Parameterized Verification under TSO with Data Types

Parosh Aziz Abdulla[1], Mohamad Faouzi Atig[1], Florian Furbach[1](✉), Adwait A. Godbole[3], Yacoub G. Hendi[1], Shankara N. Krishna[2], and Stephan Spengler[1]

[1] Uppsala University, Uppsala, Sweden
florian.furbach@it.uu.se
[2] Indian Institute of Technology Bombay, Mumbai, India
[3] UC Berkeley, Berkeley, USA

We consider parameterized verification of systems executing according to the total store ordering (TSO) semantics. The processes manipulate abstract data types over potentially infinite domains. We present a framework that translates the reachability problem for such systems to the reachability problem for register machines enriched with the given abstract data type. We use the translation to obtain tight complexity bounds for TSO-based parameterized verification over several abstract data types, such as push-down automata, ordered multi push-down automata, one-counter nets, one-counter automata, and Petri nets. We apply the framework to get complexity bounds for higher order stack and counter variants as well.

1 Introduction

A *parameterized system* consists of a fixed but arbitrary number of identical processes that execute in parallel. The goal of *parameterized verification* is to prove the correctness of the system regardless of the number of processes. Examples for such systems are sensor networks, leader election protocols, and mutual exclusion protocols. The topic has been the subject of intensive research for more than three decades (see e.g. [10,32,13,6]), and it is the subject of one chapter of the Handbook of Model Checking [8]. Research on parameterized verification has been mostly conducted under the premise that (i) the processes run according to the classical Sequential Consistency (SC) semantics, and (ii) the processes are finite-state machines.

Under SC, the processes operate on a set of shared variables through which they communicate *atomically*, i.e., read and write operations take effect immediately. In particular, a write operation is visible to all the processes as soon as the writing process carries out its operation. Therefore, the processes always maintain a uniform view of the shared memory: they all see the latest value written on any given variable, hence we can interpret program runs as interleavings of sequential process executions. Although SC has been immensely popular as an intuitive way of understanding the behaviours of concurrent processes, it is not realistic to assume computation platforms guarantee SC anymore. The reason is that, due to hardware and compiler optimizations, most modern platforms

© The Author(s) 2023
S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 588–606, 2023.
https://doi.org/10.1007/978-3-031-30823-9_30

allow more relaxed program behaviours than those permitted under SC, leading to so-called *weak memory models*. Weakly consistent platforms are found at all levels of system design such as multiprocessor architectures (e.g., [48,47]), Cache protocols (e.g., [46,21]), language level concurrency (e.g., [41]), and distributed data stores (e.g., [17]). Therefore, in recent years, research on the parameterized verification of concurrent programs under weak memory models have started to become popular. Notable examples are the cases of the TSO semantics [4] and the Release-Acquire semantics of C11 [39].

In a parallel development, several works have extended the basic model of parameterized systems (under the SC semantics) by considering processes that are infinite-state systems. The most dominant such class has been the case where the individual processes are variants of push-down automata [36,33,28,28,40,42,30]

Parameterized verification is difficult, even under the original assumption of both SC and finite-state processes as we still need to handle an infinite state space. The extension to weakly consistent systems is even more complex due to the intricate extra process behaviours. Almost all weak memory models induce infinite state spaces even without parameterization and even when the program itself is finite-state. Therefore, performing parameterized verification under weak consistency requires handling a state space that is infinite in two dimensions; one due to parameterization and one due to the weak memory model. The same applies to the extension of parameterized verification under SC where the processes are infinite-state: in addition to infiniteness due to parameterization, we have a second source of infinity due to the infiniteness of the processes.

In this paper, we combine the above two extensions. We study parameterized verification of programs under the TSO semantics, where the processes use infinite data structures such as stacks and counters. The framework is uniform in that the manipulation can be described using an abstract data type.

We revisit the pivot abstraction technique presented in [4]. As a first contribution, we show that we can capture pivot abstraction precisely, using a class of register machines in which the registers assume values over a finite domain. We show that, for any given abstract data type A, we can reduce, in polynomial time, the parameterized verification problem under TSO and A to the reachability problem for register machines manipulating A. Furthermore, we show that the reduction also holds in the other direction: the reachability problem for register machines over A is polynomial-time reducible to the parameterized verification problem under TSO for A. In particular, the model abstracts away the semantics of TSO (in fact, it abstracts away concurrency altogether) since we are dealing with a single register machine.

We summarize the contributions of the paper as follows:

- We present a register abstraction scheme that captures the behaviour of parameterized systems under the TSO semantics.
- We translate parameterized verification under the TSO semantics when the processes manipulate an ADT A, to the reachability problem for register machines operating over A.
- We instantiate the framework for deciding the complexity of parameterized verification under TSO for different abstract data types. In particular we

show the problem is PSPACE-complete when A is a one-counter, EXPTIME-complete if A is a stack, 2-ETIME-complete if A is an ordered multi stack, and EXPSPACE-complete if A is a Petri net. We obtain further complexity bounds for higher order counter and stacks.

Related Work There has been an extensive research effort on parameterized verification since the 1980s (see [13,8] for recent surveys of the field). Early works showed the undecidability of the general problem (even assuming finite-state processes) [10], and hence the emphasis has been on finding useful special cases. Such cases are characterized by three aspects, namely the system topology (unordered, arrays, trees, graphs, rings, etc.), the allowed communication patterns (shared memory, Rendez-vous, broadcast, lossy channels, etc.), and the process types (anonymous, with IDs, with priorities, etc.) [27,20,31,24,23,43].

Another line of research to counter undecidability are over-approximations based on regular model checking [38,14,16,1], monotonic abstraction [5], and symmetry reduction [37,22,7].

A seminal work in the area is the paper by German and Sistla [32]. The authors consider the verification of systems consisting of an arbitrary number of finite-state processes interacting through Rendez-Vous communication. The paper shows that the model checking problem is EXPSPACE-complete. In a series of more recent papers, parameterized verification has been considered in the case where the individual processes are push-down automata. [36,33,28,40,42,30].All the above works assume the SC semantics.

Due to the relevance of weak memory models in parameterized verification, papers on the topic have started to appear in the last two years. The paper [4] considers parameterized verification of programs running under TSO, and shows that the reachability problem is PSPACE-complete. However, the paper assumes that the processes are finite-state and, in particular, the processes do not manipulate unbounded data domains. The model of the paper corresponds to the particular case of our framework where we take the abstract data type to be empty. In this case our framework also implies PSPACE-completeness.

The paper [39] shows PSPACE-completeness when the underlying semantics is the Release-Acquire fragment of C11. The latter semantics gives rise to different semantics compared to TSO. The paper also considers finite-state processes.

The paper [2] considers parameterized verification of programs running under TSO. However, the paper applies the framework of well-structured systems where the buffers of the processes are modeled as lossy channels, and hence the complexity of the algorithm is non-primitive recursive. In particular, the paper does not give any complexity bounds for the reachability problem (or any other verification problems). Conchon et al. [19] address the parameterized verification of programs under TSO as well. They make use of Model Checker Modulo Theories, no decidability or complexity results are given. The paper [15] considers checking the robustness property against SC for parameterized systems running under the TSO semantics. However, the robustness problem is entirely different from reachability and the techniques and results developed in the paper cannot

be applied in our setting. The paper shows that the problem is ExpSpace-hard. All these works assume finite-state processes.

In contrast to all the above works, the current paper is the first paper that studies decidability and complexity of parameterized verification under the TSO semantics when the individual processes are infinite-state.

2 Preliminaries

We denote a function f between sets A and B by $f : A \longrightarrow B$. We write $f[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for all $x \neq a$.

For a finite set A, we use $|A|$ to refer to the size of A. We also use A^* to denote the set of words over A including the empty word ϵ. For a word $w \in A^*$, we use $|w|$ to refer to the length of w. We say a word w is *differentiated* if all symbols in w are pairwise different. The set A^{diff} is the set of all differentiated words over the set A. Finally, for a differentiated word w, we define $\mathrm{pos}(w)(a)$ as the unique position of the letter a in w.

A *labelled transition system* is a tuple $\langle \mathsf{C}, \mathsf{C}_{\mathsf{init}}, \mathsf{Labs}, \longrightarrow \rangle$, where C is the set of configurations, $\mathsf{C}_{\mathsf{init}} \subseteq \mathsf{C}$ is the set of initial configurations, Labs is a finite set of labels and $\longrightarrow \subseteq \mathsf{C} \times \mathsf{Labs} \times \mathsf{C}$ is the transition relation over the set of configurations. For a transition $\langle \mathsf{c}_1, \mathsf{lab}, \mathsf{c}_2 \rangle \in \longrightarrow$, we usually write $\mathsf{c}_1 \xrightarrow{\mathsf{lab}} \mathsf{c}_2$ instead. We use $\mathsf{c}_1 \longrightarrow \mathsf{c}_2$ to denote that $\mathsf{c}_1 \xrightarrow{\mathsf{lab}} \mathsf{c}_2$ for some $\mathsf{lab} \in \mathsf{Labs}$. Furthermore, we write $\xrightarrow{*}$ to denote the transitive reflexive closure over $\longrightarrow$, and if $\mathsf{c}_1 \xrightarrow{*} \mathsf{c}_2$ then we say c_2 is *reachable from* c_1. If $\mathsf{c}_1 \in \mathsf{C}_{\mathsf{init}}$, then we just say that c_2 is *reachable*. A *run* ρ is an alternating sequence of configurations and labels and is expressed as follows: $\mathsf{c}_0 \xrightarrow{\mathsf{lab}_1} \mathsf{c}_1 \xrightarrow{\mathsf{lab}_2} \mathsf{c}_2 \ldots \mathsf{c}_{n-1} \xrightarrow{\mathsf{lab}_n} \mathsf{c}_n$. Given ρ, we write $\mathsf{c}_0 \xrightarrow{n} \mathsf{c}_n$ meaning that c_n is reachable from c_0 by n steps, and we write $\mathsf{c}_0 \xrightarrow{\rho} \mathsf{c}_n$ meaning that c_n is reachable from c_0 through the run ρ.

3 Abstract Data Types (ADT)

In this section, we introduce the notion of abstract data types (ADTs) which will be used extensively in the paper. An ADT is a labelled transition system $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$. Intuitively, this describes the behaviour of some data type such as a stack, or a counter. Vals is the set of configurations of A. It describes the possible values the data type can assume. The initial configuration is $\mathsf{val}_{\mathsf{init}} \in \mathsf{Vals}$. The set of labels Ops represents the operations that can be executed on the data type and the transition relation $\longrightarrow_{\mathsf{A}} \in \mathsf{Vals} \times \mathsf{Ops} \times \mathsf{Vals}$ describes the semantics of these operations. Below, we give some concrete examples of abstract data types.

Example 1 (Counter). We define a counter, denoted by the ADT Ct, as follows. The set of configurations $\mathsf{Vals}^{\mathrm{Ct}} = \mathbb{N}$ are the natural numbers. The initial value, denoted by $\mathsf{val}^{\mathrm{Ct}}_{\mathsf{init}}$, is 0. The set of operations is $\mathsf{Ops}^{\mathrm{Ct}} = \{\mathtt{inc}, \mathtt{dec}, \mathtt{isZero}\}$. The transition relation $\longrightarrow_{\mathrm{Ct}}$ is as follows: The operations `inc` and `dec` increase or

decrease the value of the counter by one, respectively. The latter operation is only enabled if the value of the counter is non-zero, otherwise it blocks. Finally, the transition `isZero` checks that the value of the counter is zero, i.e. it is only enabled if that condition is true.

Example 2 (Weak Counter). A weak counter differs from a counter in that it cannot be checked for zero. The ADT wCt representing a weak counter is defined as in Example 1, except the operations of wCt are reduced to $\mathsf{Ops}^{\text{wCt}} = \{\texttt{inc}, \texttt{dec}\}$.

Example 3 (Stack). Let Γ be a finite set representing the stack alphabet. A *stack* $\text{St} = \langle \mathsf{Vals}^{\text{St}}, \{\mathsf{val}^{\text{St}}_{\mathsf{init}}\}, \mathsf{Ops}^{\text{St}}, \rightarrow_{\text{St}} \rangle$ on Γ is defined as follows. The configurations of St are $\mathsf{Vals}^{\text{St}} = \Gamma^*$ and the initial configuration is the empty stack $\mathsf{val}^{\text{St}}_{\mathsf{init}} = \varepsilon$. The set of operations is $\mathsf{Ops}^{\text{St}} = \{\texttt{pop}(\gamma), \texttt{push}(\gamma), \texttt{isEmpty} \mid \gamma \in \Gamma\}$. The transition relation is as follows. For every word $w \in \Gamma^*$ and every symbol $\gamma \in \Gamma$, $\texttt{push}(\gamma)$ adds the symbol γ to the top of the stack. Similiarly, the $\texttt{pop}(\gamma)$ operation removes the topmost symbol from the stack. It is only enabled if the topmost symbol on the stack. The `isEmpty` operation does not change the stack, but can only be performed if the stack is the empty word ε.

Example 4 (Petri Nets). Given a Petri net[44], We can define a corresponding ADT Petri that models its semantics. The values are the markings, the operations are the Petri net transitions and the transition relation is given by the input and output vectors of the Petri net transitions.

Higher Order ADTs We extend the ADT St to higher order stacks referred to as n-St. This is done recursively[18,25]. The formal definition is in the full version of our paper [3]. A value of a level n higher order stack n-St is a stack of level $n-1$ stacks. For level 1, it is the standard stack St. The operations for level n are $\mathsf{Ops}^{n\text{-St}} = \{\texttt{pop}(\gamma), \texttt{push}(\gamma), \texttt{pop}_k, \texttt{push}_k, \mid \gamma \in \Gamma, 2 \leq k \leq n\}$. The operations $\texttt{pop}(\gamma)$ and $\texttt{push}(\gamma)$ are recursively applied to the top element in the stack (which consists of a stack that is one level lower) until the level of the top element is 1. Here, they have the standard stack behaviour. Operations $\texttt{pop}_k$ and $\texttt{push}_k$ are recursively applied to the top element until the level of the element is k. Then, a copy of this level k stack is pushed on top of the original.

Since a counter can be seen as a stack with an alphabet of size 1 (and a bottom element $\perp$), we can extend definitions of wCt and Ct to n-wCt and n-Ct in the same way. We add operations $\texttt{inc}_k, \texttt{dec}_k$. All operations are recursively applied to the top counter. For $\texttt{inc}, \texttt{dec}, \texttt{isZero}$, we use standard behaviour once the level is 1. For $\texttt{inc}_k, \texttt{dec}_k$, we copy/remove the top element once the level is k.

Example 5 (Ordered Multi Stack). We extend the stack to a numbered list of n many stacks n-OMSt [12]. A value of n-OMSt consists of list of stacks $\mathsf{val}^{\text{St}}_1 \ldots \mathsf{val}^{\text{St}}_n$. An operation $\mathsf{Ops}^{n\text{-OMSt}} = \{\texttt{isZero}_i, \texttt{pop}_i(\gamma), \texttt{push}_i(\gamma), \mid \gamma \in \Gamma, i \leq n\}$ works on stack number i in the standard way. One additional condition is that the stacks have to be ordered, meaning an operation $\texttt{pop}_i(\gamma)$ is only enabled if the stacks $1 \ldots i-1$ are empty.

4 TSO with an Abstract Data Type : TSO(A)

In this section, we introduce concurrent programs running under $\mathsf{TSO}(\mathsf{A})$ for an ADT $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$. These programs consist of concurrent processes where the communication between processes is performed using shared memory under the TSO semantics. In addition, each process maintains a local variable of type A.

Syntax of $\mathsf{TSO}(\mathsf{A})$. Let Dom be a finite data domain and Vars be a finite set of shared variables over Dom. Let $\mathsf{d}_{\mathsf{init}} \in \mathsf{Dom}$ be the initial value of the variables. We define the *instruction set* of $\mathsf{TSO}(\mathsf{A})$ as $\mathsf{Instrs} = \{\mathtt{rd}(\mathsf{x},\mathsf{d}), \mathtt{wr}(\mathsf{x},\mathsf{d}) \mid \mathsf{x} \in \mathsf{Vars}, \mathsf{d} \in \mathsf{Dom}\} \cup \{\mathtt{skip}, \mathtt{mf}\}$, which are called *read*, *write*, *skip* and *memory fence*, respectively.

A process is represented by a finite state transition system. It is given by the tuple $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$, where Q is a finite set of states, $\mathsf{q}_{\mathsf{init}} \in \mathsf{Q}$ is the initial state, and $\delta \subseteq \mathsf{Q} \times (\mathsf{Instrs} \cup \mathsf{Ops}) \times \mathsf{Q}$ is the transition relation. We call this tuple the *description* of the process. A concurrent program is a tuple of processes $\mathcal{P} = \langle \mathsf{Proc}_\iota \rangle_{\iota \in \mathcal{I}}$, where $\mathcal{I}$ is some finite set of process identifiers. For each $\iota \in \mathcal{I}$ we have $\mathsf{Proc}^\iota = \langle \mathsf{Q}^\iota, \mathsf{q}^\iota_{\mathsf{init}}, \delta^\iota \rangle$.

Semantics of $\mathsf{TSO}(\mathsf{A})$. We describe the semantics of a program $\mathcal{P}$ running under $\mathsf{TSO}(\mathsf{A})$ by a labelled transition system $\mathcal{T}_{\mathcal{P}} = \langle \mathsf{C}^{\mathcal{P}}, \mathsf{C}^{\mathcal{P}}_{\mathsf{init}}, \mathsf{Labs}^{\mathcal{P}}, \longrightarrow_{\mathcal{P}} \rangle$. The formal definition is given in [3]. Under $\mathsf{TSO}(\mathsf{A})$, there is an unbounded FIFO buffer of writes between each process and the memory. A configuration $\mathsf{c} \in \mathsf{C}^{\mathcal{P}}$ of the system consists of the value of each variable in the shared memory as well as for each process: its local state, its value of the ADT, and the content of the corresponding write buffer.

The labelled transitions $\longrightarrow_{\mathcal{P}}$ are as follows: A local `skip` transition simply updates the state of the corresponding process. An ADT operation additionally updates the ADT value according to ADT behaviour $\longrightarrow_{\mathsf{A}}$. When a process executes a write instruction, the operation is enqueued as a *pending write message* into its buffer. A message msg is an assignment of the form $\mathsf{msg} = \langle \mathsf{x}, \mathsf{d} \rangle$, where $\mathsf{x} \in \mathsf{Vars}$ and $\mathsf{d} \in \mathsf{Dom}$. We denote the set of all messages by $\mathsf{Msgs} = \mathsf{Vars} \times \mathsf{Dom}$. The buffer content for a process is given as a word over Msgs. The messages inside each buffer are moved non-deterministically to the main memory in a FIFO manner. Once a message reaches the memory, it becomes visible to all the other processes. When executing a read instruction on a variable $\mathsf{x} \in \mathsf{Vars}$, the process first checks its buffer for pending write messages on x. If the buffer contains such a message, then it reads the value of the most recent one. If the buffer contains no write messages on x, then the process fetches the value of x from the memory. The *initial configuration* is $\mathsf{c}^{\mathcal{P}}_{\mathsf{init}}$, where each process is in its initial state, each ADT holds its initial value, each store buffer is empty and the memory holds the initial values of all variables. Note that since FIFO buffer is unbounded, this is an infinite state transition system, even for finite ADT.

A sequence of transitions $\mathsf{c}_0 \xrightarrow{\mathsf{lab}_1}_{\mathcal{P}} \mathsf{c}_1 \xrightarrow{\mathsf{lab}_2}_{\mathcal{P}} \mathsf{c}_2 \ldots \mathsf{c}_{n-1} \xrightarrow{\mathsf{lab}_n}_{\mathcal{P}} \mathsf{c}_n$ where $\mathsf{c}_0 = \mathsf{c}^{\mathcal{P}}_{\mathsf{init}}$ is the initial configuration and $\mathsf{lab}_i \in \mathsf{Labs}^{\mathcal{P}}$ is called a run in the $\mathsf{TSO}(\mathsf{A})$ transition system. If there is a run ending in a configuration with state $\mathsf{q}_{\mathsf{final}}$, then we say $\mathsf{q}_{\mathsf{final}}$ is reachable by Proc under $\mathsf{TSO}(\mathsf{A})$.

5 Parameterized Reachability in TSO(A)

In this section, we consider the parameterized TSO setting which allows for an a priori unbounded number of processes with the same process description. We begin by formally introducing the parameterized state reachability problem, and then develop a generic construction that allows us to represent the TSO semantics (except for the ADT) in a finite manner.

The Parameterized State Reachability Problem Intuitively, parameterization allows for an arbitrary number of identical processes. The parameterized state reachability problem for TSO(A) called TSO(A)-P-Reach identifies a family of (standard) reachability problem instances. We want to determine whether we have reachability in some member of the family. We now introduce this formally.

For a given process description Proc, we consider the program instance, $\mathcal{P}^n_{\mathsf{Proc}}$ parameterized by a natural number n as follows. For $\mathcal{I} = \{1, \ldots, n\}$, let $\mathcal{P}^n_{\mathsf{Proc}} = \langle \mathsf{Proc}_1, \ldots, \mathsf{Proc}_n \rangle$ with $\mathsf{Proc}_\iota = \mathsf{Proc}$ for all $\iota \in \mathcal{I}$. That is, the n^{th} *slice* of the parameterized family of programs contains n processes, all with identical descriptions Proc. We require that all processes maintain copies of the ADT A.

TSO(A)-P-Reach:
Given: A process $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$, an ADT A, and a state $\mathsf{q}_{\mathsf{final}} \in \mathsf{Q}$,
Decide: Is there a $n \in \mathbb{N}$ s.t. $\mathsf{q}_{\mathsf{final}}$ is reachable by $\mathcal{P}^n_{\mathsf{Proc}}$ under TSO(A)?

When talking about a certain family of ADTs, e.g. the family of petri nets, we write TSO(Petri)-P-Reach and mean the restriction of TSO(A)-P-Reach to petri nets, i.e. to instances where A is a petri net.

The main difference between the non-parameterized case and the parameterized case of the problem is that in the first case the index set $\mathcal{I}$ is *a priori* fixed, while in the second case it can be *arbitrary*. This results in $\mathsf{C}^{\mathcal{P}}_{\mathsf{init}}$ being a singleton in the non-parameterized case while it becomes infinite (one initial state for each n-slice) in the parameterized case.

We determine upper and lower bounds for the complexity of the state reachability problem. The challenge of solving this problem varies with the ADT. This problem for plain TSO without an ADT has been studied in [4]. They showed that the problem can be decided in PSpace and is in fact PSpace-complete. The result is based on an abstraction technique called the *pivot* semantics. The pivot semantics is *exact* in the sense that a state q is reachable under parameterized TSO if and only if it is reachable under the pivot semantics.

We show that the dynamics underlying the pivot abstraction can be generalized to our model with ADT. We show that the pivot abstraction can be extended to obtain a register machine. We use this construction to give a general characterization of TSO(A)-P-Reach. First, we recall the pivot abstraction.
The Pivot Abstraction [4]. For a set of variables Vars and data domain Dom, processes generate pending write messages from the set $\mathsf{Msgs} = \mathsf{Vars} \times \mathsf{Dom}$ by

executing wr instructions. This set has size $|\mathsf{Vars}| \cdot |\mathsf{Dom}|$ and hence at most as many distinct (variable, value) pairs can be produced in any run. For a run ρ of the program, for each message $\mathsf{msg} = \langle \mathsf{x}, \mathsf{d} \rangle \in \mathsf{Msgs}$ we can define the first point along ρ at which some write on variable x with value d is propagated to the memory. The pivot abstraction identifies these points as *pivot* points $\mathrm{pvt}(\mathsf{msg})$, for each distinct message in Msgs. For a write message msg under ρ, the pivot point $\mathrm{pvt}(\mathsf{msg})$ is the first point of propagation of msg to the memory under ρ.

The core observation is that if at some point in ρ, a process Proc_ι propagates a message $\mathsf{msg} = \langle \mathsf{x}, \mathsf{d} \rangle$ from its buffer to the memory, then after that point, the value d will always be available to read on variable x from the shared memory. Technically, this follows from parameterization. There are arbitrarily many processes executing identical descriptions. This means transitions of the original process Proc_ι can be mimicked by a *clone* process $\mathsf{Proc}_{\iota'}$ identical to Proc_ι. Hence, $\mathsf{Proc}_{\iota'}$ can replicate the execution of Proc_ι right up to the point where the message msg is the oldest message in its buffer. Then a single propagate step updates the value of x in the shared memory to d. There can be arbitrarily many such clones and the propagate step can happen at any time. It follows that beyond the $\mathrm{pvt}(\mathsf{msg})$ point in ρ, the value d can always be read from x.

For distinct messages from Msgs, we can order the pivot points corresponding to these messages according to the order in which they appear in ρ. This gives us a *first update sequence*, denoted by ω. No two messages in ω are the same; the set of such sequences is the set of differentiated words $\mathsf{Msgs}_{\mathsf{diff}}$. A message $\mathsf{msg} \in \mathsf{Msgs}$ in ω has the *rank* k if it is the k-th pivot point in ω.

Providers. The pivot abstraction simulates a run ρ under the TSO semantics by running abstract processes called *providers* in a sequential manner. For $1 \leq k \leq |\omega| + 1$, the k-provider simulates the process that generates the write of the rank k message $\langle \mathsf{x}, \mathsf{d} \rangle$ corresponding to the k-pivot in ρ. The k-provider completes its task when it has simulated this process until the point it generates $\langle \mathsf{x}, \mathsf{d} \rangle$. At this point, it invokes the $(k+1)$-provider. With this background, we now develop the formal pivot semantics for parameterized $\mathsf{TSO}(\mathsf{A})$.

Formal Pivot semantics for Parameterized $\mathsf{TSO}(\mathsf{A})$. We define the formal operational semantics of the pivot abstraction as a labelled transition system. Given a process description $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$ and ADT $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$, a configuration of the pivot transition system represents the *view* of a provider when simulating a run of the program. A view $\mathsf{v} = \langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle$ is defined as follows. The process state is given by $\mathsf{q} \in \mathsf{Q}$. The value of the provider's ADT A is $\mathsf{val} \in \mathsf{Vals}$. The function $\mathsf{Lw} : \mathsf{Vars} \longrightarrow \mathsf{Dom} \cup \{\oslash\}$ gives for each $\mathsf{x} \in \mathsf{Vars}$, the value of the latest (i.e., most recent) write the provider has performed on x. If no such instruction exists (the process has made no writes to x) then $\mathsf{Lw}(\mathsf{x}) = \oslash$. Note that Lw abstracts the buffer in terms of read-own-write operations since the process can only read from the most recent pending write in its buffer on each variable (if it exists). We define $\mathsf{Lw}_\oslash$ such that $\mathsf{Lw}_\oslash(\mathsf{x}) = \oslash$ for all $\mathsf{x} \in \mathsf{Vars}$. The first update sequence of pivot messages is $\omega \in \mathsf{Msgs}_{\mathsf{diff}}$. It is unchanged by transitions and remains constant throughout the pivot run.

The *external pointer*, $\phi_E \in \{0, 1, \ldots, |\omega|\}$ helps the provider keep track of which messages from ω it has observed. These messages have been propagated by other processes. The external pointer is used to identify which variables are still holding their initial values in the memory. If the provider observes an external write on a variable x (by accessing the memory), then this write has overwritten the initial value of x in the memory. The *local pointer* $\phi_L : \mathsf{Vars} \longrightarrow \{0, 1, \ldots, |\omega|\}$ is a set of pointers, one for each variable $\mathsf{x} \in \mathsf{Vars}$. The function $\phi_L(\mathsf{x})$ gives the highest ranked write operation the provider itself has performed (on any variable) before it performed the latest write on x. The local pointer is necessary to know which variables lose their initial values when we need to empty the buffer. In other words, the local pointer abstracts the buffer in terms of update operations. We define $\phi_L^{\max} := \max\{\phi_L(\mathsf{x}) \mid \mathsf{x} \in \mathsf{Vars}\}$ as the highest value of a local pointer and ϕ_L^0 such that $\phi_L^0(\mathsf{x}) = 0$ for all variables $\mathsf{x} \in \mathsf{Vars}$, i.e., the pointers are all in the leftmost position. The *progress pointer* $\phi_P \in \{1, 2, \ldots, |\omega| + 1\}$ gives the rank of the process the current provider is simulating.

$$\text{skip}\ \frac{\langle \mathsf{q}, \mathtt{skip}, \mathsf{q}' \rangle \in \delta}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{skip}}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

$$\text{write(1)}\ \frac{\langle \mathsf{q}, \mathtt{wr}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) < \phi_P, \phi'_L = \phi_L[\mathsf{x} \leftarrow \max(\mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle), \phi_L^{\max})]}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{wr}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}[\mathsf{x} \leftarrow \mathsf{d}], \omega, \phi_E, \phi'_L, \phi_P \rangle}$$

$$\text{write(2)}\ \frac{\langle \mathsf{q}, \mathtt{wrx}, \mathsf{d}, \mathsf{q}' \rangle \in \delta, \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) = \phi_P}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{wr}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \mathsf{v}_{\mathsf{init}}(\omega, \phi_P + 1)}$$

$$\text{read(1)}\ \frac{\langle \mathsf{q}, \mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathsf{Lw}(\mathsf{x}) = \mathsf{d}}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

$$\text{read(2)}\ \frac{\langle \mathsf{q}, \mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathsf{d} = \mathsf{init}(\mathsf{x}), \mathsf{Lw}(\mathsf{x}) = \bot, \mathrm{pos}(\omega)(\mathsf{x}) > \phi_E}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

$$\text{read(3)}\ \frac{\langle \mathsf{q}, \mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) < \phi_P, \phi'_E = \max(\phi_E, \phi_L(\mathsf{x}), \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle))}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi'_E, \phi_L, \phi_P \rangle}$$

$$\text{memory-fence}\ \frac{\langle \mathsf{q}, \mathtt{mf}, \mathsf{q}' \rangle \in \delta, \phi'_E = \max(\phi_E, \phi_L^{\max})}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{mf}}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi'_E, \phi_L, \phi_P \rangle}$$

$$\text{data-operation}\ \frac{\langle \mathsf{q}, \mathsf{op}, \mathsf{q}' \rangle \in \delta, \mathsf{op} \in \mathsf{Ops}, \mathsf{val} \xrightarrow{\mathsf{op}}_{\mathsf{A}} \mathsf{val}'}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathsf{op}}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}', \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

Fig. 1: The transition relation of the pivot semantics for a process Proc.

Given an update sequence $\omega \in \mathsf{Msgs}^{\mathsf{diff}}$ and $1 \leq k \leq |\omega| + 1$, we define *the initial view induced by* ω *and* k denoted by $\mathsf{v}_{\mathsf{init}}(\omega, k)$, as the view $\langle \mathsf{q}^{\mathsf{init}}, \mathsf{val}_{\mathsf{init}}, \mathsf{Lw}_\bot, \omega, 0, \phi_L^0, k \rangle$. For a given ω, the k-provider starts with $\mathsf{v}_{\mathsf{init}}(\omega, k)$: $\mathsf{Lw}_\bot$ and ϕ_L^0 imply that the simulated process has not performed any writes and $\phi_E = 0$ means that it has not read/updated from/to the memory.

We define the labeled transition relation $\longrightarrow_{\mathsf{pvt}}$ on the set of views by the inference rules given in Figure 1. The set of labels is $\mathsf{Instrs} \cup \mathsf{Ops}$. We describe the inference rules briefly. The `skip` rule only changes the local state of the process. There are two inference rules, write(1) and write(2), to describe the execution of a write operation $\mathtt{wr}(\mathsf{x}, \mathsf{d})$. The rule write(1) describes the situation when the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ is strictly smaller than the progress pointer ϕ_P. In this case, we update both Lw and ϕ_L. The rule write(2) describes the situation when the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ equals the progress pointer. This means that the provider has provided the message $\langle \mathsf{x}, \mathsf{d} \rangle$ with rank ϕ_P. Hence it has completed its mission, and initiates the next provider by transitioning to $\mathsf{v}_{\mathsf{init}}(\omega, \phi_P + 1)$.

There are three inference rules that describe a read operation $\mathtt{rd}(\mathsf{x}, \mathsf{d})$. The rule read(1) describes when the last written value to x by the provider is d, $\mathsf{Lw}(\mathsf{x}) = \mathsf{d}$. In this case, the provider simply reads from its local buffer. The rule read(2) describes the read of an initial value. It ensures that the read is possible by checking that no write operation on x is executed by the provider ($\mathsf{Lw}(\mathsf{x}) = \bot$), and by checking that the initial value of the variable has not been overwritten in the memory. This is achieved by checking if the position of $\langle \mathsf{x}, \mathsf{d} \rangle$ in ω, i.e. $\mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle)$, is strictly larger than ϕ_E. The rule read(3) describes when the simulated process reads from the memory. It checks that the message $\langle \mathsf{x}, \mathsf{d} \rangle$ has been generated by some previous provider ($\mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) < \phi_P$), and then it updates the external pointer to $\max(\phi_E, \phi_L(\mathsf{x}), \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle))$. The memory fence rule describes when the simulated process does a fence action. The rule updates the external pointer to $\max(\phi_E, \phi_L^{\max})$. Finally, the data-operation rule describes when the simulated process does an ADT operation.

The set of initial views is $\mathsf{V}_{\mathsf{init}} = \{\mathsf{v}_{\mathsf{init}}(\omega, 1) \mid \omega \in \mathsf{Msgs}^{\mathsf{diff}}\}$. This is the set of initial views of the 1-provider and it is finite because $\mathsf{Msgs}^{\mathsf{diff}}$ is finite, unlike the set of initial configurations $\mathsf{C}_{\mathsf{init}}$ in the parameterized case under TSO.

6 Register Machines

Our goal is to design a general method to determine the decidability and complexity of $\mathsf{TSO}(\mathsf{A})\text{-}\mathtt{P}\text{-}\mathtt{Reach}$ depending on A. We examine the pivot abstraction introduced in the previous chapter. A view $\mathsf{v} = \langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle$ of the pivot transition system, can be partitioned into the following two components: (1) $\mathsf{q}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P$ which contains the local state and also effectively abstracts the unbounded FIFO buffers and shared memory of the TSO system and (2) val which captures the value of the ADT. The first part is finite since each component takes finitely many values. We call this the *book-keeping* state since it keeps track of the progress of the core TSO system. However, the ADT part can be infinite, depending upon the abstract data type.

We will use a register machine in order to represent the book-keeping state in a finite way using states and registers. On the other hand, we will keep the ADT component general and only later instantiate it to some interesting cases.

A *register machine* is a finite state automaton that has access to a finite set of *registers*, each holding a natural number. The register machine can execute two

operations on a register, it can write a given value or it can read a given value. A read is blocking if the given value is not in the register. We differ from most definitions of register machines in two significant ways: Since we only require a finite domain to model $\mathsf{TSO}(\mathsf{A})$ semantics, the values of the registers are bound from above by an $N \in \mathbb{N}$. This makes the register assignments finite whereas most definitions allow for an unbounded domain. Further, our register machine is augmented with an ADT.

Given an ADT $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$, let Regs be a finite set of registers and $\mathsf{Dom} = \{0, \ldots, N\}$ their domain. We define the set of actions $\mathsf{Acts} = \{\mathtt{SKP}, \mathtt{WRITE}(\mathsf{r}, \mathsf{d}), \mathtt{READ}(\mathsf{r}, \mathsf{d}) \mid \mathsf{r} \in \mathsf{Regs}, \mathsf{d} \in \mathsf{Dom}\}$. A register machine is then defined as a tuple $\mathcal{R}(\mathsf{A}) = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$, where Q is a finite set of states, $\mathsf{q}_{\mathsf{init}} \in \mathsf{Q}$ is the initial state and $\delta \subseteq \mathsf{Q} \times (\mathsf{Acts} \cup \mathsf{Ops}) \times \mathsf{Q}$ is the transition relation.

The semantics of the register machine are given in terms of a transition system. The set of configurations is $\mathsf{Q} \times \mathsf{Dom}^{\mathsf{Regs}} \times \mathsf{Vals}$. A configuration consists of a state, a register assignment $\mathsf{Regs} \longrightarrow \mathsf{Dom}$ and a value of A. The initial configuration is $\langle \mathsf{q}_{\mathsf{init}}, 0^{\mathsf{Regs}}, \mathsf{val}_{\mathsf{init}} \rangle$, where all registers contain the value 0.

The transition relation $\longrightarrow$ is described in the following. $\mathtt{SKP}$ only changes the local state, not the registers or the ADT value. $\mathtt{WRITE}(\mathsf{r}, \mathsf{d})$ sets the value of the register r to d. $\mathtt{READ}(\mathsf{r}, \mathsf{d})$ is only enabled if the value of r is d, it does not change the value. The operations in Ops work as usually, they do not change any register. We define the state reachability problem for register machines as $\mathcal{R}(\mathsf{A})$-$\mathtt{Reach}$ in the usual way. A state $\mathsf{q}_{\mathsf{final}} \in \mathsf{Q}$ is reachable if there is a run of the transition system defined by the semantics of $\mathcal{R}(\mathsf{A})$ that starts in the initial configuration and ends in a configuration with state $\mathsf{q}_{\mathsf{final}}$.

6.1 Simulating Pivot Abstraction by Register Machines

In this section we will show how to simulate the pivot abstraction by a register machine. The idea is to save the book-keeping state (except for the local state) in the registers. Given a process description $\mathsf{Proc} = \langle \mathsf{Q}^{\mathsf{Proc}}, \mathsf{q}_{\mathsf{init}}^{\mathsf{Proc}}, \delta^{\mathsf{Proc}} \rangle$ for an ADT A, we construct a register machine $\mathcal{R}(\mathsf{A}) = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$ that simulates the pivot semantics as follows. The set of registers is

$$\mathsf{Regs} := \{\mathsf{Lw}(\mathsf{x}), \mathsf{rk}_{\mathsf{Vars}}(\mathsf{x}), \mathsf{rk}_{\mathsf{Msgs}}(\mathsf{msg}), \phi_E, \phi_L(\mathsf{x}), \phi_L^{\max}, \phi_P, \mathsf{rk}_{\mathsf{nxt}} \mid \mathsf{x} \in \mathsf{Vars}, \mathsf{msg} \in \mathsf{Msgs}\} \ .$$

The registers $\mathsf{rk}_{\mathsf{Vars}}(\mathsf{x})$ and $\mathsf{rk}_{\mathsf{Msgs}}(\mathsf{msg})$ hold the rank of each variable and message, respectively. This implicitly gives rise to an update sequence. The auxiliary register $\mathsf{rk}_{\mathsf{nxt}}$ is used to initialize the other rank registers, as will be explained later on. The remaining registers correspond to their respective counterparts in the pivot abstraction. Note that the number of registers is linear in the number of messages $|\mathsf{Msgs}|$. The domain of the registers is defined to be $\mathsf{Dom} = \{0, \ldots, |\mathsf{Msgs}| + 1\}$. Since the TSO memory domain is finite, we can assume w.l.o.g. that the memory values are positive integers. If $\mathsf{Lw}(\mathsf{x}) = 0$, it means that there has been no write on x and it still holds the initial value. The set of states Q contains $\mathsf{Q}^{\mathsf{Proc}} \cup \{\mathsf{q}_{\mathsf{init}}^{\mathcal{R}}(\mathsf{A}), \mathsf{q}_{\mathsf{init}}^{\mathsf{ptr}}\}$ as well as a number of (unnamed) auxiliary states that will be used in the following.

To simplify our construction, we will use additional operations on registers, instead of just WRITE and READ. We introduce different blocking comparisons between registers and values such as $==, <, \leq, \neq$, register assignments such as $\mathsf{r} := \mathsf{r}'$, and increments by one denoted as $\mathsf{r}{+}{+}$. A more detailed description of these instructions is given in [3].

The Initializer. The pivot semantics define an exponential number of initial states: one per possible update sequence. The register machine instead guesses an update sequence at the start of the execution and stores it in the rank registers. This part of the register machine is the *rank initializer* (shown in Figure 2 (a)). It uses the auxiliary register $\mathsf{rk}_{\mathsf{nxt}}$ to keep track of the next rank that is to be assigned. In a nondeterministic manner, the rank initializer chooses a so far unranked message and then it assigns the next rank to this message. If the variable of the message has no rank assigned yet, it updates the rank of the variable. Then it increases the $\mathsf{rk}_{\mathsf{nxt}}$ register and continues. After each rank assignment, the initializer can choose to stop the rank assignment. In that case, it initializes the register ϕ_P to 1 and finishes in the initial state of Proc.

In addition to the rank initializer, we have the *pointer initializer*. It is responsible for resetting all pointers except the process pointer to zero. The process pointer is incremented by one instead. This initializer is not executed in the beginning of the simulation, but between epochs of the pivot abstraction.

The simulator. The main part of this construction handles the simulation of the pivot abstraction. It contains $\mathsf{Q}^{\mathsf{Proc}}$ as well as several auxiliary states that are described in the following. It simulates each instruction of $\mathsf{TSO}(\mathsf{A})$. The skip instruction and the data instructions are carried out unchanged. A visualization of the remaining instructions is depicted in Figure 2. In case of a write instruction $\mathtt{wr}(\mathsf{x}, \mathsf{d})$, we first compare the rank of the write message with the process pointer. If they are equal, it means that the epoch is finished and the next process should start, therefore we jump to the first state of the pointer initializer. Otherwise, we set the last write pointer $\mathsf{Lw}(\mathsf{x})$ to d. Now, we ensure that ϕ_L^{max} is at least as large as the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ and finally we update the local pointer $\phi_L(\mathsf{x})$ to be equal to ϕ_L^{max}. For the memory fence instruction, it only needs to be ensured that the external pointer is at least as large as the maximum local pointer ϕ_L^{max}. For a read instruction $\mathtt{rd}(\mathsf{x}, \mathsf{d})$, if the last write to x was of value d, we can execute the read directly. Otherwise, after checking that the write can be performed by the current provider, we ensure that the external pointer is at least as large as both the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ and the local pointer of x. For the special case that $\mathsf{d} = \mathsf{d}_{\mathsf{init}}$, there is an additional way in which the read can be performed: We can read $\mathsf{d}_{\mathsf{init}}$ from the memory if the process has neither already written to x nor observed a write that has higher or equal rank than the rank of x. This gives us the following theorem, proven in Appendix C of the full version [3]:

Theorem 1. $\mathsf{TSO}(\mathsf{A})$*-P-Reach is polynomial time reducible to* $\mathcal{R}(\mathsf{A})$*-Reach.*

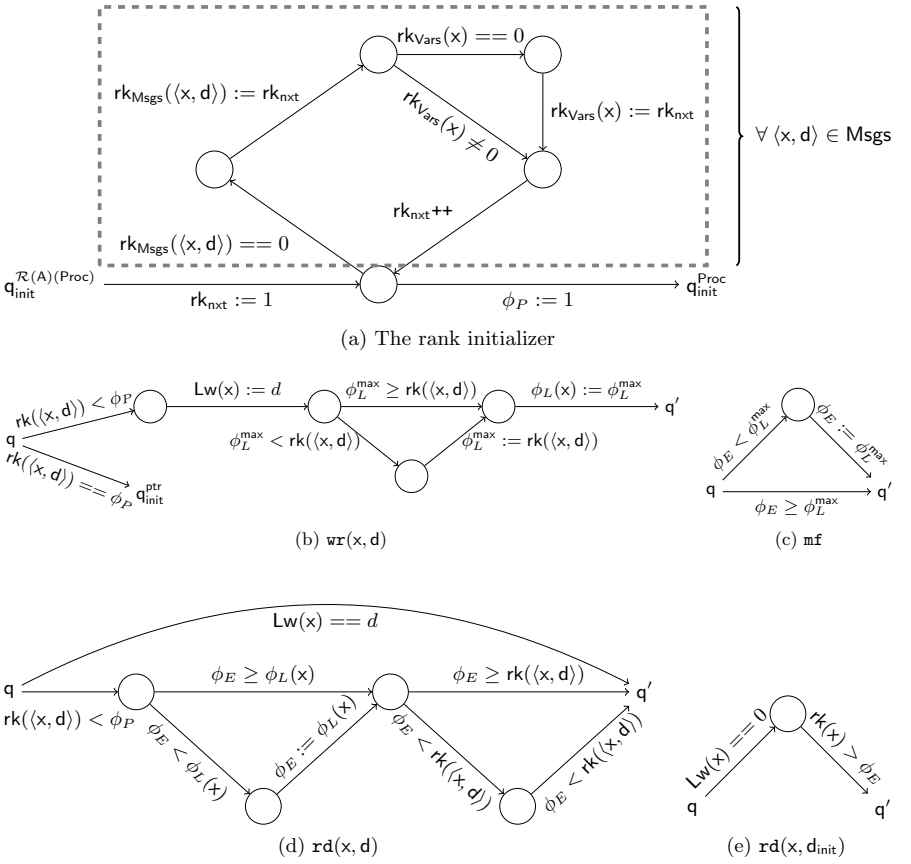

Fig. 2: The rank initializer and the simulator for some instructions instr.

6.2 Simulating Register Machines by TSO

We will now show how to simulate an ADT register machine with a parameterized program running under TSO(A). The main idea is to save the information about the registers in the last pending write operations, while making sure that not a single write operation actually hits the memory. Thus, the simulator always reads the initial value or its own writes, never writes of other processes.

The TSO program has a variable for each register, and two additional variables x_s and x_c that act as flags: x_s indicates that the verifier should start working, while x_c indicates that the verifier has successfully completed the verification. At the beginning of the execution, each process nondeterministically chooses to be either *simulator*, *scheduler*, or *verifier*. Each role will be described in the following. The complete construction is shown in Appendix C of [3].

The simulator uses the same states and transitions as $\mathcal{R}(\mathsf{A})$, but instead of reading from and writing to registers, it uses the memory. If the simulator reaches the target state $\mathsf{q}_{\mathsf{target}}$, it first checks the x_s flag. If it is already set, the simulator

stops, never reaching the final state $\mathsf{q}_{\mathsf{final}}$. Otherwise, it waits until it observes the flag x_c to be set. It then enters the final state. The scheduler's only responsibility is to signal the start of the verification process. It does so by setting the flag x_s at a nondeterministically chosen time during the execution of the program. The verifier waits until it observers the flag x_s. It then starts the verification process, which consists of checking each variable that corresponds to a register. If all of them still contain their initial value, the verification was successful. The verifier signals this to the simulator process by setting the x_c flag.

Any execution ending in $\mathsf{q}_{\mathsf{final}}$ must perform a simulation of $\mathcal{R}(\mathsf{A})$ ending in $\mathsf{q}_{\mathsf{target}}$ first, then a scheduler propagates the setting of flag x_s and afterwards a verifier executes. This ensures that the initial values are read by the verifier after the register machine has been simulated and thus the shared memory is unchanged. This means the simulator only accessed its write buffer and not writes from other threads. It follows that $\mathsf{q}_{\mathsf{target}}$ is reachable by $\mathcal{R}(\mathsf{A})$ if and only if $\mathsf{q}_{\mathsf{final}}$ is reachable by Proc under $\mathsf{TSO}(\mathsf{A})$. This gives us the following result:

Theorem 2. *$\mathcal{R}(\mathsf{A})$-Reach is polynomial time reducible to $\mathsf{TSO}(\mathsf{A})$-P-Reach.*

Theorem 1 and Theorem 2 give us a method of determining upper and lower bounds of the complexity of $\mathsf{TSO}(\mathsf{A})$-P-Reach for different instantiations of ADT. Since we have reductions in both directions, we can conclude that $\mathsf{TSO}(\mathsf{A})$-P-Reach is decidable if and only if $\mathcal{R}(\mathsf{A})$-Reach is decidable. We know $\mathsf{TSO}(\mathsf{A})$-P-Reach is PSPACE-hard for $\mathsf{TSO}(\mathtt{NoAdt})$-P-Reach where NoAdt is the trivial ADT that models plain TSO semantics [4]. We can immediately derive a lower bound for any ADT: $\mathsf{TSO}(\mathsf{A})$-P-Reach is PSPACE-hard.

7 Instantiations of ADTs

In the following, we instantiate our framework to a number of ADTs in order to show its applicability.

Theorem 3. *$\mathsf{TSO}(\mathtt{Ct})$-P-Reach and $\mathsf{TSO}(\mathtt{wCt})$-P-Reach are* PSPACE*-complete.*

We know $\mathsf{TSO}(\mathsf{A})$-P-Reach is PSPACE-hard for any ADT A including Ct and wCt. Regarding the upper bound for Ct, we can show that $\mathcal{R}(\mathtt{Ct})$-Reach can be polynomially reduced to $\mathcal{R}(\mathtt{NoAdt})$-Reach. The idea is to show that there is a bound on the counter values in order to find a witness for $\mathcal{R}(\mathtt{Ct})$-Reach. This bound is polynomial in the number of possible states and register assignments (i.e., this bound is at most exponential in the size of $\mathcal{R}(\mathrm{Ct})$.) Assume a run that contains a configuration c with a value that exceeds the bound, then certain state and register assignment are repeated in the run with different values. We can use this to shorten the run such that the counter value in c is reduced.

We can encode the counter value (up to this bound) in a binary way into registers acting as bits. The number of additional registers is polynomial in the size of $\mathcal{R}(\mathrm{Ct})$. In order to simulate an `inc` operation on this binary encoding using `WRITE` and `READ`, we only have to go through the bits starting at the least

important bit and flip them until one is flipped from 0 to 1. The dec operation works analogously. This only requires a polynomial state and transition overhead.

We know that $\mathcal{R}$(NoAdt)-Reach is in PSpace[4]. It follows from the polynomial reduction that $\mathcal{R}$(Ct)-Reach is in PSpace. Applying Theorem 1 gives us that TSO(Ct)-P-Reach is in PSpace. Since any wCt is a Ct, it follows TSO(wCt)-P-Reach is in PSpace as well. The proof is in [3].

Theorem 4. TSO(St)-*P-Reach is* ExpTime-*complete.*

For membership, we encode the registers of $\mathcal{R}$(St) in the states, which yields a finite state machine with access to a stack, i.e. a pushdown automaton. The construction has an exponential number of states. From [45], we have that checking the emptiness of a context-free language generated by a pushdown automaton is polynomial in terms of the size of the automaton. Combined, we get that state reachability of the constructed pushdown automaton is in ExpTime. It follows that $\mathcal{R}$(St)-Reach is in ExpTime (thanks to Theorem 1).

To prove the lower bound, we can reduce the problem of checking the emptiness of the intersection of a pushdown with n finite-state automata [35] to $\mathcal{R}$(St)-Reach. This problem is well-known to be ExpTime-complete. The idea is to use the stack to simulate pushdown automaton and n registers to keep track of the states of the finite-state automata. We apply Theorem 2 and get TSO(St)-P-Reach is ExpTime-hard. The formal proof is in [3]

Theorem 5. TSO(Petri)-*P-Reach is* ExpSpace-*complete.*

Proof. Petri net coverability is known to be ExpSpace complete [26]. We show hardness by reducing coverability of a marking m to $\mathcal{R}$(Petri)-Reach. The idea is to construct a register machine with a Petri net as ADT. This register machine will have two states $\mathsf{q}_{\mathsf{init}}$ and $\mathsf{q}_{\mathsf{final}}$. For every transition t of the original Petri net, we have t: $\mathsf{q}_{\mathsf{init}} \xrightarrow{t} \mathsf{q}_{\mathsf{init}}$ as a transition of the register machine (we simply simulate the original Petri net). Furthermore, we have $\mathsf{q}_{\mathsf{init}} \xrightarrow{t_m} \mathsf{q}_{\mathsf{final}}$ as a transition of the register machine. Thus, the state $\mathsf{q}_{\mathsf{final}}$ can be reached iff m can be covered.

We reduce reachability of $\mathcal{R}$(Petri) to Petri net coverability. We construct the Petri net by taking the ADT Petri and adding a place p_{q} for every state q and a place $p_{\mathsf{reg},d}$ for every register $\mathsf{reg} \in \mathsf{Regs}$ and register value $d \in \mathsf{Dom}$. The idea is that a marking with a token in p_{q} and one in $p_{\mathsf{reg},d}$ but none $p_{\mathsf{reg},d'}$ for $d' \neq d$ corresponds to a configuration of $\mathcal{R}$(Petri) with state q and $\mathsf{reg} = d$. The value of Petri is given by the remainder of the marking.

We simulate any $\mathsf{q} \xrightarrow{\mathsf{instr}} \mathsf{q}'$ with a transition t that takes one token from q and puts one in q'. If $\mathsf{instr} \in \mathsf{Ops}$, then instr is a Petri net transition. We simply add the same input and output arcs to t. To simulate a write, we add a new transition $t_{d'}$ for every $d' \in \mathsf{Dom}$ with an arc to $p_{\mathsf{reg},d}$ and an arc from $p_{\mathsf{reg},d'}$. The initial marking is consistent with $\mathsf{val}_{\mathsf{init}}^{\text{Petri}}$ and has one token in $p_{\mathsf{q}_{\mathsf{init}}}$. A state q is reachable if a marking with one token in p_{q} is coverable.

Higher Order ADTs. Let $\mathcal{M}(\mathsf{A})$-Reach problem be the restriction of $\mathcal{R}(\mathsf{A})$-Reach with no registers. The $\mathcal{M}(\mathsf{A})$-Reach problem has been studied for many ADT such as higher order counter and higher order stack variations[34,25].

Theorem 6.

- $\mathsf{TSO}(n\text{-}\mathtt{St})$-*P-Reach is* $(n-1)$-ExpTime-*hard and in* n-ExpTime.
- $\mathsf{TSO}(n\text{-}\mathtt{wCt})$-*P-Reach is* $(n-2)$-ExpTime-*hard and in* $(n-1)$-ExpTime.
- $\mathsf{TSO}(n\text{-}\mathtt{Ct})$-*P-Reach is* $(n-2)$-ExpSpace-*hard and in* $(n-1)$-ExpSpace.

Proof. $\mathcal{M}(n\text{-}\mathtt{St})\text{-}\mathtt{Reach}$ has been shown to be $(n-1)$-ExpTime-complete [25]. We know $\mathcal{M}(n\text{-}\mathtt{wCt})\text{-}\mathtt{Reach}$ is $(n-2)$-ExpTime-complete and $\mathcal{M}(n\text{-}\mathtt{Ct})\text{-}\mathtt{Reach}$ is $(n-2)$-ExpSpace-complete [34]. Since the reduction from $\mathcal{M}(\mathsf{A})$-Reach to $\mathcal{R}(\mathsf{A})$-Reach is trivial, any hardness result can be applied to $\mathsf{TSO}(\mathsf{A})$-P-Reach immediately using Theorem 2. In order to reduce $\mathcal{R}(\mathsf{A})$-Reach to $\mathcal{M}(\mathsf{A})$-Reach, we encode register assignments into the state which results in an exponential state explosion. Then we apply Theorem 1 to obtain our upper bound.

Theorem 7. $\mathsf{TSO}(n\text{-}\mathtt{OMSt})$-*P-Reach is* 2-ETime-*complete.*

Proof. We know that $\mathcal{M}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{Reach}$ is 2-ETime-complete [12] and we can apply Theorem 2 to get 2-ETime-hardness. According to Theroem 4.6 in [11], $\mathcal{M}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{Reach}$ is in $\mathcal{O}(|\mathcal{M}(\mathsf{A})|^{2^{dn}})$ for some constant $d \in \mathbb{N}$. We apply the exponential size reduction to $\mathcal{R}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{Reach}$ and Theorem 1 and get $\mathsf{TSO}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{P}\text{-}\mathtt{Reach}$ is in $\mathcal{O}((2^{|\mathcal{P}|})^{2^{dn}}) = \mathcal{O}(2^{|\mathcal{P}|\cdot 2^{dn}})$ and thus it is also in $\mathcal{O}(2^{2^{|\mathcal{P}|}\cdot 2^{dn}}) = \mathcal{O}(2^{2^{|\mathcal{P}|+dn}})$. Thus, $\mathsf{TSO}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{P}\text{-}\mathtt{Reach}$ is in 2-ETime.

We study well structured ADTs [29,9] as defined in [3]:

Theorem 8. *If ADT* A *is well structured, then* $\mathsf{TSO}(\mathsf{A})$-*P-Reach is decidable.*

A register machine for a well structured ADT A is equivalent to the composition of a well structured transition system (WSTS) modeling A and a finite transition system (and thus a WSTS) that models states and registers. According to [9], the composition is again a WSTS and reachability is decidable. The above theorem is then an immediate corollary of Theorem 1.

8 Conclusions and Future Work

In this paper, we have taken the first step to studying the complexity of parameterized verification under weak memory models when the processes manipulate unbounded data domains. Concretely, we have presented complexity results for parameterized concurrent programs running on the classical TSO memory model when the processes operate on an abstract data type. We reduce the problem to reachability for register machines enriched with the given abstract data type.

State reachability for finite automata with ADT has been extensively studied for many ADTs[34,25]. We have shown in Theorem 6 that we can apply

our framework to existing complexity results of this problem. This provides us with decidability and complexity results for the corresponding instances of TSO(A)-P-Reach. However, due to the exponential number of register assignments, the upper bound is exponentially larger than the lower bound. We aim to study these cases further and determine more refined parametric bounds.

A direction for future work is considering other memory models, such as the partial store ordering semantics, the release-acquire semantics, and the ARM semantics. It is also interesting to re-consider the problem under the assumption of having distinguished processes (so-called *leader processes*). Adding leaders is known to make the parameterized verification problem harder. The complexity/decidability of parameterized verification under TSO with a single leader is open, even when the processes are finite-state.

References

1. Parosh Aziz Abdulla. Regular model checking. *STTT*, 14(2):109–118, 2012.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. A load-buffer semantics for total store ordering. *LMCS*, 14(1), 2018.
3. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Florian Furbach, Adwait Godbole, Yacoub G. Hendi, Shankaranarayanan Krishna, and Stephan Spengler. Parameterized verification under tso with data types. *arXiv e-prints*, 2023. arXiv:2302.02163.
4. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. Parameterized verification under tso is pspace-complete. *Proc. ACM Program. Lang.*, 4(POPL), 2019.
5. Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: A CEGAR for parameterized verification. In *CONCUR 2010*, pages 86–101, 2010.
6. Parosh Aziz Abdulla and Giorgio Delzanno. Parameterized verification. *STTT*, 18(5):469–473, 2016.
7. Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Parameterized verification through view abstraction. *STTT*, 18(5):495–516, 2016.
8. Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In *Handbook of Model Checking*, pages 685–725. Springer, 2018.
9. Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160:109–127, 2000.
10. Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.
11. Mohamed Faouzi Atig. Model-Checking of Ordered Multi-Pushdown Automata. *LMCS*, Volume 8, Issue 3, 2012.
12. Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *Developments in Language Theory*, pages 121–133. Springer, 2008.
13. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016.

14. Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large (extended abstract). In *CAV*, volume 2725 of *LNCS*, pages 223–235. Springer, 2003.
15. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ETAPS*, pages 533–553, 2013.
16. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular (tree) model checking. *STTT*, 14(2):167–191, 2012.
17. Sebastian Burckhardt. Principles of eventual consistency. *FTPL*, 1(1-2):1–150, 2014.
18. Thierry Cachat and Igor Walukiewicz. The complexity of games on higher order pushdown automata. *CoRR*, abs/0705.0262, 2007.
19. Sylvain Conchon, David Declerck, and Fatiha Zaïdi. Parameterized model checking on the tso weak memory model. *J. Autom. Reason.*, 64(7):1307–1330, 2020.
20. Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR*, pages 313–327, 2010.
21. Marco Elver and Vijay Nagarajan. TSO-CC: consistency directed cache coherence for TSO. In *HPCA*, pages 165–176. IEEE, 2014.
22. E. Allen Emerson, John Havlicek, and Richard J. Trefler. Virtual symmetry reduction. In *LICS*, pages 121–131, 2000.
23. E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME*, volume 2860 of *LNCS*, pages 247–262. Springer, 2003.
24. E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL*, volume 3210 of *LNCS*, pages 325–339. Springer, 2004.
25. Joost Engelfriet. Iterated stack automata and complexity classes. *Inf. Comput.*, 95(1):21–75, 1991.
26. Javier Esparza. Decidability and complexity of petri net problems - an introduction. *LNCS*, 1491, 2000.
27. Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.
28. Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. *J. ACM*, 63(1):10:1–10:48, 2016.
29. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001. ISS.
30. Marie Fortin, Anca Muscholl, and Igor Walukiewicz. Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In *CAV*, pages 155–175, 2017.
31. Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6:1–6:48, 2012.
32. Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
33. Matthew Hague. Parameterised pushdown systems with non-atomic writes. In *FSTTCS*, pages 457–468, 2011.
34. Alexander Heußner and Alexander Kartzow. Reachability in higher-order-counters. In *MFCS*, pages 528–539. Springer, 2013.
35. Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. Reachability analysis of communicating pushdown systems. In *FOSSACS*, pages 267–281. Springer, 2010.
36. Vineet Kahlon. Parameterization as abstraction: A tractable approach to the dataflow analysis of concurrent programs. In *LICS*, pages 181–192, 2008.

37. Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 645–659. Springer, 2010.
38. Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001.
39. Shankara Narayanan Krishna, Adwait Godbole, Roland Meyer, and Soham Chakraborty. Parameterized verification under release acquire is pspace-complete. In *PODC*, pages 482–492. ACM, 2022.
40. Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In *CONCUR*, pages 72–84, 2015.
41. Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *SIGPLAN-SIGACT*, pages 649–662. ACM, 2016.
42. Anca Muscholl, Helmut Seidl, and Igor Walukiewicz. Reachability for dynamic parametric processes. In *VMCAI*, pages 424–441, 2017.
43. Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In *ETAPS*, volume 9636 of *LNCS*, pages 589–606. Springer, 2016.
44. J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.
45. Ahmed Bouajjani Rajeev Alur and Javier Esparza. *Handbook of Model Checking*, chapter Model Checking Procedural Programs, pages 547–569. Springer, 2018.
46. Alberto Ros and Stefanos Kaxiras. Racer: TSO consistency via race detection. In *MICRO*, pages 33:1–33:13. IEEE Computer Society, 2016.
47. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *ACM SIGPLAN, PLDI*, pages 175–186. ACM, 2011.
48. 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.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.