence of values written by itself to x_1. This encodes the lossyness of the channel of $\mathcal{M}$.

Formally, let $D = \cup \{\sharp\}$ be the finite data domain and $X = \{x_1, x_2\}$ be the set of two variables valued in D. The $(\mathsf{w} \to \mathsf{r})$-relaxed memory system $\mathcal{N} = (\mathcal{P}_1, \mathcal{P}_2)$ is defined from $\mathcal{M}$ as follows:

- $\mathcal{P}_1 = (P_1, \ _1)$ is a finite-state process where: (1) $P_1 = Q \cup (Q \times \{!, ?\})$ is a finite set of control states, and (2) is the smallest set of transition rules such that:
  - Nop: If $q \xrightarrow{,c:\ |\mathsf{nop}} q'$, then $q \xrightarrow{\mathsf{nop}}_1 q'$
  - Send: If $q \xrightarrow{,c:\ |c!a} q'$, then
    $$q \xrightarrow{\mathsf{w}(1,x_1,a)}_1 (q',!) \quad \text{and} \quad (q',!) \xrightarrow{\mathsf{w}(1,x_1,\sharp)}_1 q'$$
  - Receive: If $q \xrightarrow{,c:\ |c?a} q'$, then
    $$q \xrightarrow{\mathsf{r}(1,x_2,a)}_1 (q',?) \quad \text{and} \quad (q',?) \xrightarrow{\mathsf{r}(1,x_2,\sharp)}_1 q'$$
- $\mathcal{P}_2 = (P_2, \ _2)$ is a finite-state process where: (1) $P_2 = \{p_1, p_2\} \cup (D \times \{!\})$ is a finite set of control states, and (2) $_2$ is the smallest set of transition rules such that for every symbol $a \in$, we have:
  $$p_1 \xrightarrow{\mathsf{r}(2,x_1,a)}_2 (a,!) \qquad (a,!) \xrightarrow{\mathsf{w}(2,x_2,a)}_2 p_2$$
  $$p_2 \xrightarrow{\mathsf{r}(2,x_1,\sharp)}_2 (\sharp,!) \qquad (\sharp,!) \xrightarrow{\mathsf{w}(2,x_2,\sharp)}_2 p_1$$

Theorem 3 is an immediate consequence of the following lemma:

LEMMA 4. *Let $q, q' \in Q$. The control state q' is (infinity often) reachable by $\mathcal{M}$ from q iff $(\mathbf{p}', \mathbf{d})$ is (infinity often) reachable by $\mathcal{N}$ from $(\mathbf{p}, \mathbf{d})$ where $\mathbf{p} = (q, p_1)$, $\mathbf{p}' = (q', p_1)$, $\mathbf{d} = (\sharp, \sharp)$.*

The proof of the lemma above is straightforward. It consists in establishing a bisimulation relation between $\mathcal{N}$ and $\mathcal{M}$. □

From Theorem 3, [Schnoebelen 2002] and [Abdulla and Jonsson 1994], we deduce that:

THEOREM 4. *The state reachability problem for $(\mathsf{w} \to \mathsf{r})$-relaxed memory systems is non-primitive recursive, and their repeated state reachability problem is undecidable.*

The same results established above in this section still hold when we consider in addition the $\mathsf{w} \to \mathsf{w}$ relaxation. In fact, in the system $\mathcal{N}$ built for the proof of Theorem 3 each process writes to only one single variable, which implies that the behavior of $\mathcal{N}$ remains the unchanged if we also consider the $\mathsf{w} \to \mathsf{w}$ relaxation. Therefore, our results concerning the TSO model hold for its $\mathsf{w} \to \mathsf{w}$ relaxation (PSO) as well.

THEOREM 5. *The state reachability problem for $(\mathsf{w} \to \mathsf{r}/\mathsf{w})$-relaxed memory systems is non-primitive recursive, and their repeated state reachability problem is undecidable.*

8. Adding the $\mathsf{r} \to \mathsf{r}/\mathsf{w}$ relaxation

In addition to the $\mathsf{w} \to \mathsf{r}$ relaxation, we consider now that a read or a write operations on some variable x_j can overtake a read operation (by the same process) if the latter concerns a variable different from x_j. As before, we consider that atomic read-write operations cannot permute with any operation, and we also consider the cancellation rule concerning a read that immediately follows a write of the same value on the same variable (by the same processes).

8.1 An operational model

We define hereafter an operational model to capture this semantics. Our model has again one buffer per process, but this time the buffer is used to store write operations as well as read operations. Write operations are stored as before in the buffer to allow overtakes by read operations (Write to store). When a write operation to a variable x_j is present in the buffer of a process $\mathcal{P}_i$, assume that $\mathsf{w}(i, x_j, d)$ is the last of such an operation, then if a read operation $\mathsf{r}(i, x_j, d)$ is the next operation performed by $\mathcal{P}_i$ concerning x_j, this operation is validated immediately (Read own write). If the previous situation does not hold and a read operation $\mathsf{r}(i, x_j, d)$ is performed, then the read operation is stored in the buffer. This corresponds to guessing that x_j will have the value d sometime in the future (Guess). The guess is validated when $\mathsf{r}(i, x_j, d)$ becomes the first operation on x_j in the buffer, and the value assigned to x_j in the global memory at that time is precisely d (Validate). Finally, memory updates are done by executing an operation $\mathsf{w}(i, x_j, d)$ which must be the first (read or write) operation in the buffer of $\mathcal{P}_i$ concerning x_j, i.e., it can only be preceded by read operations on different variables (Update). The formal definition of the model is as follows.

Let $\mathbf{P} = P_1 \times \cdots \times P_n$ and for every $i \in [n]$, let $B_i = \{\mathsf{w}, \mathsf{r}\} \times \{i\} \times [m] \times D$. A *configuration* of $\mathcal{N}$ is a tuple $\langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle$ where $\mathbf{p} \in \mathbf{P}$, $\mathbf{d} \in M$, and $\mathbf{u} \in B_1^* \times \cdots \times B_n^*$.

We define the transition relation $\Rightarrow_{\mathcal{N}}$ on configurations of $\mathcal{N}$ to be the smallest relation such that, for every $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, for every $\mathbf{u}, \mathbf{u}' \in B_1^* \times \cdots \times B_n^*$, and for every $\mathbf{d}, \mathbf{d}' \in M$, we have $\langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle \Rightarrow_{\mathcal{N}} \langle \mathbf{p}', \mathbf{d}', \mathbf{u}' \rangle$ if there is an $i \in [n]$, and there are $p, p' \in P_i$, such that $\mathbf{p}[i] = p$, $\mathbf{p}' = \mathbf{p}[i \hookleftarrow p']$, and one of the following cases hold:

1. Nop: $p \xrightarrow{\mathsf{nop}}_i p'$, $\mathbf{d} = \mathbf{d}'$, and $\mathbf{u} = \mathbf{u}'$.
2. Write to store: $p \xrightarrow{\mathsf{w}(i,x_j,d)}_i p'$, $\mathbf{d} = \mathbf{d}'$, and $\mathbf{u}' = \mathbf{u}[i \hookleftarrow (\mathsf{w},i,j,d)\mathbf{u}[i]]$.
3. Update: $p = p'$, and $\exists j \in [m].\ \exists d \in D.\ \exists u_1,u_2 \in B_i^*$ such that:
   (a) $\mathbf{u}[i] = u_1(\mathsf{w},i,j,d)u_2$, and $\forall(\mathsf{op},i,k,d') \in u_2.$ ($\mathsf{op} = \mathsf{r}$ and $k \neq j$),
   (b) $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d]$,
   (c) $\mathbf{u}'[i] = u_1u_2$, and $\forall k \neq i.\ \mathbf{u}'[k] = \mathbf{u}[k]$
4. Read: $p \xrightarrow{\mathsf{r}(i,x_j,d)}_i p'$, $\mathbf{d} = \mathbf{d}'$, and
   - Read own write: $\mathbf{u} = \mathbf{u}'$ if $\exists u_1,u_2 \in B_i^*$ such that: (a) $\mathbf{u}[i] = u_1(w,i,j,d)u_2$, and (b) $\forall(\mathsf{op},i,k,d') \in u_1.\ k \neq j$, or
   - Guess: $\mathbf{u}' = \mathbf{u}[i \hookleftarrow (\mathsf{r},i,j,d)\mathbf{u}[i]]$ otherwise.
5. Validate: $p = p'$, $\mathbf{d} = \mathbf{d}'$, and $\exists j \in [m].\ \exists d \in D.\ \exists u_1,u_2 \in B_i^*$ s.t.
   (a) $\mathbf{u}[i] = u_1(r,i,j,d)u_2$, and $\forall(\mathsf{op},i,k,d') \in u_2.\ k \neq j$,
   (b) $\mathbf{d}[j] = d$,
   (c) $\mathbf{u}'[i] = u_1u_2$, and $\forall k \neq i.\ \mathbf{u}'[k] = \mathbf{u}[k]$.
6. ARW: $p \xrightarrow{\mathsf{arw}(i,x_j,d,d')}_i p'$, $\mathbf{u}[i] = \ $, $\mathbf{u} = \mathbf{u}'$, and $\mathbf{d}[j] = d$, and $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d']$.

We call $\{\mathsf{w} \to \mathsf{r}, \mathsf{r} \to \mathsf{r}/\mathsf{w}\}$-relaxed memory system a concurrent system $\mathcal{N}$ with the operational semantics defined by $\Rightarrow_{\mathcal{N}}$. Let $\Rightarrow_{\mathcal{N}}^*$ denote as usual the reflexive-transitive closure of $\Rightarrow_{\mathcal{N}}$. The state reachability, and the repeated state reachability problems are defined as in the case of TSO systems in Section 3.4.

8.2 Adding the $\mathsf{w} \to \mathsf{w}$ relaxation

Again, the $\mathsf{w} \to \mathsf{w}$ relaxation can be taken into account in addition to $\{\mathsf{w} \to \mathsf{r}, \mathsf{r} \to \mathsf{r}/\mathsf{w}\}$ simply by associating to each process m different buffers instead of a single one, one per variable, similarly to the relaxation from TSO to PSO.

9. (Un)decidability results

We prove in this section that the addition of the $\mathsf{r} \to \mathsf{r}/\mathsf{w}$ relaxation to either TSO or PSO models leads to the undecidability of the state reachability problem. On the other hand, if we consider models where the number of (guessed) reads stored in the buffers is always bounded, this problem becomes decidable.

9.1 The general case

We prove hereafter the following fact:

THEOREM 6. *The state reachability problem is undecidable for $\{\mathsf{w} \to \mathsf{r}, \mathsf{r} \to \mathsf{r}/\mathsf{w}\}$-relaxed memory systems.*

The proof of Theorem 6 is by a reduction of the Post's Correspondence Problem (PCP), well-known to be undecidable [Post 1946]. We recall that PCP consists in, given two finite sequences $\{u_1,\dots,u_n\}$ and $\{v_1,\dots,v_n\}$ of nonempty words over some alphabet , checking whether there is a sequence of indices $i_1,\dots,i_k \in [n]$ such that $u_{i_1}\cdots u_{i_k} = v_{i_1}\cdots v_{i_k}$.

Then, let $\{u_1,\dots,u_n\}$ and $\{v_1,\dots,v_n\}$ be an instance of PCP. We construct a system $\mathcal{N}$ with two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ sharing a set of four variables $X = \{x_1,x_2,x_3,x_4\}$ such that, two specific states in $\mathcal{M}$ are related by a run if and only if PCP has a solution for the considered instance.

The idea of the reduction is as follows: Process $\mathcal{P}_1$ guesses the solution of PCP as a sequence of indices $i_1,\dots,i_k$ and performs iteratively a sequence of operations: It (1) writes successively to x_1 the symbols of u_{i_j}, (2) reads from x_3 the symbols of u_{i_j}, (3) writes to x_2 the index i_j, and (4) reads i_j from x_4, for j ranging backward from k to 1. Moreover, each write (resp. read) operation to (resp. from) a variable is followed by a write (resp. read) operation of the marker $\sharp$. The insertion of the markers allows to ensure that a written value to a variable by one of the processes can be read at most once by the other process. In parallel, process $\mathcal{P}_2$ also guesses the solution of PCP and performs the same operations as $\mathcal{P}_1$, except that it writes (resp. reads) symbols of the words v_{i_j} and the indices i_j to x_3 and x_4 (from x_1 and x_2), respectively.

Then, we prove that PCP has a solution if and only if it is possible to reach a state of the system $\mathcal{N}$ where both store buffers are empty. In other words, a full computation of $\mathcal{N}$ checks that the two processes have guessed the same sequence of indices and that this sequence is indeed a solution for the considered PCP instance.

The "only if direction" can be shown using the fact that the ordering in the buffers between reads and writes (as well as between reads and other reads) can be relaxed, it is possible to construct a run of the $\mathcal{N}$ where the execution of each write done by one of the processes is immediately followed by its corresponding read operation done by the other process.

The argument for the reverse direction is the following: If there is a run which empties both buffers, then it can be seen that, due to the fact that a read can validate at most one write, the sequence of read symbols by process $\mathcal{P}_2$ is a subword of the sequence of written symbols by $\mathcal{P}_1$, and vice versa. The same holds also for the sequences of indices guessed by both processes. (Here again permuting operations in the buffers is necessary in order to match reads by one of the processes to writes by the other one.) These facts imply that the processes have indeed guessed the same (right) solution to the given instance of PCP.

Let us define more formally the reduction. Let $D = \ \cup \{\sharp,-\} \cup [n]$ be the set of data manipulated by processes $\mathcal{P}_1$ and $\mathcal{P}_2$.

To simplify the presentation, we need to introduce some notations. Let $i \in [2]$, $j \in [4]$, $s \in D^*$, $\mathsf{op} \in \{\mathsf{w},\mathsf{r}\}$, $m = \mathit{length}(s)$ and such that $m \geq 2$. We use the macro transition $p \xrightarrow{\mathsf{op}(i,x_j,s)}_i p'$ to denote the sequence of consecutive transitions $p \xrightarrow{\mathsf{op}(i,x_j,s(1))}_i p_1$, $p_l \xrightarrow{\mathsf{op}(i,x_j,s(l+1))}_i p_{l+1}$ for all $l \in [m-2]$, and $p_{m-1} \xrightarrow{\mathsf{op}(i,x_j,s(m))}_i p'$ where $p_1,\dots,p_m$ are extra intermediary control states of $\mathcal{P}_i$ that are not used anywhere else (and that we may omit from the set of control states of $\mathcal{P}_i$). We use also (op,i,j,s) to denote the fact that the store buffer of $\mathcal{P}_i$ contains the following sequence of consecutive operations $(\mathsf{op},i,j,s(1))\cdots(\mathsf{op},i,j,s(m))$.

Let be a mapping from * to D^* such that for every word $u = a_1\cdots a_m \in \ ^*$, $(u) = \sharp\cdot a_1\cdots\sharp\cdot a_m$.

Then, a computation of the process $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is a sequence of phases where each phase consists in the following operations:

1. Choose a number $l \in [n]$:
   $p \xrightarrow{\mathsf{nop}}_1 p_l$ (resp. $q \xrightarrow{\mathsf{nop}}_2 q_l$)
2. Write the sequence of data (u_l) (resp. (v_l)) to x_1 (resp. x_3):
   $p_l \xrightarrow{\mathsf{w}(1,x_1,\ (u_l))}_1 p_l^1$ (resp. $q_l \xrightarrow{\mathsf{w}(2,x_3,\ (v_l))}_2 q_l^1$)
3. Read the sequence of data (u_l) (resp. (v_l)) from x_3 (resp. x_1):
   $p_l^1 \xrightarrow{\mathsf{r}(1,x_3,\ (u_l))}_1 p_l^2$ (resp. $q_l^1 \xrightarrow{\mathsf{r}(2,x_1,\ (v_l))}_2 q_l^2$)
4. Write the sequence of data $\sharp\cdot l$ to x_2 (resp. x_4):
   $p_l^2 \xrightarrow{\mathsf{w}(1,x_2,\sharp\cdot l)}_1 p_l^3$ (resp. $q_l^2 \xrightarrow{\mathsf{w}(2,x_4,\sharp\cdot l)}_2 q_l^3$)
5. Read the sequence of data $\sharp\cdot l$ from x_4 (resp. x_2):
   $p_l^3 \xrightarrow{\mathsf{r}(1,x_4,\sharp\cdot l)}_1 p$ (resp. $q_l^3 \xrightarrow{\mathsf{r}(2,x_2,\sharp\cdot l)}_2 q$)

Next, we establish the link between the state reachability problem for the $\{w \to r, r \to r/w\}$-relaxed memory system $\mathcal{N}$ and the existence of a solution for the PCP.

LEMMA 5. *There is $i_1, \ldots, i_k \in [n]$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$ if and only if the configuration $\langle (p,q), (\sharp,\sharp,\sharp,\sharp),\ ^2\rangle$ is reachable in $\mathcal{N}$ from the initial configuration $\langle (p,q), (-,-,-,-),\ ^2\rangle$.*

Proof. (The if direction:) Assume that $\langle (p,q), (\sharp,\sharp,\sharp,\sharp),\ ^2\rangle$ is reachable in $\mathcal{N}$ from $\langle (p,q), (-,-,-,-),\ ^2\rangle$. This means that all the read operations of $\mathcal{P}_1$ and $\mathcal{P}_2$ have been validated.

Then, assume that $i_k, \ldots, i_1$ is the sequence of indices chosen by $\mathcal{P}_1$ and that $j_h, \ldots, j_1$ is the sequence of indices chosen by $\mathcal{P}_2$. We use the facts that (1) write and read operations by a same process to a same variable cannot be reordered, and that (2) each write operation of $\mathcal{P}_1$ can only validate a unique read operation of $\mathcal{P}_2$ and vice-versa (but of course some written values can be missed since processes are asynchronous), to show that the following relations hold:

- $u_{i_1} u_{i_2} \cdots u_{i_k} \preceq v_{j_1} v_{j_2} \cdots v_{j_h}$.
- $v_{j_1} v_{j_2} \cdots v_{j_h} \preceq u_{i_1} u_{i_2} \cdots u_{i_k}$.
- $i_1 i_2 \cdots i_k \preceq j_1 j_2 \cdots j_h$.
- $j_1 j_2 \cdots j_h \preceq i_1 i_2 \cdots i_k$.

This implies that $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{j_1} v_{j_2} \cdots v_{j_h}$ and $i_1 i_2 \cdots i_k = j_1 j_2 \cdots j_h$.

(The only-if direction:) Assume that there is a sequence of indices $i_1, \ldots, i_k \in [n]$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$. Then, we can construct the following run of $\mathcal{N}$ from the initial configuration $\langle (p,q), (-,-,-,-),\ ^2\rangle$ to the configuration $\langle (p,q), (\sharp,\sharp,\sharp,\sharp),\ ^2\rangle$:

- First, $\mathcal{P}_1$ chooses the sequence of indices $i_k, \ldots, i_1$ and stores in its buffer $seq_1 \cdots seq_k$ where, for every $l \in [k]$, seq_l is the sequence of operations stored by $\mathcal{P}_1$ during its l^{th} phase (i.e., $(r,1,4,\sharp \cdot i_l)(w,1,2,\sharp \cdot i_l)(r,1,3,\ (u_{i_l}))(w,1,1,\ (u_{i_l}))$).
- Then, $\mathcal{P}_2$ chooses the sequence of indices $i_k, \ldots, i_1$ and stores in its buffer $seq'_1 \cdots seq'_k$ where for every $l \in [k]$, seq'_l is the sequence of operations stored by $\mathcal{P}_2$ during its l^{th} phase (i.e., $(r,2,2,\sharp \cdot i_l)(w,2,4,\sharp \cdot i_l)(r,2,1,\ (v_{i_l}))(w,2,3,\ (v_{i_l}))$).
- Finally, $\mathcal{N}$ adopts the following run in order to execute the writes and validate the reads in the buffers. This run is divided into two steps:
  - In the first step, $\mathcal{N}$ performs alternately the following actions: (1) execute the first write operation in the buffer of $\mathcal{P}_1$ concerning some variable x say, and then (2) validate the corresponding read operation in the buffer of $\mathcal{P}_2$. This read operation is the first read operation on x in the buffer of $\mathcal{P}_2$. However, this read operation may occur behind some other operations in the buffer, but by construction, they are all on other variables. Therefore the read operation can be validated due to the relaxed memory semantics we consider. The relaxation rules are also necessary to be able to take successively write operations in the buffer of $\mathcal{P}_1$ since this requires overtaking the reads that occur between the writes.
  - In the second step, the role of $\mathcal{P}_1$ and $\mathcal{P}_2$ are interchanged.

This terminates the proof of Lemma 5. □

Finally, Theorem 6 is an immediate consequence of Lemma 5. Let us again mention that the same result holds when we additionally consider the $w \to w$ relaxation. In fact, the system $\mathcal{N}$ we construct in the proof of Theorem 6 can be (re)used with the more relaxed semantics. The effect of the relaxation is simply to split the buffer of each of the processes into four different buffers, but this does not affect the reasoning concerning the relations between the sequences of reads and their corresponding sequences of writes.

THEOREM 7. *The state reachability problem is undecidable for $\{w \to r/w, r \to r/w\}$-relaxed memory systems.*

9.2 Bounded-guess $r \to r/w$ relaxation

The undecidability result in the previous section uses the fact that it is possible to perform an unbounded number of guesses on read values before validating them. Therefore, a natural idea is to bound the number of guesses made by a process at each time. This corresponds to impose a bound on the number of reads stored in each buffer (without bounding the number of writes in the buffers). Let us call *bounded-guess* relaxed memory systems the so restricted systems. (It is of course straightforward to adapt the models we have defined previously to impose this restriction for a given bound on the number of reads in each buffer.)

Under the bounded-guess restriction, the state reachability problem becomes decidable. The reason is that it is possible to construct lossy channel machines for these systems by adapting the construction of Section 5. Indeed, it suffices to consider that the stored read operations in the channels are "strong symbols" (in the sense that they cannot be lost). This is clearly possible since the number of these reads is bounded by hypothesis.

THEOREM 8. *The state reachability problem for bounded-guess $\{w \to r, r \to r/w\}$-relaxed memory systems is decidable. The same holds for bounded-guess $\{w \to r/w, r \to r/w\}$-relaxed memory systems.*

10. Conclusion

We have investigated the boundary between decidability and undecidability for the verification problem of programs under various weak memory models. We have considered models classified according to the type of the order relaxations they involve (following the style of [Adve and Gharachorloo 1996]). We have shown that the reachability problem is decidable for $w \to r$-relaxed memory systems (TSO) as well as for their $w \to w$ relaxation (PSO). This result is obtained through a non-trivial translation to lossy channel machines. We have shown that, however, when the $r \to r/w$ relaxation is added to these models, the reachability problem becomes undecidable. On the other hand, if we consider that the number of read operations that are delayed (overtaken by other operations) is always bounded (i.e., at any point in time but not necessarily if we consider the whole computation), then this problem is decidable, since in this case it can again be reduced to the reachability problem for lossy channel machines.

Moreover, we have established the complexity of the reachability problem for these memory models (when it is decidable). We have proved that lossy channel machines can be simulated by TSO (as well as by its relaxations we consider such as PSO), which implies that the reachability problem for these models is non-primitive recursive. This shows that there is (in theory) an important jump in the complexity of the reachability problem when moving from the SC model (PSPACE-complete) to weak memory models. However, the high theoretical complexity is not necessarily an obstacle for exploiting our decidability results in practice. Indeed, it could be possible to use for instance efficient verification techniques, combining effective symbolic representations and iterative under/upper approximate analysis, that are complete for well-structured systems such as lossy channel machines (see, e.g., [Geeraerts et al. 2006, Ganty et al. 2006]). The design of efficient tools based on these techniques that are tailored for the automatic verification of

programs/algorithms under weak memory models is a challenging topic for future work.

The ability of relaxed memory systems to simulate lossy channel machines implies also that the repeated state reachability problem (and therefore verifying liveness properties) for these systems is undecidable. Investigating conditions under which liveness properties can be automatically and efficiently verified for weak memory systems is again an interesting topic for future work.

Acknowledgments

The two first authors were partially supported by the project ANR-06-SETI-001 AVERISS.

References

P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170. IEEE Computer Society, 1993.

P. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. In *ICALP*, LNCS 820, pages 316–327. Springer, 1994.

P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.

P. Abdulla, K. Cerans, B. Jonsson, and Y-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Logic in Computer Science (LICS)*, pages 219–228, 1996.

S. Baswana, S. Mehta, and V. Powar. Implied set closure and its application to memory consistency verification. In *Computer-Aided Verification (CAV)*, 2008.

S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer-Aided Verification (CAV)*, pages 107–120, 2008. Extended Version as Tech Report MSR-TR-2008-12, Microsoft Research.

S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.

S. Hangal et al. TSOtool: A program for verifying memory systems using the memory consistency model. In *International Symposium on Computer Architecture (ISCA)*, 2004.

A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

P. Ganty, J. F. Raskin, and L. Van Begin. A complete abstract interpretation framework for coverability properties of WSTS. In *VMCAI'06*, LNCS 3855, pages 49–64. Springer, 2006.

G. Geeraerts, J. F. Raskin, and L. Van Begin. Expand, enlarge and check: New algorithms for the coverability problem of wsts. *J. Comput. Syst. Sci.*, 72(1):180–203, 2006.

P. B. Gibbons and E. Korach. The complexity of sequential consistency. In *Parallel and Distributed Processing*, pages 317–325. IEEE, 1992.

T. Q. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods (FM)*, LNCS 4085, pages 476–491. Springer, 2006.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *OOPSLA*, page to appear, 2009.

Maged Michael, Martin Vechev, and Vijay Saraswat. Idempotent work stealing. In *PPoPP*, pages 45–54, 2009.

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge, 2009.

S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 34–41, 1995. ISBN 0-89791-717-0.

E. L. Post. A variant of a recursively unsolvable problem. *Bull. of the American Mathematical Society*, 52:264–268, 1946.

A. Roy, C. Fleckenstein, S. Zeisset, and J. Huang. Fast and generalized polynomial time memory consistency verification. In *Computer-Aided Verification (CAV)*, 2005.

Ph. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5):251–261, September 2002.

P. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Center, 1991.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004. doi: 10.1109/IPDPS.2004.1302944.