The Benefits of Duality in Verifying Concurrent Programs under TSO

Parosh Aziz Abdulla1, Mohamed Faouzi Atig2, Ahmed Bouajjani3, and Tuan Phong Ngo4

1 Uppsala University, Sweden parosh@it.uu.se
2 Uppsala University, Sweden mohamed_faouzi.atig@it.uu.se
3 IRIF, Université Paris Diderot & IUF, France abou@liafa.univ-paris-diderot.fr
4 Uppsala University, Sweden tuan-phong.ngo@it.uu.se

Abstract

We address the problem of verifying safety properties of concurrent programs running over the TSO memory model. Known decision procedures for this model are based on complex encodings of store buffers as lossy channels. These procedures assume that the number of processes is fixed. However, it is important in general to prove correctness of a system/algorithm in a parametric way with an arbitrarily large number of processes. In this paper, we introduce an alternative (yet equivalent) semantics to the classical one for the TSO model that is more amenable for efficient algorithmic verification and for extension to parametric verification. For that, we adopt a dual view where load buffers are used instead of store buffers. The flow of information is now from the memory to load buffers. We show that this new semantics allows (1) to simplify drastically the safety analysis under TSO, (2) to obtain a spectacular gain in efficiency and scalability compared to existing procedures, and (3) to extend easily the decision procedure to the parametric case, which allows to obtain a new decidability result, and more importantly, a verification algorithm that is more general and more efficient in practice than the one for bounded instances.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Weak Memory Models, Reachability Problem, Parameterized Verification

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.5

1 Introduction

Most modern processor architectures execute instructions in an out-of-order manner to gain efficiency. In the context of sequential programming, this out-of-order execution is transparent to the programmer since one can still work under the Sequential Consistency (SC) model [24]. However, this is not true when we consider concurrent processes that share the memory. In fact, it turns out that concurrent algorithms such as mutual exclusion and producer-consumer protocols may not behave correctly any more. Therefore, program verification is a relevant (and difficult) task in order to prove correctness under the new semantics. The inadequacy of the interleaving semantics has led to the invention of new program semantics, so called

* This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

Weak (or relaxed) Memory Models (WMM), by allowing permutations between certain types of memory operations [7, 20, 8]. Total Store Ordering (TSO) is one of the the most common models, and it corresponds to the relaxation adopted by Sun’s SPARC multiprocessors [28] and formalizations of the x86-tso memory model [26, 27]. These models put an unbounded perfect (non-lossy) store buffer between each process and the main memory where a store buffer carries the pending store operations of the process. When a process performs a store operation, it appends it to the end of its buffer. These operations are propagated to the shared memory non-deterministically in a FIFO manner. When a process reads a variable, it searches its buffer for a pending store operation on that variable. If no such a store operation exists, it fetches the value of the variable from the main memory. Verifying programs running on the TSO memory model poses a difficult challenge since the unboundedness of the buffers implies that the state space of the system is infinite even in the case where the input program is finite-state. Decidability of safety properties has been obtained by constructing equivalent models that replace the perfect store buffer by lossy channels [11, 12, 2]. However, these constructions are complicated and involve several ingredients that lead to inefficient verification procedures. For instance, they require each message inside a lossy channel to carry (instead of a single store operation) a full snapshot of the memory representing a local view of the memory contents by the process. Furthermore, the reductions involve non-deterministic guessing the lossy channel contents. The guessing is then resolved either by consistency checking [11] or by using explicit pointer variables (each corresponding to one process) inside the buffers [2], causing a serious state space explosion problem.

In this paper, we introduce a novel semantics which we call the dual TSO semantics. Our aim is to provide an alternative (and equivalent) semantics that is more amenable for efficient algorithmic verification. The main idea is to have load buffers that contain pending load operations (more precisely, values that will potentially be taken by forthcoming load operations) rather than store buffers (that contain store operations). The flow of information will now be in the reverse direction, i.e., store operations are performed by the processes atomically on the main memory, while values of variables are propagated non-deterministically from the memory to the load buffers of the processes. When a process performs a load operation it can fetch the value of the variable from the head of its load buffer. We show that the dual semantics is equivalent to the original one in the sense that any given set of processes will reach the same set of local states under both semantics. The dual semantics allows us to understand the TSO model in a totally different way compared to the classical semantics. Furthermore, the dual semantics offers several important advantages from the point of view of formal reasoning and program verification. First, the dual semantics allows transforming the load buffers to lossy channels without adding the costly overhead that was necessary in the case of store buffers. This means that we can apply the theory of well-structured systems [6, 5, 21] in a straightforward manner leading to a much simpler proof of decidability of safety properties. Second, the absence of extra overhead means that we obtain more efficient algorithms and better scalability (as shown by our experimental results). Finally, the dual semantics allows extending the framework to perform parameterized verification which is an important paradigm in concurrent program verification. Here, we consider systems, e.g., mutual exclusion protocols, that consist of an arbitrary number of processes. The aim of parameterized verification is to prove correctness of the system regardless of the number of processes. It is not obvious how to perform parameterized verification under the classical semantics. For instance, extending the framework of [2], would involve an unbounded number of pointer variables, thus leading to channel systems with unbounded message alphabets. In contrast, as we show in this paper, the simple nature of the dual
semantics allows a straightforward extension of our verification algorithm to the case of
parametrized verification. This is the first time a decidability result is established for the
parametrized verification of programs running over WMM. Notice that this result is taking
into account two sources of infinity: the number of processes, and the size of the buffers.

Based on our framework, we have implemented a tool and applied it to a large set of
benchmarks. The experiments demonstrate the efficiency of the dual semantics compared to
the classical one (by two order of magnitude in average), and the feasibility of parametrized
verification in the former case. In fact, besides its theoretical generality, parametrized
verification is practically crucial in this setting: as our experiments show, it is much more
efficient than verification of bounded-size instances (starting from a number of components
of 3 or 4), especially concerning memory consumption (which is the critical resource).

Related Work. There have been a lot of works related to the analysis of programs running
under WMM (e.g., [25, 22, 23, 17, 2, 15, 16, 13, 14, 29]). Some of these works propose precise
analysis techniques for checking safety properties or stability of finite-state programs under
WMM (e.g., [2, 13, 19, 4]). Others propose stateless model-checking techniques for programs
under TSO and PSO (e.g., [1, 2, 18]). Different other techniques based on monitoring and
testing have also been developed during these last years (e.g., [15, 16, 25]). There are also a
number of efforts to design bounded model checking techniques for programs under WMM
(e.g., [9, 29, 14]) which encode the verification problem in SAT/SMT.

The closest works to ours are those presented in [2, 11, 3, 12] which provide precise and
sound techniques for checking safety properties for finite-state programs running under TSO.
However, as stated in the introduction, these techniques are complicated and can not be
extended, in a straightforward manner, to the verification of parameterized systems (as it is
the case of the developed techniques for the dual TSO semantics).

In Section 7, we experimentally compare our techniques with Memorax [2, 3] which is the
only precise and sound tool for checking safety properties for programs under TSO.

2 Preliminaries

Let \( \Sigma \) be a finite alphabet. We use \( \Sigma^* \) (resp. \( \Sigma^+ \)) to denote the set of all words (resp.
non-empty words) over \( \Sigma \). Let \( \varepsilon \) be the empty word. The length of a word \( w \in \Sigma^* \) is denoted
by \( |w| \) (and in particular \( |\varepsilon| = 0 \)). For every \( i : 1 \leq i \leq |w| \), let \( w(i) \) be the symbol at position
\( i \) in \( w \). For \( a \in \Sigma \), we write \( a \in w \) if \( a \) appears in \( w \), i.e., \( a = w(i) \) for some \( i : 1 \leq i \leq |w| \).

Given two words \( u \) and \( v \) over \( \Sigma \), we use \( u \preceq v \) to denote that \( u \) is a (not necessarily
contiguous) subword of \( v \) (i.e., if there is an injection \( h : \{1, \ldots, |u|\} \rightarrow \{1, \ldots, |v|\} \) such that:
(1) \( h(i) < h(j) \) for all \( i < j \), and (2) for every \( i \in \{1, \ldots, |u|\} \), we have \( u(i) = v(h(i)) \)).

Given a subset \( \Sigma' \subseteq \Sigma \) and a word \( w \in \Sigma^* \), we use \( w|_{\Sigma'} \) to denote the projection of \( w \)
over \( \Sigma' \), i.e., the word obtained from \( w \) by erasing all the symbols that are not in \( \Sigma' \).

Let \( A \) and \( B \) be two sets and let \( f : A \rightarrow B \) be a total function from \( A \) to \( B \). We use
\( f[a \leftarrow b] \) to denote the function \( f' \) such that \( f'(a') = b \) and \( f'(a') = f(a') \) for all \( a' \neq a \).

A transition system \( T \) is a tuple \((C, \text{Init}, \text{Act}, \cup_{a \in \text{Act}} \xrightarrow{a} \) where \( C \) is a (potentially infinite)
set of configurations, \( \text{Init} \subseteq C \) is a set of initial configurations, \( \text{Act} \) is a set of actions, and for
every \( a \in \text{Act}, \xrightarrow{a} \subseteq C \times C \) is a transition relation. We use \( \xrightarrow{a} \) to denote that \((c, c') \in \xrightarrow{a} \).

Let \( \rightarrow = \cup_{a \in \text{Act}} \xrightarrow{a} \). A run \( \pi \) of \( T \) is of the form \( c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} c_n \), where \( c_i \xrightarrow{a_{i+1}} c_{i+1} \)
for all \( i : 0 \leq i < n \). Then, we write \( c_0 \xrightarrow{*} c_n \). We use \( \text{target}(\pi) \) to denote the configuration \( c_n \).
The run \( \pi \) is said to be a computation if \( c_0 \in \text{Init} \). Two runs \( \pi_1 = c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_m} c_m \) and
\( \pi_2 = c_{m+1} \xrightarrow{a_{m+2}} c_{m+2} \xrightarrow{a_{m+3}} \cdots \xrightarrow{a_n} c_n \) are compatible if \( c_m = c_{m+1} \). Then, we write
5:4 The Benefits of Duality in Verifying Concurrent Programs under TSO

\[ \pi_1 \cdot \pi_2 \] to denote the run \( \pi = \pi_0 \overset{a_1}{\rightarrow} \pi_1 \overset{a_2}{\rightarrow} \cdots \overset{a_n}{\rightarrow} \pi_n \). For two configurations \( c \) and \( c' \), we use \( c \xrightarrow{\tau} c' \) to denote that \( c \xrightarrow{\tau} c' \) for some run \( \pi \). A configuration \( c \) is said to be reachable in \( T \) if \( c_0 \xrightarrow{\tau} c \) for some \( c_0 \in \text{Init} \), and a set \( C \) of configurations is said to be reachable in \( T \) if some \( c \in C \) is reachable in \( T \).

3 Concurrent Systems

In this section, we define the syntax we use for concurrent programs, a model for representing communicating concurrent processes. Communication between processes is performed through a shared memory that consists of a finite number of shared variables (over finite domains) to which all processes can read and write. Then we recall the classical TSO semantics including the transition system it induces and its reachability problem. We introduce the Dual TSO semantics and its induced transition system. Finally, we state the equivalence between the two semantics (for a given concurrent program, we can reduce its reachability problem under the classical TSO to its reachability problem under Dual TSO and vice-versa).

3.1 Syntax

Let \( V \) be a finite data domain and \( X \) be a finite set of variables. We assume w.l.o.g. that \( V \) contains the value 0. Let \( \Omega(X, V) \) be the smallest set of memory operations that contains (1) “no operation” \( \text{nop} \), (2) read operation \( r(x, v) \), (3) write operation \( w(x, v) \), (4) fence operation \( \text{fence} \), and (5) atomic read-write operation \( \text{arw}(x, v, v') \), where \( x \in X \), and \( v, v' \in V \).

A concurrent system is a tuple \( \mathcal{P} = (A_1, A_2, \ldots, A_n) \) where for every \( p : 1 \leq p \leq n \), \( A_p \) is a finite-state automaton describing the behavior of the process \( p \). The automaton \( A_p \) is defined as a triple \( (Q_p, q_p^{\text{init}}, \Delta_p) \) where \( Q_p \) is a finite set of local states, \( q_p^{\text{init}} \in Q_p \) is the initial local state, and \( \Delta_p \subseteq Q_p \times \Omega(X, V) \times Q_p \) is a finite set of transitions. We define \( \mathcal{P} = \{1, \ldots, n\} \) to be the set of process IDs, \( \mathcal{Q} := \cup_{p \in \mathcal{P}} Q_p \) to be the set of all local states and \( \Delta := \cup_{p \in \mathcal{P}} \Delta_p \) to be the set of all transitions.

3.2 Classical TSO semantics

In the following, we recall the semantics of concurrent systems under the classical TSO model as formalized in [26, 27]. To do that, we define the set of configurations and the induced transition relation. Let \( \mathcal{P} = (A_1, A_2, \ldots, A_n) \) be a concurrent system.

Configurations. A TSO-configuration \( c \) is a triple \( (q, b, \text{mem}) \) where (1) \( q : \mathcal{P} \mapsto Q \) is the global state of \( \mathcal{P} \) mapping each process \( p \in \mathcal{P} \) to a local state in \( Q_p \) (i.e., \( q(p) \in Q_p \)), (2) \( b : \mathcal{P} \mapsto (X \times V)^* \) gives the content of the store buffer of each process, and (3) \( \text{mem} : X \mapsto V \) defines the value of each shared variable. Observe that the store buffer of each process contains a sequence of write operations, where each write operation is defined by a pair, namely a variable \( x \) and a value \( v \) that is assigned to \( x \). The initial TSO-configuration \( c^{\text{init}} \) is defined by the tuple \( (q^{\text{init}}, b^{\text{init}}, \text{mem}^{\text{init}}) \) where, for all \( p \in \mathcal{P} \) and \( x \in X \), we have that \( q^{\text{init}}(p) = q_p^{\text{init}} \), \( b^{\text{init}}(p) = \epsilon \) and \( \text{mem}^{\text{init}}(x) = 0 \). In other words, each process is in its initial local state, all the buffers are empty, and all the variables in the shared memory are initialized to 0. We use \( \mathcal{C}_{\text{TSO}} \) to denote the set of TSO-configurations.

Transition Relation. The transition relation \( \rightarrow_{\text{TSO}} \) between TSO-configurations is given by a set of rules, described in Figure 1. Here we informally explain these rules. A \( \text{nop} \) transition \( (q, \text{nop}, q') \in \Delta_p \) changes only the local state of the process \( p \) from \( q \) to \( q' \). A
The TSO Reachability Problem. A global state $q_{\text{target}}$ is said to be reachable in $T_{TSO}$ iff there is a TSO-configuration $c$ of the form $(q_{\text{target}}, b, \text{mem})$, with $b(p) = \epsilon$ for all $p \in P$, such that $c \xrightarrow{t} T_{TSO} c'$ for some $t \in \Delta \cup \Delta'$ where $\Delta' := \{\text{update}_p | p \in P\}$. The transition system induced by $P$ under the classical TSO semantics is then given by $T_{TSO} = (C_{TSO}, \{c_{\text{init}}\}, \Delta \cup \Delta', \rightarrow_{TSO})$.

The transition relation $\rightarrow_{TSO}$ under TSO. Here $p \in P$ and $t \in \Delta_p \cup \{\text{update}_p\}$ where $\text{update}_p$ is an operation that updates the memory using the oldest message in the buffer of process $p$.

- **Nop**
  $$t = (q, \text{nop}, q') \quad q(p) = q$$
  $$(q, b, \text{mem}) \rightarrow_{TSO} (q[p \leftarrow q'], b, \text{mem})$$

- **Write**
  $$t = (q, w(x, v), q') \quad q(p) = q$$
  $$(q, b, \text{mem}) \rightarrow_{TSO} (q[p \leftarrow q'], b, \text{mem})$$

- **Update**
  $$t = \text{update}_p$$
  $$(q, b[p \leftarrow b(p) \cdot (x, v)], \text{mem}) \rightarrow_{TSO} (q, b, \text{mem} [x \leftarrow v])$$

- **Read-Own-Write**
  $$t = (q, r(x, v), q') \quad q(p) = q$$
  $$(q, b, \text{mem}) \rightarrow_{TSO} (q[p \leftarrow q'], b, \text{mem})$$

- **Read Memory**
  $$t = (q, \text{r}(x, v), q') \quad q(p) = q \quad b(p)(x) = \epsilon \quad \text{mem}(x) = v$$
  $$(q, b, \text{mem}) \rightarrow_{TSO} (q[p \leftarrow q'], b, \text{mem})$$

- **ARW**
  $$t = (q, \text{arw}(x, v, v'), q') \quad q(p) = q \quad b(p) = \epsilon \quad \text{mem}(x) = v$$
  $$(q, b, \text{mem}) \rightarrow_{TSO} (q[p \leftarrow q'], b, \text{mem} [x \leftarrow v'])$$

- **Fence**
  $$t = (q, \text{fence}, q') \quad q(p) = q$$
  $$(q, b, \text{mem}) \rightarrow_{TSO} (q[p \leftarrow q'], b, \text{mem})$$

We use $c \rightarrow_{TSO} c'$ to denote that $c \xrightarrow{t} T_{TSO} c'$ for some $t \in \Delta \cup \Delta'$ where $\Delta' := \{\text{update}_p | p \in P\}$. The transition system induced by $P$ under the classical TSO semantics is then given by $T_{TSO} = (C_{TSO}, \{c_{\text{init}}\}, \Delta \cup \Delta', \rightarrow_{TSO})$.
3.3 Dual TSO semantics

In this section, we define the Dual TSO semantics. The model has a perfect FIFO load buffer between the main memory and each process. The load buffer is used to store potential read operations that will be performed by the process. Each message in the load buffer of a process \( p \) is either a pair of the form \((x, v)\) or a triple of the form \((x, v, \text{own})\) where \( x \in X \) and \( v \in V \). A message of the form \((x, v)\) corresponds to the fact that \( x \) had the value \( v \) in the shared memory. While a message \((x, v, \text{own})\) corresponds to the fact that the process \( p \) has written the value \( v \) to \( x \). A write operation \( w(x, v) \) of the process \( p \) immediately updates the shared memory and a message of the form \((x, v, \text{own})\) is then appended to the tail of the load buffer of \( p \). Read propagation is then performed by non-deterministically choosing a variable (let’s say \( x \) and its value \( v \) in the shared memory) and appending the message \((x, v)\) to the tail of the load buffer of \( p \). This propagation operation speculates on a read operation of \( p \) on \( x \) that will be performed later on. Moreover, any message at the head of the load buffer can be removed at any time. A read operation \( r(x, v) \) of the process \( p \) can be executed if the message at the head of the load buffer (i.e., the oldest one) of \( p \) is of the form \((x, v)\) and there is no pending message of the form \((x, v', \text{own})\). In the case that the load buffer contains a message belonging to \( p \) (i.e., of the form \((x, v', \text{own})\)), the read value must correspond to the value of the most recent message belonging to \( p \) (implicitly, this allows to simulate the Read-Own-Write transitions). A fence means that the load buffer of \( p \) must be empty before \( p \) can continue. Let \( \mathcal{P} = \{A_1, A_2, \ldots, A_n\} \) be a concurrent system.

**Configurations.** A DTSO-configuration \( c \) is a triple \((q, b, \text{mem})\) where \( q : P \mapsto Q \) is the global state of \( \mathcal{P} \), \( b : P \mapsto ((X \times V) \cup (X \times V \times \{\text{own}\}))^* \) is the content of the load buffer, and \( \text{mem} : X \mapsto V \) gives the value of each variable. The initial DTSO-configuration \( c^{\text{init}} \) is defined by \((q^{\text{init}}, b^{\text{init}}, \text{mem}^{\text{init}})\) where, for all \( p \in P \) and \( x \in X \), we have that \( q^{\text{init}}(p) = q_p^{\text{init}}, b^{\text{init}}(p) = \epsilon \) and \( \text{mem}^{\text{init}}(x) = 0 \). We use \( \mathcal{C}_{\text{DTSO}} \) to denote the set of DTSO-configurations.

**Transition Relation.** The transition relation \( \rightarrow_{\text{DTSO}} \) induced by the Dual TSO semantics is given in Figure 2. This relation is induced by members of \( \Delta \) and \( \Delta_{\text{aux}} := \{\text{propagate}^p, \text{delete}^p | p \in P, x \in X\} \). \( \text{propagate}^p \) is an operation that speculates on a read operation of \( p \) over \( x \) that will be executed later. This is done by appending the message \((x, v)\) to the tail of the load buffer of \( p \) where \( v \) is the current value of \( x \) in the shared memory. The operation \( \text{delete}^p \) removes the oldest message in the load buffer of process \( p \). A write operation \( w(x, v) \) updates the memory and appends the message \((x, v, \text{own})\) to the tail of the load buffer. A read operation \( r(x, v) \) checks first if the load buffer contains a message of the form \((x, v', \text{own})\), and in that case the read value \( v \) should correspond to the value of the most recent message of that form. If there is no message on the variable \( x \) belonging to \( p \) in its load buffer then the value \( v \) of \( x \) is fetched from the message at the head of its load buffer.

We use \( c \rightarrow_{\text{DTSO}} c' \) to denote that \( c \xrightarrow{t} c' \) for some \( t \in \Delta \cup \Delta_{\text{aux}} \). The transition system induced by \( \mathcal{P} \) under the Dual TSO semantics is then given by \( \mathcal{T}_{\text{DTSO}} = (\mathcal{C}_{\text{DTSO}}, \{c^{\text{init}}\}, \Delta \cup \Delta_{\text{aux}}, \rightarrow_{\text{DTSO}}) \).

**The Dual TSO Reachability Problem.** The reachability problem for \( \mathcal{P} \) under the Dual TSO semantics is defined in a similar manner to the case of TSO. A global state \( q_{\text{target}} \) is said to be reachable in \( \mathcal{T}_{\text{DTSO}} \) iff there is a DTSO-configuration \( c \) of the form \((q_{\text{target}}, b, \text{mem})\), with \( b(p) = \epsilon \) for all \( p \in P \), such that \( c \) is reachable in \( \mathcal{T}_{\text{DTSO}} \). Then, the reachability problem
• Figure 2 The induced transition relation $\rightarrow_{\text{DTSO}}$ under the Dual TSO semantics. Here $p \in P$ and $t \in \Delta_p \cup \Delta'_p$ where $\Delta'_p := \{\text{propagate}^p, \text{delete}^p | x \in X\}$. 

consists in checking whether $q_{\text{target}}$ is reachable in $T_{\text{DTSO}}$. The following theorem states equivalence of the reachability problems under the TSO and Dual TSO semantics.

\textbf{Theorem 1.} A global state $q_{\text{target}}$ is reachable in $T_{\text{TSO}}$ iff $q_{\text{target}}$ is reachable in $T_{\text{DTSO}}$.

\section{The Dual TSO Reachability Problem}

In this section, we show the decidability of the Dual TSO reachability problem by making use of the framework of \textit{Well-Structured Transition Systems} (WSTS) \cite{Abdulla:2010a,Atig:2011}. First, we briefly recall the framework of WSTS and then we instantiate it to show the decidability of the Dual TSO reachability problem.

\subsection{Well-Structured Transition Systems}

Let $\mathcal{T} = (\mathcal{C}, \text{Init}, \text{Act}, \cupact, \rightarrow)$ be a transition system. Let $\sqsubseteq$ be a \textit{well-quasi ordering} on $\mathcal{C}$. Recall that a well-quasi ordering on $\mathcal{C}$ is a binary relation over $\mathcal{C}$ that is reflexive and transitive and for every infinite sequence $(c_i)_{i \geq 0}$ of elements in $\mathcal{C}$ there exist $i, j \in \mathbb{N}$ such that $i < j$ and $c_i \sqsubseteq c_j$. A set $\mathcal{U} \subseteq \mathcal{C}$ is called \textit{upward closed} if for every $c \in \mathcal{U}$ and $c' \in \mathcal{C}$ with $c \sqsubseteq c'$, we have $c' \in \mathcal{U}$. It is known that every upward closed set $\mathcal{U}$ can be characterised by a finite \textit{minor set} $\mathcal{M} \subseteq \mathcal{U}$ such that: (i) for every $c \in \mathcal{U}$ there is $c' \in \mathcal{M}$ such that $c' \sqsubseteq c$, and (ii) if $c, c' \in \mathcal{M}$ and $c \sqsubseteq c'$ then $c = c'$. We use $\min$ to denote the function which for a given upward closed set $\mathcal{U}$ returns its minor set.
Let $D \subseteq C$. The upward closure of $D$ is defined as $D \uparrow := \{ c' \in C \mid \exists c \in D \text{ with } c \subseteq c' \}$. We also define the set of predecessors of $D$ as $\text{Pre}_T(D) := \{ c \mid \exists c_1 \in D, c \rightarrow c_1 \}$. For a finite set of configurations $M \subseteq C$, we use $\text{minpre}(M)$ to denote $\min(\text{Pre}_T(M \uparrow) \cup M \uparrow)$.

The transition relation $\rightarrow$ is said to be monotonic wrt. $\sqsubseteq$ if, given $c_1, c_2, c_3 \in C$ s.t. $c_1 \rightarrow c_2$ and $c_1 \sqsubseteq c_3$, we can compute a configuration $c_4 \in C$ and a run $\pi$ s.t. $c_3 \overset{\pi}{\rightarrow} c_4$ and $c_2 \sqsubseteq c_4$. The pair $(T, \sqsubseteq)$ is called monotonic transition system if $\rightarrow$ is monotonic wrt. $\sqsubseteq$.

Given a finite set of configurations $M \subseteq C$, the coverability problem of $M$ in the monotonic transition system $(T, \sqsubseteq)$ asks whether the set $M \uparrow$ is reachable in $T$. For the decidability of this problem the following three conditions are sufficient: (i) For every two configurations $c_1$ and $c_2$, it is decidable whether $c_1 \sqsubseteq c_2$, (ii) for every $c \in C$, we can check whether $\{ c \} \uparrow \cap \text{Init} \neq \emptyset$, and (iii) for every $c \in C$, the set $\text{minpre}(c)$ is finite and computable.

The solution for the coverability problem as suggested in [5, 21] is based on a backward analysis approach. It is shown that starting from a finite set $M_0 \subseteq C$, the sequence $(M_i)_{i \geq 0}$ with $M_{i+1} := \text{minpre}(M_i)$, for $i \geq 0$ reaches a fixpoint and is computable.

4.2 Dual TSO Transition System is a WSTS

In this section, we instantiate the framework of WSTS to show the following result:

**Theorem 2.** The Dual TSO reachability problem is decidable.

The rest of this section is devoted to the proof of the above theorem. Let $P = (A_1, A_2, \ldots, A_n)$ be a concurrent system (as defined in Section 3). Let $T_{DTSO} = (C_{DTSO}, \{ C_{\text{init}}^D \} \cup \Delta_{\text{aux}}) \rightarrow_{DTSO}$ be the transition system induced by $P$ under the Dual TSO semantics (as defined in Section 3.3). In the following, we will first define a well-quasi ordering $\sqsubseteq$ on the set of DTSO-configurations (Lemma 4) such that for every two configurations $c_1$ and $c_2$, it is decidable whether $c_1 \sqsubseteq c_2$. Then we show that the transition system induced under the Dual TSO semantics is monotonic wrt. to $\sqsubseteq$ (Lemma 5). We will show also that the Dual TSO reachability problem for $P$ can be reduced to the coverability problem in the monotonic transition system $(T_{DTSO}, \sqsubseteq)$ (Lemma 6). (Observe that this reduction is needed since we require that the load buffers are empty when defining the Dual TSO reachability problem.) The second sufficient condition (i.e., checking whether the upward closed set $\{ c \} \uparrow$, with $c$ is a DTSO-configuration, contains an initial configuration) for the decidability of the coverability problem is trivial. This check boils down to verifying whether $c$ is an initial configuration. Finally, the computability of the set of minimal configurations for the set of predecessors of any upward closed set is stated by the following lemma:

**Lemma 3.** For any configuration $c$, we can compute $\text{minpre}(\{ c \})$.

**Well-quasi Ordering.** In the following, we define a well-quasi ordering $\sqsubseteq$ on $C_{DTSO}$. Let us first introduce some notations and definitions. Consider a word $w \in ((X \times V) \cup (X \times V \times \{\text{own}\}))^*$ representing the content of a load buffer. We define an operation that divides $w$ into a number of fragments according to the most-recent own-messages concerning each variable. We define $[w]_{\text{own}} := (w_1, (x_1, v_1, \text{own}), w_2, \ldots, w_m, (x_m, v_m, \text{own}), w_{m+1})$, where the following conditions are satisfied: (1) $x_i \neq x_j$ if $i \neq j$, (2) if $(x, v, \text{own}) \in w_i$ then $x = x_j$ for some $j < i$ (i.e., the most recent own-message on $x_j$ occurs at position $j$), and (3) $w = w_1 \cdot (x_1, v_1, \text{own}) \cdot w_2 \cdot \cdots \cdot w_m \cdot (x_m, v_m, \text{own}) \cdot w_{m+1}$ (i.e., the fragments correspond to the given word $w$).
Let $w, w' \in ((X \times V) \cup (X \times \{\text{own}\}))^*$ be two words. Let us assume that $[w]_{\text{own}} = (w_1, (x_1,v_1,\text{own}), w_2, \ldots, w_r, (x_r,v_r,\text{own}), w_{r+1})$, and $[w']_{\text{own}} = (w'_1, (x'_1,v'_1,\text{own}), w'_2, \ldots, w'_{m'}, (x'_{m'},v'_{m'},\text{own}), w'_{m'+1})$. We write $w \sqsubseteq w'$ to denote that the following conditions are satisfied: (i) $r = m'$, (ii) $x'_i = x_i$ and $v'_i = v_i$ for all $i : 1 \leq i \leq m$, and (iii) $w_i \leq w'_i$ for all $i : 1 \leq i \leq m + 1$.

Consider two DTSO-configurations $c = (q, b, mem)$ and $c' = (q', b', mem')$, we extend the ordering $\sqsubseteq$ to configurations as follows: $c \sqsubseteq c'$ iff the following conditions are satisfied: (i) $q = q'$, (ii) $(p(p) \sqsubseteq b'(p)$ for all process $p \in P$, and (iii) $\text{mem}' = \text{mem}$. The following lemma shows that $\sqsubseteq$ is indeed a well-quasi ordering.

**Lemma 4.** The relation $\sqsubseteq$ is a well-quasi ordering over $\mathcal{C}_{\text{DTSO}}$. Furthermore, for every two DTSO-configurations $c_1$ and $c_2$, it is decidable whether $c_1 \sqsubseteq c_2$.

**Monotonicity.** Given configurations $c_1 = (q_1, b_1, \text{mem}_1), c_2 = (q_2, b_2, \text{mem}_2), c_3 = (q_3, b_3, \text{mem}_3) \in \mathcal{C}_{\text{DTSO}}$ such that $c_1 \xrightarrow{t}_{\text{DTSO}} c_2$ for some transition $t \in \Delta_p \cup \{\text{propagate}, \text{delete}\} | x \in X\}$, with $p \in P$, and $c_1 \sqsubseteq c_3$, we will show that it is possible to compute a configuration $c_4 \in \mathcal{C}_{\text{DTSO}}$ and a run $\pi$ such that $c_3 \xrightarrow{\pi}_{\text{DTSO}} c_4$ and $c_2 \sqsubseteq c_4$. To that aim, we first show that it is possible from $c_3$ to reach a configuration $c'_3$ by performing a certain number of $\text{delete}_p$ operations, such that the process $p$ will have the same last message in its load buffer in the configurations $c_1$ and $c'_3$ while $c_1 \sqsubseteq c'_3$. Then, from the configuration $c'_3$, the process $p$ can perform the same transition $t$ as $c_1$ did in order to reach the configuration $c_4$ such that $c_2 \sqsubseteq c_4$. Let us assume that $[b_3(p)]_{\text{own}}$ is of the form $(w_1, (x_1,v_1,\text{own}), w_2, \ldots, w_m, (x_m,v_m,\text{own}), w_{m+1})$, and $[b_3(p)]_{\text{own}}$ is of the form $(w'_1, (x'_1,v'_1,\text{own}), w'_2, \ldots, w'_{m'}, (x'_{m'},v'_{m'},\text{own}), w'_{m'+1})$. We define the word $w \in (X \times V) \cup (X \times \{\text{own}\})^*$ to be the longest word such that $w_{m+1} = w' \cdot w$ with $w_{m+1} \leq w'$. Observe that in this case we have either $w_{m+1} = w' = \epsilon$ or $w'([w]) = w_{m+1}([w_{m+1}])$. Then, after executing a certain number $|w|$ of $\text{delete}_p$ operations from the configuration $c_3$, one can obtain a configuration $c'_3 = (q_3, b'_3, \text{mem}_3)$ such that $b_3 = b'_3[p \leftarrow b'_3(p) \cdot w]$. As a consequence, we have $c_1 \sqsubseteq c'_3$. Furthermore, since $c_1$ and $c'_3$ have the same global state, the same memory valuation, the same sequence of most-recent own messages concerning each variable, and the same last message in the load buffer of $p$, $c'_3$ can perform the transition $t$ and reaches a configuration $c_4$ such that $c_2 \sqsubseteq c_4$. The following lemma shows that $(\mathcal{T}_{\text{DTSO}}, \sqsubseteq)$ is a monotonically system.

**Lemma 5.** The relation $\xrightarrow{\pi}_{\text{DTSO}}$ is monotonically wrt. $\sqsubseteq$.

**From Reachability to Coverability.** Let $q_{\text{target}}$ be a global state of $\mathcal{P}$ and $m_{\text{target}}$ be the set of DTSO-configurations of the form $c = (q_{\text{target}}, b, \text{mem})$ with $b(p) = \epsilon$ for all $p \in P$. Next, we show that the reachability problem of $q_{\text{target}}$ in $\mathcal{T}_{\text{DTSO}}$ can be reduced to the coverability problem of $m_{\text{target}}$ in $(\mathcal{T}_{\text{DTSO}}, \sqsubseteq)$. Recall that $q_{\text{target}}$ in $\mathcal{T}_{\text{DTSO}}$ is reachable in $\mathcal{T}_{\text{DTSO}}$. Let us assume that $m_{\text{target}} \uparrow$ is reachable in $\mathcal{T}_{\text{DTSO}}$. This means that there is a configuration $c \in m_{\text{target}} \uparrow$ which is reachable in $\mathcal{T}_{\text{DTSO}}$. Let us assume that $c$ is of the form $(q_{\text{target}}, b, \text{mem})$. Then, from the configuration $c$, it is possible to reach the configuration $c' = (q_{\text{target}}, b', \text{mem})$, with $b'(p) = \epsilon$ for all $p \in P$, by performing a sequence of $\text{delete}_p$ operations to empty the load buffer of each process. It is then easy to see that $c' \in m_{\text{target}}$ and so $m_{\text{target}}$ is reachable in $\mathcal{T}_{\text{DTSO}}$. The other direction of the following lemma is trivial since $m_{\text{target}} \subseteq m_{\text{target}} \uparrow$.

**Lemma 6.** $m_{\text{target}} \uparrow$ is reachable in $\mathcal{T}_{\text{DTSO}}$ iff $m_{\text{target}}$ is reachable in $\mathcal{T}_{\text{DTSO}}$. 

**CONCUR 2016**
5 Parameterized Concurrent Systems

Let $V$ be a finite data domain and $X$ be a finite set of variables ranging over $V$. A parameterized concurrent system (or simply a parameterized system) consists of an unbounded number of identical processes running under the Dual TSO semantics. Formally, a parameterized system $S$ is defined by a finite-state automaton $A = (Q, q^\text{init}, \Delta)$ uniformly describing the behavior of each process. An instance of $S$ is a concurrent system $P = (A_1, A_2, \ldots, A_n)$, for some $n \in \mathbb{N}$, where for every $p : 1 \leq p \leq n$, we have $A_p = A$. In other words, it consists of a finite set of processes each running the same code defined by $A$. Observe that considering that all processes run the same code is not a real restriction. In fact, the case where the processes run (finitely many) different finite-state automata $A_1, A_2, \ldots, A_m$ can be easily encoded in our model by constructing an extended automaton $A$ which represents the union of these automata $A_1, A_2, \ldots, A_m$. We use $\text{Inst}(S)$ to denote all possible instances of $S$. We use $\text{Inst}(P)$ to denote the transition system induced by an instance $P$ of $S$ under the Dual TSO semantics.

A parameterized configuration $\alpha$ is a pair $(P, c)$ where $P = \{1, \ldots, n\}$, with $n \in \mathbb{N}$, is the set of process IDs and $c$ is a DTSO-configuration of an instance $P$ of $S$ of the form $(A_1, A_2, \ldots, A_n)$. The parameterized configuration $\alpha = (P, c)$ is said to be initial if $c$ is an initial configuration of $P$ (i.e., $c \in \text{Init}(P)$). We use $\text{C}$ (resp. $\text{Init}$) to denote the set of all the parameterized configurations (resp. initial configurations) of $S$.

Let $\text{Act}$ denote the set of actions of all possible instances of $S$ (i.e., $\text{Act} = \cup_{P \in \text{Inst}(S)} \text{Act}(P)$). We define a transition relation $\rightarrow$ on parameterized configurations such that $(P, c) \xrightarrow{t} (P', c')$ for some action $t \in \text{Act}$ iff $P' = P$ and there is an instance $P$ of $S$ such that $t \in \text{Act}(P)$ and $c \xrightarrow{P} c'$. The transition system induced by $S$ is given by $T = (\text{C}, \text{Init}, \text{Act}, \rightarrow)$.

In the following we extend the definition of the Dual TSO reachability problem to the case of parameterized systems. A global state $q_{\text{target}} : P' \rightarrow Q$ is said to be reachable in $T$ if there exists a parameterized configuration $\alpha = (P, (q, b, \text{mem}))$, with $b(p) = \epsilon$ for all $p \in P$, such that $\alpha$ is reachable in $T$ and $q_{\text{target}}(1) \cdots q_{\text{target}}(|P'|) \preceq q(1) \cdots q(|P|)$. Then, the reachability problem consists in checking whether $q_{\text{target}}$ is reachable in $T$. In other words, the Dual TSO reachability problem for parameterized systems asks whether there is an instance of the parameterized system that reaches a configuration with a number of processes in certain given local states.

6 Decidability of the Parameterized Verification Problem

We prove hereafter the following fact:

**Theorem 7.** The Dual TSO reachability problem for parameterized systems is decidable.

Let $S = (Q, q^\text{init}, \Delta)$ be a parameterized system and $(\text{C}, \text{Init}, \text{Act}, \rightarrow)$ be its induced transition system. The proof of Theorem 7 is done by instantiating the framework of WSTS. Following that framework, we will first define an ordering that we denote by $\preceq$ on the set of parameterized configurations and show the monotonicity of the relation $\rightarrow$ wrt. this ordering (see Lemma 9 and Lemma 10). Then we will show that the Dual TSO reachability problem for $S$ can be reduced to the coverability problem in the monotonic transition system $(T, \preceq)$ (Lemma 11). The second sufficient condition (i.e., checking whether the upward closed set $\{\alpha\} \uparrow$, with $\alpha$ is a parameterized configuration, contains an initial configuration) for the decidability of the coverability problem is trivial. This check boils down to whether the configuration $\alpha$ is initial. Finally, the last sufficient condition (i.e., computing the set
of minimal configurations for the set of predecessors of any upward closed set) for the decidability of the coverability problem is stated by the following lemma:

**Lemma 8.** For any parameterized configuration $\alpha$, we can compute $\text{minpre}(\{\alpha\})$.

**Well-quasi Ordering.** Let $\alpha = (P, (q, b, \text{mem}))$ and $\alpha' = (P', (q', b', \text{mem}'))$ be two parameterized configurations. We define the ordering $\preceq$ on the set of parameterized configurations as follows: $\alpha \preceq \alpha'$ iff the following conditions are satisfied: (1) $\text{mem} = \text{mem}'$ and (2) there is an injection $h : \{1, \ldots, |P|\} \to \{1, \ldots, |P'|\}$ such that (i) $p < p'$ implies $h(p) < h(p')$, and (ii) for every $p \in \{1, \ldots, |P|\}$, $q(p) = q'(h(p))$ and $b(p) \subseteq b'(h(p))$. The following lemma states that $\preceq$ is indeed a well-quasi ordering.

**Lemma 9.** The relation $\preceq$ is a well-quasi ordering over $C$. Furthermore, for every two parameterized configurations $\alpha$ and $\alpha'$, it is decidable whether $\alpha \preceq \alpha'$.

**Monotonicity.** Let $\alpha_1 = (P, (q_1, b_1, \text{mem}_1))$, $\alpha_2 = (P, (q_2, b_2, \text{mem}_2))$ and $\alpha_3 = (P', (q_3, b_3, \text{mem}_3))$ be parameterized configurations. Furthermore, we assume that $\alpha_1 \preceq \alpha_2 \alpha_3 \delta$ for some transition $\delta$. Since $\alpha_1 \preceq \alpha_3$, there is an injection function $h : \{1, \ldots, |P|\} \to \{1, \ldots, |P'|\}$ such that (i) $p < p'$ implies $h(p) < h(p')$, and (ii) for every $p \in \{1, \ldots, |P|\}$, $q(p) = q_3(h(p))$ and $b_3(p) \subseteq b_3(h(p))$. We define the parameterized configuration $\alpha'$ from $\alpha_3$ by only keeping the local state and load buffers of processes in $h(P)$. Formally, $\alpha' = (P, (q', b', \text{mem}'))$ is defined as follows: (i) $\text{mem}' = \text{mem}_3$ and (ii) for every $p \in \{1, \ldots, |P|\}$, $q'(p) = q_3(h(p))$ and $b'(p) = b_3(h(p))$. (Observe that $(q_1, b_1, \text{mem}_1) \subseteq (q', b', \text{mem}')$. Since the relation $\rightarrow_{\text{DTSO}}$ is monotonic wrt. the ordering $\preceq$ (see Lemma 5), there is a Dual TSO-configuration $(q'', b'', \text{mem}'')$ such that $(q', b', \text{mem}') \rightarrow_{\text{DTSO}} (q'', b'', \text{mem}'')$ and $(q_2, b_2, \text{mem}_2) \subseteq (q'', b'', \text{mem}'')$. Consider now the parameterized configuration $\alpha_4 = (P', (q_4, b_4, \text{mem}_4))$ such that $\text{mem}'' = \text{mem}_4$, (i) for every $p \in \{1, \ldots, |P|\}$, $q''(p) = q_4(h(p))$ and $b''(p) = b_4(h(p))$, and (iii) for $p \in ((1, \ldots, |P'|) \setminus \{h(1), \ldots, h(|P|)\})$, we have $q_4(p) = q_3(p)$ and $b_4(p) = b_3(p)$. It is easy then to see that $\alpha_2 \preceq \alpha_4$ and $\alpha_3 \rightarrow \alpha_4$. Hence, we obtain the following result:

**Lemma 10.** The relation $\rightarrow$ is monotonic wrt. $\preceq$.

**From Reachability to Coverability.** Let $q_{\text{target}} : P' \to Q$ be a global state. Let $\mathcal{M}_{\text{target}}$ be the set of parameterized configurations of the form $\alpha = (P', (q_{\text{target}}, b, \text{mem}))$ with $b(p) = \epsilon$ for all $p \in P'$. In the following, we show that $\mathcal{M}_{\text{target}} \uparrow$ is reachable in $T$ iff there is a parameterized configuration $\alpha = (P, (q, b, \text{mem}))$, with $b(p) = \epsilon$ for all $p \in P$, such that $\alpha$ is reachable in $T$ and $q_{\text{target}}(1) \cdots q_{\text{target}}(|P'|) \preceq q(1) \cdots q(|P|)$.

Let us assume that there is a parameterized configuration $\alpha = (P, (q, b, \text{mem}))$, with $b(p) = \epsilon$ for all $p \in P$, such that $\alpha$ is reachable in $T$ and $q_{\text{target}}(1) \cdots q_{\text{target}}(|P'|) \preceq q(1) \cdots q(|P|)$. It is then easy to show that $\alpha \in \mathcal{M}_{\text{target}} \uparrow$.

Now let us assume that there is a $\alpha' = (P'', (q', b', \text{mem}')) \in \mathcal{M}_{\text{target}} \uparrow$ which is reachable in $T$. From the configuration $\alpha'$, it is possible to reach the configuration $\alpha'' = (P', (q', b', \text{mem}'))$, with $b'(p) = \epsilon$ for all $p \in P'$, by performing a sequence of delete operations to empty the load buffer of each process. Since $\alpha' \in \mathcal{M}_{\text{target}} \uparrow$, we have $q_{\text{target}}(1) \cdots q_{\text{target}}(|P'|) \preceq q'(1) \cdots q'(|P'|)$. Hence, $\alpha''$ is a witness of the state reachability problem.

**Lemma 11.** $q_{\text{target}}$ is reachable in $T$ iff $\mathcal{M}_{\text{target}} \uparrow$ is reachable in $T$. 

---

**CONCUR 2016**

P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo
The Benefits of Duality in Verifying Concurrent Programs under TSO

Table 1: Comparison between Dual-TSO and Memorax: The columns #P, #T and #C give the number of processes, the running time in seconds and the number of generated configurations, respectively. If a tool runs out of time, we put TO in the #T column and • in the #C column.

<table>
<thead>
<tr>
<th>Program</th>
<th>#P</th>
<th>Dual-TSO #T</th>
<th>Dual-TSO #C</th>
<th>Memorax #T</th>
<th>Memorax #C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>5</td>
<td>0.3</td>
<td>10641</td>
<td>559.7</td>
<td>10515914</td>
</tr>
<tr>
<td>LB</td>
<td>3</td>
<td>0.0</td>
<td>2048</td>
<td>71.4</td>
<td>1499475</td>
</tr>
<tr>
<td>WRC</td>
<td>4</td>
<td>0.0</td>
<td>1507</td>
<td>63.3</td>
<td>1398393</td>
</tr>
<tr>
<td>ISA2</td>
<td>3</td>
<td>0.0</td>
<td>509</td>
<td>21.1</td>
<td>226519</td>
</tr>
<tr>
<td>RWC</td>
<td>5</td>
<td>0.1</td>
<td>4277</td>
<td>61.5</td>
<td>1196988</td>
</tr>
<tr>
<td>W+RWC</td>
<td>4</td>
<td>0.0</td>
<td>1713</td>
<td>83.6</td>
<td>1389099</td>
</tr>
<tr>
<td>IRIW</td>
<td>4</td>
<td>0.0</td>
<td>520</td>
<td>34.4</td>
<td>358057</td>
</tr>
<tr>
<td>Nbw_w_wr</td>
<td>2</td>
<td>0.0</td>
<td>222</td>
<td>10.7</td>
<td>200844</td>
</tr>
<tr>
<td>Sense_rev_bar</td>
<td>2</td>
<td>0.1</td>
<td>1704</td>
<td>0.8</td>
<td>20577</td>
</tr>
<tr>
<td>Dekker</td>
<td>2</td>
<td>0.1</td>
<td>5053</td>
<td>1.1</td>
<td>19788</td>
</tr>
<tr>
<td>Dekker_simple</td>
<td>2</td>
<td>0.0</td>
<td>98</td>
<td>0.0</td>
<td>595</td>
</tr>
<tr>
<td>Peterson</td>
<td>2</td>
<td>0.1</td>
<td>5442</td>
<td>5.2</td>
<td>90301</td>
</tr>
<tr>
<td>Peterson_loop</td>
<td>2</td>
<td>0.2</td>
<td>7632</td>
<td>5.6</td>
<td>100082</td>
</tr>
<tr>
<td>Szymanski</td>
<td>2</td>
<td>0.6</td>
<td>29018</td>
<td>1.0</td>
<td>26003</td>
</tr>
<tr>
<td>MP</td>
<td>4</td>
<td>0.0</td>
<td>883</td>
<td>TO</td>
<td>•</td>
</tr>
<tr>
<td>Ticket_spin_lock</td>
<td>3</td>
<td>0.9</td>
<td>18963</td>
<td>TO</td>
<td>•</td>
</tr>
<tr>
<td>Bakery</td>
<td>2</td>
<td>2.6</td>
<td>82050</td>
<td>TO</td>
<td>•</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>2</td>
<td>0.2</td>
<td>8324</td>
<td>TO</td>
<td>•</td>
</tr>
<tr>
<td>Lamport_fast</td>
<td>3</td>
<td>17.7</td>
<td>292543</td>
<td>TO</td>
<td>•</td>
</tr>
<tr>
<td>Burns</td>
<td>4</td>
<td>124.3</td>
<td>2762578</td>
<td>TO</td>
<td>•</td>
</tr>
</tbody>
</table>

7 Experimental Results

We have implemented our techniques described in Section 4 and Section 6 in an open-source tool called Dual-TSO\(^1\). The tool checks the state reachability problems for (parameterized) concurrent systems under the Dual TSO semantics. We compare our tool with Memorax \([2, 3]\) which is the only precise and sound tool for deciding the state reachability problem of concurrent systems under TSO. Observe that Memorax cannot handle parameterized verification. All experiments are performed on an Intel x86-32 Core2 2.4 Ghz machine and 4GB of RAM.

In the following, we present two sets of results. The first set concerns the comparison of Dual-TSO with Memorax (see Table 1). The second set shows the benefit of the parameterized verification compared to the use of the state reachability when increasing the number of processes (see Figure 3 and Table 2). Our examples are from \([2, 10, 13, 4, 25]\). In all experiments, we set up the time out to 600 seconds.

Table 1 presents a comparison between Dual-TSO and Memorax on a representative sample of 20 benchmarks. In all these examples, Dual-TSO and Memorax return the same result for the state reachability problem (except 6 examples where Memorax runs out of time). In the examples where the two tools return, Dual-TSO out-performs Memorax and generates fewer configurations (and so uses less memory). Indeed, Dual-TSO is 600 times faster than

\(^1\) [https://www.it.uu.se/katalog/tuang296/dual-tso](https://www.it.uu.se/katalog/tuang296/dual-tso)
Table 2 Parameterized verification with Dual-TSO.

<table>
<thead>
<tr>
<th>Program</th>
<th>Dual-TSO</th>
<th>#T</th>
<th>#C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>0.0</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>LB</td>
<td>0.6</td>
<td>1028</td>
<td></td>
</tr>
<tr>
<td>MP</td>
<td>0.0</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>WRC</td>
<td>0.8</td>
<td>618</td>
<td></td>
</tr>
<tr>
<td>ISA2</td>
<td>4.3</td>
<td>1539</td>
<td></td>
</tr>
<tr>
<td>RWC</td>
<td>0.2</td>
<td>293</td>
<td></td>
</tr>
<tr>
<td>W+RWC</td>
<td>1.5</td>
<td>828</td>
<td></td>
</tr>
<tr>
<td>IRIW</td>
<td>4.6</td>
<td>648</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3 Running time of Memorax and Dual-TSO by increasing number of processes. The x axis is number of processes, the y axis is running time in seconds.

Memorax and generates 277 times fewer configurations on average.

The second set compares the scalability of Memorax and Dual-TSO while increasing the number of processes. The results are given in Fig. 3. We observe that Dual-TSO scales better than Memorax in all these examples. In fact, Memorax can only handle the examples with at most 5 processes. Table 2 presents the running time and the number of generated configurations when checking the state reachability problem for the parameterized version of these examples. We observe that the verification of these parameterized systems is much more efficient than verification of bounded-size instances (starting from a number of processes of 3 or 4), especially concerning memory consumption (which is given in terms of number of generated configurations). The reason behind is that the size of the generated minor sets in the analysis of a parameterized system is usually smaller than the size of the generated configurations during the analysis of an instance of the system with a large number of processes.

8 Conclusion

In this paper, we have presented an alternative (yet equivalent) semantics to the classical one for the TSO model. This new semantics allows us to understand the TSO model in a totally different way compared to the classical semantics. Furthermore, the proposed semantics offers several important advantages from the point of view of formal reasoning and program verification. First, the dual semantics allows transforming the load buffers to lossy channels without adding the costly overhead that was necessary in the case of store buffers. This means that we can apply the theory of well-structured systems [6, 5, 21] in a
The Benefits of Duality in Verifying Concurrent Programs under TSO

A straightforward manner leading to a much simpler proof of decidability of safety properties. Second, the absence of extra overhead means that we obtain more efficient algorithms and better scalability (as shown by our experimental results). Finally, the dual semantics allows extending the framework to perform parameterized verification which is an important paradigm in concurrent program verification.

In the future, we plan to apply our techniques to more memory models and to combine with predicate abstraction for handling programs with unbounded data domain.

References