Parameterized Verification under The Total Store Order Memory Model is EXPTIME-Complete

Yacoub Hendi
Abstract

Parameterized Verification under The Total Store Order Memory Model is EXPTIME-Complete

Yacoub Hendi

In this paper, we study the problem of parameterized verification of a concurrent program running under the Total Store Order (TSO) memory model. A concurrent program is a finite set of processes that are instances of the same pushdown system and which communicate through a set of shared variables. A memory model is a set of rules that decides in what order the memory operations done by one process are observed by the other processes. TSO is a weak memory model. In TSO, a FIFO buffer is inserted between each process and the shared memory, where a write operation is added to the buffer and it then gets non-deterministically updated to the shared memory. This causes the order of read and write operations inside a process to be distorted when it is observed by other processes. The main result we show is that the parameterized state reachability problem of a concurrent program running under TSO is EXPTIME-complete. Note that this is an extension of a previous result which showed that the problem is PSPACE-complete when the processes are instances of a finite state system. The main advantage of extending the analysis to pushdown systems is that the concurrent programs can then use recursion.
Acknowledgment
I would like to express my deepest appreciation to my supervisor Parosh Abdulla and my reviewer Mohammed Atig who introduced this field to me and always supported me with advices and ideas to finish this thesis.
## Contents

1 Introduction .................................................. 1

2 Preliminaries .................................................. 3

3 TSO Semantics .................................................. 4
   3.1 Classical TSO Semantics .................................. 5
   3.2 Parameterized TSO Semantics .......................... 7

4 The Pivot With Buffer Abstraction .......................... 9
   4.1 Definitions ............................................... 10
   4.2 The Pivot With Buffer Transition System ............ 11
   4.3 From $T_{PTSO}$ To $T_{PB}$ ............................ 14
   4.4 From $T_{PB}$ To $T_{PTSO}$ ............................ 16

5 The Pivot Abstraction ......................................... 19
   5.1 Definitions ............................................... 19
   5.2 The Pivot Transition System ............................ 20
   5.3 From $T_P$ To $T_{PB}$ .................................... 22
   5.4 From $T_{PB}$ To $T_P$ .................................... 23

6 EXPTIME-Completeness ....................................... 24
   6.1 Register Machine ........................................ 24
   6.2 From $T_P$ to $T_R$ ...................................... 25
   6.3 EXPTIME-hard ........................................... 29

7 Conclusion .................................................... 32

A Proof of Lemma 4.2 .......................................... 32

B Proof of Lemma 4.3 .......................................... 33

C Proof of Lemma 4.4 .......................................... 37

D Proof of Lemma 4.5 .......................................... 38

E Proof of Lemma 5.3 .......................................... 42

F Proof of Lemma 5.4 .......................................... 48

G Proof of Lemma 6.1 .......................................... 53
1 Introduction

In software development, it is a key step to check that a program is safe and correct, and this usually consists of constructing unit tests or other type of tests. However, when we deal with concurrent programs, testing becomes vulnerable due to concurrency issues such as data races and race conditions, and other issues as heisenbugs. The other alternative to testing, and which is more suitable to concurrent programs, is verification, and to verify a program it means to mathematically prove that the program is correct with respect to its specification. To this end, different techniques have been invented to verify program and one of them is called Hoare Logic. In Hoare Logic, the specification of the program is translated to formal logical statements which represent pre-conditions and post-conditions on inputs and outputs, respectively, while each instruction in the program is converted to a logical implication formula. Hoare Logic defines the correct program to the program where the pre-conditions imply the post-conditions. Another verification technique is model checking, which is a vast field that covers many different techniques. In general, the main idea is to represent a program by some diagram such as an automaton, and the specification is then translated to specific property on the diagram. The main advantage of verification, in contrast to testing, is that it proves the absence of bugs in programs. Nevertheless, this requires developing several different algorithms according the type of programs and the type of properties we want to verify. This creates the need to study algorithmically verification problems, namely, study the decidability and the complexity of different classes of verification problems.

In this paper, we consider the verification of parameterized systems. A parameterized system is any system that its definition relies on some parameter; and here we fix the definition of a parameterized system to be a concurrent program that consists of an arbitrary number of processes, where the parameter is the number of processes. Parameterized verification is to prove that a given property holds for the system regardless of the value of the parameter. For example, the verification of a mutual exclusion protocol is a parameterized verification because we want to verify that no two processes are in their critical sections at the same time regardless of the number of the processes involved in the system. In this paper, we focus on a specific problem in parameterized verification which is the state reachability problem. The problem is to decide, for a given state $q$ of a process $P$, whether the parameterized system induced by $P$ has a run such that one process in the system reaches the state $q$. To be clearer, if the answer is yes then for some $n$ there is a concurrent system consisting of $n$ instances of $P$, and it has a run such that one of the instances reaches the state $q$.

It is known that the general problem of parameterized verification is undecidable. Therefore in the last three decades, research focused on parameterized verification of special classes of systems hoping to prove decidability and construct efficient algorithms for these specific classes. To this end, systems are classified according to three factors: the topology, namely how processes are
ordered in relation to each others (in array, tree, ... etc); the communication
mechanisms, namely how processes communicates (via channels, rendezvous,
shared variables, ... etc) and the types of processes (finite-state automata,
push-down automata, leader, ... etc). Many results were shown for such special
systems and here we mention few of them. It was shown that the parameterized
verification of a system that consists of finite-state automata communicating
via rendezvous is decidable, and more precisely it is EXSPACE-complete [1].
Later on, the reachability problem for a parameterized system that consists of
a single leader and an arbitrary number of identical pushdown automata which
communicate over a register with bounded value was shown to be PSPACE-
complete [3]. Other results have also been shown, nevertheless, the majority
of these results are to some extent impractical for the modern computers be-
cause these results assume that the systems adopt Sequential Consistency as a
memory model, which is not the case for modern computers.

A memory model is a set of rules that decides in what order the sequence of
memory operations done by a process is viewed by other processes. The most
intuitive memory model is the Sequential Consistency model(SC) defined by
Lamport [4]. Under SC, the memory operations are observed by all processes
in the same order as they were issued. Another way to define SC, and where
its name comes from, is when "the result of any execution is the same as if
the operations of all the processes were executed in some sequential order, and
the operations of each individual process appear in this sequence in the order
specified by its program" [4]. We say SC preserves all four possible orderings
of memory operations which are specifically, read:read, read:write, write:write
and write:read. Even though SC depicts how usually programmers reason about
their programs, however, it has a major drawback represented in its expensiveness
as SC exposes the latency of write operations. For this reason, SC is not
adopted by modern processors. To understand why SC is very expensive, con-
sider a CPU that adopts SC and that executes a program which does a write
operation on \textit{x} followed by a read operation on \textit{y}. In this example, the CPU
must finish the write operation to \textit{x} in the main memory before it starts reading
\textit{y}. Since the CPU is much faster than the main memory, most of the time spent
to execute the write operation is used by the memory to respond or update
while the CPU stays idle because it cannot starts the execution of the next
instruction.

The Total Store Order model(TSO) allows for more optimization than SC.
In TSO, an unbounded FIFO store buffer is added between each process and the
main memory. When a process does a write operation a pending message of the
write operation gets added to the buffer. Then, the message at the top of the
buffer gets non-deterministically updated to the memory. When a process reads
the value of a variable it tries to read it from its own buffer in what we call a
read-own-write operation, and if it fails then it brings the value from the main
memory. Since the process does not need to write to the main memory directly,
many, it means that the process might do a subsequent read before the write operation
reaches the main memory. Therefore, TSO does not preserve the ordering of
write:read. It is clear that TSO is at least as expressive as SC because any run
in SC can be simulated by TSO by just adding an update operation after each write. However, the extra expressiveness of TSO can result into extra behaviors of which some are bad behaviors.

In [5], it was shown that the parameterized state reachability problem under TSO is PSPACE-complete, where the system consists of identical finite-state systems. To prove this result, the authors designed a novel semantics called the pivot semantics that collapses the FIFO buffers into bounded pointers and proved that the pivot semantics is exact w.r.t to the parameterized state reachability problem i.e a state is reachable under the pivot semantics iff it is reachable under the parameterized TSO semantics (PTSO). The definition of the pivot semantics relies on two observations. The first one is the unbounded supply principle which states that if an assignment \( \langle x, d \rangle \), where \( x \) is a variable and \( d \) is a value, hits the shared memory then there is an unbounded supply of this update which can be read by other processes. This allows us to only focus on the first instances of updates. The second observation is that in a run, a write operation depends only on previous write operations. By using the pivot semantics, the authors showed that the state reachability problem under PTSO can be reduced from/to the reachability problem of 1-safe Petri nets which shows that the state reachability problem under PTSO is PSPACE-complete.

In this paper, we consider the parameterized state reachability problem under TSO where the program consists of identical pushdown systems. Notice that the main reason we consider pushdown systems instead of finite-state automata as it was done in [5], is because pushdown systems simulate recursion. In other words, we study the parameterized state reachability problem under TSO for recursive programs. We start with introducing the classical TSO semantics, and the parameterized version of it. Then, we discuss the unbounded supply principle in PTSO semantics. We then use this principle to remove the first source of unboundedness in the system, namely the unbounded number of process, by simulating a run in PTSO by a run of a bounded number of abstract processes called suppliers. In the next step, we convert the suppliers into providers which are processes in the pivot semantics. In total, this shows the exactness of the pivot semantics w.r.t the parameterized state reachability problem under TSO.

In the last section, we reduce the state reachability problem under the pivot semantics to the state reachability problem under a PDS with an exponential size, which shows that the problem is at most ExpTime. Then we reduce the problem of checking the emptiness of intersection between a context free language and a number of regular languages to the parameterized state reachability problem under TSO. This shows that last problem is ExpTime-hard, and by its upper bound it becomes ExpTime-complete.

2 Preliminaries

Let \( f \) be a function, we define \( f[a \leftarrow b] \) to be the function \( f' \) such that \( f'(a') = b \) if \( a = a' \), and \( f'(a') = f(a') \), otherwise. We also use the symbol \( \perp \) in \( f(a) = \perp \) to express that \( f \) is undefined at \( a \). For two natural numbers \( n_1 \leq n_2 \), the set
\{n_1, \ldots, n_2\}\) contains all natural numbers between \(n_1\) and \(n_2\), inclusive themselves. Also for a finite set \(A\) we define \(|A|\) to be the size of the set, and \(A^*\) to be the set of finite words over \(A\). In addition, we define \(P(A)\) to be the power set of \(A\) which contains all the subsets of \(A\).

A labeled transition system (LTS) is the tuple \(\langle C, C_{\text{init}}, L, \rightarrow \rangle\), where \(C\) is the set of configurations and \(C_{\text{init}} \subseteq C\) is the set of initial configurations. \(L\) is a set of labels, or actions. The symbol \(\rightarrow\) is a transition relation with the signature \(C \times L \times C\). A transition in \(\rightarrow\) is the triple \(\langle c, l, c' \rangle \in \rightarrow\), which we also express by the notation \(c \xrightarrow{l} c'\). We write \(c \rightarrow c'\) to denote that for some label \(l\) we have \(c \xrightarrow{l} c'\). Also, for \(C_1, C_2 \subseteq C\) we write \(C_1 \rightarrow C_2\) to denote that there are \(c_1 \in C_1\) and \(c_2 \in C_2\) such that \(c_1 \rightarrow c_2\). We define \(\rightarrow\) to be the reflexive transitive closure on \(\rightarrow\). If \(C_1 \rightarrow C_2\) we say \(C_2\) is reachable from \(C_1\). Sometimes, we write \(C_1 \twoheadrightarrow C_2\) instead of \(C_1 \rightarrow \{c_2\}\), and say that \(c_2\) is reachable from \(C_1\). A run in the transition system is an alternating sequence of configurations and actions and it can be expressed as follows

\[c_0 \xrightarrow{l_1} c_1 \ldots \xrightarrow{l_n} c_n.\]

We write \(c_0 \xrightarrow{n} c_n\) to denote that \(c_n\) can be reached from \(c_0\) through \(n\) steps and \(c_0 \xrightarrow{l} c_n\) to denote that \(c_n\) is reachable from \(c_0\) through the run \(\rho\). We write \(c \in \rho\) to denote that \(c\) is reached somewhere in the run \(\rho\), and we write \(l \in \rho\) to denote that a transition with label \(l\) is used somewhere in the run \(\rho\). We define \(\#\rho = n\); namely the number of steps in \(\rho\). We define \(\text{start}(\rho) = c_0\) and \(\text{end}(\rho) = c_n\). A run \(\rho\) is initialized if \(\text{start}(\rho) \in C_{\text{init}}\). We define \(\rho[i \ldots j]\) to be the sub-run from \(c_i\) to \(c_j\). For two runs \(\rho_1\) and \(\rho_2\), where \(\text{end}(\rho_1) = \text{start}(\rho_2)\), we define \(\rho_1 \cdot \rho_2\) to be the concatenation of the two runs. Also if \(\text{end}(\rho_1) = c_1\), \(\text{start}(\rho_2) = c_2\) and \(c_1 \xrightarrow{l} c_2\) then \([\rho_1][\rho_2]\) denotes the concatenation \(\rho_1 \cdot c_1 \xrightarrow{l} c_2 \cdot \rho_2\).

### 3 TSO Semantics

In this section we introduce the semantics of a concurrent program that consists of a fixed number of identical pushdown systems (PDS) which communicate between each other through a set of shared variables under TSO. Then we introduce the parameterized case; namely the case where the number of processes is unbounded.

We define \(X\) to be the finite set of shared variables ranging over the finite domain \(\mathbb{D} \subseteq \mathbb{N}\). We also define a PDS \(P\) to be a tuple \((Q, q_{\text{init}}, \Gamma, Act, \Delta)\), where \(Q\) is a finite set of states, \(q_{\text{init}} \in Q\) is the initial state, \(\Gamma\) is a finite stack alphabet, \(Act\) is a finite set of actions and \(\Delta\) is a finite set of transitions such that \(\Delta \subseteq (Q \times Act \times Q)\). We also define the size of the PDS \(P\), denoted by \(|P|\), to be equal to \(|Q| + |\Delta|\). A transition \(\delta \in \Delta\) induced by an action \(\alpha \in Act\) is of the form \(q \xrightarrow{\alpha} q'\), and it means that the PDS \(P\) can move from the state \(q\) to state \(q'\) after it executes the action \(\delta\).

Finally, we fix the set of actions \(Act\) such that for \(x \in X\), \(d \in \mathbb{D}\) and \(\gamma \in \Gamma\), the set \(Act\) contains the actions \(w(x, d)\), \(r(x, d)\), \(\text{pop}(\gamma)\), \(\text{push}(\gamma)\), \(\text{skip}\) and \(\text{mf}\).
3.1 Classical TSO Semantics

We define now the operational semantics of a concurrent program that consists of a finite instances of PDS $P$ running under TSO. The semantics is defined via a labeled transition system $\mathcal{T}_{TSO}$. In TSO, there is an unbounded FIFO buffer between each process and the main shared memory. The main task of the buffer is to hold the sequence of the write operations done by the process but which are not updated to the memory yet. Each write operation in the buffer is denoted by an assignment $\langle x, d \rangle$ which means that the process assigns the value $d$ to the variable $x$. Let $A = \mathbb{X} \times \mathbb{D}$, and since the buffer is a sequence of assignments, a buffer state is a finite word on $A$.

A configuration in $\mathcal{T}_{TSO}$ is a tuple $\langle I, Q, S, B, M \rangle$, where $I$ is an index set that contains an index for each process in the system and it is fixed. $Q : I \rightarrow \mathbb{Q}$ is a total function that stores the current state for each process. $S : I \rightarrow \Gamma^* \mathbb{Γ}$ is a total function that stores the current state of the stack for each process. $B : I \rightarrow A^*$ is a total function that stores the buffer state of each process. Finally, $M : \mathbb{X} \rightarrow \mathbb{D}$ is a total function that stores the memory state. For a configuration $c = \langle I, Q, S, B, M \rangle$, we define $c(Q) = Q$ and similarly for other components in $c$. We define $C_{\mathcal{T}_{TSO}}$ to be the set of all possible configurations in $\mathcal{T}_{TSO}$.

We define the initial configuration to be $c_{\text{init}} = \langle I, Q_{\text{init}}, S_{\text{init}}, B_{\text{init}}, M_{\text{init}} \rangle$, where $Q_{\text{init}}(i) = q_{\text{init}}$, $S(i) = \epsilon$ and $B_{\text{init}}(i) = \epsilon$ for all $i \in I$. Also, $M_{\text{init}}(x) = 0$; here we assume that $0 \in \mathbb{D}$ and that the initial value of any variable $x$ in the memory before any update is made to it is 0. This means that in the initial configuration, all processes are in their initial states with empty stacks and empty buffers, and the memory still holds the initial values of all variables. We define $C_{\mathcal{T}_{TSO}}$ to be the set of initial configurations in $\mathcal{T}_{TSO}$ and since $I$ is fixed we get that $C_{\mathcal{T}_{TSO}}$ contains only a single element.

We also define the function $\text{LVal} : A^* \rightarrow [\mathbb{X} \rightarrow \mathbb{D} \cup \{\emptyset\}]$ such that $\text{LVal}(w)(x) = d$ if $\langle x, d \rangle$ is the last assignment for $x$ that occurs in $w$, and if there is no assignment for $x$ in $w$ then $\text{LVal}(w)(x) = \emptyset$, where $\emptyset$ is some natural number that does not belong to $\mathbb{D}$. For example, $\text{LVal}(B(i))(x) = d$ if $w(x, d)$ is the last write operation done by $i$ on the variable $x$ and the write message has not been updated to the memory yet, and $\text{LVal}(B(i))(x) = \emptyset$ if the process $i$ has no pending write message in its buffer regarding the variable $x$. We define now the transition relation of $\mathcal{T}_{TSO}$ denoted by $\rightarrow$ and introduce its set of labels.

In Fig. we notice that each inference rule is induced by some event represented as a tuple containing an action and the process that does the action. For an event $\lambda$ represented by tuple $\langle i, opr \rangle$, we define $\lambda.pro = i$ and $\lambda.opr = opr$. We also notice that there is one rule for the write operation, and in this rule the process pushes a write message to its buffer rather writing directly to the memory. From the inference rule ”memory-update”, we make two observations.
(i) a pending write message in the buffer reaches the memory only after all the messages in front of it have already reached, (ii) the update of a pending write message on the top of a buffer is non-deterministic. We also notice that there are two inference rules for reading. In the rule "read-own-write", the process reads the last pending write message on the variable \( x \) in its buffer and checks if the value is equal to \( d \). In the rule "read-from-memory", the process first checks that it has no pending write message on the variable \( x \) in its buffer and therefore it can not read \( x \) from the buffer, then it reads \( x \) from the memory and checks if the value is equal to \( d \). In the inference rule "fence", we check if the process has no pending message in its buffer. The inference rules "push" and "pop" are the only rule that modify the stack of a process, and the rule "pop" is the only rule that sets a condition on the state of the stack.
The State Reachability Problem under $T_{TSO}$ (SRP). For an LTS $T_{TSO}$ induced by a PDS $P$ and an index set $I$, and for a state $q \in Q$. We ask whether there is a configuration $c$ reachable from $C_{init}$ such that there is some $i \in I$ and $Q(i) = q$.

Example. We give here examples of runs in $T_{TSO}$. Consider the state reachability problem for the concurrent program $P$ induced by the automaton $P$ in Fig.2, and the state $q_{tar} \in Q$. Assume that $P$ consists of a single process, then we can immediately see that the state $q_{tar}$ is not reachable. This is because there is only one path in $P$ that reaches $q_{tar}$ and this path requires the reading of the assignment $\langle y, 1 \rangle$, which is not a reading of an initial value and the single process does not execute the operation $w(y, 1)$ on this path. Assume now that $P$ has two processes ($p_1$ and $p_2$), then $q_{tar}$ becomes reachable. A possible run is to let $p_1$ read $\langle x, 0 \rangle$, which is a read of an initial value, and move to the state $q_1$. Then let $p_1$ execute $w(x, 1)$ and move to the state $q_2$. The program then non-deterministically can choose to update the pending write message $\langle x, 1 \rangle$ to the memory. Now $p_2$ can move from $q_{init}$ to the state $q_3$ by reading $\langle x, 1 \rangle$ from the memory. Then let $p_2$ execute the write operation $w(y, 1)$ and move to the state $q_4$. The program then can non-deterministically choose to update the pending write message $\langle y, 1 \rangle$ to the memory. Finally, $p_1$ reads the assignment $\langle y, 1 \rangle$ from the memory and moves to the state $q_{tar}$. Note that for any number of instances greater or equal to 2, we can repeat the same run using two instances only while keeping all the other processes in their initial states. This implies that if the state reachability problem is satisfied for $n$ instances then it is satisfied for any $n' > n$.

3.2 Parameterized TSO Semantics

In this section, we introduce the parameterized case of verification under TSO. In the parameterized case, we investigate if the state reachability problem under TSO is satisfiable for any size of the index set $I$, rather than for a fixed bounded size of $I$. As we already illustrated, parameterized verification is relevant because in many cases we want to verify some property of a concurrent program regardless of the number of processes it includes. For example, a mutual exclusion protocol must guarantee that no two processes are in their critical section at the same time, regardless of the number of processes in the program.

We have shown that if a state $q$ is reachable for an index set $I$ of size $n$ then it is also reachable for any index set with a larger size. Therefore, one way to introduce parameterization formally is by letting the index set $I$ be unbounded. We do this by defining a new labelled transition system called parameterized TSO (PTSO) and denoted by $T_{PTSO}$. The set of configurations $C$, the set of initial configurations $C_{init}$, the transition relation $\rightarrow$ and the set of labels in $T_{PTSO}$ are the same as their correspondents in $T_{TSO}$, with the only difference in the interpretation of the index set $I$.

The interpretation of $I$ under $T_{PTSO}$ gives rise to the unbounded supply
principle, which states that if a state \( q \) is reached by one process then \( q \) can be reached by an unbounded number of processes that all have the same buffer and the same stack. It follows directly that if an assignment \( a \) hits the memory then there is an unbounded number of processes that have \( a \) at the bottom of their buffers, which can non-deterministically update \( a \) to the memory. This means that when a value of a variable \( x \) reaches the memory then it stays there for ever. However, this does not apply for the initial value of \( x \), because this value is not provided by any process. Therefore, unlike the values of \( x \) updated to the memory, the initial value of \( x \) disappears when it is overwritten by an update.

<table>
<thead>
<tr>
<th></th>
<th>( P1 )</th>
<th>( P2 )</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( r(x,0) )</td>
<td>( x = 0; y = 0 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( w(x,1) )</td>
<td>( x = 0; y = 0 )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( u(x,1) )</td>
<td>( x = 1; y = 0 )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( r(x,1) )</td>
<td>( x = 1; y = 0 )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( w(y,1) )</td>
<td>( x = 1; y = 0 )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( w(y,2) )</td>
<td>( x = 1; y = 1 )</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( u(y,1) )</td>
<td>( x = 1; y = 1 )</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( r(y,2) )</td>
<td>( x = 1; y = 1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>( u(y,2) )</td>
<td>( x = 1; y = 2 )</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>( u(y,1) )</td>
<td>( x = 1; y = 1 )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: To the left is an invalid run in \( T_{TSO} \). To the right is the same run in \( T_{PTSO} \) made valid by the unbounded supply principle

**Example.** We show here an example of how an invalid run in the classical semantics can be modified through the use of the unbounded supply principle to be valid in the parameterized semantics. In Fig.3 to the left we have a run in a program containing two processes \( P1 \) and \( P2 \). We assume that the program starts from its initial configuration, which means all the buffers and the stacks are empty, \( P1 \) and \( P2 \) are in their initial states and all the shared variables hold their initial values which we assume to be 0 for all of them. At the first step, \( P1 \) reads \( x \) from the memory, as its buffer is empty. It then writes \( x,1 \) which is pushed to its buffer and in the next step it updates \( x,1 \) to the memory. At step 4, \( P2 \) reads \( x,1 \) from the memory and in the next two steps it pushes at first \( y,1 \) then \( y,2 \) to its buffer. \( P2 \) then updates \( y,1 \) to the memory, this is because \( y,1 \) is the last assignment in its buffer. In the next step, \( P2 \) reads the value \( y,2 \) from its buffer, although the value of \( y \) is 1 in the memory. This is because in TSO, the process always reads its latest write operation on the variable that is still in its buffer and it reads from the memory only if no such
write is found. At step 9, \( P2 \) updates \( \langle y, 2 \rangle \) to the memory. At step 10, \( P1 \) tries to read \( \langle y, 1 \rangle \), but it fails because it has no such write in its buffer and the value of \( y \) is 2 in the memory. However, if the program is parameterized then \( P1 \) will be able to read \( \langle y, 1 \rangle \) due the principle of unbounded supply. We notice that \( P2 \) updates \( \langle y, 1 \rangle \) before \( P1 \) tries to read it. In the table to the right in Fig.3, process \( P2' \) simulates \( P2 \) until it writes \( \langle y, 1 \rangle \), and since no other writes is executed before then \( \langle y, 1 \rangle \) sits at the top of the buffer and can be updated at any we want to the memory. The process \( P2' \) updates the assignment \( \langle y, 1 \rangle \) to memory at step 12 which allows \( P1 \) to read \( \langle y, 1 \rangle \) from the memory. In this example we showed how in \( T_{PTS0} \), due to the unbounded supply principle, if an update is made by some process then it is possible to add a copy of the process in order to replicate the update. From now on, we will use this result of the unbounded supply principle without explicitly adding more copies like the process \( P2' \). We define now the parameterized state reachability problem under TSO.

**The State Reachability Problem under PTSO (PSRP).** For an LTS \( T_{PTS0} \) induced by a PDS \( \mathcal{P} \), and a state \( q \in Q \). We ask whether there is a configuration \( c \) reachable from \( C_{init} \) such that there is some \( \iota \in \mathcal{I} \) and \( Q(\iota) = q \).

If we want to analyse the space and time complexity of PSRP then we immediately face two problems characterized by the unbounded nature of system. The unbounded nature of the system has two sources, the first is the number of processes and the second is the size of the buffers. In order tackle these problems, we will define new abstractions that simulate PTSO and that uses a bounded number of processes and bounded data structures.

### 4 The Pivot With Buffer Abstraction

An abstraction of a system is some system that simulates given properties in the original system to a certain extent. According to the extent of simulation, abstractions are divided into three categories: an over-approximation, an under-approximation and exact. We say an abstraction is an under-approximation if the property satisfaction in the abstraction implies that it is satisfied in the original system. We say an abstraction is an over-approximation if the property satisfaction in the original system implies that it is satisfied in the abstraction. Finally, an abstraction is exact if it is both an under-approximation and an over-approximation.

In the paper \[5\], the authors introduced the pivot abstraction as an exact abstraction of the PTSO w.r.t the state reachability problem. The pivot abstraction relies on identifying certain points called pivots in a run in PTSO. A pivot corresponds to an assignment \( a \), where it refers to the first update of \( a \) under the run. These pivots induce a ranking on the assignments where an assignment \( a \) gets a lower rank than an assignment \( a' \) if the pivot of \( a \) occurs before the pivot of \( a' \). Furthermore, the ranking of assignments induce a ranking on the variables and the processes. The crucial observation is that in order to
produce the pivot of $a$, we only need to produce the pivots of the assignments with lower ranks. The pivot abstraction relies on this observation and simulates the behavior of the processes in a run in PTSO by a sequence of abstract processes called providers that execute one after the other. The main task of $k$th provider is to simulate the process in the run in PTSO that produces the $k$th pivot and since the number of pivots is bounded by $|X| \times |D|$ we get that the number of providers is bounded. The other main advantage of the pivot abstraction is that the providers do not use unbounded buffers. Instead the providers use variables and functions that are defined over sets of bounded size. This enabled the authors to show that the state reachability problem under PTSO is PSPACE-complete, where the processes are instances of finite automata. We replace the finite state automata with PDS and use the pivot abstraction to show that the state reachability problem under PTSO is EXPSPACE-complete.

In order to simplify the pivot abstraction, we introduce it in two steps. In each step we tackle a single source of unboundedness. In the first step, and specifically in this section, we introduce the pivot with buffer abstraction of PTSO which is exact w.r.t the state reachability problem. The pivot with buffer abstraction uses the pivots in a run in PTSO to convert the run into a sequential execution of a bounded number of abstract processes called the suppliers. Hence it removes the first source of unboundedness, however, the suppliers still use unbounded buffers. In the second step, and specifically in the next section, we introduce the pivot abstraction as abstraction of the pivot with buffer system. The pivot abstraction relies on observations about the behavior of buffers in the pivot with buffer system and simulates the sequential execution of a bounded number of suppliers with the sequential execution of a bounded number of providers, where the providers use only bounded data structures, and thus it removes the second source of unboundedness.

In the next subsection, we formalize our observations about PTSO in order to introduce the pivot with buffer abstraction.

### 4.1 Definitions

Let us fix $\rho$ to be an initialized run in $T_{PTSO}$ defined by the following sequence:

$$c_{init} \xrightarrow{\lambda_1} c_1 \cdots \xrightarrow{\lambda_n} c_n.$$

We define $A(\rho)$ to be the subset of $A$ that contains all the assignments updated to the memory under $\rho$. We also define $\text{first}(\rho)(a)$, for $a \in A(\rho)$, to be the position of the first update of the assignment $a$ in $\rho$, namely $\text{first}(\rho)(a) = \min\{i \mid \lambda_i.\text{opr} = u(a)\}$. If $a \notin A(\rho)$ then we define $\text{first}(\rho)(a) = \bot$. Recall that by the unbounded supply principle once an update is made then it is always accessible by all processes. This means that for each $a \in A(\rho)$ the value $\text{first}(\rho)(a)$ refers to the first position in $\rho$ where the read of $a$ from the memory is possible after it.

Based on this, we define the concept of ranks. We say that an assignment $a$ has a lower rank than assignment $a'$ in $\rho$, if $\text{first}(\rho)(a) < \text{first}(\rho)(a')$. 
We define a new function \( \text{rank}(\rho) : A \to \{1, \ldots, |A(\rho)|\} \), such that if \( a \notin A(\rho) \) then \( \text{rank}(\rho)(a) = \bot \), otherwise \( \text{rank}(\rho)(a) = \min\{\alpha \in A(\rho) | \text{first}(\rho)(\alpha) \leq \text{first}(\rho)(a)\} \). Based on the function \( \text{rank}(\rho) \), we define the function \( \text{rank}_X(\rho) : X \to \{1, \ldots, |A(\rho)|\} \), which for a given \( x \) returns the smallest rank of an assignment of \( x \). For a variable \( x \), we define \( \text{rank}_X(\rho)(x) = \min\{\text{rank}(\rho)((x,d)) | d \in D, (x,d) \in A\} \), and if \( x \) is not updated under \( \rho \) then \( \text{rank}_X(\rho)(x) = \bot \). We say that a read action of an assignment \( a \) is of rank \( k \) if \( \text{rank}(\rho)(a) = k \), similarly for the ranks of the write actions and the update actions.

The positions defined by the function \( \text{first}(\rho) \) are called the pivots of \( \rho \). We define the function \( \text{pivots}(\rho) : \{1, \ldots, |A(\rho)|\} \to \{1, \ldots, n\} \) such that \( \text{pivots}(\rho)(k) = i \) where \( \text{first}(\rho)(a) = i \) and \( \text{rank}(\rho)(a) = k \). We also call the position \( \text{pivots}(\rho)(k) \) to be the \( k \)-pivot in \( \rho \). Based on the function \( \text{pivots}(\rho) \), we define a ranking function for the processes \( \text{rank}_I(\rho) : I \to P\{1, \ldots, |A(\rho)| + 1\} \) such that \( i \in \text{rank}_I(\rho)(\cdot) \) iff \( \lambda_{\text{pivots}(\rho)(k)} \cdot \text{pro} = i \). For technical reasons, we let \( |A(\rho)| + 1 \in \text{rank}_I(\rho)(\cdot) \) only if \( \lambda_n \cdot \text{pro} = i \).

In order to formalize the unbounded supply principle we define a new concept called the external pointer of \( \rho \), denoted by \( \phi_E(\rho) \). The external pointer of \( \rho \) is a function \( \phi_E(\rho) : \{0, \ldots, n\} \to \{0, \ldots, |A(\rho)|\} \), where for each position \( i \) in \( \rho \) it returns the highest rank of a pivot that occurs before \( i \). Formally, \( \phi_E(\rho)(i) = \max\{k \mid \text{pivots}(\rho)(k) \leq i\} \). Clearly, \( \phi_E(\rho) \) is an increasing monotonic function. Moreover, \( \phi_E(\rho) \) can be computed recursively as follows: \( \phi_E(\rho)(0) = 0 \) and \( \phi_E(\rho)(i + 1) = \max(\phi_E(\rho)(i), \text{rank}(\rho)(a)) \) if \( \lambda_{i+1} \cdot \text{opr} = u(a) \), otherwise \( \phi_E(\rho)(i + 1) = \phi_E(\rho)(i) \). The purpose of the external pointer is that it tells us what updates have reached the memory up to date, and hence it tells us what read-from-memory actions are possible.

We observe that a read-from-memory of an initial value \( \text{rfm}(x, \text{init}(x)) \) at position \( i \) is possible only if \( \phi_E(\rho)(i) < \text{rank}_X(\rho)(x) \), as this basically means that no update of \( x \) has reached the memory, yet. We also observe that a read-from-memory of an assignment \( a \) that is updated by some process is possible at position \( i \) only if \( \phi_E(\rho)(i) \geq \text{rank}(\rho)(a) \), as this basically means that some process has updated \( a \) to the memory before the position \( i \) and therefore by the unbounded supply principle \( \text{rfm}(a) \) at position \( i \) is possible.

Note that all the functions defined here such as \( \text{rank}(\rho) \), \( \text{pivots}(\rho) \), etc are definable only after \( \rho \) completes its execution and therefore cannot be used in the rules of \( T_{PTSO} \). Nevertheless, the importance of these functions is that they help us to simulate runs in \( T_{PTSO} \).

### 4.2 The Pivot With Buffer Transition System

In this section, we introduce the pivot with buffer abstraction described by the LTS \( T_B \). The LTS \( T_B \) simulates the behavior of processes in \( T_{PTSO} \) by a sequence of a bounded number of abstract processes called the suppliers that run one after the other. The task of the \( k \)-th supplier is to simulate the execution of the process \( \iota \) in \( \rho \) from the beginning until it executes the update at the \( k \)-pivot, given that \( k \in \text{rank}_I(\rho)(\iota) \).
Given the task of the $k$-supplier, we derive what type of information a supplier must have in order to carry out the simulation. The supplier must know the process state and the stack state in addition to know what values of variables are readable by the process. For the state and the stack we can let the supplier store them locally. For the readable values, we recall that a process tries to read a value of a variable from its own buffer at first, and to simulate this we let the supplier stores the buffer locally. A process also reads a value of a variable from the shared memory if no assignment of the variable exists in the buffer. We observe that $k$-supplier only reads assignments of ranks less than $k$ because it simulates a process up to the $k$-pivot. To simulate this, we let the supplier stores the rank of the process it simulates. Also for reads of initial values, we let the supplier keep track of what updates have reached the memory, whether done by the process it simulates or other processes.

Based on the information needed by the supplier we define the configuration in $\mathcal{T}_{PB}$. A configuration $s$ in $\mathcal{T}_{PB}$ is called a scene and it is a tuple $\langle q, S, B, \text{rank}, \phi_E, \phi_P \rangle$, where $q \in Q$ is a process state; $S \in \Gamma^*$ a stack state; $B \in A^*$ a buffer state; $\text{rank} : A \rightarrow \{1, \ldots, n\}$, where $n$ can be any natural number, is a ranking function and it simulates the ranking function $\text{rank}(\rho)$ and $\rho$ is the run in $\mathcal{T}_{PTSO}$ to be simulated; $\phi_E$ is an index that is called the external pointer, too, and it is crucial to simulate reads of initial values; and $\phi_P$ is an index that is called the progress pointer and it holds the rank of the process it is simulated. The progress pointer is crucial to simulate reads from the shared memory. For a scene $s$, we let $s(q)$ denote the process state at $s$. We use similar notation, such as $s(S), s(B)$ etc, to refer to the other components in the scene $s$. We denote the set of all scenes by $\mathcal{C}_{PB}$.

A run in $\mathcal{T}_{PB}$ is a sequence of executions of suppliers, where the last supplier is called the verifier and its task is not to supply an update but to reach a specific process state. For the function $\text{rank}$ and the progress pointer $\phi_P$, we define the initial scene $s_{init}(\text{rank}, \phi_P) = \langle q_{init}, \epsilon, \epsilon, \text{rank}, 0, \phi_P \rangle$, where $q_{init}$ is the initial process state. The set of initial configurations in $\mathcal{T}_{PB}$, denoted by $\mathcal{C}_{init}$, contains only the initial scenes of the first suppliers, namely $\mathcal{C}_{init}$ contains $s_{init}(\text{rank},1)$ for any possible $\text{rank}$.

The transition relation of $\mathcal{T}_{PB}$, denoted by $\rightarrow_{PB}$, is given in Fig.4. The aim of the transition relation $\rightarrow_{PB}$ is to simulate the transition relation $\rightarrow_{PT}$. The rules for skip, write, fence, push and pop simulate in a direct manner the corresponding rules in the transition relation $\rightarrow_{PT}$. The rule read(1) in $\rightarrow_{PB}$ simulates the rule read-own-write in $\rightarrow_{PT}$. The other two remaining read rules in $\rightarrow_{PB}$ are read(2) and read(3) and combined they simulate the read-from-memory rule $\rightarrow_{PT}$. In both rules, we first check that a read from buffer is not possible by reading $\text{LVal}(B)(x) = \emptyset$. The rule read(2) simulates a read from memory of the initial value of a variable $x$, therefore the value of $x$ must not have been overwritten. Since the scene does not have a memory component, to check this, the rule read(2) relies on the components $\text{rank}$ and $\phi_E$ and it checks $\text{rank}(x) > \phi_E$. The rule read(3) in $\rightarrow_{PB}$ simulates a read from memory of an assignment $a$ that is provided by some process, and it does so by checking
In read(3), the external pointer is also updated to max(φ_E, rank(a)) because from the supplier perspective such a read can happen only if it is preceded by an update of a. There are also two update rules in \( \rightarrow \), update(1) and update(2) which combined they simulate the update rule in \( \rightarrow \). Since the scene does not include a memory component, therefore the update applies to the external pointer instead. In rule update(1), the rule checks that the buffer has the required assignment at its bottom and that the rank of the updated assignment is less than the progress pointer. This means that the update is possible and this update is not the final event in the current supplier. The rule also updates the external supplier to max(φ_E, rank(a)). In the rule update(2), the rank of the updated assignment a must be equal to the progress pointer. This means that the supplier has finished its execution and therefore it moves the execution to the next supplier.

An initialized run \( \rho \) in \( T_{PB} \) is of the form:

\[
s_{init}(\text{rank}, 1) \xrightarrow{\psi_1^1} s_1^1 \xrightarrow{\psi_2^1} s_2^1 \cdots s_{m_1}^1 \xrightarrow{u(2)(a_1)} s_{init}(\text{rank}, 2) \xrightarrow{\psi_2^2} s_2^2 \cdots \\
\vdots
\]

\[
s_{init}(\text{rank}, n) \xrightarrow{u(2)(a_n)} s_{init}(\text{rank}, n + 1) \xrightarrow{\psi_{n+1}^1} s_1^{n+1} \cdots s_m^{n+1} \xrightarrow{\psi_{m+n+1}^k} s_{m+n+1}^k.
\]

Where \( \text{rank}(a_i) = i \), and \( \psi_i^j \neq u(2)(a) \). We define the k-supplier of the run \( \rho \) formally to be the sub-run:

\[
s_{init}(\text{rank}, k) \xrightarrow{\psi_k^1} s_k^1 \xrightarrow{\psi_k^2} s_2^k \cdots s_{m_k}^k.
\]
4.3 From $T_{PTSO}$ To $T_{PB}$

Also for a run $\rho$ in $T_{PTSO}$, not necessarily initial, we define $\text{last}(\rho)$ to be the last scene in $\rho$.

The State Reachability Problem under $T_{PB}$. For an LTS $T_{PB}$ induced by a PDS $P$, and a process state $q_{\text{tar}} \in Q$, the problem asks whether there is a scene $s$ that is reachable from $c_{\text{init}}^{\infty}$ in $T_{PB}$ and such that $s(q) = q_{\text{tar}}$.

Lemma 4.1. A process state $q$ in a PDS $P$ is reachable in $T_{PTSO}$ iff it is reachable in $T_{PB}$.

The next two subsections are devoted to the proof of this lemma. In the first subsection we design Algorithm 1 which takes a run $\rho'$ in $T_{PTSO}$ and produces a run $\rho$ in $T_{PB}$ where both runs reach the same process state in their final configurations. The rest of the work consists of proving that $\rho$ is actually valid in $T_{PB}$, and that there is a process in $\rho$ that reaches the same process state as the one reached to by $\rho'$ in its final scene.

4.3 From $T_{PTSO}$ To $T_{PB}$

We explain here briefly and informally Algorithm 1. The algorithm takes a valid run $\rho'$ in $T_{PTSO}$ and returns a valid run $\rho$ in $T_{PB}$. At line 1, it initializes $\text{rank}$ to be the function $\text{rank}(\rho')$ where $\text{rank}$ will be used in the scenes in $\rho$. At line 2, it initializes $n$ to be equal to the number of pivots in $\rho'$. At lines 3, 4 and 5, it initializes $n+1$ suppliers where each supplier corresponds to a pivot in $\rho'$ and includes only its initial scene. At line 7, it initializes $n$ Boolean values to $\text{True}$. The boolean $\text{Bool}_k$ is switched to $\text{False}$ when the $k$-supplier is fully constructed.

After it finishes constructing the suppliers in the for-loop, it connects them at line 46.

In Algorithm 1, we say that an event $\psi$ and a scene $s$ in $\rho$ simulate an event $\lambda_i$ and a configuration $c_i$ in $\rho$, respectively, if the event $\psi$ and the scene $s$ get added to $\rho$ under an iteration induced by the event $\lambda_i$ in $\rho'$. Also for technical
Algorithm 1: How to construct a run $\rho$ in $T_{\text{PB}}$ from a run $\rho'$ in $T_{\text{PTSO}}$.

Input: $\rho'$ in $T_{\text{PTSO}}$.
Output: $\rho$ in $T_{\text{PB}}$.

1. $\text{rank} \leftarrow \text{rank}(\rho')$;
2. $n \leftarrow \max(\text{rank})$;
3. Let $Pr$ be an array of size $n + 1$;
4. for $i = 1$ to $i = n + 1$ do
5. \hspace{1em} $\text{Sup}[i] = s_{\text{init}}(\text{rank}, i)$
6. end
7. $\text{Bool}_1, \ldots, \text{Bool}_n \leftarrow \text{True}$;
8. for $i = 1$ to $i = \#\rho'$ do
9. \hspace{1em} $i \leftarrow \lambda_i: \text{pro}$;
10. \hspace{2em} $\alpha \leftarrow \lambda_i: \text{opr}$;
11. \hspace{2em} $q \leftarrow c_i(Q)(i); \ S \leftarrow c_i(S)(i); \ B \leftarrow c_i(B)(i)$;
12. \hspace{2em} for $k \in \text{rank}_E(\rho')(i)$ do
13. \hspace{3em} if $\text{Bool}_k$ then
14. \hspace{4em} $\phi_E \leftarrow \text{last}(\text{Sup}[k])(\phi_E)$;
15. \hspace{4em} if $\alpha \in \{\text{skip}, w(a), \text{push}(\gamma), \text{pop}(\gamma), \text{mf} \mid a \in A, \gamma \in \Gamma\}$ then
16. \hspace{5em} $s \leftarrow \langle q, S, B, \text{rank}, \phi_E, k \rangle$;
17. \hspace{5em} $\text{Sup}[k] \leftarrow \text{Sup}[k]. \overset{\alpha}{\rightarrow} \text{Sup}[s]$;
18. \hspace{4em} end
19. \hspace{4em} if $\alpha \in \{\text{row}(a) \mid a \in A\}$ then
20. \hspace{5em} $s \leftarrow \langle q, S, B, \text{rank}, \phi_E, k \rangle$;
21. \hspace{5em} $\text{Sup}[k] \leftarrow \text{Sup}[k]. \overset{r(1)(a)}{\rightarrow} \text{Sup}[s]$;
22. \hspace{4em} end
23. \hspace{4em} if $\alpha \in \{\text{rfm}(a) \mid a \in A\}$ then
24. \hspace{5em} if $a \in \{(x, 0) \mid x \in X\}$ then
25. \hspace{6em} $s \leftarrow \langle q, S, B, \text{rank}, \phi_E, k \rangle$;
26. \hspace{6em} $\text{Sup}[k] \leftarrow \text{Sup}[k]. \overset{r(2)(a)}{\rightarrow} \text{Sup}[s]$;
27. \hspace{5em} end
28. \hspace{5em} else
29. \hspace{6em} $\phi'_E \leftarrow \max(\phi_E, \text{rank}(a))$;
30. \hspace{6em} $s \leftarrow \langle q, S, B, \text{rank}, \phi'_E, k \rangle$;
31. \hspace{6em} $\text{Sup}[k] \leftarrow \text{Sup}[k]. \overset{r(3)(a)}{\rightarrow} \text{Sup}[s]$;
32. \hspace{5em} end
33. \hspace{4em} end
34. \hspace{4em} if $\alpha \in \{u(a) \mid a \in A\}$ then
35. \hspace{5em} if $\text{rank}(a) < k$ then
36. \hspace{6em} $\phi'_E \leftarrow \max(\phi_E, \text{rank}(a))$;
37. \hspace{6em} $s \leftarrow \langle q, S, B, \text{rank}, \phi'_E, k \rangle$;
38. \hspace{6em} $\text{Sup}[k] \leftarrow \text{Sup}[k]. \overset{u(1)(a)}{\rightarrow} \text{Sup}[s]$;
39. \hspace{5em} end
40. \hspace{5em} if $\text{rank}(a) == k$ then
41. \hspace{6em} $\text{Bool}_k \leftarrow \text{False}$;
42. \hspace{5em} end
43. \hspace{4em} end
44. \hspace{4em} end
45. \hspace{4em} end
46. \hspace{1em} $\rho \leftarrow \text{Sup}[1]. \overset{u(2)(a_1)}{\rightarrow} \text{Sup}[2] \ldots \text{Sup}[n]. \overset{u(2)(a_n)}{\rightarrow} \text{Sup}[n + 1]$;
4.4 From $T_{PB}$ To $T_{PTSO}$

reasons we take the initial scenes in $\rho$ to simulate the configuration $c_{\text{init}}$ in $\rho'$, and we denote $c_{\text{init}}$ by $c_0$.

Lemma 4.2. If the scene $s$ in $\rho$, constructed by Algorithm 1, simulates a configuration $c_i$ in $\rho'$ then:

1. $s(q) = c_i(Q)(i)$.
2. $s(B) = c_i(B)(i)$.
3. $s(S) = c_i(S)(i)$.
4. $s(\phi_E) \leq \phi_E(\rho')(i)$.

Proof. Appendix A. $\square$

Lemma 4.3. The run $\rho$ constructed in Algorithm 1 is a valid run in $T_{PB}$.

Proof. Appendix B. $\square$

Corollary 4.3.1. For a PDS $P$, if a process state $q_{\text{tar}} \in Q$ is reachable in $T_{PTSO}$ then $q_{\text{tar}}$ is also reachable in $T_{PB}$.

Proof. Since $q_{\text{tar}}$ is reachable in $T_{PTSO}$, then there is a run $\rho'$ in $T_{PTSO}$, such that for some $i \in I$, $\lambda_{#\rho'} \cdot \text{pro} = i$ and $c_{#\rho'}(Q)(i) = q_{\text{tar}}$. By lemma 4.3 we get that the run $\rho$ in $T_{PB}$ constructed from $\rho'$ by Algorithm 1 is valid. By Algorithm 1 and the definition of $\text{rank}_I(\rho')$ we get that the scene $\text{last}(\rho)$ simulates $c_{#\rho'}$. By lemma 4.2 we get $\text{last}(\rho)(q) = q_{\text{tar}}$. $\square$

4.4 From $T_{PB}$ To $T_{PTSO}$

We explain here informally what Algorithm 2 does and make few observations on the constructed run $\rho$. At line 1, the algorithm initiates a process for each supplier in $\rho'$; the idea is to let the process with id $k$ to simulate the $k$-supplier in $\rho'$. At line 2, it initiates the run $\rho$ to include only the initial configuration $c_{\text{init}}$. At line 3, it initiates the variable $\text{index}$ to be 0. At line 4 the algorithm starts a for loop that iterates over all the events in $\rho'$. For each event $\psi$, it stores the rank of the supplier, the process state, the buffer state and the stack state before and after the execution of $\psi$ in $\rho'$ in multiple variables. Then according to the type of the event $\psi$, the algorithm chooses a position $\text{prev}$ in the up-to-date $\rho$ and constructs a configuration $c$. It then chooses an event $\lambda$ in $T_{PTSO}$ that is equivalent to $\psi$ and inserts $\lambda$ and $c$ at position $\text{prev} + 1$, where the semantic of $\text{insert}$ is defined as follows. If $\rho$ is a run in $T_{PTSO}$ defined by the sequence:

\[
\begin{array}{c}
\text{c}_{\text{init}} \xrightarrow{\lambda_1} c_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_m} c_m \xrightarrow{\lambda_{m+1}} \cdots \xrightarrow{\lambda_n} c_n,
\end{array}
\]

then $\text{insert}(\rho, (k, \alpha), c, m + 1)$ is the run:

\[
\begin{array}{c}
\text{c}_{\text{init}} \xrightarrow{\lambda_1} c_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_m} c_m \xrightarrow{(k, \alpha)} c \xrightarrow{\lambda_{m+1}} \cdots \xrightarrow{\lambda_n} c_n,
\end{array}
\]
Algorithm 2: How to construct a run $\rho$ in $T_{PTSO}$ from a run $\rho'$ in $T_{PB}$.

**Input:** $\rho'$ in $T_{PB}$.

**Output:** $\rho$ in $T_{PTSO}$.

1. Let $\{1, \ldots, \max(\text{rank})\} \subseteq I$;
2. $\rho \leftarrow c_{\text{init}}$;
3. $\text{index} \leftarrow 0$;
4. for $i = 1$ to $i = \#\rho'$ do
5.   $k \leftarrow s_i(\phi_p)$; $q \leftarrow s_i(q)$; $S \leftarrow s_i(S)$; $B \leftarrow s_i(B)$;
6.   $q' \leftarrow s_{i-1}(q)$; $S' \leftarrow s_{i-1}(S)$; $B' \leftarrow s_{i-1}(B)$; $s_0 = s_{\text{init}}(\text{rank}, 1)$;
7.   if $\psi_i \in \{\text{skip}, u(a), \text{push}(\gamma), \text{pop}(\gamma), \text{mf} \mid a \in A, \gamma \in \Gamma\}$ then
8.     prev $\leftarrow \text{index}$;
9.     $c \leftarrow (I, c_{\text{prev}}(Q)[k \leftarrow q], c_{\text{prev}}(S)[k \leftarrow S], c_{\text{prev}}(B)[k \leftarrow B], c_{\text{prev}}(M))$;
10.    $\rho \leftarrow \text{insert}(\rho, (k, \psi_i), c, \text{prev} + 1)$;
11.    $\text{index} \leftarrow \text{prev} + 1$;
12.   end
13. if $\psi_i \in \{r(1)(a) \mid a \in A\}$ then
14.   prev $\leftarrow \text{index}$;
15.   $c \leftarrow (I, c_{\text{prev}}(Q)[k \leftarrow q], c_{\text{prev}}(S)[k \leftarrow S], c_{\text{prev}}(B)[k \leftarrow B], c_{\text{prev}}(M))$;
16.    $\rho \leftarrow \text{insert}(\rho, (k, \text{row}(a)), c, \text{prev} + 1)$;
17.    $\text{index} \leftarrow \text{prev} + 1$;
18. end
19. if $\psi_i \in \{r(2)(x, 0) \mid x \in X\}$ then
20.   prev $\leftarrow \text{index}$;
21.   $c \leftarrow (I, c_{\text{prev}}(Q)[k \leftarrow q], c_{\text{prev}}(S)[k \leftarrow S], c_{\text{prev}}(B)[k \leftarrow B], c_{\text{prev}}(M))$;
22.    $\rho \leftarrow \text{insert}(\rho, (k, \text{rfm}(x, 0)), c, \text{prev} + 1)$;
23.    $\text{index} \leftarrow \text{prev} + 1$;
24. end
25. if $\psi_i \in \{r(3)(a) \mid a \in A\}$ then
26.   prev $\leftarrow \max(\text{index}, \text{first}(\rho)(a))$;
27.   $c \leftarrow (I, c_{\text{prev}}(Q)[k \leftarrow q], c_{\text{prev}}(S)[k \leftarrow S], c_{\text{prev}}(B)[k \leftarrow B], c_{\text{prev}}(M))$;
28.    $\rho \leftarrow \text{insert}(\rho, (k, \text{rfm}(a)), c, \text{prev} + 1)$;
29.    $\text{index} \leftarrow \text{prev} + 1$;
30. end
31. if $\psi_i \in \{u(1)(x, d) \mid x \in X, d \in D\}$ then
32.   prev $\leftarrow \max(\text{index}, \text{first}(\rho)(a))$;
33.   $c \leftarrow (I, c_{\text{prev}}(Q)[k \leftarrow q], c_{\text{prev}}(S)[k \leftarrow S], c_{\text{prev}}(B)[k \leftarrow B], c_{\text{prev}}(M)[x \leftarrow d])$;
34.    $\rho \leftarrow \text{insert}(\rho, (k, u(x, d)), c, \text{prev} + 1)$;
35.    $\text{index} \leftarrow \text{prev} + 1$;
36. end
37. if $\psi_i \in \{u(2)(x, d) \mid x \in X, d \in D\}$ then
38.   prev $\leftarrow \#\rho$;
39.   $c \leftarrow (I, c_{\text{prev}}(Q), c_{\text{prev}}(S), c_{\text{prev}}(B)[k \leftarrow B''], c_{\text{prev}}(M)[x \leftarrow d])$;
40.    $\%\%$ where $B' = B'' \cdot (x, d)$;
41.    $\rho \leftarrow \text{insert}(\rho, (k, u(x, d)), c, \text{prev} + 1)$;
42.    $\text{index} \leftarrow 0$;
43. end
4.4 From $T_{PB}$ To $T_{PTSO}$

Where $c_i' = (I, c_i(Q)[k ← c(Q)(k)], c_i(S)[k ← c(S)(k)], c_i(B)[k ← c(B)(k)], c_i(M))$, and $m < i \leq n$.

Let $\rho$ be a run in $T_{PTSO}$ constructed by Algorithm 2 based on a run $\rho'$ in $T_{PB}$. We say that an event $\lambda$ in $\rho$ simulates an event $\psi$ in $\rho'$ if $\lambda$ is added to $\rho$ under the iteration induced by $\psi$. We observe that the variable index in Algorithm 2 always holds the position of the event in $\rho$ that simulates the last considered event in the $k$-supplier. Since $\text{prev} \geq \text{index}$, we get that if two events $\psi_1$ and $\psi_2$ belong to the same supplier in $\rho'$ and $\psi_1$ precedes $\psi_2$ then the event $\lambda_1$ precedes the event $\lambda_2$ in $\rho$ where $\lambda_1$ and $\lambda_2$ simulate $\psi_1$ and $\psi_2$, respectively. Also for a configuration $c$ in $\rho$, if $c$ is added to $\rho$ under the iteration induced by the event $\psi^k$ then we say $c$ simulates the scene $s^k$ in $\rho'$. For technical reasons we take $c_{\text{init}}$ to simulate $s^k_0$ for all possible $k$.

We also observe that the $k$-pivot in $\rho$ is the position of the event in $\rho$ that simulates the event $u(2)(a_k)$. This follows from the fact that in a run $\rho'$ in $T_{PB}$ and an assignment $a \in \mathcal{A}$, an event $u(1)(a)$ must be preceded by an event $u(2)(a)$. This means that for any assignment $a$, the event that simulates $u(2)(a)$ is inserted before the event that simulates an event $u(1)(a)$. Moreover, in Algorithm 2 by line 32 an event that simulates $u(1)(a)$ is inserted in $\rho$ after $\text{first}(\rho)(a)$. As an additional result we get, $\text{rank} = \text{rank}(\rho)$ where $\text{rank}$ is the ranking function in $\rho$. Another result is that since the pivots in $\rho$ are the positions of events that simulates events of the form $u(2)(a_k)$, and since these events are always inserted at the end of the up-to-date $\rho$, therefore the insertion of new pivots does not change the value of the external pointer at events inserted in previous iterations. In other words, the value of the external pointer at any event $\lambda$ in the final version of $\rho$ is the same as the value of the external pointer at $\lambda$ when it was just inserted. We use now these observations to prove the next two lemmata.

**Lemma 4.4.** If the configuration $c_m$ in $\rho$, constructed by Algorithm 1, simulates the scene $s^k$ in the $k$-supplier in $\rho'$ then:

1. $c_m(Q)(k) = s^k(q)$.
2. $c_m(B)(k) = s^k(B)$.
3. $c_m(S)(k) = s^k(S)$.
4. $\phi_E(\rho)(m) \leq s^k(\phi_E)$.

*Proof.* Appendix C. □

**Lemma 4.5.** The run $\rho$ constructed by Algorithm 2 is a valid run in $T_{PTSO}$.

*Proof.* Appendix D. □

**Corollary 4.5.1.** For a PDS $\mathcal{P}$, if a process state $q_{\text{tar}} \in Q$ is reachable in $T_{PB}$ then $q_{\text{tar}}$ is also reachable in $T_{PTSO}$. □
Proof. Assume $q_{\text{tar}}$ is reachable in $T_{PB}$, then there is a an initial run $\rho'$ in $T_{PB}$ such that $\text{last}(\rho')(q) = q_{\text{tar}}$. By lemma 4.3, we know that the run $\rho$ constructed in Algorithm 2 based on $\rho'$ is a valid initial run in $T_{P_TSO}$. We also know that there is a configuration $c$ in $\rho$ such that $c$ simulates the scene $\text{last}(\rho')$. By lemma 4.4, we get that for some $\iota \in I$, $c(Q)(\iota) = q_{\text{tar}}$. Thus, $q_{\text{tar}}$ is also reachable in $T_{P_TSO}$.

5 The Pivot Abstraction

In this section, we introduce the pivot abstraction. We observe the buffer in the scene in $T_{PB}$ serves two functionalities. For a variable $x$, the buffer is used in the rules read(1), (2), (3) and memory-fence to check for the last written value to the variable $x$ that is still in the buffer. In the rule update(1) the buffer is used to update the value of the external pointer. The pivot abstraction builds on the observation that the first functionality can be served without a buffer by using a function $L$ that stores the last written value to any variable. The second functionality which consists of updating the external pointer can also be served without the buffer by using a new set of pointers which are collectively called the local pointer and is denoted by $\phi_L$. For each variable $x$, the local pointer stores the highest rank of a write operation done before the last write operation on $x$. Therefore, the local pointer is a function with a domain $X$.

Note that both $L$ and $\phi_L$ are bounded data structures. In the next subsection, we formalize these observations.

5.1 Definitions

In this subsection, we fix $\rho$ to be following initial run in $T_{PB}$:

$s_{\text{init}}(\text{rank},1) \xrightarrow{\psi_1^1} s_1^1 \xrightarrow{\psi_1^2} s_2^1 \cdots \xrightarrow{\psi_1^i} s_m^1 \xrightarrow{\text{u(2)}(a_1)} s_{\text{init}}(\text{rank},2) \xrightarrow{\psi_2^1} s_2^2 \cdots$

$s_m^n \xrightarrow{\text{u(2)}(a_n)} s_{\text{init}}(\text{rank},n+1) \xrightarrow{\psi_{n+1}^1} s_1^{n+1} \cdots \xrightarrow{\psi_{n+1}^{m+1}} s_{m,n+1}.$

For the $i$th event in the $k$-supplier in $\rho$, we define $L(\rho)(k)(i)$ to be the function $L(\rho)(k)(i) : X \rightarrow D \cup \{\emptyset\}$. This function is meant to hold the last written value to any variable inside the $k$-supplier up to the $i$th event in the $k$-supplier. If a variable $x$ is not written to inside the $k$-supplier up to the $i$th event then $L(\rho)(k)(i)(x) = \emptyset$. Therefore, for all $x$ and $k$, $L(\rho)(k)(0)(x) = \emptyset$. The main difference between $L(\rho)(k)(i)$ and $LVal$ is that $L(\rho)(k)(i)$ holds the last values written to variables whether these values are still in the buffer or not.

A crucial observation is that if $L(\rho)(k)(i)(x) = d$ and the supplier wants to read the assignment $\langle x,d \rangle$ in the event $\psi_{i+1}^1$, then the external pointer does not change in the next scene. This is because if the $\langle x,d \rangle$ is still in the buffer then the supplier can use the read(1) rule to do the read and this does not change the external pointer. Otherwise, if the $\langle x,d \rangle$ is not in the buffer then the
The Pivot Transition System

The pivot labeled transition system, denoted by \( T_P \), simulates the behavior of the suppliers in a run \( T_{PB} \) by a sequence of abstract processes called the providers which only use bounded data structures. The task of the \( k \)-th provider is to simulate the \( k \)-th supplier from its initial scene until it executes the first write operation of rank \( k \), or until its last operation if it does not do such a write operation.

Given the task of the \( k \)-provider, we derive the type of information needed to carry out the simulation. The provider must know the process state, the stack state, the ranking function, the external pointer and the progress pointer of \( k \)-supplier. It must also know information about of the functions \( \mathcal{L} \) and \( \phi_L \) in order to replace the buffer. Based on what the \( k \)-provider must know we define the configuration in \( T_P \).

A configuration in \( T_P \) is a called a view and it is a tuple \( \langle q, S, \text{rank}, \mathcal{L}, \phi_E, \phi_L, \phi_P \rangle \), where \( q \), \( \text{rank} \), \( \phi_E \) and \( \phi_P \) are defined in the same manner as in the scenes in \( T_{PB} \). The component \( \mathcal{L} \) is function \( \mathcal{L} : \mathbb{X} \rightarrow \mathbb{D} \sqcup \{ \emptyset \} \) where \( \mathcal{L}(x) = d \) if the last written assignment of \( x \) by the provider is \( \langle x, d \rangle \), and if no assignment of \( x \) is written under the provider then \( \mathcal{L}(x) = \emptyset \). The component \( \phi_L \) is called the local pointer and it is a function where for each variable \( x \) it holds the highest rank among the write operation that are executed before the last write operation on \( x \). For a view \( v \), we let \( v(q) \) denote the process state at
v. We use similar notation, such as \( s(S), s(\text{rank}) \) etc., to refer to the other
components in the view \( v \). We denote the set of all possible views by \( \mathcal{C}^\emptyset \). For
the function \( \text{rank} \) and the progress pointer \( \phi_p \) we define the initial view
\( v_{\text{init}}(\text{rank}, \phi_p) = (v_{\text{init}}, \phi, \text{rank}, \mathcal{L}_o, \phi_L, \phi_E, \phi_p) \), where for all \( x \), \( \mathcal{L}_o(x) = \emptyset \) and \( \phi_L(x) = 0 \). The \( k \)-provider always starts with \( v_{\text{init}}(\text{rank}, k) \). The set of
initial configurations in \( T_P \), denoted by \( \mathcal{C}_{\text{init}}^\emptyset \), contains only the initial views of
the first providers, namely \( \mathcal{C}_{\text{init}}^\emptyset \) contains \( v_{\text{init}}(\text{rank}, 1) \) for any possible \( \text{rank} \).

The transition relation of \( T_P \), denoted by \( \rightarrow \emptyset \), is given in Fig. 5. The aim of
the transition relation \( \rightarrow \emptyset \) is to simulate the transition relation in \( T_{PB} \). Since
the views do not have buffers, therefore the transition relation does not have any
rules for updates. Therefore, the external pointer is updated in the rules of
\( \text{read}(3) \) and memory-fence based on the observations we made in the previous
subsection.

The rules for skip, push, pop, and \( \text{read}(2) \) in \( \rightarrow \emptyset \) simulate in a direct manner
the corresponding rules in the transition relation in \( T_{PB} \).

The rule \( \text{read}(3) \) in \( \rightarrow \emptyset \) simulates the rule \( \text{read}(3) \) in \( T_{PB} \) as both require
the rank of the read assignment to be less than the progress pointer. Also, in
\( \text{read}(3) \) in \( T_P \) the external pointer \( \phi_E \) is updated to \( \max(\phi_E, \phi_L(x), \text{rank}(x, d)) \),
the elements \( \phi_E \) and \( \text{rank}(x, d) \) exist in the max function here for the same
reasons as in the rule \( \text{read}(3) \) in \( T_{PB} \). The element \( \phi_L(x) \) exists in the max
function in order to simulate the fact that in \( T_{PB} \) before any application of the
rule \( \text{read}(3) \) the supplier must empty its buffer from all pending messages of the
variable \( x \).

The rule \( \text{read}(1) \) in \( \rightarrow \emptyset \) does not simulate directly any read rule in \( T_{PB} \). It
nevertheless simulates the event in \( T_{PB} \) where the supplier reads the last written
value to a variable whether this value is still in the buffer or not. Based on the
observation in the previous subsection the external pointer does not change after
such a read and therefore in the rule \( \text{read}(1) \) in \( \rightarrow \emptyset \) the external pointer is not
changed either.

The memory-fence rule in \( \rightarrow \emptyset \) simulates the memory-fence rule in \( T_{PB} \).
The memory-fence rule in \( \rightarrow \emptyset \) updates the external pointer to \( \max(\phi_E, \phi_{L_{\text{max}}} \) in
order to simulate the fact that in \( T_{PB} \) before any application of the memory-
fence rule the supplier must empty its buffer completely.

The write(1) rule in \( \rightarrow \emptyset \) simulates the write rule in \( T_{PB} \) if \( \text{rank}(x, d) < k \).
In the rule write(1) in \( \rightarrow \emptyset \) the last written value to \( x \), \( \mathcal{L}(x) \), is updated and the
local pointer \( \phi_L(x) \) is updated to \( \max(\phi_L, \text{rank}(x, d)) \). The write(2) rule in
\( \rightarrow \emptyset \) simulates indirectly the rule update(2) in \( T_{PB} \), in the sense that it is the
final event in the \( k \)-provider and it moves the execution to the \((k+1)\)-provider.

An initialized run \( \rho \) in \( T_P \) is of the form:

\[
v_{\text{init}}(\text{rank}, 1) \xrightarrow{\cdot} \emptyset v_1^1 \xrightarrow{\cdot} v_2^1 \ldots \xrightarrow{w(2)(a_1)} \emptyset v_{\text{init}}(\text{rank}, 2) \xrightarrow{\cdot} \emptyset v_1^2 \ldots
\]

\[
v_{m_n}^n \xrightarrow{w(2)(a_n)} \emptyset v_{\text{init}}(\text{rank}, n + 1) \xrightarrow{\cdot} \emptyset v_1^{n+1} \ldots \xrightarrow{\cdot} \emptyset v_{m_n+1}^{n+1}.
\]
5.3 From $\mathcal{T}_P$ To $\mathcal{T}_{PB}$

The State Reachability Problem under $\mathcal{T}_P$. For an LTS $\mathcal{T}_P$, induced by a PDS $\mathcal{P}$, and a process state $q_{\text{tar}} \in Q$, the problem asks whether there is a view $v$ that is reachable from $C_{\text{init}}$ in $\mathcal{T}_P$ and such that $v(q) = q_{\text{tar}}$.

**Lemma 5.1.** A process state $q$ in a PDS $\mathcal{P}$ is reachable in $\mathcal{T}_{PB}$ iff it is reachable in $\mathcal{T}_P$.

Each one of the next two subsections proves an implication in lemma 5.1.

5.3 From $\mathcal{T}_P$ To $\mathcal{T}_{PB}$

In this subsection we prove the following lemma:

**Lemma 5.2.** If a process state $q$ in a PDS $\mathcal{P}$ is reachable in $\mathcal{T}_P$ then it is reachable in $\mathcal{T}_{PB}$.

The proof of this lemma relies on the following simulation relation.
The Simulation Relation. For a run \( \rho \) in \( \mathcal{T}_P \) and a scene \( s = \langle q, S, B, \text{rank}, \phi_E, \phi_P \rangle \), we say that \( s \) simulates the view \( v^k_i \) in \( \rho \), denoted by \( \rho \models^k_i s \) if the following holds for any variable \( x \) and any value \( d \):

1. \( q = v^k_i(q) \), \( S = v^k_i(S) \), \( \phi_P = v^k_i(\phi_P) \) and \( \text{rank} = v^k_i(\text{rank}) \).
2. If \( v^k_i(L)(x) = d \) then either \( \text{LVal}(B)(x) = d \) or, \( \text{LVal}(B)(x) = \emptyset \) and \( \text{rank}(x, d) \leq \phi_E \).
3. If \( v^k_i(L)(x) = \emptyset \) and \( \text{rank}_X(x) > v^k_i(\phi_E) \) then \( \text{LVal}(B)(x) = \emptyset \) and \( \text{rank}_X(x) > \phi_E \).
4. For any variable \( x \), \( \max(\phi_E, \phi_L(s)(x)) \leq \max(v^k_i(\phi_E), v^k_i(\phi_L)(x)) \).

**Lemma 5.3.** If \( \rho \) is an initial run in \( \mathcal{T}_P \) then for each view \( v^k_i \) in \( \rho \) there is a scene \( s \) that is reachable in \( T_{PB} \) such that \( \rho \models^k_i s \).

**Proof.** Appendix E.

**Proof of lemma 5.3** Assume that \( q \) is reachable in \( \mathcal{T}_P \) then there is an initial run \( \rho \) in \( \mathcal{T}_P \) such that \( \text{last}(\rho)(q) = q \). By lemma 5.3 we get there is a scene \( s \) that is reachable in \( T_{PB} \) and that \( \rho \models^k_i s \), where \( v^k_i = \text{last}(\rho) \). By the definition of the simulation relation we get \( s(q) = q \), and thus \( q \) is reachable in \( T_{PB} \).

### 5.4 From \( T_{PB} \) To \( \mathcal{T}_P \)

In this subsection we prove the following lemma:

**Lemma 5.4.** If a process state \( q \) in a PDS \( \mathcal{P} \) is reachable in \( T_{PB} \) then it is reachable in \( \mathcal{T}_P \).

The proof of this lemma relies on the following simulation relation.

The Simulation Relation. For a run \( \rho \) in \( T_{PB} \) and a scene \( v = \langle q, S, \text{rank}, L, \phi_E, \phi_L, \phi_P \rangle \), we say that \( v \) simulates the scene \( s^k_i \), in \( \rho \), denoted by \( \rho \models^k_i v \) if the following holds for any variable \( x \) and any value \( d \):

1. \( q = s^k_i(q) \), \( S = s^k_i(S) \), \( \phi_P = s^k_i(\phi_P) \) and \( \text{rank} = s^k_i(\text{rank}) \).
2. If \( \text{LVal}(s^k_i(B))(x) = d \) then \( L(x) = d \).
3. If \( \text{LVal}(s^k_i(B))(x) = \emptyset \) and \( \text{rank}_X(x) > s^k_i(\phi_E) \) then \( L(x) = \emptyset \) and \( \text{rank}_X(x) > \phi_E \).
4. \( \max(\phi_E, \phi_L(x)) \leq \max(s^k_i(\phi_E), \phi_L(s^k_i)(x)) \).

**Lemma 5.5.** Let \( \rho \) be an initial run in \( T_{PB} \) with a ranking function \( \text{rank} \) and let \( s^k_i \) be a scene in the \( k \)-supplier in \( \rho \) where for all \( j \leq i \) we have \( v^k_j \neq w(x, d) \) if \( \text{rank}(x, d) = k \). Then there is a view \( v \) that is reachable in \( \mathcal{T}_P \) such that \( \rho \models^k_i v \).

**Proof.** Appendix F. 

\[ \square \]
Proof of lemma 6.4 Assume that \( q \) is reachable in \( T_{PB} \) then there is a run \( \rho \) in \( T_p \) such that \( \text{last}(\rho)(q) = q \). Given that the last supplier is the \( k \)-supplier, we know that it cannot have a write operation of rank \( k \). By lemma 6.3 we get there is a view \( v \) that is reachable in \( T_p \) and that \( \rho \vDash^k v \), where \( s_i^k = \text{last}(\rho) \). By the definition of the simulation relation we get \( v(q) = q \), and thus \( q \) is reachable in \( T_p \).

6 ExpTime-Completeness

In this section we study the complexity of the state reachability problem under PTSo and show that it is ExpTime-complete. For the upper bound we show PSRP is ExpTime by simulating the pivot semantics with a register machine of exponential size in terms of \( \mathcal{A} \). For the lower bound, we consider the problem of checking the emptiness of the intersection of a context-free language with a fixed number of regular languages. This problem is ExpTime-complete, and by reducing it to PSRP we show that PSRP is ExpTime-complete.

6.1 Register Machine

A register machine \( \mathcal{R} \) is a tuple \( \mathcal{R} = (Q, q_{init}, \Gamma, R, V, \text{Act}_R, \Delta) \), where \( Q \) is a finite set of states, \( q_{init} \in Q \) is the initial state, \( \Gamma \) is the stack alphabet, \( R \) is a finite set of registers, \( V \subset \mathbb{N} \) is a finite set of values, \( \text{Act}_R \) is a finite set of actions and \( \Delta \subseteq Q \times \text{Act}_R \times Q \) is a transition relation. For \( r_1, r_2 \in R, v \in V \) and \( \gamma \in \Gamma \) we define the action set \( \text{Act}_R \) to be the smallest set that contains the actions \( \text{skip}, \text{push}(\gamma), \text{pop}(\gamma), \text{r}(r_1, v), \text{w}(r_1, v), \text{comp}(r_1, r_2), \text{compg}(r_1, r_2), \text{compg}\{r_1, r_2\}, \text{rl}(r_1, v), \text{addr}(r_1) \) and \( \text{wfr}(r_1, r_2) \).

The meaning of the actions \( \text{skip}, \text{push}(\gamma), \text{pop}(\gamma), \text{r}(r_1, v) \) and \( \text{w}(r_1, v) \) are as usual. The action \( \text{comp}(r_1, r_2) \) checks if the values in the registers \( r_1 \) and \( r_2 \) are the same, \( \text{compg}(r_1, r_2) \) checks if the value in the register \( r_1 \) is strictly greater than the value in the register \( r_1 \), \( \text{compg}(r_1, r_2) \) checks if the value in the register \( r_1 \) is greater than or equal to the value in the register \( r_1 \), \( \text{rl}(r_1, v) \) checks if the value in the register \( r_1 \) is less than \( v \), \( \text{addr}(r_1) \) adds 1 to the value of \( r_1 \) and \( \text{wfr}(r_1, r_2) \) reads the values in the register \( r_2 \) and writes it to the register \( r_1 \). We also assume that \( \emptyset \in V \) and use it to denote the initial value of registers.

We give the operational semantics of a register machine by a labeled transition system \( T_R \). A configuration in \( T_R \) is a tuple \( (q, S, \mathcal{R}) \), where \( q \in Q \) is the control state, \( S \in \Gamma^* \) represents the stack content and \( \mathcal{R} : R \rightarrow V \) is a total function that represents the registers content, and it is called the registers configuration. The initial configuration is the tuple \( (q_{init}, \epsilon, \mathcal{R}_\emptyset) \) where \( \mathcal{R}_\emptyset(r) = \emptyset \) for all \( r \in R \) meaning that no value has been written to any register, yet. The transition relation denoted by \( \rightarrow \) in a register machine is given in Fig.6 by a set of labeled inference rules, where the labels are exactly the ones in \( \text{Act}_R \) and the rules explain how these actions are executed.
For a state \( q \in Q \), we say \( q \) is reachable under the register machine semantics if the configuration \( \langle q, S, R \rangle \), for some \( S \) and \( R \) is reachable from the initial configuration \( \langle q_{\text{init}}, \epsilon, R_{\emptyset} \rangle \) under the register machine semantics.

\[
\begin{align*}
\text{skip} & \quad \langle q, \text{skip}, q' \rangle \in \Delta \\
\text{read} & \quad \langle q, r(v), q' \rangle \in \Delta, \ R(v) = v \\
\text{write} & \quad \langle q, w(v), q' \rangle \in \Delta \\
\text{push} & \quad \langle q, \text{push}(\gamma), q' \rangle \in \Delta \\
\text{pop} & \quad \langle q, \text{pop}(\gamma), q' \rangle \in \Delta, \ S = \gamma \cdot S' \\
\text{read-less} & \quad \langle q, \text{rl}(r, v), q' \rangle \in \Delta \\
\text{write-from-register} & \quad \langle q, \text{wfr}(r_1, r_2), q' \rangle \in \Delta \\
\text{add-to-register} & \quad \langle q, \text{addr}(r), q' \rangle \in \Delta
\end{align*}
\]

Figure 6: The Transition Relation of \( T_R \).

The State Reachability problem under \( T_R \). For a register machine \( \mathcal{R} = (Q, q_{\text{init}}, \Gamma, R, V, Act_R, \Delta_R) \) and a control state \( q \in Q \), the problem asks whether there is an initial run in \( T_R \) that reaches a configuration \( c \), where \( c(q) = q \).

**Lemma 6.1.** For a register machine \( \mathcal{R} = (Q, q_{\text{init}}, \Gamma, R, V, Act_R, \Delta) \), the state reachability problem under \( T_R \) has a time complexity of the order \( O(|Q| + |\Delta| \cdot |V|^{|R|}) \).

**Proof.** Appendix G.

### 6.2 From \( T_P \) to \( T_R \)

In this section we consider the state reachability problem under \( T_P \) and reduce it to the state reachability problem under \( T_R \). The idea is simple, we save the pointers and the functions in the view in registers, and then use the actions on registers in order to simulate the events in the views.

Let us assume we have a \( T_P \) induced by a set of variables \( X \) and a set of value \( D \) and a PDS \( \mathcal{P} = (Q, q_{\text{init}}, \Gamma, Act, \Delta) \). We want to build a register machine...
\( R = (Q', q'_{\text{init}}, \Gamma, R, V, \text{Act}_R, \Delta') \) such that for a certain \( q \in Q \) there is \( q' \in Q' \) and \( q \) is reachable under \( T_P \) iff \( q' \) is reachable under \( T_R \).

**The Construction.** We start by constructing the set of registers and their values. Given the sets \( X \) and \( D \), let the set of registers \( R = \{ L(x), \phi_L(x), \text{rank}_X(x), \text{rank}(a), a, \phi_{L_{\text{max}}}, \phi_E, \phi_P, \text{rank}_{\text{next}} \mid x \in X, a \in \mathcal{A} \} \). Let the set of values \( V = D \cup \{0, 1, \ldots, |\mathcal{A}| \} \cup \{ \emptyset \} \) where \( \emptyset \notin D \) and \( \emptyset \notin \{0, 1, \ldots, |\mathcal{A}|\} \). As a result we get that both \(|R|\) and \(|V|\) are \( O(|\mathcal{A}|) \).

The register machine consists of two components the initializer and the simulator. We start by constructing the initializer. The initializer has two phases. In the first phase, it writes the values 0, 0, 1 and 1 to the registers \( \phi_E, \phi_{L_{\text{max}}}, \text{rank}_{\text{next}} \) and \( \phi_P \), respectively. It also writes 0 to the register \( \phi_L(x) \) for every \( x \in X \). In the second phase, it initializes the values of the registers labeled \( \text{rank}_X(x) \) and \( \text{rank}(a) \). Notice that the labels \( \text{rank}(a) \) and \( \text{rank}_X(x) \) simulate the functions \( \text{rank} \) and \( \text{rank}_X \) for some sequence of pivots. In other words, in the second phase the initializer tries to choose a sequence of ranking for the assignments.

To do this, the initializer needs to satisfy four constraints: (i) the initializer must distribute the assignments in a strict ascending order, which means that if \( r_1 < r_2 \) then rank \( r_1 \) must be given to some assignment before \( r_2 \). (ii) The initializer is able to consider the assignments in any random order when it assigns to them their ranks. (iii) The initializer can leave any assignment without ranks, however no assignment is allowed to be ranked twice. (iv) Finally, the initializer must initiates the labels \( \text{rank}_X(x) \) in accordance to the values given to the labels \( \text{rank}(a) \) such that the value of \( \text{rank}_X(x) \) is \( \emptyset \) iff all the assignments of \( x \) are not ranked, otherwise the value of \( \text{rank}_X(x) \) must be equal to the smallest value of a register \( \text{rank}(a) \) where \( a \) is an assignment of \( x \). To satisfy all these constraints, we let the initializer in its second phase have a state for each assignment in \( \mathcal{A} \). We build a click such that any transition toward the state \( a_n \) must go through a sub-automaton \( P_n \). The automaton \( P_n \), drawn in Fig. 8 checks that the assignment \( a_n \) has not yet been considered by the initializer in its second phase, and it does so by reading the value of the check register \( a_n \) to be equal to \( \emptyset \), it then writes 1 to the register \( a_n \). The automaton \( P_n \) then gives the assignment \( a_n \) the current available rank stored in the register \( \text{rank}_{\text{next}} \). It then checks the value of the register \( \text{rank}_X(x) \) which can either be \( \emptyset \) or be strictly less than the value in \( \text{rank}(a_n) \) and this is because the initializer gives ranks in an ascending order so if \( \text{rank}_X(x) \) has been updated by the initializer in a previous iteration then it must have got a smaller value than the value given to \( \text{rank}(a_n) \). If the value of \( \text{rank}_X(x) \) is \( \emptyset \) then the register \( \text{rank}_X(x) \) is given the same value given to the register \( \text{rank}(a_n) \). The initializer finishes this iteration by adding 1 to the value in the register \( \text{rank}_{\text{next}} \), which always stores the next available rank. Notice that the first constraint is satisfied because the next available rank is stored in \( \text{rank}_{\text{next}} \) and is increased by 1; therefore no rank is skipped. The second constraint is satisfied by the click construction. The third and the fourth constraints are satisfied by the sub-automaton \( P_n \). At
any point, the initializer can decide to stop ranking assignments and take the
skip transition to the end state as shown in Fig. 7.

The size of the initializer’s first phase is $O(|X|)$. Since the size of a click of $n$
vertices is $O(n^2)$, we get that the size of the initializer’s second phase is $O(|\mathcal{A}|^2)$,
and therefore the size of the initializer is $O(|\mathcal{A}|^2)$.

We state formally the construction of the simulator component based on the
definition of $\mathcal{P}$.

1. For each control state $q \in Q$ add a copy of it $q^c$ to $Q'$.

2. Add a state $q_{default}$ to $Q'$, and add a sequence of transitions that starts at
$q_{default}$ and ends at $q_{init}'$ to $\Delta'$. Let this sequence of transitions repeats the
first phase in the initializer, except for that it replaces the action $w(\phi_P, 1)$
by the action $\text{addr}(\phi_P)$. This sequence is necessary to simulate the rule
$\text{write}(2)$ in the pivot semantics.

3. For each transition $\langle q, \text{act}, q' \rangle \in \Delta$, if $\text{act.opr} \in \{\text{skip}, \text{pop}, \text{push}\}$ then
6.2 From $\mathcal{T}_P$ to $\mathcal{T}_R$

add the transition $\langle q', act, q'' \rangle$ to $\Delta'$.

4. For each transition $\langle q, r(x, 0), q' \rangle \in \Delta$ add a new control state $q_1$ to $Q'$, and add two transitions, $\langle q', r(L(x), \emptyset), q_1 \rangle$ and $\langle q_1, \text{compge}(\text{rank}(x), E), q'' \rangle$ to $\Delta'$. Note that this simulates read(2) in the pivot semantics.

5. For each transition $\langle q, r(x, d), q' \rangle \in \Delta$ add 6 new states $\{q_1, \ldots, q_6\}$ to $Q'$. To simulate read(1) in the pivot semantics, add the transition $\langle q', r(L(x), d), q''' \rangle$ to $\Delta'$. To simulate read(3) in the pivot semantics add 9 new transitions to $\Delta'$. The first transition is $\langle q', \text{compge}(\phi_P, \text{rank}(x, d)), q_1 \rangle$. The remaining 8 transitions ensures that $\phi_E$ is updated to $\text{max}(\phi_E, \phi_L(x), \text{rank}(x, d))$, and they are as follows:

- $\langle q_1, \text{compge}(\phi_E, \phi_L(x)), q_2 \rangle$,
- $\langle q_2, \text{compge}(\phi_E, r(E)), q'' \rangle$,
- $\langle q_1, \text{compge}(\phi_L(x), \phi_E), q_3 \rangle$,
- $\langle q_3, \text{compge}(\phi_L(x), \text{rank}(x, d)), q_4 \rangle$,
- $\langle q_4, \text{wfr}(\phi_E, \phi_L(x)), q''' \rangle$,
- $\langle q_1, \text{compge}(\phi_L(x), \phi_E), q_5 \rangle$,
- $\langle q_5, \text{compge}(\phi_L(x), \phi_E), q_6 \rangle$,
- $\langle q_6, \text{wfr}(\phi_E, \phi_L(x), d), q'''' \rangle$.

6. For each transition $\langle q, w(x, d), q' \rangle \in \Delta$, to simulate write(2) add the transition $\langle q', \text{compge}(\text{rank}(x, d), \phi_P), q_{\text{default}} \rangle$. To simulate write(1), add 4 new states $\{q_1, \ldots, q_4\}$ to $Q'$, and 6 new transitions to $\Delta'$ which simulate the check that the rank of the write operation is less than the progress pointer $\phi_P$, and simulates the update of the $L(x)$ and the update of the local pointer $\phi_L(x)$. The 6 transitions are as follows:

- $\langle q', \text{compge}(\phi_P, \text{rank}(x, d)), q_1 \rangle$,
- $\langle q_1, w(L(x), d), q_2 \rangle$,
- $\langle q_2, \text{compge}(\text{rank}(x, d), \phi_{\text{max}}), q_3 \rangle$,
- $\langle q_3, \text{wfr}(\phi_{\text{max}}, \text{rank}(x, d)), q_4 \rangle$,
- $\langle q_4, \text{compge}(\phi_{\text{max}}, \text{rank}(x, d)), q_5 \rangle$,
- $\langle q_5, \text{wfr}(\phi_L(x), \phi_{\text{max}}), q_6 \rangle$.

7. For each transition $\langle q, mf, q' \rangle \in \Delta$, to simulate the memory-fence rule, add a new state $q_1$ to $Q'$ and 3 new transitions to $\Delta'$. The transitions are as follows:

- $\langle q', \text{compge}(\phi_E, \phi_{\text{max}}), q''' \rangle$,
- $\langle q', \text{compge}(\phi_E, \phi_{\text{max}}), q'''' \rangle$,
- $\langle q', \text{wfr}(\phi_{\text{max}}, \phi_E), q''' \rangle$.

The size of the simulator is $O(|\mathcal{P}|)$. Consequently, $|Q'| + |\Delta'|$ is $O(|\mathcal{P}| + |A|^2)$. Since the size of both $\mathcal{R}$ and $\mathcal{V}$ is $O(|A|)$.

**Lemma 6.2.** Let $\mathcal{T}_P$ be induced by a PDS $\mathcal{P}$ with a set of variables $\mathbb{X}$ and a value domain $\mathbb{D}$. Then, there is a register machine $\mathcal{R} = (Q', q'_0, \Gamma, R, V, \text{Act}_R, \Delta')$ such that:

1. $|R|$ and $|V|$ are $O(|A|)$, and $|\mathcal{P}'|$ is $|Q'| + |\Delta'|$ is $O(|\mathcal{P}| + |A|^2)$.

2. For certain $q$ in $Q'$, there is a state $q'$ in $Q'$ such that $q$ is reachable in $\mathcal{T}_P$ iff $q'$ is reachable in $\mathcal{T}_R$.

**Corollary 6.2.1.** The state reachability problem under $\mathcal{T}_P$ induced by a PDS $\mathcal{P}$ with a set of variables $\mathbb{X}$ and a value domain $\mathbb{D}$, has an upper bound $O(|\mathcal{P}| \cdot 2^{|A|^2})$. 

Proof. By lemma 6.1 and lemma 6.2, we get that the parameterized state reachability problem under TSO has a time complexity $O(|P| + |A|^2 \cdot |A|^{|A|}) \subseteq O(|P| \cdot |A|^{|A|}) \subseteq O(|P| \cdot 2^{|A|^2})$.

6.3 Exptime-hard

In this section we consider the problem of checking the emptiness of the intersection between a context-free language and a fixed number of regular languages defined over an alphabet $A$. This problem is Exptime-complete [2], and we reduce it to the state reachability problem under $T_{PSO}$, which shows that PSRP is Exptime-hard. Since we have already showed that PVSRP is Exptime, we get that PSRP is Exptime-complete.

We define a context-free language $L$ through pushdown automata $P$ that accepts it. A pushdown automata (PA) is a tuple $(Q, q_{init}, q_{acc}, \Sigma, \Gamma, \Delta)$, where $Q$ is a finite set of states, $q_{init}$ is the initial state and $q_{acc}$ is called the accepting state. $\Sigma$ is the automata alphabet, sometimes it is called either input or output alphabet, and $\Gamma$ is the stack alphabet. $\Delta$ is the transition relation and it is divided into two disjoint subsets $\Delta_s$ and $\Delta_p$. $\Delta_s$ is called the stack transitions subset and it is a subset of $Q \times \{\text{push}(\Gamma), \text{pop}(\Gamma)\} \times Q$. $\Delta_p$ is called the production transitions subset and it is a subset of $Q \times \Sigma \times Q$. The size of the PA $P$ denoted by $|P|$ is the sum of the number of all states and the number of all transitions i.e $|Q| + |\Delta|$. We define the semantics of a pushdown automata through a LTS $T_{PA}$, where a configuration in $T_{PA}$ is a tuple $(q, S, w)$. The configuration means that, at the start, the automaton is at its initial state, the stack is empty and no symbol has been produced. The transition relation in $T_{PA}$, denoted by $\rightarrow_{T_{PA}}$, is given in Fig.9, where the set of labels contains push($\gamma$), pop($\gamma$) and $\sigma$ for $\gamma \in \Gamma$ and $\sigma \in \Sigma$. The rules of push and pop just simulate the action of pushing and popping a stack symbol to and from the stack. The rule prod simulates the automata producing a new alphabet symbol.

We say that a word $w \in \Sigma^*$ is accepted by the PA $P$ if there is an initial run $\rho$ in $T_{PA}$ that ends with a configuration with the state $q_{acc}$ and the produced
string is $w$. We say the language $L$ is characterized by a PA $P$ if $w \in L$ iff $w$ is accepted by $P$. A finite state automata (FA) is a special case of PA where the transition relation $\Delta$ consists only of the subset $\Delta_p$. A language characterized by an FA is called a regular language, while a language characterized by a PA is called a context-free language. Clearly, the set of regular languages is a subset of the set of context-free languages.

The Intersection of a Context-free and Regular Languages Problem. Given a PA $P$ and a set of $n$ different FAs $\{F_1, \ldots, F_n\}$, we ask whether the intersection of the all the languages characterized by $P, F_1, \ldots, F_n$ is empty.

In the next lemma we construct $\mathcal{T}_{PTSO}$ induced a PDS $P$ with a target state $q_{\text{tar}}$. The PDS $P$ simulates the PA $P$ and simultaneously checks if the last produced symbol by $P$ can be produced by all the other FAs from their current states. It keeps the simulation going until the PA $P$ reaches its accepting state and chooses non-deterministically to stop producing new symbols. The PDS then checks that all other FAs are also in their accepting states, which means that all of them were able to produce and accept the same word as the PA $P$. In the last phase, a second instance of the PDS $P$ does a sanity check which consists of checking that none of the variables in the memory got updated, and if the verification is successful then it moves to the target state $q_{\text{tar}}$. The sanity check is necessary due to the unbounded supply principle which implies that if a value is updated to the memory then it will remain there forever. We explain the construction of $P$ in $\mathcal{T}_{PTSO}$ with further details in the next lemma.

Lemma 6.3. For a PA $P = (Q^P, q_{\text{init}}^P, q_{\text{acc}}^P, \Sigma^P, \Gamma, \Delta^P)$ and FAs $\{F_1, \ldots, F_n\}$, there is a PDS $P = (Q^P, q_{\text{init}}^P, \text{Act}, \Gamma, \Delta^P)$ and target state $q_{\text{tar}}$ in $Q^P$ such that:

1. $|P|$ is polynomial in terms of $|P|, |F_1|, \ldots, |F_n|$.
2. There is a state $q_{\text{tar}}$ in $P$ such that $q_{\text{tar}}$ is reachable in $\mathcal{T}_{PTSO}$ iff the intersection of the languages characterized by $P$ and $\{F_1, \ldots, F_n\}$ is not empty.

Proof. The PDS $P$ consists of two components, a simulator and a sanity checker. The simulator is responsible for simulating the production of the same string by each of one of the automata. The sanity checker is responsible for checking that no value is updated to the memory. This is important because due to the unbounded supply principle in PTSO, if a value reaches a memory then it is always available.

We start by defining the the set of variables $X$ and the value domain $D$. The set of variables $X$ contains $n + 3$ variables $\{x^P, x^{F_1}, \ldots, x^{F_n}, x_{\text{next}}, x_{\text{check}}\}$. The variables $x^P, x^{F_1}, \ldots, x^{F_n}$ will store the current state in $P, F_1, \ldots, F_n$, respectively. The variable $x_{\text{next}}$ will store the next symbol to be produced, and the variable $x_{\text{check}}$ will be used to trigger the sanity checker. The value domain $D$ is $\{0, 1\} \cup \Sigma_{\text{all}} \cup Q_{\text{all}}$, where $\Sigma_{\text{all}}$ is the union of all the automata alphabets and $Q_{\text{all}}$ is the union of all the sets of states. The initial values of the variables are $\{q_{\text{init}}^P, q_{\text{init}}^{F_1}, \ldots, q_{\text{init}}^{F_n}, 0, 0\}$.
Corollary 6.3.1. The parameterized state reachability problem under TSO is EXPSPACE-complete.
7 Conclusion

In this paper, we have extended the pivot semantics to consider the case of pushdown systems which communicate through a set of shared variables under TSO. We have more specifically shown that the parameterized state reachability problem under TSO is decidable and it is EXPTIME-complete when the processes are instances of the same push down system. This work is relevant because it shows that it is straight forward to extend the pivot semantics to simulate any concurrent program in the parameterized case under TSO as long as the processes inside the program communicate only through a set of shared variables. Therefore, for future works we believe it is possible to use the pivot semantics to extract a general result which given the type of processes in the concurrent program, the result gives the the upper bound on the complexity of the parameterized state reachability problem under TSO.

A Proof of Lemma 4.2

Proof. We prove this by induction over $i$ the index of the configuration $c_i$. For the base case, where $i = 0$, the four conclusions follows trivially from the definitions of the initial scenes in $T_{PB}$ and the initial configuration in $T_{PTSO}$.

For the inductive step, assume that the four conclusions holds for $0 \leq i < m$. We show that the four conclusions must also hold for $c_{m+1}$. Let $\lambda_{m+1} \cdot pro = \text{c}'$ and $\lambda_{m+1} \cdot opr = \alpha'$. The first three conclusions follow immediately from line 11 in Algorithm 1. For the fourth conclusion, we consider the scene $s$ that will be added to $Sup[k]$ under an iteration induced by $\lambda_{m+1}$. We observe that $\text{last}(Sup[k])$ must simulate a configuration $c_i$. By our assumption and the fact that $i < m+1$ we get that $\text{last}(Sup[k])(\phi_E) \leq \phi_E(\rho')(i) \leq \phi_E(\rho')(m+1)$. We consider the different cases for $\alpha'$:

- $\alpha' \in \{\text{skip, row(a), w(a), push(\gamma), pop(\gamma), mf} \mid a \in \mathcal{A}, \gamma \in \Gamma\}$: by lines 16 and 20 in Algorithm 1, we get $s(\phi_E) = \text{last}(Sup[k])(\phi_E)$. Therefore, $s(\phi_E) \leq \phi_E(\rho')(m+1)$.

- $\alpha' \in \{\text{rfm(a)} \mid a \in \mathcal{A}\}$: there are two cases to consider:
  - $a \in \{x, \text{init}(x) \mid x \in \mathcal{X}\}$. By line 25, we get $s(\phi_E) = \text{last}(Sup[k])(\phi_E)$. Therefore, $s(\phi_E) \leq \phi_E(\rho')(m+1)$.
  - Otherwise, $\text{rfm(a)}$ is a read from the memory of a non-initial value. Therefore, $\lambda_i \cdot opr = u(a)$, for some $i : 1 \leq i \leq m$, and as result we get $\phi_E(\rho')(m+1) \geq \text{rank}(\rho')(a)$. Since $\text{rank} = \text{rank}(\rho')$ by line 1, we get $\phi_E(\rho')(m+1) \geq \text{rank}(a)$. From lines 30 and 29, we get $s(\phi_E) = \max(\text{last}(Sup[k])(\phi_E), \text{rank}(a))$, and therefore $s(\phi_E) \leq \phi_E(\rho')(m+1)$.

- $\alpha' \in \{u(a) \mid a \in \mathcal{A}\}$: if $\text{rank} = k$ then no scene is added. If $\text{rank}(a) < k$ then by line 36 we get $s(\phi_E) = \max(\text{last}(Sup[k])(\phi_E), \text{rank}(a))$. We
also have $\phi_E(\rho')(m+1) = \max(\phi_E(\rho')(m), \text{rank}(\rho')(a))$. By line 1 we get $\text{rank}(\rho')(a) = \text{rank}(a)$. By our inductive assumption we get that $\text{last}(\text{Sup}[k])|\phi_E| < \phi_E(\rho')(i)$, for some $i : 1 \leq i \leq m$, and consequently we get $\text{last}(\text{Sup}[k])|\phi_E| < \phi_E(\rho')(m)$. Therefore, $s(\phi_E) < \phi_E(\rho')(m+1)$.

\[ \square \]

B Proof of Lemma 4.3

Proof. To prove this lemma we need to show that any transition $s' \xrightarrow{\psi} \Theta s$ in $\rho$ is a valid transition $T_{PB}$. There are two cases to consider, the first case is when $\psi$ simulates some event $\lambda_i$ in $\rho$. The second case is when $\psi = u(2)(a_k)$.

For the first case, assume $\psi$ simulates an event $\lambda_i$, where $\lambda_i \cdot \text{opr} = \alpha$ and $\lambda_i \cdot \text{pro} = \iota$, and $\psi$ is in the $k$-supplier, namely is in $\text{Sup}[k]$. We make two important observations, the first is that there is $i'$ such that $0 \leq i' < i$ and $s'$ simulates $c_{\iota}$. The second observation is that for $0 < i' < i'' < i$, if $\lambda_i \cdot \text{pro} = \iota$ then $\lambda_{i'} \cdot \text{pro} = \iota \neq \lambda_{i''} \cdot \text{pro}$. It follows that if $\lambda_i \cdot \text{pro} = \iota$ then $c_{\iota}(\Omega)(i) = c_{i-1}(\Omega)(i)$, $c_{\iota}(\mathbf{S})(i) = c_{i-1}(\mathbf{S})(i)$ and $c_{\iota}(\mathcal{B})(i) = c_{i-1}(\mathcal{B})(i)$. For each different case of $\alpha$, we show that $s' \xrightarrow{\psi} \Theta s$ is a valid transition in $T_{PB}$.

- $\alpha = \text{skip}$. By line 16 we have $\psi = \text{skip}$. In order for $s' \xrightarrow{\psi} \Theta s$ to be valid in $T_{PB}$, we must show that:
  1. $\langle s'(q), \text{skip}, s(q) \rangle \in \Delta$.
  2. $s(S) = s'(S)$.
  3. $s(B) = s'(B)$.
  4. $s(\phi_E) = s'(\phi_E)$.
  5. $s(\phi_P) = s'(\phi_P)$.

From $\lambda_i$, we know that $\langle c_{i-1}(\Omega)(i), \text{skip}, c_i(\Omega)(i) \rangle \in \Delta$. $c_{i-1}(\mathbf{B})(i) = c_i(\mathbf{B})(i)$ and $c_{i-1}(\mathbf{S})(i) = c_i(\mathbf{S})(i)$. By lemma 4.2 and our two observations, we get (i) $s'(q) = c_{\iota}(\Omega)(i) = c_{i-1}(\Omega)(i)$ and $s(q) = c_i(\Omega)(i)$; (ii) $s'(S) = c_{\iota}(\mathbf{S})(i) = c_{i-1}(\mathbf{S})(i)$ and $s(S) = c_i(\mathbf{S})(i)$; (iii) $s'(B) = c_{\iota}(\mathcal{B})(i) = c_{i-1}(\mathcal{B})(i)$ and $s(B) = c_i(\mathcal{B})(i)$. Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Also, condition 4 follows from line 13 and condition 5 follows from line 16.

- $\alpha \in \{w(a) \mid a \in A\}$. By line 16 we have $\psi = w(a)$. In order for $s' \xrightarrow{w(a)} \Theta s$ to be valid in $T_{PB}$, we must show that:
  1. $\langle s'(q), w(a), s(q) \rangle \in \Delta$.
  2. $s(S) = s'(S)$.
  3. $s(B) = a \cdot s'(B)$.
  4. $s(\phi_E) = s'(\phi_E)$.
5. $s(\phi_P) = s'(\phi_P)$.

From $\lambda_i$ we know that $\langle c_{i-1}(Q)(\iota), w(a), c_i(Q)(\iota) \rangle \in \Delta$, $c_i(B)(\iota) = a \cdot c_{i-1}(Q)(\iota)$ and $c_{i-1}(S)(\iota) = c_i(S)(\iota)$. By Lemma 4.2 and our two observations, we get (i) $s'(q) = c_{i'}(Q)(\iota) = c_{i-1}(Q)(\iota)$ and $s(q) = c_i(Q)(\iota)$; (ii) $s'(S) = c_{i'}(S)(\iota) = c_{i-1}(S)(\iota)$ and $s(S) = c_i(S)(\iota)$; (iii) $s'(B) = c_{i'}(B)(\iota) = c_{i-1}(B)(\iota)$ and $s(B) = c_i(B)(\iota)$; Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Also, condition 4 follows from line 13 and condition 5 follows from line 16.

- $\alpha \in \{\text{push}(\gamma) \mid \gamma \in \Gamma\}$. By line 16 we have $\psi = \text{push}(\gamma)$. In order for $s' \xrightarrow{\mathcal{G}} s$ to be valid in $\mathcal{T}_{PB}$, we must show that:

1. $\langle s'(q), \text{push}(\gamma), s(q) \rangle \in \Delta$.
2. $s(S) = \gamma \cdot s'(S)$.
3. $s(B) = s'(B)$.
4. $s(\phi_E) = s'(\phi_E)$.
5. $s(\phi_P) = s'(\phi_P)$.

From $\lambda_i$ we know that $\langle c_{i-1}(Q)(\iota), \text{push}(\gamma), c_i(Q)(\iota) \rangle \in \Delta$, $c_i(B)(\iota) = c_{i-1}(Q)(\iota)$ and $c_{i-1}(S)(\iota) = \gamma \cdot c_i(S)(\iota)$. By Lemma 4.2 and our two observations, we get (i) $s'(q) = c_{i'}(Q)(\iota) = c_{i-1}(Q)(\iota)$ and $s(q) = c_i(Q)(\iota)$; (ii) $s'(S) = c_{i'}(S)(\iota) = c_{i-1}(S)(\iota)$ and $s(S) = c_i(S)(\iota)$; (iii) $s'(B) = c_{i'}(B)(\iota) = c_{i-1}(B)(\iota)$ and $s(B) = c_i(B)(\iota)$; Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Also, condition 4 follows from line 13 and condition 5 follows from line 16.

- $\alpha \in \{\text{pop}(\gamma) \mid \gamma \in \Gamma\}$. By line 16 we have $\psi = \text{pop}(\gamma)$. In order for $s' \xrightarrow{\mathcal{G}} s$ to be valid in $\mathcal{T}_{PB}$, we must show that:

1. $\langle s'(q), \text{pop}(\gamma), s(q) \rangle \in \Delta$.
2. $s'(S) = \gamma \cdot s(S)$.
3. $s(B) = s'(B)$.
4. $s(\phi_E) = s'(\phi_E)$.
5. $s(\phi_P) = s'(\phi_P)$.

From $\lambda_i$ we know that $\langle c_{i-1}(Q)(\iota), \text{pop}(\gamma), c_i(Q)(\iota) \rangle \in \Delta$, $c_i(B)(\iota) = c_{i-1}(Q)(\iota)$ and $c_{i-1}(S)(\iota) = \gamma \cdot c_i(S)(\iota)$. By Lemma 4.2 and our two observations, we get (i) $s'(q) = c_{i'}(Q)(\iota) = c_{i-1}(Q)(\iota)$ and $s(q) = c_i(Q)(\iota)$; (ii) $s'(S) = c_{i'}(S)(\iota) = c_{i-1}(S)(\iota)$ and $s(S) = c_i(S)(\iota)$; (iii) $s'(B) = c_{i'}(B)(\iota) = c_{i-1}(B)(\iota)$ and $s(B) = c_i(B)(\iota)$; Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Also, condition 4 follows from line 13 and condition 5 follows from line 16.

- $\alpha = \text{mf}$. By line 16 we have $\psi = \text{mf}$. In order for $s' \xrightarrow{\mathcal{G}} s$ to be valid in $\mathcal{T}_{PB}$, we must show that:
\[ \alpha \in \{ \text{row}(x,d) \mid x \in X, d \in \mathbb{D} \}. \] By line 21 we have \( \psi = r(1)(x,d) \). In order for \( s' \xrightarrow{r(1)(x,d)} s \) to be valid in \( T_{PB} \), we must show that

1. \( \langle s'(q), r(x,d), s(q) \rangle \in \Delta \).
2. \( s(S) = s'(S) \).
3. \( s(B) = s'(B) = \epsilon \).
4. \( s(\phi_E) = s'(\phi_E) \).
5. \( s(\phi_P) = s'(\phi_P) \).

From \( \lambda \), we know that \( \langle c_{i-1}(Q)(i), r(x,d), c_i(Q)(i) \rangle \in \Delta \), \( c_{i-1}(B)(i) = c_i(B)(i) = \epsilon \) and \( c_{i-1}(S)(i) = c_i(S)(i) \). By lemma 4.2 and our two observations, we get (i) \( s'(q) = c_i(Q)(i) = c_{i-1}(Q)(i) \) and \( s(q) = c_i(Q)(i) \); (ii) \( s'(S) = c_i(S)(i) \) and \( s(S) = c_i(S)(i) \); (iii) \( s'(B) = c_i(B)(i) = c_{i-1}(B)(i) \) and \( s(B) = c_i(B)(i) \). Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Also, condition 4 follows from line 13 and condition 5 follows from line 16.

- \( \alpha \in \{ \text{rfm}(a) \mid a \in A \} \). There are two cases to consider:

  - \( a \in \{ x, \text{init}(x) \} \). By line 26 we have \( \psi = r(2)(x, \text{init}(x)) \).

    In order for \( s' \xrightarrow{r(2)(x, \text{init}(x))} s \) to be valid in \( T_{PB} \), we must show that:

1. \( \langle s'(q), r(x, \text{init}(x)), s(q) \rangle \in \Delta \).
2. \( s(S) = s'(S) \).
3. \( s(B) = s'(B) = \emptyset \).
4. \( s(\phi_E) = s'(\phi_E) \) where \( \text{rank}_X(x) > s'(\phi_E) \).
5. \( s(\phi_P) = s'(\phi_P) \).
From $\lambda$, we know that $(c_{i-1}(Q)(i), r(x, \text{init}(x)), c_i(Q)(i)) \in \Delta$, $c_{i-1}(B)(i) = c_i(B)(i)$ where $\text{LVal}(s'(B)(i))(x) = \emptyset$, and $c_{i-1}(S)(i) = c_i(S)(i)$. By lemma 12 and our two observations, we get (i) $s'(q) = c_i(Q)(i) = c_{i-1}(Q)(i)$ and $s(q) = c_i(Q)(i)$; (ii) $s'(S) = c_i(S)(i) = c_{i-1}(S)(i)$ and $s(S) = c_i(S)(i)$; (iii) $s'(B) = c_i(B)(i) = c_{i-1}(B)(i)$ and $s(B) = c_i(B)(i)$. Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. For condition 4, we notice that $\text{rank}_\xi(x) = \text{rank}_\xi(\rho')(x)$ by line 1. By the definition of external pointers in $T_{PSO}$, we have $\text{rank}_\xi(\rho')(x) > \phi_E(\rho')(i - 1) \geq \phi_E(\rho')(i')$, . By lemma 12, we have $s'(\phi_E) \leq \phi_E(\rho')(i')$, and therefore we get $s'(\phi_E) \leq \text{rank}_\xi(x)$. The rest of condition 4 follows from line 13, and condition 5 follows from line 25.

$- \text{rfm}(a)$ is a read from the memory of a non-initial value. By line 31, $\psi = r(3)(a)$. In order for $s' \overset{r(3)(a)}{\rightarrow} \emptyset s$ to be valid in $T_{PB}$, we must show that:

1. $(s'(q), r(a), s(q)) \in \Delta$.
2. $s(S) = s'(S)$.
3. $s(B) = s'(B)$ and $\text{LVal}(s'(B))(x) = \emptyset$.
4. $s(\phi_E) = \max(s'(\phi_E), \text{rank}(a))$.
5. $s(\phi_P) = s'(\phi_P)$ where $\text{rank}(a) < s'(\phi_P)$.

From $\lambda$, we know that $(c_{i-1}(Q)(i), r(a), c_i(Q)(i)) \in \Delta$, $c_{i-1}(B)(i) = c_i(B)(i)$ where $\text{LVal}(s'(B)(i))(x) = \emptyset$, and $c_{i-1}(S)(i) = c_i(S)(i)$. By lemma 12 and our two observations, we get (i) $s'(q) = c_i(Q)(i) = c_{i-1}(Q)(i)$ and $s(q) = c_i(Q)(i)$; (ii) $s'(S) = c_i(S)(i) = c_{i-1}(S)(i)$ and $s(S) = c_i(S)(i)$; (iii) $s'(B) = c_i(B)(i) = c_{i-1}(B)(i)$ and $s(B) = c_i(B)(i)$. Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Condition 4 follows from line 29. To show that $\text{rank}(a) < s'(\phi_P)$, we do a proof by contradiction. Since $s'$ is in the $k$-supplier then $s'(\phi_P) = k$. Assume $\text{rank}(a) \geq k$ then $\text{rank}(\rho')(a) \geq k$ and $i > \text{pivots}(\rho')(k)$. By lines 14 and 41 we see that the $k$-supplier in $\rho$ does not include an event $\psi$ that simulate an event $\lambda_i$ where $i \geq \text{pivots}(\rho')(k)$. Therefore, $\text{rank}(a) < k$. The rest of condition 5 follows from line 30.

$\bullet \alpha \in \{u(a) \mid a \in A\}$. If $\text{rank}(a) \geq k$ then as we argued earlier, we get $i \geq \text{pivots}(\rho')(k)$ which is impossible. Therefore, we only consider the case where $\text{rank}(a) < k$. By line 38, $\psi = u(1)(a)$. In order for $s' \overset{u(3)(a)}{\rightarrow} \emptyset s$ to be valid in $T_{PB}$, we need to show that:

1. $s(q) = s'(s)$.
2. $s(S) = s'(S)$.
3. $s'(B) = s(B) \cdot a$.
4. $s(\phi_E) = \max(s'(\phi_E), \text{rank}(a))$. 


5. $s(\phi_P) = s'(\phi_P)$.

From $\lambda_i$ we know that $c_{i-1}(Q)(i) = c_i(Q)(i)$, $c_{i-1}(B)(i) = c_i(B)(i) \cdot a$ where, and $c_{i-1}(S)(i) = c_i(S)(i)$. By lemma 4.2 and our two observations, we get (i) $s'(q) = c_i(Q)(i) = c_{i-1}(Q)(i)$ and $s(q) = c_i(Q)(i)$; (ii) $s'(S) = c_i(S)(i) = c_{i-1}(S)(i)$ and $s(S) = c_i(S)(i)$; (iii) $s'(B) = c_i(B)(i) = c_{i-1}(B)(i)$ and $s(B) = c_i(B)(i)$. Therefore, the conditions 1, 2 and 3 follow from (i), (ii) and (iii), respectively. Condition 4 follows from line 36, and condition 5 follows from line 37.

We consider now the second case where $\psi = u(2)(a_k)$. If $s' \xrightarrow{u(2)(a_k)} \sigma$, $s$ is in $\rho$, then $s'$ is the last scene added to the $k$-supplier $S^{\rho}[k]$. In order for $s' \xrightarrow{u(2)(a_k)} \sigma$ to be valid in $T_{PB}$, we need to show that $s$ is $s_{init}(\text{rank}, k + 1)$, which follows from lines 6 and 46. We also need to show $\text{rank}(a_k) = k$ and this follows from the definition of $a_k$. Finally we need to show that $s'(B) = w \cdot a_k$, where $w \in A^\ast$. Assume that $k \in \text{rank}(\rho')(i)$ and $\lambda_i$ is the last event done by $i$ before $\text{pivots}(\rho')(i)$, then we get that $s'$ simulate $c_i$ and $c_i(B(i)) = w \cdot a_k$. By lemma 4.2 we get $s'(B) = w \cdot a_k$.

C Proof of Lemma 4.4

Proof. For the base case assume $i = 0$, then by definition we have $c_m = c_{init}$. The four conclusions follow from the definitions of the initial configuration in $T_{PTSO}$ and the initial scenes in $T_{PB}$, in addition to the fact that $\phi_E(\rho)(0) = 0$.

For the inductive step, assume that the lemma holds for $0 \leq i \leq j$. We prove that the lemma holds for $j + 1$. The first three conclusions follow immediately from line 5 in Algorithm 2. For the fourth conclusion we consider the different cases for $\psi_{j+1}^k$. To make the argument clear, we let $\rho_1$ be the run constructed by Algorithm 2 before the iteration induced by the event $\psi_{j+1}^k$, and we let $\rho_2$ be the constructed run after the iteration and finally we let $\rho$ be the final constructed run. We denote the position of the configuration that simulates $s_{j+1}^k$ by $m$ and $m'$ in $\rho$ and $\rho_2$, respectively. We also notice that $c_{\text{index}}$ simulates $s_j^k$ in $\rho_1$ and $\rho_2$, and we let $\text{index}'$ be the position of the configuration that simulates $s_j^k$ in $\rho$. Furthermore from previous observations we know that $\phi_E(\rho)(m) = \phi_E(\rho_2)(m')$ and $\phi_E(\rho_2)(\text{index}) = \phi_E(\rho)(\text{index}')$. We have

- $\psi_{j+1}^k \in \{\text{skip}, r(1)(x, d), r(2)(x, init(x)), u(x, d), \text{push}(\gamma), \text{pop}(\gamma), \text{mf} \mid x \in \mathbb{X}, d \in \mathbb{D}, \gamma \in \Gamma\}$: we want to show $\phi_E(\rho)(m) \leq s_{j+1}^k(\phi_E)$. To do this we show $\phi_E(\rho)(m) = \phi_E(\rho)(\text{index}')$ and $s_{j+1}^k(\phi_E) = s_j^k(\phi_E)$. The result follows from the inductive assumption which states that $\phi_E(\rho)(\text{index}') \leq s_j^k(\phi_E)$.

By the semantics of $T_{PB}$ we get $s_{j+1}^k(\phi_E) = s_j^k(\phi_E)$, since $\psi_{j+1}^k$ is not an update neither an $r(3)$ action. By lines 10, 16 and 22 in Algorithm 2 we get that $m' = \text{index} + 1$. Since the event inserted at $\text{index} + 1$ in all cases is not
an update we get that $\phi_E(\rho_2)(index) = \phi_E(\rho_2)(index + 1) = \phi_E(\rho_2)(m')$.
From previous observations we have $\phi_E(\rho_2)(index) = \phi_E(\rho)(index')$ and $\phi_E(\rho)(m) = \phi_E(\rho_2)(m')$. Consequently, we have $\phi_E(\rho)(m) = \phi_E(\rho)(index')$ as required.

- $\psi^{k+1}_{j+1} \in \{r(3)(a), u(1)(a) \mid a \in \mathcal{A}\}$: we want to show $\phi_E(\rho)(m) \leq s^k_{j+1}(\phi_E)$.
Since $\psi^{k+1}_{j+1} = r(3)(a)$ or $\psi^{k+1}_{j+1} = u(1)(a)$ then by the semantics of $T_{PB}$ we get $s^k_{j+1}(\phi_E) = \max(s^k_j(\phi_E), \text{rank}(a))$. By lines 26 and 32, we have $\text{prev} = \max(index, \text{first}(\rho_1)(a))$, and by lines 28 and 34 we have $m' = \text{prev} + 1$. Since the inserted action at position $m'$ is either not an update or it is an update of $a$ we get that $\phi_E(\rho_1)(\text{prev}) = \phi_E(\rho_2)(\text{prev}) = \phi_E(\rho_2)(m') = \phi_E(\rho)(m)$. Therefore to show our result, it is enough to show that $\phi_E(\rho_1)(\text{prev}) \leq s^k_{j+1}(\phi_E) = \max(s^k_j(\phi_E), \text{rank}(a))$.

Since $\text{prev} = \max(index, \text{first}(\rho_1)(a))$, we get $\phi_E(\rho_1)(\text{prev}) = \max(\phi_E(\rho_1)(index), \text{rank}(\rho_1)(a))$. From previous observations we have $\phi_E(\rho_1)(index) = \phi_E(\rho)(index') \leq s^k_j(\phi_E)$. We also have $\text{rank}(\rho_1)(a) = \text{rank}(\rho)(a) = \text{rank}(a)$. Therefore, $\max(\phi_E(\rho_1)(index), \text{rank}(\rho_1)(a)) \leq \max(s^k_j(\phi_E), \text{rank}(a))$ and thus $\phi_E(\rho)(m) \leq s^k_{j+1}(\phi_E)$ as required.

\qed

\section*{D Proof of Lemma 4.5}

\emph{Proof.} Let $\rho'$ be:

$s_0 \xrightarrow{\psi_1} s_1 \xrightarrow{\psi_2} s_2 \cdots s_m$.

Let $\rho$ be:

$c_{init} \xrightarrow{\lambda_1} c_1 \cdots \xrightarrow{\lambda_{index}} c_{index} \cdots \xrightarrow{\lambda_{prev}} c_{prev} \cdots \xrightarrow{\lambda_m} c_m$.

We do a proof by induction over the length of $\rho'$. For the base case, let $\#\rho' = 0$ then $\rho' = s_{init}(\text{rank}, 1)$ and $\rho = c_{init}$, which is a valid run in $T_{PSO}$. For the inductive step assume that $\rho$ is valid if $\#\rho' = m$. We show that if $\rho'$ is extended by one event $\psi$ to be the run $\rho'_1$:

$s_0 \xrightarrow{\psi_1} s_1 \xrightarrow{\psi_2} s_2 \cdots s_m \xrightarrow{\psi} s$,

then the constructed run $\rho_1 = \text{insert}(\rho, \lambda, c, \text{prev} + 1)$:

$c_{init} \xrightarrow{\lambda_1} c_1 \cdots \xrightarrow{\lambda_{index}} c_{index} \cdots \xrightarrow{\lambda_{prev}} c_{prev} \xrightarrow{\lambda} c \cdots \xrightarrow{\lambda_m} c'_m$

is also valid. Notice that to prove $\rho_1$ is valid, it is enough to show that the sub-run $c_{prev} \xrightarrow{\lambda} c \cdots \xrightarrow{\lambda_m} c'_m$ is valid. There are two possibilities here, it is either that the scenes $s_m$ and $s$ belong to the same supplier in $\rho'_1$, or they belong to different suppliers. We begin with the first possibility and assume that $s_m$
and $s$ belong to the same $k$-supplier. In this case, we immediately get that $c_{\text{index}}$ simulates $s_m$ and $c$ simulates $s$. Furthermore, since $\lambda_i \cdot \text{proc} \neq k$ for $i > \text{index}$ and $\lambda_i$ in $\rho$, we get $c_{\text{index}}(Q)(k) = c_{\text{prev}}(Q)(k)$, $c_{\text{index}}(S)(k) = c_{\text{prev}}(S)(k)$ and $c_{\text{index}}(B)(k) = c_{\text{prev}}(B)(k)$. By lemma 4.5 we get, $c_{\text{prev}}(Q)(k) = s_m(q)$, $c_{\text{prev}}(S)(k) = s_m(S)$ and $c_{\text{prev}}(B)(k) = s_m(B)$; we refer to this result by remark 1. We have a case to consider for each possible type of $\psi$. 

- $\psi = \text{skip}$. By line 10, we have $\lambda = \langle k, \text{skip} \rangle$. To show that $c_{\text{prev}} \xrightarrow{(k, \text{skip})} c$ is a valid transition, we need to show that:
  
  1. $c(Q) = c_{\text{prev}}(Q)[k \leftarrow q]$, where $\langle c_{\text{prev}}(Q)(k), \text{skip}, q \rangle \in \Delta$.
  2. $c(S) = c_{\text{prev}}(S)$.
  3. $c(B) = c_{\text{prev}}(B)$.
  4. $c(M) = c_{\text{prev}}(M)$.

  By lines 5 and 9 we have $c(Q) = c_{\text{prev}}(Q)[k \leftarrow s(q)]$, $c(S) = c_{\text{prev}}(S)[k \leftarrow s(S)]$, $c(B) = c_{\text{prev}}(B)[k \leftarrow s(B)]$ and $c(M) = c_{\text{prev}}(M)$. By $\psi$ we have, $\langle s_m(q), \text{skip}, s(q) \rangle \in \Delta$. Since $s_m(S) = s(S)$ and $s_m(B) = s(B)$. Thus the four conditions follow by remark 1.

  For the the rest of the run we notice that all subsequent events are done by processes different from $k$, and therefore the changes in the state, the buffer and the stack of the process $k$ do not matter. Moreover, the event $\lambda = \langle k, \text{skip} \rangle$ does not change the external pointer at subsequent events and therefore the same values are readable from memory before and after the insertion of $\lambda$. As result the rest of the run is also valid. The same argument applies for the cases where $\lambda \in \{\text{w}(a), \text{push}(\gamma), \text{pop}(\gamma), \text{row}(a), \text{rfm}(a), \text{mf}\}$. If $\lambda = \text{w}(a)$ then $\psi = \text{w}(1)(a)$ and by line 32 we have that $\lambda$ is inserted after $\text{first}(\rho)(a)$ and therefore the external pointer does not change for all subsequent events.

- $\psi \in \{\text{w}(a) \mid a \in A\}$. By line 10, we have $\lambda = \langle k, \text{w}(a) \rangle$. To show that $c_{\text{prev}} \xrightarrow{(k, \text{w}(a))} c$ is a valid transition, we need to show that:
  
  1. $c(Q) = c_{\text{prev}}(Q)[k \leftarrow q]$, where $\langle c_{\text{prev}}(Q)(k), \text{w}(a), q \rangle \in \Delta$.
  2. $c(S) = c_{\text{prev}}(S)$.
  3. $c(B) = c_{\text{prev}}(B)[k \leftarrow a \cdot c_{\text{prev}}(B)(k)]$.
  4. $c(M) = c_{\text{prev}}(M)$.

  By lines 5 and 9 we have $c(Q) = c_{\text{prev}}(Q)[k \leftarrow s(q)]$, $c(S) = c_{\text{prev}}(S)[k \leftarrow s(S)]$, $c(B) = c_{\text{prev}}(B)[k \leftarrow s(B)]$ and $c(M) = c_{\text{prev}}(M)$. By $\psi$ we have, $\langle s_m(q), \text{w}(a), s(q) \rangle \in \Delta$. Since $s_m(S) = s(S)$ and $s(B) = a \cdot s_m(B)$. Thus the four conditions follow by remark 1.

- $\psi \in \{\text{push}(\gamma) \mid \gamma \in \Gamma\}$. By line 10, we have $\lambda = \langle k, \text{push}(\gamma) \rangle$. To show that $c_{\text{prev}} \xrightarrow{(k, \text{push}(\gamma))} c$ is a valid transition, we need to show that:
D PROOF OF LEMMA 4.5

1. \( c(Q) = c_{prev}(Q)[k \leftarrow q] \), where \( \langle c_{prev}(Q)(k), \text{push} (\gamma), q \rangle \in \Delta \).
2. \( c(S) = c_{prev}(S)[k \leftarrow \gamma \cdot c_{prev}(S)(k)] \).
3. \( c(B) = c_{prev}(B) \).
4. \( c(M) = c_{prev}(M) \).

By lines 5 and 9 we have \( c(Q) = c_{prev}(Q)[k \leftarrow s(q)] \), \( c(S) = c_{prev}(S)[k \leftarrow s(S)] \), \( c(B) = c_{prev}(B)[k \leftarrow s(B)] \) and \( c(M) = c_{prev}(M) \). By \( \psi \) we have, \( \langle s_m(q), \text{push} (\gamma), s(q) \rangle \in \Delta \), \( s(S) = \gamma \cdot s_m(S) \) and \( s_m(B) = s(B) \). Thus the four conditions follow by remark 1.

- \( \psi \in \{ \text{pop} (\gamma) \mid \gamma \in \Gamma \} \). By line 10, we have \( \lambda = \langle k, \text{push} (\gamma) \rangle \). To show that \( c \) is a valid transition, we need to show that:
  1. \( c(Q) = c_{prev}(Q)[k \leftarrow q] \), where \( \langle c_{prev}(Q)(k), \text{pop} (\gamma), q \rangle \in \Delta \).
  2. \( c(S) = c_{prev}(S)[k \leftarrow S] \) where \( \gamma \cdot S = c_{prev}(S)(k) \).
  3. \( c(B) = c_{prev}(B) \).
  4. \( c(M) = c_{prev}(M) \).

By lines 5 and 9 we have \( c(Q) = c_{prev}(Q)[k \leftarrow s(q)] \), \( c(S) = c_{prev}(S)[k \leftarrow s(S)] \), \( c(B) = c_{prev}(B)[k \leftarrow s(B)] \) and \( c(M) = c_{prev}(M) \). By \( \psi \) we have, \( \langle s_m(q), \text{pop} (\gamma), s(q) \rangle \in \Delta \), \( s_m(S) = \gamma \cdot s(S) \) and \( s_m(B) = s(B) \). Thus the four conditions follow by remark 1.

- \( \psi = \text{mf} \). By line 10, we have \( \lambda = \langle k, \text{mf} \rangle \). To show that \( c \) is a valid transition, we need to show that:
  1. \( c(Q) = c_{prev}(Q)[k \leftarrow q] \), where \( \langle c_{prev}(Q)(k), \text{mf}, q \rangle \in \Delta \).
  2. \( c(S) = c_{prev}(S) \).
  3. \( c(B) = c_{prev}(B) \), where \( c_{prev}(B)(k) = \epsilon \).
  4. \( c(M) = c_{prev}(M) \).

By lines 5 and 9 we have \( c(Q) = c_{prev}(Q)[k \leftarrow s(q)] \), \( c(S) = c_{prev}(S)[k \leftarrow s(S)] \), \( c(B) = c_{prev}(B)[k \leftarrow s(B)] \) and \( c(M) = c_{prev}(M) \). By \( \psi \) we have, \( \langle s_m(q), \text{mf}, s(q) \rangle \in \Delta \), \( s_m(S) = s(S) \) and \( s_m(B) = s(B) = \epsilon \). Thus the four conditions follow by remark 1.

- \( \psi \in \{ \text{r}(1)(x,d) \mid x \in \mathcal{X}, d \in \mathbb{D} \} \). By line 16, we have \( \lambda = \langle k, \text{r}(x,d) \rangle \).

To show that \( c \) is a valid transition, we need to show that:
  1. \( c(Q) = c_{prev}(Q)[k \leftarrow q] \), where \( \langle c_{prev}(Q)(k), \text{r}(x,d), q \rangle \in \Delta \).
  2. \( c(S) = c_{prev}(S) \).
  3. \( c(B) = c_{prev}(B) \), where \( \text{LVal}(c_{prev}(B)(k))(x) = d \).
  4. \( c(M) = c_{prev}(M) \).
By lines 5 and 15 we have $c(Q) = c_{\text{prev}}(Q)[k \leftarrow s(q)]$, $c(S) = c_{\text{prev}}(S)[k \leftarrow s(S)]$, $c(B) = c_{\text{prev}}(B)[k \leftarrow s(B)]$, $c(M) = c_{\text{prev}}(M)$. By Lemma 4.5.3 we have, $(s_m(q), r(x, d), s(q)) \in \Delta$, $s_m(S) = s(S)$ and $s_m(B) = s(B)$ where $\text{LVal}(s_m(B))(x) = d$. Thus the four conditions follow by remark 1.

- $\psi \in \{r(2)(x, \text{init}(x)) \mid x \in X\}$. By line 22, we have $\lambda = (k, \text{rfm}(x, \text{init}(x)))$. To show that $c_{\text{prev}}\frac{(k, \text{rfm}(x, \text{init}(x)))}{\varnothing} c$ is a valid transition, we need to show that:
  1. $c(Q) = c_{\text{prev}}(Q)[k \leftarrow q]$, where $(c_{\text{prev}}(Q)(k), r(x, \text{init}(x)), q) \in \Delta$.
  2. $c(S) = c_{\text{prev}}(S)$.
  3. $c(B) = c_{\text{prev}}(B)$, where $\text{LVal}(c_{\text{prev}}(B)(k))(x) = \varnothing$.
  4. $c(M) = c_{\text{prev}}(M)$.
  5. $\phi_E(\rho)(\text{prev}) < \text{rank}_X(\rho)(x)$.

By lines 5 and 15 we have $c(Q) = c_{\text{prev}}(Q)[k \leftarrow s(q)]$, $c(S) = c_{\text{prev}}(S)[k \leftarrow s(S)]$, $c(B) = c_{\text{prev}}(B)[k \leftarrow s(B)]$ and $c(M) = c_{\text{prev}}(M)$. By Lemma 4.5.3 we have, $(s_m(q), r(x, d), s(q)) \in \Delta$, $s_m(S) = s(S)$ and $s_m(B) = s(B)$ where $\text{LVal}(s_m(B))(x) = \varnothing$. Thus the first four conditions follow by remark 1. By line 20 we have $c_{\text{prev}} = \text{index}$ in $\rho$. Since $\text{index}$ simulates $s_m$ then by lemma 4.5.4 we get $\phi_E(\rho)(\text{index}) \leq s_m(\phi_E)$. By Lemma 4.5.3 we have $s_m(\phi_E) < \text{rank}_X(\rho)(x)$, Thus, the fifth condition follows.

- $\psi \in \{r(3)(x, d) \mid x \in X, d \in \mathbb{D}\}$. By line 28, we have $\lambda = (k, \text{rfm}(x, d))$. To show that $c_{\text{prev}}\frac{(k, \text{rfm}(x, d))}{\varnothing} c$ is a valid transition, we need to show that:
  1. $c(Q) = c_{\text{prev}}(Q)[k \leftarrow q]$, where $(c_{\text{prev}}(Q)(k), r(x, d), q) \in \Delta$.
  2. $c(S) = c_{\text{prev}}(S)$.
  3. $c(B) = c_{\text{prev}}(B)$, where $\text{LVal}(c_{\text{prev}}(B)(k))(x) = \varnothing$.
  4. $c(M) = c_{\text{prev}}(M)$.
  5. $\phi_E(\rho)(\text{prev}) \geq \text{rank}_X(\rho)(x, d)$.

By lines 5 and 27 we have $c(Q) = c_{\text{prev}}(Q)[k \leftarrow s(q)]$, $c(S) = c_{\text{prev}}(S)[k \leftarrow s(S)]$, $c(B) = c_{\text{prev}}(B)[k \leftarrow s(B)]$ and $c(M) = c_{\text{prev}}(M)$. By Lemma 4.5.3 we have, $(s_m(q), r(x, d), s(q)) \in \Delta$, $s_m(S) = s(S)$ and $s_m(B) = s(B)$ where $\text{LVal}(s_m(B))(x) = \varnothing$. Thus the first four conditions follow by remark 1. By line 26 we have $\text{prev} \geq \text{first}(\rho)(x, d)$. Thus, the fifth condition follows.

- $\psi \in \{u(1)(x, d) \mid x \in X, d \in \mathbb{D}\}$. By line 28, we have $\lambda = (k, u(x, d))$. To show that $c_{\text{prev}}\frac{(k, u(x, d))}{\varnothing} c$ is a valid transition, we need to show that:
  1. $c(Q) = c_{\text{prev}}(Q)$.
  2. $c(S) = c_{\text{prev}}(S)$. 


3. \( c(B) = c_{prev}(B)[k \leftarrow B] \), where \( c_{prev}(B)(k) = \langle x, d \rangle \cdot B \).

4. \( c(M) = c_{prev}(M)[x \leftarrow d] \).

By lines 5 and 33 we have \( c(Q) = c_{prev}(Q)[k \leftarrow s(q)] \), \( c(S) = c_{prev}(S)[k \leftarrow s(S)] \), \( c(B) = c_{prev}(B)[k \leftarrow s(B)] \) and \( c(M) = c_{prev}(M)[x \leftarrow d] \). By \( \psi \) we have, \( s_m(q) = s(q) \), \( s_m(S) = s(S) \) and \( s_m(B) = \langle x, d \rangle \cdot s(B) \). Thus the four conditions follow by remark 1.

We now consider the second case, namely when \( s_m \) belongs to the \( k \)-supplier and \( s \) belongs to the \((k + 1)\)-supplier. In this case, \( \psi = u(2)(a_k) \), where \( rank(a_k) = k \). Let \( a_k = \langle x, d \rangle \). By line 38, we have \( \lambda = \langle k, u(x, d) \rangle \). By line 40, we have \( c_{prev} = c_m \) and therefore in order to show \( \rho_1 \) to be valid we only need to show that \( c_{prev}(k,u(x,d)) \cap c \). To show this we need to show:

1. \( c(Q) = c_{prev}(Q) \).
2. \( c(S) = c_{prev}(S) \).
3. \( c(B) = c_{prev}(B)[k \leftarrow B] \), where \( c_{prev}(B)(k) = \langle x, d \rangle \cdot B \).
4. \( c(M) = c_{prev}(M)[x \leftarrow d] \).

By \( \psi \) we have \( s_m(B) = B'' \cdot \langle x, d \rangle \). Also by lemma 4.4 we have \( c_{prev}(B)(k) = s_m(B) \) because \( c_{index} \) simulates \( s_m \) and \( prev \geq index \). By lines 6 and 39, we have \( c(Q) = c_{prev}(Q) \), \( c(S) = c_{prev}(S) \), \( c(B) = c_{prev}(B)[k \leftarrow B''] \) and \( c(M) = c_{prev}(M)[x \leftarrow d] \). Thus the four conditions follow.

\[ \square \]

E  Proof of Lemma 5.3

Proof. We fix the function \( rank \) to be the ranking function in \( \rho \). We do a proof by induction over all the views in \( \rho \). For the base case, we consider \( i = 0 \) and \( k = 1 \). Since \( \rho \) is an initial run we get that \( v^{1}_0 = v_{init}(rank, 1) \).

We also know that the scene \( s_{init}(rank, 1) \) is reachable in \( TP_B \). To show that \( \rho \models^{1}_0 s_{init}(rank, 1) \), we check the four conditions. Let \( v = v_{init}(rank, 1) \) and \( s = s_{init}(rank, 1) \) in the following:

1. The first condition follows as, \( s(q) = q_{init} = v(q) \), \( s(S) = \epsilon = v(S) \) \( s(\phi_P) = 1 = v(\phi_P) \) and \( s(rank) = rank = v(rank) \).

2. The second condition follows as \( v(L) = L_{\emptyset} \) where \( L_{\emptyset}(x) = \emptyset \) for all \( x \in X \).

3. The third condition follows as \( s(\phi_E) = 0 = v(\phi_E) \) and \( v(L)(x) = \emptyset = L_{\emptyset}(\epsilon)(x) \) for all \( x \in X \)

4. The fourth condition follows as \( s(\phi_E) = 0 = v(\phi_E) \) and \( \phi_L(\epsilon)(x) = 0 = \phi_{L0}(x) \) for all \( x \in X \) where \( s(B) = \epsilon \) and \( v(\phi_L) = \phi_{L0} \).
For the inductive step we assume that there is a scene \( s = \langle q, S, B, \text{rank}, \phi_E, \phi_P \rangle \) such that \( s \) is reachable in \( \mathcal{T}_{PB} \) and \( \rho \models s \). Let \( v \) be the view \( v_i^k \) in \( \rho \) and let \( v_i^{k+1} \) be the view that follows after the event \( \zeta_i^k \) in \( \rho \). Notice that \( v_i^{k+1} \) is either \( v_i^{k+1} = v_{init}(\text{rank}, k+1) \). We show that there is a scene \( s' \) such that \( s \xrightarrow{\Omega} s' \) and \( \rho \models s' \). We consider the different cases for \( \zeta_i^k \). In the following we let \( v = v_i^k \) and \( v' = v_i^{k+1} \):

\begin{itemize}
  \item \( \zeta_i^k = \text{skip} \). By the relation \( \xrightarrow{\Omega} \), we know \( \langle v(q), \text{skip}, v'(q) \rangle \in \Delta \). By the assumption \( \rho \models s \), we have \( q = v(q) \). Let \( s' = \langle v'(q), S, B, \text{rank}, \phi_E, \phi_P \rangle \).

  By the relation \( \xrightarrow{\Omega} \), we know \( s \xrightarrow{\text{skip}} s' \). We show \( \rho \models s' \) by checking the four conditions for any variable \( x \) and any value \( d \):

  1. By the definition of \( s' \) we have \( s'(q) = v'(q) \). Also by \( v \xrightarrow{\text{skip}} v' \) and \( \rho \models s \) we get \( s'(S) = v'(S) = S \), \( s'(\phi_P) = v'(\phi_P) = \phi_P \), and \( s'(\text{rank}) = v'(\text{rank}) = \text{rank} \).

  2. We want to show if \( v'(L)(x) = d \) then either \( \text{LVal}(s'(B))(x) = d \) or, \( \text{LVal}(s'(B))(x) = \emptyset \) and \( \text{rank}(x, d) \leq s'(\phi_E) \). By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( \text{LVal}(s'(B))(x) = \text{LVal}(B)(x) \). By \( v \xrightarrow{\text{skip}} v' \), we get \( v'(L) = v(L) \). The required statement follows from the assumption \( \rho \models s \).

  3. We want to show if \( v'(L)(x) = \emptyset \) and \( \text{rank}_x(x) > v'(\phi_E) \) then \( \text{LVal}(s'(B))(x) = \emptyset \) and \( \text{rank}_x(x) > s'(\phi_E) \). By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( \text{LVal}(s'(B))(x) = \text{LVal}(B)(x) \). By \( v \xrightarrow{\text{skip}} v' \) we get \( v'(L) = v(L) \). The required statement follows from the assumption \( \rho \models s \).

  4. We want to show \( \text{max}(s'(\phi_E), \phi_L(s')(x)) \leq \text{max}(v'(\phi_E), v'(\phi_L)(x)) \). By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( \text{max}(s'(\phi_E), \phi_L(s')(x)) = \text{max}(\phi_E, \phi_L(s)(x)) \). By \( v \xrightarrow{\text{skip}} v' \) we get \( v'(\phi_E) = v(\phi_E) \) and \( v'(\phi_L) = v(\phi_L) \). Therefore, \( \text{max}(v'(\phi_E), v'(\phi_L)(x)) = \text{max}(v(\phi_E), v(\phi_L)(x)) \). The required statement follows from the assumption \( \rho \models s \).

\item \( \zeta_i^k = \text{push}(\gamma) \). By the relation \( \xrightarrow{\Omega} \), we know \( \langle v(q), \text{push}(\gamma), v'(q) \rangle \in \Delta \) and \( v'(S) = \gamma \cdot v(S) \). By the assumption \( \rho \models s \), we have \( q = v(q) \) and \( S = v(S) \). Let \( s' = \langle v'(q), v'(S), B, \text{rank}, \phi_E, \phi_P \rangle \). By the relation \( \xrightarrow{\Omega} \), we know \( s \xrightarrow{\text{push}(\gamma)} s' \). We show \( \rho \models s' \) by checking the four conditions for any variable \( x \) and any value \( d \):

  1. By the definition of \( s' \) we have \( s'(q) = v'(q) \) and \( s'(S) = v'(S) \). Also by \( v \xrightarrow{\text{push}(\gamma)} v' \) and \( \rho \models s \) we get \( s'(\phi_P) = v'(\phi_P) = \phi_P \) and \( s'(\text{rank}) = v'(\text{rank}) = \text{rank} \).

  2. We want to show if \( v'(L)(x) = d \) then either \( \text{LVal}(s'(B))(x) = d \) or, \( \text{LVal}(s'(B))(x) = \emptyset \) and \( \text{rank}(x, d) \leq s'(\phi_E) \). By the definition of
s' we get \( s'(B) = B \) and \( s' (\phi_E) = \phi_E \). Therefore, \( LVal(s'(B))(x) = LVal(B)(x) \). By \( v \xrightarrow{\text{push}(\gamma)} \Theta \), \( \gamma \), \( v' \) we get \( v'(\mathcal{L}) = v(\mathcal{L}) \). The required statement follows from the assumption \( \rho \models^k s \).

3. We want to show if \( v'(\mathcal{L})(x) = \emptyset \) and \( \text{rank}_x(x) > v' (\phi_E) \) then \( LVal(s'(B))(x) = \emptyset \) and \( \text{rank}_x(x) > s'(\phi_E) \). By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( LVal(s'(B))(x) = LVal(B)(x) \). By \( v \xrightarrow{\text{push}(\gamma)} \Theta \), \( v' \) we get \( v'(\mathcal{L}) = v(\mathcal{L}) \) and \( v'(\phi_E) = v(\phi_E) \). The required statement follows from the assumption \( \rho \models^k s \).

4. We want to show if \( \max(s'(\phi_E), \phi_L(s')(x)) \leq \max(v'(\phi_E), v'(\phi_L)(x)) \).

By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( \max(s'(\phi_E), \phi_L(s')(x)) = \max(v'(\phi_E), v'(\phi_L)(x)) \).

The required statement follows from the assumption \( \rho \models^k s \).

- \( \zeta^k = \text{pop}(\gamma) \).

By the relation \( \rightarrow_\Theta \), we get \( \langle v(q), \text{pop}(\gamma), v'(q) \rangle \in \Delta \) and \( v(S) = \gamma \cdot v(S) \). By the assumption \( \rho \models^k s \), we get \( q = v(q) \) and \( S = v(S) \).

Let \( s' = \langle v'(q), v'(S), B, \text{rank}, \phi_E, \phi_P \rangle \). By the relation \( \rightarrow_\Theta \), we know \( s \xrightarrow{\text{pop}(\gamma)} s' \).

We show \( \rho \models^k s' \) by checking the four conditions for any variable \( x \) and any value \( d \):

1. By the definition of \( s' \) we have \( s'(q) = v'(q) \) and \( s'(S) = v'(S) \). Also, by \( s \xrightarrow{\text{pop}(\gamma)} s' \), \( v \xrightarrow{\text{pop}(\gamma)} \Theta \) \( \gamma \), \( v' \) and \( \rho \models^k s \) we get \( s'(\phi_P) = v'(\phi_P) \) and \( s'(\text{rank}) = v'(\text{rank}) = \text{rank} \).

2. We want to show if \( v'(\mathcal{L})(x) = d \) then either \( LVal(s'(B))(x) = d \) or \( LVal(s'(B))(x) = \emptyset \) and \( \text{rank}(x, d) \leq s'(\phi_E) \). By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( LVal(s'(B))(x) = LVal(B)(x) \). By \( v \xrightarrow{\text{pop}(\gamma)} \Theta \), \( v' \) we get \( v'(\mathcal{L}) = v(\mathcal{L}) \) and \( v'(\phi_E) = v(\phi_E) \). The required statement follows from the assumption \( \rho \models^k s \).

3. We want to show if \( v'(\mathcal{L})(x) = \emptyset \) and \( \text{rank}_x(x) > v'(\phi_E) \) then \( LVal(s'(B))(x) = \emptyset \) and \( \text{rank}_x(x) > s'(\phi_E) \). By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( LVal(s'(B))(x) = LVal(B)(x) \). By \( v \xrightarrow{\text{pop}(\gamma)} \Theta \), \( v' \) we get \( v'(\mathcal{L}) = v(\mathcal{L}) \) and \( v'(\phi_E) = v(\phi_E) \). The required statement follows from the assumption \( \rho \models^k s \).

4. We want to show if \( \max(s'(\phi_E), \phi_L(s')(x)) \leq \max(v'(\phi_E), v'(\phi_L)(x)) \).

By the definition of \( s' \) we get \( s'(B) = B \) and \( s'(\phi_E) = \phi_E \). Therefore, \( \max(s'(\phi_E), \phi_L(s')(x)) = \max(v'(\phi_E), v'(\phi_L)(x)) \).

The required statement follows from the assumption \( \rho \models^k s \).

- \( \zeta^k = \text{w}(1)(x, d) \).

By the relation \( \rightarrow_\Theta \), we know \( \langle v(q), w(x, d), v'(q) \rangle \in \Delta \).

By the assumption \( \rho \models^k s \), we have \( q = v(q) \). Let \( s' = \langle v'(q), S, (x, d) \rangle \).
By the relation $\rightarrow_{\widehat{\omega}}$, we know $s \xrightarrow{(x,d)} \widehat{\omega} s'$. We show $\rho \models_{\widehat{\omega}} s'$ by checking the four conditions for any variable $x'$ and any value $d'$:

1. By the definition of $s'$ we have $s'(q) = v'(q)$. Also by $v' \xrightarrow{(1)(x,d)} \widetilde{\omega} v'$ and $\rho \models_{\widehat{\omega}} s$ we get $s'(S) = v'(S) = S$, $s'(\phi_E) = v'(\phi_E) = \phi_E$, and $s'(\text{rank}) = v'(\text{rank}) = \text{rank}$.

2. We want to show if $v'(\mathcal{L})(x') = d'$ then either $\text{LVal}(s'(B))(x') = d'$ or, $\text{LVal}(s'(B))(x') = \emptyset$ and $\text{rank}(x',d') \leq s'(\phi_E)$. By the definition of $s'$ we get $s'(\phi_E) = \phi_E$ and $s'(B) = (x,d) \cdot B$. As a result, we have $\text{LVal}(s'(B)) = \text{LVal}(B)[x \leftarrow d]$. By $v' \xrightarrow{(1)(x,d)} \widetilde{\omega} v'$, we get $v'(\mathcal{L}) = v(\mathcal{L})[x \leftarrow d]$. It follows if $x'$ is the variable $x$ then we have $v'(\mathcal{L})(x) = d = \text{LVal}(s'(B))(x)$. If $x'$ a variable different from $x$, then $v'(\mathcal{L})(x') = \mathcal{L}(x')$ and $\text{LVal}(s'(B))(x') = \text{LVal}(B)(x)$. The required statement then follows from the assumption $\rho \models_{\widehat{\omega}} s$.

3. We want to show if $v'(\mathcal{L})(x') = d'$ then either $\text{rank}_x(x') > v'(\phi_E)$ then $\text{LVal}(s'(B))(x') = \emptyset$ and $\text{rank}_x(x') > s'(\phi_E)$. By the definition of $s'$ we get $s'(\phi_E) = \phi_E$ and $s'(B) = (x,d) \cdot B$. As a result, we have $\text{LVal}(s'(B)) = \text{LVal}(B)[x \leftarrow d]$. By $v' \xrightarrow{(1)(x,d)} \widetilde{\omega} v'$, we get $v'(\phi_E) = v(\phi_E)$ and $v'(\mathcal{L}) = v(\mathcal{L})[x \leftarrow d]$. It follows if $x'$ is the variable $x$ then we have $v'(\mathcal{L})(x) = d$, which trivially satisfies the required statement. If $x'$ a variable different from $x$, then $v'(\mathcal{L})(x') = \mathcal{L}(x')$ and $\text{LVal}(s'(B))(x') = \text{LVal}(B)(x)$. The required statement then follows from the assumption $\rho \models_{\widehat{\omega}} s$.

4. We want to show $\max(s'(\phi_E), \phi_L(s')(x')) \leq \max(v'(\phi_E), v'(\phi_L)(x'))$. Notice first that by the assumption $\rho \models_{\widehat{\omega}} s$, we have for all $x \in \mathbb{X}$

$$\max(\phi_E, \phi_L(s)(x)) \leq \max(v(\phi_E), v(\phi_L)(x))$$

which implies

$$\max(\phi_E, \phi_L(s))(x) \leq \max(v(\phi_E), v(\phi_L(s))) \tag{2}$$

By the definition of $s'$ we get $s'(\phi_E) = \phi_E$ and $s'(B) = (x,d) \cdot B$. Therefore, $\phi_L(s'(x)) = \phi_L(s)[x \leftarrow \text{rank}(x,d), \phi_L(s)]$. By $v' \xrightarrow{(1)(x,d)} \widetilde{\omega} v'$ we get $v'(\phi_E) = v(\phi_E)$ and $v'(\phi_L) = v(\phi_L)[x \leftarrow \text{rank}(x,d), \phi_L(s)]$. If $x'$ is the variable $x$ then we have $\max(s'(\phi_E), \phi_L(s')(x')) = \max(s(\phi_E), \phi_L(s)[x \leftarrow \text{rank}(x,d), \phi_L(s)])$ and $\max(v'(\phi_E), v'(\phi_L)(x')) = \max(v(\phi_E), \phi_L(s)[x \leftarrow \text{rank}(x,d), \phi_L(s)])$. The required statement then follows from equation \ref{eq:2}. If $x'$ a variable different from $x$ then we have $\max(s'(\phi_E), \phi_L(s')(x')) = \max(s(\phi_E), \phi_L(s)[x \leftarrow \text{rank}(x,d), \phi_L(s)])$ and $\max(v'(\phi_E), v'(\phi_L)(x')) = \max(v(\phi_E), v(\phi_L)(x'))$. The required statement then follows equation \ref{eq:1}.

- $\zeta^k = \omega(2)(x,d)$. By the relation $\rightarrow_{\widehat{\omega}}$, we know $\langle v(q), \omega(x,d), q' \rangle \in \Delta$, $\text{rank}(x,d) = k$ and $v' = v_{\text{init}}(\text{rank}, k + 1)$. Let $s' = s_{\text{init}}(\text{rank}, k + 1)$.
and let \( s'' = (q', S, \langle x, d \rangle \cdot B, \text{rank}, \phi_E, \phi_P) \). By the relation \( \rightarrow_\varnothing \), we have \( s \xrightarrow{v(x,d)} \varnothing \ s' \). Also by the relation \( \rightarrow_\varnothing \), we have \( s'' \xrightarrow{\rho} \varnothing \ s' \). To show \( \rho \models_x^k \ s' \) we check the four conditions for any variable \( x \) and any value \( d \) in the same way as in the base case.

- \( \zeta_i^k = \text{mf} \). By the relation \( \rightarrow_\varnothing \), we know \( (v(q), \text{mf}, v'(q)) \in \Delta \). By the assumption \( \rho \models_x^k \ s \), we have \( q = v(q) \) and \( \phi_P = \phi_E \). Notice that for any reachable view we have \( \max(v(\phi_E), v(\phi_{Lmax}) < v(\phi_P)) \). Therefore by equation (2), we get for the scene \( s \),

\[
\max(\phi_E, \phi_{Lmax}(s)) < \phi_P. \tag{3}
\]

This means that the buffer in \( s \) has no assignment with a rank higher or equal to \( \phi_P \). Therefore, the buffer in \( s \) can be emptied by updates of type \( u(1) \). By the relation \( \rightarrow_\varnothing \), for the scene \( s'' = (q, S, \epsilon, \text{rank}, \max(\phi_E, \phi_{Lmax}(s)), \phi_P) \) we have \( s \xrightarrow{\rho} \varnothing \ s'' \). Also by the relation \( \rightarrow_\varnothing \), for the scene \( s' = (v'(q), S, \epsilon, \text{rank}, \max(\phi_E, \phi_{Lmax}(s)), \phi_P) \) we have \( s'' \xrightarrow{\text{mf}} \varnothing \ s' \). We show \( \rho \models_x^k \ s' \) by checking the four conditions for any variable \( x \) and any value \( d \):

1. By the definition of \( s' \) we have \( s'(q) = v'(q) \). Also by \( v \xrightarrow{\text{mf}} \varnothing \ v' \) and \( \rho \models_x^k \ s \), we get \( s'(S) = v'(S) = S, s'(\phi_P) = v'(\phi_P) = \phi_P \), and \( s'(\text{rank}) = v'(\text{rank}) = \text{rank} \).

2. We want to show if \( v'(L)(x) \in \rho \) then \( \text{LVal}(s'(B))(x) = \emptyset \) or, \( \text{LVal}(s'(B))(x) = \emptyset \) and \( \text{rank}(x, d) \leq \phi_E \). By the definition of \( s' \) we get \( s'(B) = \epsilon \) and \( s'(\phi_E) = \max(\phi_E, \phi_{Lmax}(s)) \). By \( v \xrightarrow{\rightarrow_\varnothing} v' \), we have \( v'(L) = v(L) \). Therefore, we need to show if \( v'(L)(x) = d \) then \( \text{rank}(x, d) \leq \max(\phi_E, \phi_{Lmax}(s)) \). By the assumption \( \rho \models_x^k \ s \), we have \( \text{LVal}(B)(x) = d \) or \( \text{LVal}(B)(x) = \emptyset \) and \( \text{rank}(x, d) \leq \phi_E \). If \( \text{LVal}(B)(x) = d \) then \( \text{rank}(x, d) \leq \phi_{Lmax}(s) \) and therefore \( \text{rank}(x, d) \leq \max(\phi_E, \phi_{Lmax}(s)) \). If \( \text{LVal}(B)(x) = \emptyset \) then \( \text{rank}(x, d) \leq \phi_E \) and therefore \( \text{rank}(x, d) \leq \max(\phi_E, \phi_{Lmax}(s)) \).

3. We want to show if \( v'(L)(x) = \emptyset \) and \( \text{rank}_X(x) > \phi_E \) then \( \text{LVal}(s'(B))(x) = \emptyset \) or \( \text{rank}_X(x) > \phi_{Lmax}(s) \). By the definition of \( s' \) we get \( s'(B) = \epsilon \) and therefore we only need to show if \( v'(L)(x) = \emptyset \) and \( \text{rank}_X(x) > \phi_{Lmax}(s) \) then \( \text{rank}_X(x) > \max(\phi_E, \phi_{Lmax}(s)) \). By \( v \xrightarrow{\text{mf}} \varnothing \ v' \), we have \( v'(\phi_E) = \max(v(\phi_E), v(\phi_{Lmax})) \). The required statement follows from equation (2).

4. We want to show \( \max(s'(\phi_E), \phi_{L}(s')(x)) \leq \max(v'(\phi_E), v'(\phi_{L}(x))) \). By the definition of \( s' \), we have \( s'(B) = \epsilon \) and \( s'(\phi_E) = \max(\phi_E, \phi_{Lmax}(s)) \). Therefore, we get \( \max(s'(\phi_E), \phi_{Lmax}(s')) = \max(\phi_E, \phi_{Lmax}(s)) \). By \( v \xrightarrow{\rightarrow_\varnothing} v' \), we have \( v'(\phi_E) = \max(v(\phi_E), v(\phi_{Lmax})) \) and \( v'(\phi_{Lmax}) = v(\phi_{Lmax}) \). Therefore, we get \( \max(v'(\phi_E), v'(\phi_{Lmax})) = \max(v(\phi_E), v(\phi_{Lmax})) \). The required statement follows then from equation (2).
\[ \zeta^k = \mathbf{r}(1)(x,d). \] By the relation \( \rightarrow_\varnothing \), we know \( \langle \nu(q), \mathbf{r}(x,d), v'(q) \rangle \in \Delta \) and \( v(L)(x) = d \). By the assumption \( \nu \models^k s \), we have \( v = v(q) \) and it is either \( \text{LVal}(B)(x) = d \) or \( \text{LVal}(B)(x) = \emptyset \) and \( \text{rank}(x,d) \leq \phi_E \). Let \( s' = \langle v'(q), S, B, \text{rank}, \phi_E, \phi_P \rangle \). If \( \text{LVal}(B)(x) = d \) then by the relation \( \rightarrow_\varnothing \), we know \( s \xrightarrow{r(1)(x,d)}_\varnothing s' \). If \( \text{LVal}(B)(x) = \emptyset \) and \( \text{rank}(x,d) \leq \phi_E \) then by the relation \( \rightarrow_\varnothing \), we know \( s \xrightarrow{r(3)(x,d)}_\varnothing s' \). To show \( \nu \models^{k'} s' \), we check the four conditions for any variable \( x' \) and any value \( d' \) in the same way we did for the case \( \zeta^k = \text{skip} \).

\[ \zeta^k = \mathbf{r}(2)(x,d). \] By the relation \( \rightarrow_\varnothing \), we know \( \langle \nu(q), \mathbf{r}(x,d), v'(q) \rangle \in \Delta \), \( d = \text{init}(x) v(L)(x) = \emptyset \) and \( \text{rank}_x(x) > v(\phi_E) \). By \( \nu \models^k s \), we have \( v = v(q) \), \( \text{LVal}(B)(x) = \emptyset \) and \( \text{rank}_x(x) > \phi_E \). Let \( s' = \langle v'(q), S, B, \text{rank}, \phi_E, \phi_P \rangle \). By the relation \( \rightarrow_\varnothing \), we know \( s \xrightarrow{\mathbf{r}(2)(x,d)}_\varnothing s' \). To show \( \nu \models^{k'} s' \), we check the four conditions for any variable \( x' \) and any value \( d' \) in the same way we did for the case \( \zeta^k = \text{skip} \).

\[ \zeta^k = \mathbf{r}(3)(x,d). \] By the relation \( \rightarrow_\varnothing \), we know \( \langle \nu(q), \text{skip}, v'(q) \rangle \in \Delta \), \( \text{rank}(x,d) < v(\phi_P) \), and \( v'(\phi_E) = \max(v(\phi_E), v(\phi_L(x)), \text{rank}(x,d)) \). By the assumption \( \nu \models^k s \), we have \( v = v(q) \). By equation (??), we get that \( B \) has no assignment of a rank equal or higher to \( \phi_P \). Therefore, we can empty \( B \) from all the assignments on \( x \) by a sequence of updates of type \( \text{u}(1) \). Let \( B' \) be the buffer state after emptying \( B \) from all the assignments on \( x \), and let \( s'' = \langle q, S, B', \text{rank}, \max(\phi_E, \phi_L(s)(x)), \phi_P \rangle \). By the relation \( \rightarrow_\varnothing \), we know \( s \xrightarrow{\mathbf{r}(3)(x,d)}_\varnothing s'' \). Let \( s' = \langle v'(q), S, B', \text{rank}, \max(\phi_E, \phi_L(s)(x)), \text{rank}(x,d), \phi_P \rangle \) then, by the relation \( \rightarrow_\varnothing \), we get \( s'' \xrightarrow{\mathbf{r}(3)(x,d)}_\varnothing s' \) and hence \( s \xrightarrow{\mathbf{r}(3)(x,d)}_\varnothing s' \). We show \( \nu \models^{k'} s' \) by checking the four conditions for any variable \( x' \) and any value \( d' \):

1. By the definition of \( s' \) we have \( s'(q) = v'(q) \). Also by \( v \xrightarrow{r(3)(x,d)}_\varnothing v' \) and \( \nu \models^k s \) we get \( s'(S) = v(S) = S \), \( s'(\phi_P) = v'(\phi_P) = \phi_P \), and \( s'(\text{rank}) = v'(\text{rank}) = \text{rank} \).
2. We want to show if \( v'(L)(x') = d' \) then either \( \text{LVal}(s'(B))(x') = d' \) or, \( \text{LVal}(s'(B))(x') = \emptyset \) and \( \text{rank}(x',d') \leq s'(\phi_E) \). By the definition of \( s' \) we get \( s'(\phi_E) = \max(\phi_E, \text{rank}(x,d)) \) and \( s'(B) = B' \). By \( v \xrightarrow{r(3)(x,d)}_\varnothing v' \), we have \( v'(L) = v(L) \). As a result, we need to show if \( v'(L)(x') = d' \) then either \( \text{LVal}(B')(x') = d' \) or, \( \text{LVal}(B')(x') = \emptyset \) and \( \text{rank}(x',d') \leq \max(\phi_E, \text{rank}(x,d)) \). Assume \( v(L)(x') = d' \), then according to the assumption \( \nu \models^k s \), it is either \( \text{LVal}(B)(x') = d \) or \( \text{LVal}(B)(x') = \emptyset \) and \( \text{rank}(x',d') \leq \phi_E \). There are three cases to consider:

- If \( \text{LVal}(B)(x') = d' \) and \( \text{rank}(x',d') > \phi_L(s)(x) \) then \( \text{LVal}(B')(x') = d' \).

- If \( \text{LVal}(B)(x') = \emptyset \) then either \( \text{rank}(x',d') > \phi_E \) or \( \text{rank}(x',d') > \phi_L(s)(x) \) then \( \text{LVal}(B')(x') = \emptyset \).

- If \( \text{LVal}(B)(x') = \emptyset \) then either \( \text{rank}(x',d') > \phi_L(s)(x) \) or \( \text{rank}(x',d') > \phi_E \) then \( \text{LVal}(B')(x') = \emptyset \).
– If $\text{LVal}(B)(x') = d'$ and $\text{rank}(x', d') \leq \phi_L(s)(x)$ then it is either $\text{LVal}(B')(x') = d'$ or $\text{LVal}(B')(x') = \emptyset$ and $\text{rank}(x', d') \leq \phi_L(s)(x) \leq s'(\phi_E)$.
– If $\text{LVal}(B)(x') = \emptyset$ then $\text{LVal}(B')(x') = \emptyset$ and $\text{rank}(x', d') \leq \phi_E \leq s'(\phi_E)$.

3. We want to show if $v'(\mathcal{L})(x') = \emptyset$ and $\text{rank}_X(x') > v'(\phi_E)$ then $\text{LVal}(s'(B))(x') = \emptyset$ and $\text{rank}_X(x') > s'(\phi_E)$. By $\nu \frac{\text{rank}(x,d)}{\text{init}} v'$, we have $v'(\phi_E) = \max(v(\phi_E), v(\phi_L)(x), \text{rank}(x, d))$ and $v'(\mathcal{L}) = v(\mathcal{L})$.
By the definition of $s'$, we have $s'(\phi_E) = \max(\phi_E, \phi_L(s)(x), \text{rank}(x, d))$.
Therefore, we need to show if $\mathcal{L}(x') = \emptyset$ and $\text{rank}_X(x') > \max(v(\phi_E), v(\phi_L)(x), \text{rank}(x, d))$ then $\text{LVal}(s'(B))(x') = \emptyset$ and $\text{rank}_X(x') > \max(\phi_E, \phi_L(s)(x), \text{rank}(x, d))$. By the definition of $s'$, we have $s'(B) = B$ and therefore if $\text{LVal}(B)(x') = \emptyset$ then $\text{LVal}(B')(x') = \emptyset$. Also by equation $[\text{init}]$, we get $\max(v(\phi_E), v(\phi_L)(x), \text{rank}(x, d)) \geq \max(\phi_E, \phi_L(s)(x), \text{rank}(x, d))$. The required statement then follows from the assumption $\rho \models^k s$.

4. We want to show $\max(s'(\phi_E), \phi_L(s')(x')) \leq \max(v'(\phi_E), v'(\phi_L)(x'))$.
By the definition of $s'$, we have $s'(B) = B'$ and therefore we get $\phi_L(s')(x') \leq \phi_L(s)(x')$. Also by the definition of $s'$, we have $s'(\phi_E) = \max(\phi_E, \phi_L(s)(x), \text{rank}(x, d))$ and therefore we get $\max(s'(\phi_E), \phi_L(s')(x')) \leq \max(\phi_E, \phi_L(s)(x), \text{rank}(x, d))$.
By $\nu \frac{\text{rank}(x,d)}{\text{init}} v'$, we have $v'(\phi_E) = \max(v(\phi_E), v(\phi_L)(x), \text{rank}(x, d))$ and $v'(\phi_L) = v(\phi_L)$. Therefore we get $\max(v'(\phi_E), v'(\phi_L)(x')) = \max(v(\phi_E), v(\phi_L)(x), \text{rank}(x, d), v(\phi_L)(x'))$.
The required statement then follows by equation $[\text{init}]$.

$\square$

### F Proof of Lemma 5.4

**Proof.** We do a proof of induction over all the scenes $s^k$ in $\rho$ that are not preceded by a write of rank $k$ inside their suppliers, namely where for $j \leq i$ we have $\nu_j^i \neq w(x, d)$ if $\text{rank}(x, d) = k$. For the base case we take $i = 0$ and $k = 1$. Since $\rho$ is an initial run we have $s^0_1 = s_{\text{init}}(\text{rank}, 1)$. We have $v_{\text{init}}(\text{rank}, 1)$ is reachable in $\mathcal{T}_\rho$. We show $\rho \models^k v_{\text{init}}(\text{rank}, 1)$ by checking the four conditions for any variable $x$ and any value $d$.In the following we let $s = s^1_1$ and $v = v_{\text{init}}(\text{rank}, 1)$:

1. We have $v(q) = q_{\text{init}} = s(q), v(\phi_P) = 1 = s(\phi_E)$ and $v(\text{rank}) = \text{rank} = s(\text{rank})$.
2. If $\text{LVal}(s(B))(x) = d$ then $v(\mathcal{L})(x) = d$, follows immediately from $s(B) = \epsilon$.
3. If $\text{LVal}(s(B))(x) = \emptyset$ and $\text{rank}_X(x) > s(\phi_E)$ then $\mathcal{L}(x) = \emptyset$ and $\text{rank}_X(x) > \phi_E$, follows from $v(\mathcal{L}) = \mathcal{L}_\emptyset$ and $v(\phi_E) = s(\phi_E) = 0$. 


4. max(v(ϕE), v(ϕL)(x)) ≤ max(s(ϕE), ϕL(s)(x)), follows from v(ϕE) = ϕLo, s(B) = ε and v(ϕE) = s(ϕE) = 0.

For the inductive step, we assume si is a scene in the k supplier where it is not preceded by any write of rank k and where there is a view \( v = \langle s, \text{rank}, \mathcal{L}, \phi_E, \phi_L, \phi_P \rangle \) such that \( \rho \vdash^i v \). Let also \( s_{i+1}^k \) be the next scene in \( \rho \) after \( s_i^k \) that is not preceded by write of the rank of its supplier. Notice that \( s_{i+1}^k \) it is either \( s_{i+1}^k = s_{i+1}^0 \) or it is \( s_{i+1}^k = s_{i+1}^{k+1} = s_{\text{init}}(\text{rank}, k+1) \). For all the different cases of \( \psi_i^k \), we want to show that there is a view \( v' \) such that \( v \xrightarrow{s} v' \) and where \( \rho \vdash^i v' \). In the following we let \( s = s_i^k \) and \( s' = s_{i+1}^k \):

- \( \psi_i^k = \text{skip} \). We get \( s' = s_{i+1}^k \). By the relation \( \rightarrow_{\mathcal{O}} \), we get \( \langle s(q), \text{skip}, s'(q) \rangle \) ∈ \( \Delta \). By the assumption \( \rho \vdash^i v \), we have \( q = s(q) \). Let \( v' = \langle s'(q), \text{rank}, \mathcal{L}, \phi_E, \phi_L, \phi_P \rangle \), then by the skip rule in the relation \( \rightarrow_{\mathcal{O}} \) we get \( v \xrightarrow{\text{skip}} v' \). We show \( \rho \vdash^i v' \) by checking the four conditions for any variable \( x \) and any value \( d \):

  1. By the definition of \( v' \) we have \( v'(q) = q = s'(q) \). Also by \( s \xrightarrow{\text{skip}}_{\mathcal{O}} s' \) and \( \rho \vdash^i v \), we get \( v'(S) = s'(S) \) \( v'(\phi_P) = \phi_P = s'(\phi_P) \) and \( v'(\text{rank}) = \text{rank} = s'(\text{rank}) \).

  2. We want to show if LVal(s'(B))(x) = d then \( v'(\mathcal{L})(x) = d \). This follows from \( s'(B) = s(B) \), \( v'(\mathcal{L}) = \mathcal{L} \) and the assumption \( \rho \vdash^i v \).

  3. We want to show if LVal(s'(B))(x) = \( \emptyset \) and rank\( x ) > s'(\phi_E) \) then \( v'(\mathcal{L})(x) = \emptyset \) and rank\( x ) > v'(\phi_E) \). This follows from \( s'(B) = s(B) \), \( s'(\phi_E) = s(\phi_E) \), \( v'(\mathcal{L}) = \mathcal{L} \), \( v'(\phi_E) = \phi_E \) and the assumption \( \rho \vdash^i v \).

  4. We want to show max(v'(\phi_E), v'(\phi_L)(x)) ≤ max(s'(\phi_E), \phi_L(s')(x)). This follows from \( s'(B) = s(B) \), \( s'(\phi_E) = s(\phi_E) \), \( v'(\phi_E) = \phi_E \), \( v'(\phi_L) = \phi_L \), rank\( x ) ≤ \( s'(\phi_E)(x) \).

- \( \psi_i^k = \text{push}(\gamma) \). We get \( s' = s_{i+1}^k \). By the relation \( \rightarrow_{\mathcal{O}} \), we get \( \langle s(q), \text{push}(\gamma), s'(q) \rangle \) ∈ \( \Delta \) and \( s'(S) = \gamma \cdot s(S) \). By the assumption \( \rho \vdash^i v \), we have \( q = s(q) \) and \( S = s(S) \). Let \( v' = \langle s'(q), s'(S), \text{rank}, \mathcal{L}, \phi_E, \phi_L, \phi_P \rangle \), then by the push rule in the relation \( \rightarrow_{\mathcal{O}} \) we get \( v \xrightarrow{\text{push}(\gamma)} v' \). We show \( \rho \vdash^i v' \) by checking the four conditions for any variable \( x \) any value \( d \):

  1. By the definition of \( v' \) we have \( v'(q) = q = s'(q) \) and \( v'(S) = s'(S) \). Also by \( s \xrightarrow{\text{push}(\gamma)}_{\mathcal{O}} s' \) and \( \rho \vdash^i v \), we get \( v'(\phi_P) = \phi_P = s'(\phi_P) \) and \( v'(\text{rank}) = \text{rank} = s'(\text{rank}) \).

  2. We want to show if LVal(s'(B))(x) = d then \( v'(\mathcal{L})(x) = d \). This follows from \( s'(B) = s(B) \), \( v'(\mathcal{L}) = \mathcal{L} \) and the assumption \( \rho \vdash^i v \).

  3. We want to show if LVal(s'(B))(x) = \( \emptyset \) and rank\( x ) > s'(\phi_E) \) then \( v'(\mathcal{L})(x) = \emptyset \) and rank\( x ) > v'(\phi_E) \). This follows from \( s'(B) = s(B) \), \( s'(\phi_E) = s(\phi_E) \), \( v'(\mathcal{L}) = \mathcal{L} \), \( v'(\phi_E) = \phi_E \) and the assumption \( \rho \vdash^i v \).
4. We want to show \( \max(v'(\phi_E), v'(\phi_L)(x)) \leq \max(s'(\phi_E), \phi_L(s')(x)) \).

This follows from \( s'(B) = s(B), s'(\phi_E) = s(\phi_E), v'(\phi_L) = \phi_L, v'(\phi_E) = \phi_E \) and the assumption \( \rho \vdash^k v \).

- \( \psi^k_i = \text{pop}(\gamma) \). We get \( s' = s^k_i \). By the relation \( \rightarrow_{\Theta} \), we get \( (s(q), \text{pop}(\gamma), s'(q)) \in \Delta \) and \( s(S) = \gamma \cdot s'(S) \).

By the assumption \( \rho \vdash^k v \), we have \( q = s(q) \) and \( S = s(S) \). Let \( v' = (s'(q), s'(S), \text{rank}, \mathcal{L}, \phi_E, \phi_L, \phi_P) \), then by the pop rule in the relation \( \rightarrow_{\Theta} \) we get \( v \xrightarrow{\text{pop}(\gamma)}_{\Theta} v' \). We show \( \rho \vdash^k v' \) by checking the four conditions for any variable \( x \) any value \( d \):

1. By the definition of \( v' \) we have \( v'(q) = q = s'(q) \) and \( v'(S) = s'(S) \).
   Also by \( s \xrightarrow{\text{pop}(\gamma)}_{\Theta} s' \) and \( \rho \vdash^k v \), we get \( s'(S) = \gamma \cdot s(S), v'(\phi_P) = \phi_P = s'(\phi_P) \) and \( v'(\text{rank}) = \text{rank} = s'(\text{rank}) \).

2. We want to show if \( \text{LVal}(s'(B))(x) = d \) then \( v'(\mathcal{L})(x) = d \). This follows from \( s'(B) = s(B), v'(\mathcal{L}) = \mathcal{L} \) and the assumption \( \rho \vdash^k v \).

3. We want to show if \( \text{LVal}(s'(B))(x) = \emptyset \) and \( \text{rank}_X(x) > s'(\phi_E) \) then \( v'(\mathcal{L})(x) = \emptyset \) and \( \text{rank}_X(x) > v'(\phi_E) \).
   This follows from \( s'(B) = s(B), s'(\phi_E) = s(\phi_E), v'(\mathcal{L}) = \mathcal{L}, v'(\phi_E) = \phi_E \) and the assumption \( \rho \vdash^k v \).

4. We want to show \( \max(v'(\phi_E), v'(\phi_L)(x)) \leq \max(s'(\phi_E), \phi_L(s')(x)) \).
   This follows from \( s'(B) = s(B), s'(\phi_E) = s(\phi_E), v'(\phi_L) = \phi_L, v'(\phi_E) = \phi_E \) and the assumption \( \rho \vdash^k v \).

- \( \psi^k_i = \text{mf} \). We get \( s' = s^k_i \). By the relation \( \rightarrow_{\Theta} \), we get \( (s(q), \text{mf}, s'(q)) \in \Delta \).

By the assumption \( \rho \vdash^k v \), we have \( q = s(q) \). Let \( v' = (s'(q), S, \text{rank}, \mathcal{L}, \text{max}(\phi_E, \phi_{L\max}), \phi_L, \phi_P) \), then by the memory-fence rule in the relation \( \rightarrow_{\Theta} \) we get \( v \xrightarrow{\text{mf}}_{\Theta} v' \). We show \( \rho \vdash^k v' \) by checking the four conditions for any variable \( x \) any value \( d \):

1. By the definition of \( v' \) we have \( v'(q) = q = s'(q) \). Also by \( s \xrightarrow{\text{mf}}_{\Theta} s' \) and \( \rho \vdash^k v \), we get \( v'(S) = S = s'(S), v'(\phi_P) = \phi_P = s'(\phi_P) \) and \( v'(\text{rank}) = \text{rank} = s'(\text{rank}) \).

2. We want to show if \( \text{LVal}(s'(B))(x) = d \) then \( v'(\mathcal{L})(x) = d \). This follows from \( s'(B) = s(B) = \epsilon \).

3. We want to show if \( \text{LVal}(s'(B))(x) = \emptyset \) and \( \text{rank}_X(x) > s'(\phi_E) \) then \( v'(\mathcal{L})(x) = \emptyset \) and \( \text{rank}_X(x) > v'(\phi_E) \).
   Notice that by \( \rho \vdash^k v \) we have for any \( x \),
   \[
   \max(\phi_E, \phi_L(x)) \leq \max(s(\phi_E), \phi_L(s)(x)) \tag{4}
   \]
   which implies
   \[
   \max(\phi_E, \phi_{L\max}(x)) \leq \max(s(\phi_E), \phi_{L\max}(s)). \tag{5}
   \]

By \( s \xrightarrow{\text{mf}}_{\Theta} s' \), we get \( s'(B) = s(B) \) and by the definition of \( v' \), we get \( v'(\mathcal{L}) = \mathcal{L} \). Therefore, the required statement then follows
from showing \( v'(\phi_E) \leq s'(\phi_E) \). By the definition of \( v' \) we have
\[
v'(\phi_E) = \max(\phi_E, \phi_{L_{\text{max}}}).
\]
Also by \( s \nrightarrow_{\Delta} s' \), we have \( s(B) = \epsilon \)
and \( s'(\phi_E) = s(\phi_E) \). Therefore, \( \phi_{L_{\text{max}}}(s) = 0 \)
and thus \( s'(\phi_E) = \max(s(\phi_E), \phi_{L_{\text{max}}}(s)) \). The statement \( v'(\phi_E) \leq s'(\phi_E) \)
follows from equation (4).

4. We want to show \( \max(v'(\phi_E), v'(\phi_L)(x)) \leq \max(s'(\phi_E), \phi_L(s')(x)) \).
By the definition of \( v' \) we have \( \max(v'(\phi_E), v'(\phi_L)(x)) = \max(\phi_E, \phi_{L_{\text{max}}}, \phi_L(x)) = 
max(\phi_E, \phi_{L_{\text{max}}}, \phi_L(x)) \). By \( s \nrightarrow_{\Delta} s' \), we have \( s(B) = s'(B) = \epsilon \)
and \( s'(\phi_E) = s(\phi_E) \). Therefore, \( \phi_L(s)(x) = \phi_L(s')(x) = 0 \)
and thus \( \max(s'(\phi_E), \phi_L(s')(x)) = \max(s(\phi_E), \phi_L(s)(x)) \). The statement
follows by equation (4).

- \( \psi^k \equiv w(x, d) \). There are two cases to consider:
  - \( \text{rank}(x, d) = k \). We get \( s' = s^{k+1}_0 = s_{\text{init}}(\text{rank}, k+1) \). By the relation
\( \rightarrow_{\Delta} \) we get \( (s(q), w(x, d), q') \in \Delta \). By the assumption \( \rho \vdash^k v \), we have \( q = s(q) \).
Let \( v' = s_{\text{init}}(\text{rank}, k+1) \), then by the write(2) rule in the
relation \( \rightarrow_{\Delta} \) we get \( v \xrightarrow{(w(2)(x, d))}_{\Delta} v' \). To show \( \rho \vdash^k v' \), we can check
the four conditions for any variable \( x' \) and any value \( d' \) in the same
way as in the base case.
  - \( \text{rank}(x, d) < k \). We get \( s' = s^{k+1}_0 \). By the relation \( \rightarrow_{\Delta} \) we get
\( (s(q), w(x, d), s'(q)) \in \Delta \). By the assumption \( \rho \vdash^k v \), we have \( q = s(q) \).
Let \( v' = (s'(q), S, \text{rank}, L[x \leftarrow d], \phi_E, \phi_L[x \leftarrow \max(\text{rank}(x, d), \phi_{L_{\text{max}}}]), \phi_P] \),
then by the write(1) rule in the relation \( \rightarrow_{\Delta} \) we get \( v \xrightarrow{(w(1)(x, d))}_{\Delta} v' \).
We show \( \rho \vdash^k v' \) by checking the four conditions for any variable \( x' \)
and any value \( d' \):
  1. By the definition of \( v' \) we have \( v'(q) = q = s'(q) \). Also by
\( s \nrightarrow_{\Delta} s' \) and \( \rho \vdash^k v \), we get \( v'(S) = S = s'(S) \).
  2. We want to show if \( \text{LVal}(s'(B))(x') = d' \) then \( v'(L)(x') = d' \).
By \( s \nrightarrow_{\Delta} s' \), we have \( s'(B) = (x, d) \cdot s(B) \) and therefore
\( \text{LVal}(s'(B)) = \text{LVal}(s(B))[x \leftarrow d] \). Also by the definition of \( v' \), we have \( v'(L) = L[x \leftarrow -] \). Therefore, the required statement
follows from the assumption \( \rho \vdash^k v \).
  3. We want to show if \( \text{LVal}(s'(B))(x') = \emptyset \) and \( \text{rank}_X(x') > s'(\phi_E) \)
then \( v'(L)(x') = \emptyset \) and \( \text{rank}_X(x') > v'(\phi_E) \). By \( s \nrightarrow_{\Delta} s' \), we have \( s'(\phi_E) = s(\phi_E) \) and \( s'(B) = (x, d) \cdot s(B) \) and therefore
\( \text{LVal}(s'(B)) = \text{LVal}(s(B))[x \leftarrow d] \). As a result, we have if
\( \text{LVal}(s'(B))(x') = \emptyset \) and \( \text{rank}_X(x') > s'(\phi_E) \) then \( \text{LVal}(s'(B))(x') = \emptyset \) and \( \text{rank}_X(x') > s(\phi_E) \). Also by the definition of \( v' \), we have \( v'(\phi_E) = \phi_E \) and \( v'(L) = L[x \leftarrow -] \). As result, we have
if \( v'(L)(x') = \emptyset \) and \( \text{rank}_X(x') > v'(\phi_E) \) then \( L(x') = \emptyset \) and
rank\(\beta(x') > \phi_E\). Thus, the required statement follows from the assumption \(\rho \vdash^k_i v\).

4. We want to show \(\max(v'(\phi_E), v'(\phi_L(x')) \leq \max(s'(\phi_E), \phi_L(s')(x'))\).

By \(s \xrightarrow{w(x,d)} s'\), we have \(s'(\phi_E) = s(\phi_E)\) and \(s'(B) = (x,d) \cdot s(B)\), and therefore we have \(\phi_L(s') = \phi_L(s)[x \leftarrow \max(\text{rank}(x,d), \phi_L(s'))]\).

As a result for the variable \(x\), we have \(\max(s'(\phi_E), \phi_L(s')(x')) = \max(s(\phi_E), \phi_L(s)(x'))\).

Also by the definition of \(v'\), we have \(v'(\phi_L) = \phi_L[x \leftarrow \max(\text{rank}(x,d), \phi_Lmax)\).

As a result for the variable \(x\), we have \(\max(v'(\phi_E), v'(\phi_L(x')) = \max(\phi_E, \text{rank}(x,d), \phi_Lmax)\), and for a variable \(x'\) different from \(x\), we have \(\max(v'(\phi_E), v'(\phi_L(x'))) = \max(\phi_E, \phi_L(x'))\).

The required statement follows from the assumption \(\rho \vdash^k_i v\).

- \(\psi_i^k = r(1)(x,d)\). We get \(s' = s_{i+1}^k\). By the relation \(\rightarrow_O\), we get \(s(q), r((x,d), s'(q)) \in \Delta\) and \(\text{LVal}(s(B)(x)) = d\). By the assumption \(\rho \vdash^k_i v\), we have \(q = s(q)\) and \(L(x) = d\). Let \(v' = \langle s'(q), S, \text{rank}, \phi_E, \phi_L, \phi_P, \rangle\), then by the read(1) rule in the relation \(\rightarrow_O\) we get \(v \xrightarrow{r(1)(x,d)} v'\). To show \(\rho \vdash^k_i v'\) we can check the four conditions for any variable \(x'\) and any value \(d'\) in the same way as in the skip case.

- \(\psi_i^k = r(2)(x,d)\). We get \(s' = s_{i+1}^k\). By the relation \(\rightarrow_O\), we get \(s(q), r((x,d), s'(q)) \in \Delta\), \(\text{LVal}(s(B)(x)) = \emptyset\) and \(\text{rank}_\beta(x) > s(\phi_E)\). By the assumption \(\rho \vdash^k_i v\), we have \(q = s(q)\), \(L(x) = \emptyset\) and \(\text{rank}_\beta(x) > \phi_E\). Let \(v' = \langle s'(q), S, \text{rank}, \phi_E, \phi_L, \phi_P, \rangle\), then by the read(2) rule in the relation \(\rightarrow_O\) we get \(v \xrightarrow{r(2)(x,d)} v'\). To show \(\rho \vdash^k_i v'\) we can check the four conditions for any variable \(x'\) and any value \(d'\) in the same way as in the skip case.

- \(\psi_i^k = r(3)(x,d)\). We get \(s' = s_{i+1}^k\). By the relation \(\rightarrow_O\), we get \(s(q), r((x,d), s'(q)) \in \Delta\), \(\text{rank}(x,d) < s(\phi_P)\). By the assumption \(\rho \vdash^k_i v\), we have \(q = s(q)\) and \(\phi_P = s(\phi_P)\). Let \(v' = \langle s'(q), S, \text{rank}, \phi_E, \phi_L(x), \text{rank}(x,d), \phi_L, \phi_P, \rangle\), then by the read(3) rule in the relation \(\rightarrow_O\) we get \(v \xrightarrow{r(3)(x,d)} v'\). We show \(\rho \vdash^k_i v'\) by checking the four conditions for any variable \(x'\) and any value \(d'\):

1. By the definition of \(v'\) we have \(v'(q) = q = s'(q)\). Also by \(s \xrightarrow{r(3)(x,d)} s'\) and \(\rho \vdash^k_i v\), we get \(v'(S) = s'(S) v'(\phi_P) = s'(\phi_P)\) and \(v'(\text{rank}) = \text{rank} = s'((\text{rank})\).

2. We want to show if \(\text{LVal}(s'(B))(x') = d'\) then \(v'(L)(x') = d'\). This follows from \(s'(B) = s(B), v'(L) = L\) and the assumption \(\rho \vdash^k_i v\).

3. We want to show if \(\text{LVal}(s'(B))(x') = \emptyset\) and \(\text{rank}_\beta(x') > s'(\phi_E)\) then \(v'(L)(x') = \emptyset\) and \(\text{rank}_\beta(x') > v'(\phi_E)\). By \(s \xrightarrow{r(3)(x,d)} s'\), we have \(s'(B) = s(B)\) and \(s'(\phi_E) = \max(s(\phi_E), \text{rank}(x,d))\). Therefore if \(\text{LVal}(s'(B))(x') = \emptyset\) then \(\phi_L(s')(x') = 0\) and \(\phi_L(s)(x') = 0\). As a
result, we have \( s'(\phi_E) = \max(s(\phi_E), \phi_L(s)(x'), \text{rank}(x,d)) \). From the definition of \( v' \), we have \( v'(L) = L \) and \( v'(\phi_E) = \max(\phi_E, \phi_L(x), \text{rank}(x,d)) \).

The required statement follows from the assumption \( \rho \vdash^k_v \).

4. We want to show \( \max(v'(\phi_E), v'(\phi_L))(x') \leq \max(s'(\phi_E), \phi_L(s')(x')) \).

From the definition of \( v' \), we have \( v'(\phi_E) = \max(\phi_E, \phi_L(x), \text{rank}(x,d)) \) and \( v'(\phi_L) = \phi_L \).

As a result, we have \( \max(v'(\phi_E), v'(\phi_L))(x') = \max(\phi_E, \phi_L(x), \text{rank}(x,d), \phi_L(x')). \)

By \( s \xrightarrow{\tau(3)(x,d)} s' \), we have \( s'(B) = s(B), s'(\phi_E) = \max(s(\phi_E), \text{rank}(x,d)) \) and \( \phi_L(s)(x) = 0 \). As a result, we have \( \max(s'(\phi_E), \phi_L(s')(x')) = \max(s(\phi_E), \text{rank}(x,d), \phi_L(s)(x), \phi_L(s)(x')). \)

The required statement follows from the assumption \( \rho \vdash^k_v \).

- \( \psi^k_i = u(1)(x,d) \). We get \( s' = s^k_{i+1} \). We show \( \rho \vdash^k_v \) by checking the four conditions for any variable \( x' \) and any value \( d' \):

1. The first condition follows from \( s \xrightarrow{u(1)(x,d)} s' \) and the assumption \( \rho \vdash^k_v \).

2. We want to show if \( \text{LVal}(s'(B))(x') = d' \) then \( L(x') = d' \). By \( s \xrightarrow{u(1)(x,d)} s' \), we have \( s(B) = s'(B) \cdot (x,d) \) and therefore if \( \text{LVal}(s'(B))(x') = d' \) then \( \text{LVal}(s(B))(x') = d' \). The required statement follows from the assumption \( \rho \vdash^k_v \).

3. We want to show if \( \text{LVal}(s'(B))(x') = \emptyset \) and \( \text{rank}_X(x') > s'(\phi_E) \) then \( L(x') = \emptyset \) and \( \text{rank}_X(x') > \phi_E \). By \( s \xrightarrow{u(1)(x,d)} s' \), we have \( s'(\phi_E) = \max(\phi_E, \text{rank}(x,d)) \) and \( s(B) = s'(B) \cdot (x,d) \). If \( x' \) is the variable \( x \) then required statement follows because \( \text{rank}_X(x) \leq \text{rank}(x,d) \leq s'(\phi_E) \). If \( x' \) is different from \( x \) then \( \text{LVal}(s'(B))(x') = \text{LVal}(s(B))(x') \) and therefore the required statement follows from \( s'(\phi_E) \geq s(\phi_E) \) and the assumption \( \rho \vdash^k_v \).

4. We want to show \( \max(\phi_E, \phi_L(x')) \leq \max(s(\phi_E), \phi_L(s')(x')) \).

By \( s \xrightarrow{u(1)(x,d)} s' \), we have \( s'(\phi_E) = \max(\phi_E, \text{rank}(x,d)) \) and \( s(B) = s'(B) \cdot (x,d) \). Observe then \( \phi_L(s)(x') = \max(\text{rank}(x,d), \phi_L(s')(x')) \). Thus, \( \max(s(\phi_E), \phi_L(s)(x')) = \max(s(\phi_E), \text{rank}(x,d), \phi_L(s')(x')) = \max(s(\phi_E), \phi_L(s')(x')) \).

The required statement follows from the assumption \( \rho \vdash^k_v \).

\[\square\]

G Proof of Lemma 6.1

Let \( \text{Act}' = \{\text{skip}, \text{push}(\gamma), \text{pop}(\gamma) \mid \gamma \in \Gamma\} \).

Lemma G.1. For a register machine \( R = (Q, q_{\text{init}}, \Gamma, R, V, \text{Act}, \Delta) \), there is a PDS \( \mathcal{P} = (Q', q'_{\text{init}}, \Gamma, \text{Act}', \Delta') \) such that:

1. \( |\mathcal{P}| = \mathcal{O}(|Q| + |\Delta| \cdot |V||R|) \).
2. For each control state $q$ in $Q$ there is a unique set $C$ of control states in $Q'$ such that $q$ is reachable in $T_R$ iff the set $C$ is reachable in $P$. 

Proof. The main idea of the proof is to compensate the use of the registers by using $|V|^{|R|}$ copies of $Q$ and $\Delta$ where each copy simulates a unique register configuration. Inside a copy we remove all the read and comparison actions that do not agree with the corresponding register configuration and replace the other read and comparison actions by skip actions. We also replace all the write actions by moves between different copies using also skip actions which corresponds to changes in the register configuration. We construct $P'$ as follows:

1. For each control state $q$ in $Q$, except for $q_{init}$, and each register configuration $R$, add the control state $q^R$ to $Q'$.

2. Let $q_{init}$ correspond to $q_{init}^{R_s}$.

3. For each transition $\langle q, act, q' \rangle \in \Delta$, where $act \in Act'_R$, add the transition $\langle q^R, act, q'^R \rangle$ to $\Delta'$ for each register configuration $R$ such that $q^R, q'^R \in Q'$.

4. For each transition $\langle q, w(r, v), q' \rangle \in \Delta$, add the transition $\langle q^{R_1}, skip, q'^{R_2} \rangle$ to $\Delta'$ for each two register configurations $R_1$ and $R_2$ such that $R_2 = R_1[r \leftarrow v]$, and $q^{R_1}, q'^{R_2} \in Q'$.

5. For each transition $\langle q, addr(r), q' \rangle \in \Delta$, add the transition $\langle q^{R_1}, skip, q'^{R_2} \rangle$ to $\Delta'$ for each two register configurations $R_1$ and $R_2$ such that $R_2 = R_1[r \leftarrow R(r) + 1]$, and $q^{R_1}, q'^{R_2} \in Q'$.

6. For each transition $\langle q, wfr(r_1, r_2), q' \rangle \in \Delta$, add the transition $\langle q^{R_1}, skip, q'^{R_2} \rangle$ to $\Delta'$ for each two register configurations $R_1$ and $R_2$ such that $R_2 = R_1[r_2 \leftarrow R_1(r_1)]$, and $q^{R_1}, q'^{R_2} \in Q'$.

7. For each transition $\langle q, r(r, v), q' \rangle \in \Delta$, add the transition $\langle q^R, skip, q'^R \rangle$ to $\Delta'$ for each register configuration $R$ such that $R(r) = v$ and $q^R \in Q'$.

8. For each transition $\langle q, comp(r_1, r_2), q' \rangle \in \Delta$, add the transition $\langle q^R, skip, q'^R \rangle$ to $\Delta'$ for each register configuration $R$ such that $R(r_1) = R(r_2)$ and $q^R \in Q'$.

9. For each transition $\langle q, compg(r_1, r_2), q' \rangle \in \Delta$, add the transition $\langle q^R, skip, q'^R \rangle$ to $\Delta'$ for each register configuration $R$ such that $R(r_1) > R(r_2)$ and $q^R \in Q'$.

10. For each transition $\langle q, compg(r_1, r_2), q' \rangle \in \Delta$, add the transition $\langle q^R, skip, q'^R \rangle$ to $\Delta'$ for each register configuration $R$ such that $R(r_1) \geq R(r_2)$ and $q^R \in Q'$.

From the construction, a state $q$ in $Q$ is reachable in $T_R$ iff at least one of its copies in $Q'$ is reachable in $P$. Also, the number of different registers configurations is $|V|^{|R|}$. It follows that $|Q'| = \mathcal{O}(|Q| \cdot |V|^{|R|})$ from steps 1 and 2 in the construction, and $|\Delta'| = \mathcal{O}(|\Delta| \cdot |V|^{|R|})$ follows from steps 3-10. \qed
Proof of lemma 6.1 Let the register machine $R = (Q, q'_{init}, \Gamma, R, V, Act_R, \Delta)$. By lemma G.1 we get that there is a PDS $P = (Q', q'_{init}, \Gamma, Act', \Delta')$ where $|P| = O((|Q| + |\Delta|) \cdot |V|^{|R|})$, and the state reachability problem under $T_R$ is reducible to the state reachability problem under $P$. The lemma then follows from the result given in $R$, which states that the time complexity of the state reachability problem under $P$ is polynomial in terms of $|P|$.

References


