On Verifying TSO Robustness for Event-Driven Asynchronous Programs *

Ahmed Bouajjani[1], Constantin Enea[1], Madhavan Mukund[2], and Rajarshi Roy[2]

[1] IRIF, University Paris Diderot, France
[2] Chennai Mathematical Institute, India

Abstract. We present a method for checking whether an event-driven asynchronous program running under the Total Store Ordering (TSO) memory model is robust, i.e., all its TSO computations are equivalent to computations under the Sequential Consistency (SC) semantics. We show that this verification problem can be reduced in polynomial time to a reachability problem in a program with two threads, provided that the original program satisfies a criterion called robustness against concurrency, introduced recently in the literature. This result allows to avoid explicit handling of all concurrent executions in the analysis, which leads to an important gain in complexity.

1 Introduction

Asynchronous event-driven programming allows procedures to be executed asynchronously (after their invocation), e.g., as callbacks handling the occurrences of external events. In particular, modern user interface (UI) frameworks in Android, iOS, and Javascript, are instances of asynchronous event-driven programming. These frameworks dedicate a distinguished main thread, called UI thread, to handling user interface events. Since responsiveness to user events is a key concern, common practice is to let the UI thread perform only short-running work in response to each event, delegating to asynchronous tasks the more computationally demanding part of the work. These asynchronous tasks are in general executed in parallel on different background threads, depending on the computational resources offered by the execution platform. The apparent simplicity of UI programming models is somewhat deceptive. The difficulty of writing safe programs given the concurrency of the underlying execution platform is still all there.

Bouajjani et al. [6] have proposed a correctness criterion for such programs which requires that their standard (multi-thread) semantics is a *refinement* of a single-thread semantics where user events are executed until completion in a serial manner, one after the other, and the asynchronous tasks created by an event handler (and recursively, by its callee) are executed asynchronously

* This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177), and the Franco-Indian CNRS-DST/CEFIPRA collaborative project AVECSO.

(once the execution of the creator finishes), but serially and in the order of their invocation. The multi-thread semantics being a refinement of the single-thread implies that the sets of observable reachable states of the program w.r.t. both semantics are exactly the same (see Section 4 for the exact definition). While the multi-thread semantics provides greater performance and responsiveness, the single-thread semantics is simpler to apprehend. The inherent non-determinism due to concurrency and asynchronous task dispatching from the multi-thread semantics is not present in the context of the single-thread one. A program that satisfies this refinement condition is said to be *robust against concurrency*. The same work has shown that violations of this criterion correspond in practice to undesirable behaviors, and that this criterion can be checked efficiently (in polynomial time for Boolean programs), using a linear time reduction to the state reachability problem in *sequential programs*.

Robustness against concurrency assumes that the programs are executed under Sequential Consistency [15] (SC), where the actions of different threads are interleaved while the program order between actions of each thread is preserved. For performance reasons, modern multiprocessors implement weaker memory models, e.g., Total Store Ordering (TSO) [18] in x86 machines, which relax the program order. For instance, the main feature of TSO is the write-to-read relaxation, which allows reads to overtake writes. This relaxation reflects the fact that writes are buffered before being flushed non-deterministically to the main memory. In this work, we consider asynchronous event-driven programs that are executed under TSO, and investigate the relationship between robustness against concurrency and *robustness against TSO* [3, 7, 8, 9], which requires that a TSO program admits the same behaviors as if it was run under SC.

We first show that robustness against concurrency (which concerns only the SC semantics) doesn't imply robustness against TSO (see Section 2), i.e., even if the SC semantics doesn't allow interference between events and asynchronous invocations (they can be seen as atomic), the TSO semantics can still introduce new behaviors which are not possible under SC (therefore, breaking the atomicity of the events and asynchronous invocations). However, we show that checking robustness against TSO for programs satisfying robustness against concurrency is more efficient than in the general case. Using the approach in [5], we show that such a program is not robust against TSO iff it admits a robustness violation which can be simulated using just *two* threads. This implies that checking robustness against TSO for a program with an unbounded number of threads (that is robust against concurrency) can be reduced in polynomial time to the problem of checking TSO robustness for a program with just two threads. The latter has been proved to be polynomial time for Boolean programs in [5].

Our work leads in particular to an efficient approach for the verification of functional correctness of event-driven asynchronous programs running under TSO that consists in solving three separate problems: (1) showing that the program is functionally correct w.r.t the single-thread semantics, (2) showing that the program is robust against concurrency, and (3) showing that the program is

```
// Event 1
void searchForNews(String key) {
  new SearchTask.execute(key);
  new SaveTask.execute(key); }

// Event 2
void showDetail(int id) {
  // show detail of the idth news
  new DownloadTask.execute(id); }

class SaveTask extends AsyncTask {
  void doInBackground(String key) {
    // write key to the database } }
```

```
class SearchTask extends AsyncTask {
  List result = null;
  void doInBackground(String key) {
    result = ...
// get from the network
  }
  void onPostExecute() {
    list = result;
    // display the list of titles } }

class DownloadTask extends AsyncTask {
   String content = null;
   void doInBackground(int id) {
      content = ... // get from the network
   }
   void onPostExecute() {
      // display the content } }
```

Fig. 1: A program which is robust against concurrency.

robust against TSO. These problems can be solved efficiently by considering only particular types of computations captured by sequential or two thread programs.

Related Work. The weakest correctness criterion that enables SC reasoning for proving invariants of programs running under TSO is *state-robustness against TSO* i.e., the reachable set of states is the same under both SC and TSO. However, this problem has high complexity (at least non-primitive recursive for programs with a finite number of threads and a finite data domain [4]). Therefore, it is difficult to come up with an efficient and precise solution. A symbolic decision procedure is presented in [1] and over-approximate analyses are proposed in [13, 14]. Due to the high complexity of state-robustness, stronger correctness criteria with lower complexity have been proposed. Trace-robustness (that we call simply robustness against TSO in our paper) is one of the most studied criteria in the literature. Trace-robustness is PSPACE-complete for a finite number of threads and a finite data domain [7] and EXPSPACE-complete for an unbounded number of threads. Besides trace-robustness, there are other correctness criteria like triangular race freedom (TRF) and persistence that are stronger than state-robustness. Persistence ([2]) is incomparable to trace-robustness and TRF [16] is stronger than both trace-robustness and persistence. Our work considers the specific case of event-driven asynchronous programs and shows that checking trace-robustness has a much lower complexity (polynomial time), provided the programs are robust against concurrency.

The works in [10, 12, 11, 17] target exploring interesting subsets of executions and schedules for asynchronous programs running under SC, that offer a large coverage of the execution space. This is orthogonal to the focus of our paper which is to analyze behaviors of these programs running under TSO.

2 Motivation

We briefly discuss the relevance of robustness against concurrency using the program in Figure 1, which is extracted from an Android application (we assume it is executed under SC). This program has two event handlers `searchForNews`

and `showDetail` which can be invoked by the user to search for news containing a keyword and to display the details of a selected news respectively. Robustness against concurrency can be characterized as the conjunction of *event-serializability* and *event determinism*, which are variants of the classical notions of serializability and determinism, adapted to our context. Intuitively, since the single-thread semantics defines a unique execution, given a set of external events (partially ordered w.r.t. some causality relation imposed by the environment), then (1) the executions of the event handlers must be serializable (to an order compatible with their causality relation), i.e., the execution of each event handler and its subtasks can be seen as an atomic transaction, and (2) the execution of each event handler is deterministic, i.e., it always leads to the same state, for any possible scheduling of its parallel subtasks.

The procedure `searchForNews` creates two `AsyncTask` objects `SearchTask` and `SaveTask` whose `execute` method will invoke asynchronously `doInBackground` followed by `onPostExecute`, in the case of the former. Under the multi-thread semantics, `doInBackground` is invoked on a new thread and `onPostExecute` is invoked on the main thread. When the user input to search for news is triggered, the invocation `doInBackground` of `searchTask` connects to the network, searches for the keyword and fetches the list of resulting news titles. Then, the invocation `onPostExecute` displays the list of titles to the user. `SaveTask` saves the keyword to a database representing the search history in the background. The background tasks `SearchTask.doInBackground` and `SaveTask.doInBackground` might interfere but any interleaving produces the same result, i.e., it can be assumed `searchForNews` is deterministic.

The second event, to show the details of a title, can be triggered once the list of titles are displayed on the screen. It invokes an asynchronous task to download the contents of the news in the background and then displays it. In this case, the tasks are executed in a fixed order and the event is trivially deterministic.

Concerning serializability, the invocation of `SaveTask` in the first event and the second event might interleave (under the concrete semantics). However, assuming that the second event is triggered once the results are displayed, any such interleaving results in the same state as a serial execution of these events.

To show that robustness against concurrency doesn't imply robustness against TSO consider the two programs in Figure 2. The program on the left of Figure 2 consists of a single event which is deterministic, but it is not robust against TSO. The TSO memory model admits a computation where both `a` and `b` are 0 at the end of the program which is not possible under SC. The program on the right of Figure 2 consists of n events e_i with $1 \leq i \leq n$, which are all deterministic and serializable (therefore the program is robust against concurrency) but its semantics under TSO admits a computation where all the $\mathtt{a}_i$ variables are 0 at the end of the program (which is again not possible under SC).

```
procedure e1() {
  x:=1;
  a:=y;
  async[any] p();
}
procedure p() {
  y:=1;
  b:=x;
}
```

```
// for every 1 ≤ i ≤ n
procedure ei() {
  if(yi−1=1)
    async[any] pi();
}
procedure pi() {
  xi:=1;
  ai:=x(i+1) mod n;
  async[any] qi();
}
procedure qi() {
  yi:=1;
}
```

Fig. 2: Asynchronous programs which are not robust against TSO. All variables are initially set to 0 except for $\mathtt{y}_0$ in the second program which is set to 1.

3 Programs

In order to give a generic definition of robustness, which doesn't depend on any particular asynchronous-programming platform or syntax, we frame our discussion around the abstract notion of programs defined in § 3.1. Two alternative *multi-thread* and *single-thread* semantics to programs *under the SC memory model* are given in § 3.2 and § 3.4. A version of the multi-threaded semantics *under the TSO memory model* is given in §3.3.

3.1 Asynchronous Event-Driven Programs

We define an event handler as a procedure which is invoked in response to a user or a system input. For simplicity, we assume that inputs can arrive in any order. Event handlers may have some asynchronous invocations of other procedures, to be executed later on the same thread or on a background thread.

We fix sets G and L of global and local program states. Local states $\ell \in L$ represent the code and data of an asynchronous procedure or event-handler invocation, including the code and data of all nested synchronous procedure calls. A program is defined as a mapping between pairs of global and local states which gives the semantics of each statement in the code of a procedure (the association between threads, local states, and procedure invocations is defined in § 3.2 and § 3.4). To formalize the notions of robustness, this mapping associates with each statement a label called *program action* that records the set of accessed global variables and the asynchronous invocations in that statement. An *event set* $E \subset L$ is a set of local states; each $e \in E$ represents the code and data for a single event handler invocation (called event for short).

Formally, a *program* $P : G \times L \to G \times L \times B$ maps global states $g \in G$ and local states $\ell \in L$ to new states and program actions; each $P(g, \ell)$ represents a single program transition. Supposing that the global states $g \in G$ are maps from program variables x to values $g(x)$, and that local states $\ell \in L$ map program variables a to values $\ell(a)$ and a program counter variable pc to program

$x := a \quad a := x \quad a := \mathit{expr} \quad \texttt{call}\ p(\boldsymbol{y}) \quad \texttt{async}[w]\ p(\boldsymbol{y}) \quad \texttt{return}$

Fig. 3: Basic statements. The metavariables x and a range over global and local variable names, respectively, *expr* is an expression over local variables, p ranges over procedure names, and w over the symbols "main" and "any."

statements $\ell(\mathrm{pc})$, we give an interpretation to the standard program syntax listed in Figure 3. Besides assignments and synchronous procedure calls, $\texttt{async}[w]\ p(\boldsymbol{y})$ represents an asynchronous invocation of $p(\boldsymbol{y})$ run on a distinguished main thread when $w = \mathrm{main}$, and on an arbitrary thread when $w = \mathrm{any}$. For instance, writing ℓ^+ to denote $\ell[\mathrm{pc} \mapsto \ell(\mathrm{pc})+1]$, then $P(g, \ell)$ is

- $\langle g[x \mapsto \ell(a)], \ell^+, \mathrm{wr}(x, \ell(a))\rangle$ when $\ell(\mathrm{pc})$ is a global-variable write x `:=` a,
- $\langle g, \ell^+[a \mapsto g(x)], \mathrm{rd}(x, g(x))\rangle$ when $\ell(\mathrm{pc})$ is a global-variable read a `:=` x,
- $\langle g, \ell^+[a \mapsto \ell(\mathit{expr})], \epsilon\rangle$ when $\ell(\mathrm{pc})$ is a local computation a `:=` *expr* (here, $\ell(\mathit{expr})$ is the standard extension of ℓ to expressions *expr* over local variables),
- $\langle g, \ell^+, \mathrm{invoke}(\ell', w)\rangle$ when $\ell(\mathrm{pc})$ is an asynchronous invocation $\texttt{async}[w]\ p(\boldsymbol{y})$, where ℓ' maps the parameters of procedure p to the invocation arguments $\boldsymbol{y}$ and pc to the initial statement of p, and
- $\langle g, \ell, \mathrm{return}\rangle$ when $\ell(\mathrm{pc})$ is the `return` statement.

The semantics of other statements, including synchronous procedure calls $\texttt{call}\ p(\boldsymbol{y})$, if-then-else conditionals, while loops, or goto statements, etc., is standard, and yield the empty program action ε.

An event is called *sequential* when its code doesn't contain asynchronous invocations $\texttt{async}[w]\ p(\boldsymbol{y})$. Also, a program P with event set E is called *sequential* when every event $e \in E$ is sequential. Otherwise, P is called *concurrent*.

3.2 SC Multi-thread Asynchronous Semantics

A *task* $u = \langle \ell, i, j, k\rangle$ is a local state $\ell \in L$ along with invocation, event and thread identifiers $i, j, k \in \mathbb{N}$, and U denotes the set of tasks. We write $\mathrm{invoc}(u)$, $\mathrm{event}(u)$, and $\mathrm{thread}(u)$ to refer to i, j, and k, respectively. A *configuration* $c = \langle g, t, q\rangle$ is a global state $g \in G$ along with sets $t, q \subseteq U$ of running and waiting tasks such that: (1) invocation identifiers are unique, i.e., $\mathrm{invoc}(u_1) \neq \mathrm{invoc}(u_2)$ for all $u_1 \neq u_2 \in t \cup q$, and (2) threads run one task at a time, i.e., $\mathrm{thread}(u_1) \neq \mathrm{thread}(u_2)$ for all $u_1 \neq u_2 \in t$. The set of configurations is denoted by C_m. We say that a thread k is *idle* in c when $k \notin \{\mathrm{thread}(u) : u \in t\}$, and that an identifier i, j, k is *fresh* when $i, j, k \notin \{\alpha(u) : u \in (t \cup q)\}$ for $\alpha \in \{\mathrm{invoc}, \mathrm{event}, \mathrm{thread}\}$, respectively. A configuration is *idle* when all threads are *idle*.

To define robustness against concurrency, we expose the following set A of actions in execution traces:

$$A = \{\mathrm{start}(j), \mathrm{end}(j) : j \in \mathbb{N}\} \cup B \cup \{\mathrm{invoke}(i), \mathrm{begin}(i), \mathrm{return}(i) : i \in \mathbb{N}\}$$

By convention, we denote asynchronous procedure invocation, event, and thread identifiers, respectively, with the symbols i, j, k. The $\mathrm{start}(j)$ and $\mathrm{end}(j)$ actions represent the start and end of event j; the $\mathrm{invoke}(i)$, $\mathrm{begin}(i)$, and $\mathrm{return}(i)$

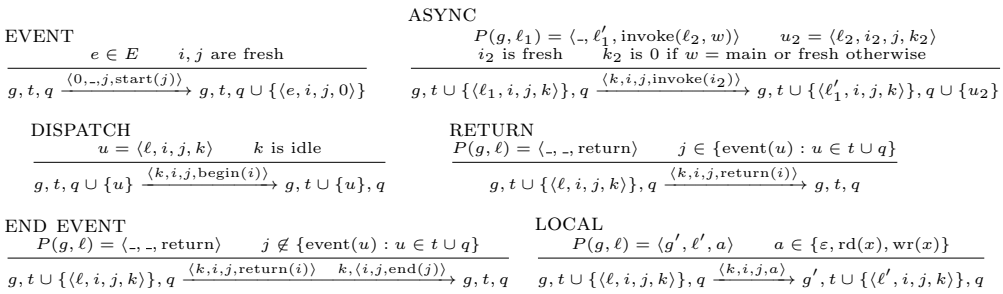

Fig. 4: The multi-thread transition function $\rightarrow$ for a program P with event set E.

actions represent an asynchronous procedure invocation (when it is added to the queue of pending invocations), the start of i's execution (when it is removed from the queue), and return of i, respectively. The set X of memory accesses is defined as in the program actions of Section 3.1.

The transition function $\rightarrow$ in Figure 4 is determined by a program P and event set E, and maps a configuration $c_1 \in C_m$ and thread identifier $k \in \mathbb{N}$ to another configuration $c_2 \in C_m$ and label $\lambda = \langle k, i, j, a\rangle$ where i and j are invocation and event identifiers, and $a \in A$ is an action — we write thread(λ), invoc(λ), event(λ), and act(λ) to refer to k, i, j, and a, respectively. Let Λ_{SC} denote the set of such transition labels λ. EVENT transitions mark the beginnings of events. We assume that all events are initiated on thread 0, which is also referred to as the *main* thread. Also, for simplicity, we assume that events can be initiated arbitrarily at any time. Adding causality constraints between events, e.g., one event can be initiated only when a certain action has been executed, is possible but tedious. ASYNC transitions create pending asynchronous invocations, DISPATCH transitions begin the execution of pending invocations, and RETURN transitions signal their end (the condition in the right ensures that this is not a return from an event). END EVENT transitions mark the end of an event and by an abuse of notation, they map c_1 and k to a configuration c_2 and two labels, return(i) denoting the end of the asynchronous invocation and end(j) denoting the end of the event. All other transitions are LOCAL.

An *execution* of a program P under the SC multi-thread semantics with event set E to configuration c_n is a configuration sequence $c_0 c_1 \dots c_n$ such that $c_m \xrightarrow{\lambda_{m+1}} c_{m+1}$ for $0 \leq m < n$. We call the sequence $\lambda_1 \dots \lambda_n$ the *trace* of $c_0 c_1 \dots c_n$. The set of traces of P with E under the SC multi-thread semantics is denoted by $[\![P, E]\!]_m^{SC}$. We may omit P when it is understood from the context.

The *call tree* of a trace τ is a ranked tree $\mathit{CallTree}_\tau = \langle V, E, O\rangle$ where V are the invocation identifiers in τ, and the set of edges E contains an edge from i_1 to i_2 whenever i_2 is invoked by i_1, i.e., τ contains a label $\langle i_1, \_, \text{invoke}(i_2)\rangle$. The function $O : E \to \mathbb{N}$ labels each edge (i_1, i_2) with an integer n whenever i_2 is the nth invocation made by i_1, i.e., $\langle i_1, \_, \text{invoke}(i_2)\rangle$ is the nth label of the form $\langle i_1, \_, \text{invoke}(\_)\rangle$ occurring in τ (reading τ from left to right).

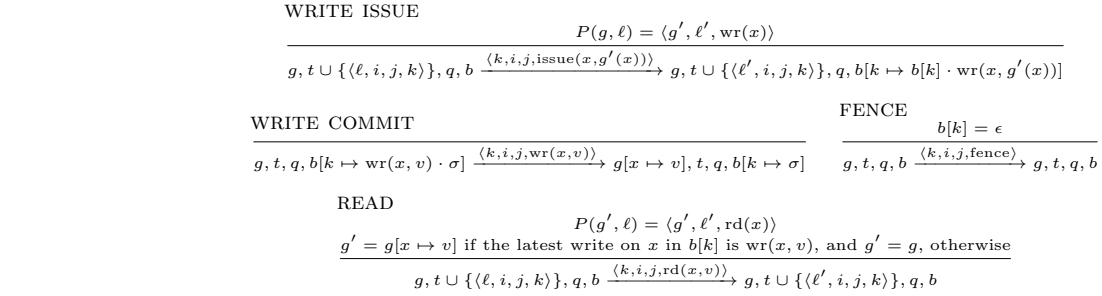

Fig. 5: The TSO multi-thread transition function $\rightarrow$ for a program P with event set E.

3.3 TSO Multi-thread Asynchronous Semantics

The extension of the SC multi-thread semantics of Section 3.2 to the TSO memory model is obtained by adding write buffers to each thread, such that each write on a global variable is first stored in a write buffer before being flushed non-deterministically to the global memory, and each read takes a value from the write buffer, if a write on the corresponding global variable exists, or the global memory, otherwise. To deal with TSO memory effects, we also extend the program syntax of Section3.1 by adding a statement `fence` which ensures that all the writes in the buffer have been flushed to the global memory.

A configuration $c = \langle g, t, q, b \rangle$ extends an SC configuration $\langle g, t, q \rangle$ with a set b of write buffers, one for each thread. The write buffer of a thread k is denoted by $b[k]$. The transition function $\rightarrow$ for local actions that access global variables is given in Figure 5 (the transitions corresponding to the rest of the actions are defined exactly as in the SC case). WRITE ISSUE adds a global variable write to a write buffer, WRITE COMMIT executes the oldest write in a buffer on the global memory, READ and FENCE give the semantics of global variable read and fence statements. Let Λ_{TSO} be the set of transition labels in the TSO semantics, i.e., the union of Λ_{SC} and all labels $\langle k, i, j, \mathrm{issue}(x, v) \rangle$ and $\langle k, i, j, \mathrm{fence} \rangle$ representing write issue and fence transitions, respectively.

The set of traces of P with E under the TSO multi-thread semantics is denoted by $[\![P, E]\!]_m^{TSO}$.

3.4 Single-thread Asynchronous Semantics

Conversely to the multi-thread semantics of Section 3.2, our single-thread semantics minimizes the set of possible program behaviors by executing all events and asynchronous invocations on the main thread, the asynchronous procedure invocations being executed in a *fixed* order (in this context, the memory model is not important since SC and TSO produce the same behaviors on a single thread).

We explain the order in which asynchronous invocations are executed using the event handler `searchForNews` in Figure 1. This event handler is supposed to add the keyword to the search history only after the fetching of the news containing that keyword succeeds. This expectation corresponds to executing the asynchronous procedures according to the DFS traversal of the call tree.

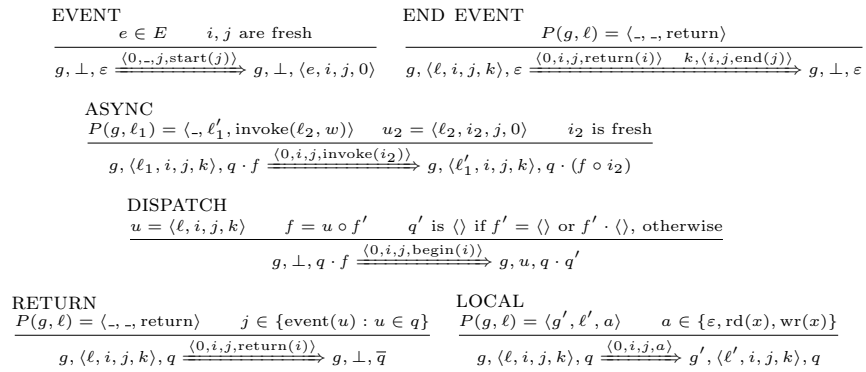

Fig. 6: The single-thread transition function $\Rightarrow$ for a program P with events E (ε and $\langle\rangle$ are the empty sequence and tuple, resp.,). Also, f and f' are tuples, and $\overline{q}$ is obtained by popping a queue from q if this queue is empty, or $\overline{q} = q$, otherwise.

In general, this traversal is relevant because it preserves causality constraints which are imprinted in the structure of the code, like in the case of standard synchronous procedure calls. Note however that this semantics is not equivalent to interpreting asynchronous invocations as synchronous, since the caller finishes before the callee starts. In the formalization of this semantics, the DFS traversal is modeled using a stack of FIFO queues for storing the pending invocations.

The formalization of the single-thread semantics reuses the notions of task and label in §3.2. Let U_0 be the set of tasks $u = \langle \ell, i, j, 0\rangle$ executing on thread 0. We overload the term *configuration* which in this context is a tuple $c = \langle g, u, q\rangle$ where $g \in G$, $u \in (U_0 \cup \{\bot\})$ is a possibly-empty task placeholder (at most one task is running at any moment), and $q \in (\mathsf{Tuples}(U_0))^*$ is a sequence of tuples of tasks (a tuple, resp., a sequence, denotes a FIFO queue, resp., a stack). C_s is the set of configurations of the single-thread semantics. We call $c \in C_s$ *idle* if $u = \bot$.

The transition function $\Rightarrow$ in Fig. 6 is essentially a restriction of $\rightarrow$ where all the procedures run on the main thread, an event begins when there are no pending invocations, and the rules ASYNC and DISPATCH use a stack of FIFO queues for storing pending invocations. The effect of pushing/popping a queue to the stack or enqueuing/dequeueing a task to a queue is represented using the concatenation operation $\cdot$, resp.,$\circ$, for sequences, resp., tuples. Every task created by ASYNC is posted to the *main* thread and it is enqueued in the queue on the top of the stack q. DISPATCH dequeues a pending task from the queue f on the top of q, and pushes a new *empty* queue to q (for storing the tasks created during the newly started invocation) if f doesn't become empty. Moreover, the rules RETURN and END EVENT pop the queue on the top of q if it is empty.

An *execution* of a program P under the single-thread semantics with event set E to configuration c_n is a sequence $c_0 c_1 \dots c_n$ s.t. $c_m \xrightarrow{0,\lambda_{m+1}} c_{m+1}$ for $0 \leq m < n$. We call the sequence $\lambda_1 \dots \lambda_n$ the *trace* of $c_0 c_1 \dots c_n$.

The set of traces of P with E under the single-thread semantics is denoted by $[\![P, E]\!]_s$ (P may be omitted when it is understood from the context).

4 Robustness Criteria

We introduce the notions of robustness against concurrency [6] and robustness against TSO [3, 7, 8, 9] which imply that every "final" [3] state of a program P reachable under a weak semantics, the SC multi-thread semantics and respectively, the TSO multi-thread semantics, is also reachable in P under a strong semantics, the single-thread semantics and respectively, the SC multi-thread semantics. Since reasoning about sets of reachable states is difficult in general, these robustness notions are defined on traces and require that roughly, every trace of the weak semantics is 'equivalent" to a trace of the same program under the strong semantics. A trace is equivalent to another if it is a permutation that preserves the order between "conflicting" labels, e.g., accesses to the same global variable.

Let $\prec \subseteq \Lambda_{TSO} \times \Lambda_{TSO}$ be a *conflict relation* defined by

$$\begin{aligned}\lambda_1 \prec \lambda_2 \text{ iff } & act(\lambda_1), act(\lambda_2) \in \{\mathrm{wr}(x,v), \mathrm{rd}(x,v')\} \text{ for some } x, v, v', \text{ and} \\ & \mathrm{act}(\lambda_1) = \mathrm{wr}(x,v) \text{ or } \mathrm{act}(\lambda_2) = \mathrm{wr}(x,v) \\ & \text{or} \\ & \mathrm{thread}(\lambda_1) = \mathrm{thread}(\lambda_2)\end{aligned}$$

that relates any two labels accessing the same variable, one of them being a write (commit), or any two labels associated to the same thread. Given a trace $\tau = \tau_1 \cdot \lambda_1 \cdot \lambda_2 \cdot \tau_2$, we say that the trace $\tau' = \tau_1 \cdot \lambda_2 \cdot \lambda_1 \cdot \tau_2$ is derived from τ by a *$\prec$-valid swap* iff $\lambda_1 \not\prec \lambda_2$. A permutation τ' of a trace τ is *conflict-preserving* when τ' can be derived from τ through a sequence of $\prec$-valid swaps.

Robustness against concurrency states that every trace of the SC multi-thread semantics of a program P with event set E has a conflict-preserving permutation where events and asynchronous invocations don't interleave (they are executed serially, but maybe not until completion) and asynchronous invocations execute according to the DFS traversal of the call tree. Such conflict-preserving permutations can be simulated by a sequential program $\mathsf{seq}(P)$ where asynchronous invocations are rewritten to regular procedure calls which however, are not necessarily executed until completion. Incomplete executions of the procedures are simulated by adding a boolean flag `skip` to each procedure, which is non-deterministically turned to `false` (it is initially `true`) and which guards every statement in the original program (i.e., the statement can be executed only if `skip` is `true`). Since asynchronous invocations are rewritten to regular procedure calls, different events cannot interleave and invocations execute according to the DFS traversal of the call tree exactly as in the single-thread semantics (the latter requires that asynchronous invocations are not followed by accesses to global variables; see Bouajjani et al. [6] for more details.). Therefore, the SC multi-thread semantics of $\mathsf{seq}(P, E)$ coincides with its single-thread semantics. By an abuse of terminology, we say that a trace τ belongs to the single-thread semantics $[\![\mathsf{seq}(P), E]\!]_s$ even if the trace τ involves multiple threads, but substituting every thread id with 0 results in a trace in $[\![\mathsf{seq}(P), E]\!]_s$.

[3] Here, "final" means that there are no pending invocations.

Definition 1. *A program P with event set E is* robust against concurrency *if there is a conflict preserving permutation $\tau' \in [\![\mathsf{seq}(P), E]\!]_s$ for every trace $\tau \in [\![P, E]\!]_m^{SC}$.*

The following theorem shows that the problem of checking robustness against concurrency is polynomial time for boolean programs. It is a consequence of the fact that this problem can be reduced in linear time to a reachability problem in sequential programs (even for infinite-state programs).

Theorem 1 ([6]). *Checking robustness against concurrency for a program P with event set E, a fixed number of variables which are all boolean, and a fixed number of procedures, each procedure containing a fixed number of asynchronous invocations, is polynomial time decidable.*

While robustness against concurrency shows that there is no interference between events and asynchronous invocations under an SC semantics, robustness against TSO holds when the *non-atomic* writes allowed by the TSO memory model (that can be delayed and executed later on the global memory) introduce no behavior which is not also possible under the SC semantics. To simplify the exposition, we say that a trace $\tau \in [\![P, E]\!]_m^{TSO}$ (under the TSO memory model) belongs to the SC semantics $[\![P, E]\!]_m^{SC}$ of a program P with event set E even if the trace τ contains write issue and fence transition labels, but every write issue $\langle k, i, j, \mathrm{issue}(x, v)\rangle$ is immediately followed by the corresponding write commit $\langle k, i, j, \mathrm{wr}(x, v)\rangle$ and removing all the write issue and fence transition labels results in a trace in $[\![P, E]\!]_m^{SC}$.

Definition 2. *A program P with event set E is* robust against TSO *if there is a conflict preserving permutation $\tau' \in [\![P, E]\!]_m^{SC}$ for every trace $\tau \in [\![P, E]\!]_m^{TSO}$.*

Bouajjani et al. [5] have shown that checking robustness against TSO is EXPSPACE-complete. The upper bound relies on a polynomial time reduction to a reachability problem in a concurrent program running under SC.

5 Checking Robustness Against TSO

While Section 2 shows that robustness against concurrency doesn't imply robustness against TSO, we show however that for programs which are robust against concurrency, the problem of checking robustness against TSO can be solved more efficiently than in the general case. More precisely, we show that the latter can be reduced in polynomial time to the problem of checking robustness against TSO for a program with only two threads, which is itself polynomial time for boolean programs (since by the results of Bouajjani et al. [5], it can be reduced to a reachability problem in a boolean program with 2 threads).

Let P be a program with event set E that is robust against concurrency. We show that all its *minimal* TSO robustness violations, if any, can be simulated by a program $\mathsf{2\text{-}threads}(P)$ similar to $\mathsf{seq}(P)$ except that exactly one invocation

which was asynchronous in P remains asynchronous in $\mathsf{2\text{-}threads}(P)$ as well (this asynchronous invocation is chosen non-deterministically).

Following the results in [5], if P is not robust against TSO, then there exists a *minimal* TSO robustness violation which is a trace of the form

$$\tau = \tau_1 \cdot \langle k, i, j, \mathrm{issue}(x, v)\rangle \cdot \tau_2 \cdot \lambda \cdot \langle k, i, j, \mathrm{wr}(x, v)\rangle \text{ where}$$

1. the writes of only one thread k are delayed (i.e., not flushed from the write buffer immediately after the write issue) and all the other threads behave like in the SC semantics (i.e., for all the other threads, the write issue is followed immediately by the corresponding write commit),
2. $\langle k, i, j, \mathrm{wr}(x, v)\rangle$ is the first write of thread k which is delayed, i.e., τ_2 doesn't contain any write commit action of thread k,
3. τ_2 contains a transition label λ_1 which conflicts with $\langle k, i, j, \mathrm{issue}(x, v)\rangle$ (necessarily, a read of thread k) such that every label following λ_1 conflicts with its predecessor and the last action of τ_2 conflicts with λ, i.e., there exists a suffix of τ_2 of the form $\lambda_1 \cdot \ldots \cdot \lambda_n$ such that $\langle k, i, j, \mathrm{issue}(x, v)\rangle \prec \lambda_1$, $\lambda_m \prec \lambda_{m+1}$ for every $1 \leq m < n$, and $\lambda_n \prec \lambda$, and
4. the label λ conflicts with the last write commit in τ, i.e., $\lambda \prec \langle k, i, j, \mathrm{wr}(x, v)\rangle$ (which together with the above, implies that τ doesn't have a conflict-preserving permutation admitted by the SC semantics), and
5. the trace τ without the last write commit is a *minimal* trace satisfying the above conditions, i.e., extending $\tau_1 \cdot \langle k, i, j, \mathrm{issue}(x, v)\rangle \cdot \tau_2$ with all the write commits corresponding to write issues of thread k has a conflict-preserving permutation admitted by the SC semantics.

We show that such a trace τ has a conflict-preserving permutation that is admitted by a program with only two threads. For simplicity, assume that k is not the main thread. By the minimality assumption (5) and since P is robust against concurrency, the trace $\tau_1 \cdot \langle k, i, j, \mathrm{issue}(x, v)\rangle \cdot \tau_2 \cdot \tau_3$, where τ_3 contains a write commit action for every write issue of thread k in τ_2, has a conflict-preserving permutation τ' which belongs to $[\![\mathsf{seq}(P), E]\!]_s$. Let τ'' be a trace obtained from τ' by removing again all the write commit actions that were present in τ_3. This trace is admitted by the TSO multi-thread semantics since the values written by the write-commits in τ_3 are not read. Since τ'' preserves the order between conflicting labels in the original trace τ, the trace $\tau'' \cdot \lambda \cdot \langle k, i, j, \mathrm{wr}(x, v)\rangle$ still satisfies properties (1-5). Also, since τ' was a trace of the sequential program $\mathsf{seq}(P)$, all the transitions which are not performed by thread k can be executed by the main thread (they belong to events and invocations which don't interleave). The transitions of thread k, the reads in particular, cannot be executed on the main thread since they can access the values of the writes of thread k which are only issued but not committed. These values are not visible to other threads. Therefore, the trace obtained from $\tau'' \cdot \lambda \cdot \langle k, i, j, \mathrm{wr}(x, v)\rangle$ by substituting every thread id $k' \neq k$ with 0 (the id of the main thread) is admitted by the TSO multi-thread semantics of P. This trace can be simulated by a program $\mathsf{2\text{-}threads}(P)$ obtained from P by replacing every asynchronous invocation $\texttt{async}[\texttt{any}]\ p(\boldsymbol{y})$ with the following code:

```
if ( * & fork )
  async[any] p(y);
  fork = false;
else
  if (PID == 0)
    call p(y);
  else
    async[main] p(y);
```

where `fork` is a global boolean flag which is initially set to `true`. The code above can invoke a procedure $p(\boldsymbol{y})$ either asynchronously, provided that `flag` is still true, or synchronously, or on the main thread if it is invoked from another thread (the test `PID == 0` checks whether the executing thread is the main thread). Note that exactly one invocation of P is asynchronous (since after the first asynchronous invocation, `fork` is turned to `false`) and that this invocation is chosen non-deterministically (the expression `*` returns a randomly-chosen Boolean value). Also, since any thread different from the main thread executes a single invocation (in the multi-thread semantics), any invocation made during the asynchronous invocation is on the main thread (it cannot be transformed to a regular procedure call since it will be executed on the same thread).

The following theorem states the correctness of the construction.

Theorem 2. *Let P be a program with event set E. If P is robust against concurrency, then P is robust against TSO iff* 2-threads(P) *is robust against TSO.*

As a corollary of the results in [5] which state that checking TSO robustness for a program with N threads can be reduced to a reachability problem in a program with N threads under the SC semantics, we get that checking TSO robustness for Boolean programs which are already robust against concurrency is polynomial time.

Corollary 1. *Checking robustness against TSO for a program P with event set E, a fixed number of variables which are all boolean, and a fixed number of procedures, each procedure containing a fixed number of asynchronous invocations, is polynomial time decidable, provided that P is robust against concurrency.*

6 Conclusions

We have presented an approach for checking robustness against TSO for event-driven asynchronous programs (with an unbounded number of threads), that avoids explicit handling of all concurrent executions. This approach reduces TSO robustness checking to a reachability problem in a program with only two threads, provided that the original program is robust against concurrency. Besides yielding an important gain in asymptotic complexity, leading to a polynomial-time TSO robustness checking procedure (for boolean programs), our reduction enables the use of existing safety-verification tools for TSO robustness checking.

Bibliography

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under tso. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–219. Springer, 2012.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for tso. In *European Symposium on Programming Languages and Systems*, pages 308–332. Springer, 2015.

[3] Jade Alglave and Luc Maranget. Stability in weak memory models. In *Computer Aided Verification*, pages 50–66. Springer, 2011.

[4] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. *ACM Sigplan Notices*, 45(1):7–18, 2010.

[5] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 533–553. Springer, 2013.

[6] Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. Verifying robustness of event-driven asynchronous programs against concurrency. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 170–200. Springer, 2017.

[7] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 2011.

[8] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *International Conference on Computer Aided Verification*, pages 107–120. Springer, 2008.

[9] Jabob Burnim, Koushik Sen, and Christos Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 11–25. Springer, 2011.

[10] Michael Emmi, Akash Lal, and Shaz Qadeer. Asynchronous programs with prioritized task-buffers. In *Proc. of the Int. Symp. on Foundations of Software Engineering*, FSE '12, pages 48:1–48:11. ACM, 2012.

[11] Michael Emmi, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. Exploiting synchronization in the analysis of shared-memory asynchronous programs. In *Proc. of the Int. SPIN Symp. on Model Checking of Software*, pages 20–29. ACM, 2014.

[12] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. *SIGPLAN Not.*, 46(1):411–422, January 2011.

[13] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Notices*, volume 46, pages 187–198. ACM, 2011.

[14] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.

[15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[16] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*, volume 6183, pages 478–503. Springer, 2010.

[17] Burcu Kulahcioglu Ozkan, Michael Emmi, and Serdar Tasiran. Systematic asynchrony bug exploration for android apps. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA*, pages 455–461, 2015.

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