Xiao LL, Zhu HB, Xu QW. Trace semantics and algebraic laws for total store order memory model. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(6): 1269–1290 Nov. 2021. DOI 10.1007/s11390-021-1616-1

Trace Semantics and Algebraic Laws for Total Store Order Memory Model

Li-Li Xiao[1], Hui-Biao Zhu[1,*], *Member*, *CCF*, and Qi-Wen Xu[2]

[1] *Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China*

[2] *Department of Computer and Information Science, Faculty of Science and Technology, University of Macau Macau, China*

E-mail: lilixiao@stu.ecnu.edu.cn; hbzhu@sei.ecnu.edu.cn; qwxu@um.edu.mo

Received May 27, 2021; accepted November 7, 2021.

Abstract Modern multiprocessors deploy a variety of weak memory models (WMMs). Total Store Order (TSO) is a widely-used weak memory model in SPARC implementations and x86 architecture. It omits the store-load constraint by allowing each core to employ a write buffer. In this paper, we apply Unifying Theories of Programming (abbreviated as UTP) in investigating the trace semantics for TSO, acting in the denotational semantics style. A trace is expressed as a sequence of snapshots, which records the changes in registers, write buffers and the shared memory. All the valid execution results containing reorderings can be described after kicking out those that do not satisfy program order and modification order. This paper also presents a set of algebraic laws for TSO. We study the concept of head normal form, and every program can be expressed in the head normal form of the guarded choice which is able to model the execution of a program with reorderings. Then the linearizability of the TSO model is supported. Furthermore, we consider the linking between trace semantics and algebraic semantics. The linking is achieved through deriving trace semantics from algebraic semantics, and the derivation strategy under the TSO model is provided.

Keywords weak memory model, Total Store Order (TSO), trace semantics, algebraic law, Unifying Theories of Programming (UTP)

1 Introduction

When considering parallel programming, there are mainly two paradigms, namely message passing and shared memory [1]. The latter one accesses shared data through reading from and writing to the shared memory, and many consistency models are applied to this paradigm [2]. The strongest and most intuitive memory consistency model is sequential consistency (SC) [3]. It states that the operations from different threads take effect in an interleaving manner, and the operations from the same thread appear in the order specified by their program order. A weak memory model refers to a model being optimized but producing the behaviors which do not conform to SC. Among weak (or relaxed) memory models, Total Store Order (TSO) is a widely implemented model [4].

The TSO memory model is supported by the x86 architecture and SPARC implementations [5]. As shown in Fig.1, every memory write (or store) works on its thread's write buffer firstly, and the information in the write buffer will be written to the shared memory at some point in the future. Thus, the effect of one memory write may be delayed. Since the write buffer conforms to the principle named First-In-First-Out (FIFO) [6], the order of store-store can be guaranteed. A read from location x (aka load) always demands to first check whether the processor's private

Regular Paper

Special Section on Software Systems 2021—Theme: Dependable Software Engineering

This work was partly supported by the National Key Research and Development Program of China under Grant No. 2018YFB2101300, the National Natural Science Foundation of China under Grant Nos. 61872145 and 62032024, and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things under Grant No. ZF1213.

*Corresponding Author

buffer contains such a write to the same location. If so, the latest value is returned and the read operation terminates successfully. Otherwise, the read will access the main memory. Consequently, as execution proceeds, the store-load order can be broken because each core in this model has a private write buffer. However, this model still maintains load-load, load-store and store-store constraints: 1) two loads or stores cannot be reordered, and 2) reordering an earlier load with a later store causes incorrect behaviors. Also, this memory model introduces fence instructions to ensure the absolute order of a process [7].

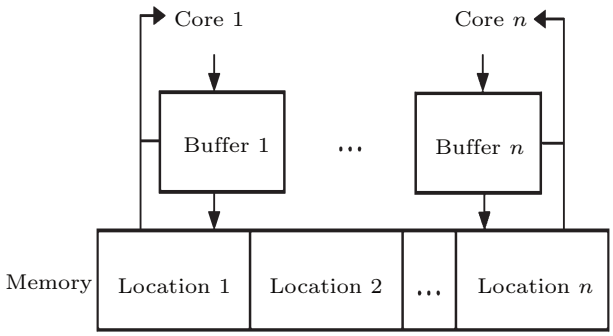

Fig.1. Total Store Order architecture.

Now we use a typical parallel program below in the store buffering (SB) [8] to help to understand how the store-load constraint is relaxed in the TSO memory model through the operations writing to the write buffer and propagating to the shared memory. The notation ";" denotes the sequential composition operator, while "||" stands for the parallel composition operator. The analysis of the program is given as follows.

$$\begin{pmatrix} x := 1; \\ a := y \end{pmatrix} \parallel \begin{pmatrix} y := 1; \\ b := x \end{pmatrix}.$$

Let variables x and y hold 0 initially. When performing $x := 1$ and $y := 1$, the values of x and y are written to both write buffers respectively, and then the reads from y and x may happen before the writes from buffers to the shared memory. Therefore, it is possible for variables a and b to both obtain 0 in the same execution.

As described in Hoare and He's Unifying Theories of Programming [9], three different mathematical models are often used to represent a theory of programming, namely, the operational [10], the denotational [11], and the algebraic approaches [12]. Each of these representations has its distinctive advantages for theories of programming. For instance, the operational semantics provides a set of transition rules that model how a program performs step by step. The denotational semantics indicates what a program does. The algebraic semantics is well suited in the symbolic calculation of parameters and structures of an optimal design. The algebraic approach has been successfully applied in provably correct compilation [13, 14]. However, there are few studies on studying both the denotational semantics and the algebraic semantics of the TSO memory model, which give the intuitive description of all the program executions containing various reorderings.

In this paper, we consider the denotational semantics for the TSO memory model, where our approach is based on UTP and the trace structure is applied. Compared with the technique called pomset presented in [15], our investigation for denotational semantics with trace semantics focuses on the results of execution sequences. Then, for the concept of linearizability [16, 17] can be reflected by our trace semantics, our approach can provide the precise understanding of the TSO model in a simpler way. In our semantic model, a trace is expressed as a sequence of snapshots which record the changes of the variables brought by different types of actions. To calculate all the possible traces, we use two main functions named *po* and *mo* which depict the program order and the modification order to filter out those illegal sequences respectively.

This paper also investigates algebraic laws including a set of sequential and parallel expansion laws. We introduce the concept of guarded choice for TSO, which is made up of a variety of guarded components. The concept of head normal form is explored, and every program can be expressed in the head normal form of guarded choice. Hence, from the perspective of the algebraic semantics, our approach can support the linearizability of the TSO memory model. In addition, we provide the strategy for deriving trace semantics from algebraic semantics. Based on the derivation strategy, the trace semantics of each program can be calculated.

The reminder of this paper is organized as follows. We study the trace semantics of TSO in Section 2. Section 3 lists a set of algebraic laws including sequential and parallel expansion laws. Section 4 discusses related work about the TSO memory model, the UTP approach and semantic linking. We conclude the paper and present the future work in Section 5. We leave some technical definitions in Appendix.

2 Trace Semantics

2.1 Syntax of TSO

The language of the TSO model which is adapted and extended from [15] is presented in the following, where v ranges over real numbers, e over arithmetic expressions on real numbers, h over Boolean expressions and p over programs. Fence is a barrier, and two memory accesses separated by it cannot be reordered. We have analyzed a simple example program in Section 1.

$v ::= ..., -2, -1, 0, 1, 2, ...$
$e ::= v \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid ...$
$h ::= \text{true} \mid \text{false} \mid e_1 = e_2 \mid \neg h \mid h_1 \vee h_2 \mid h_1 \wedge h_2 \mid ...$
$p ::= x := e \mid fence \mid \text{if } h \text{ then } p_1 \text{ else } p_2 \mid \text{while } h \text{ do } p \mid p_1; p_2 \mid p_1 || p_2$

Here:

- The variable x can be global or local.
- The operator "=" means that the value on the left of "=" is equal to that on the right.
- $\neg h$, $h_1 \vee h_2$ and $h_1 \wedge h_2$ are logic notations. $\neg h$ denotes the negation of h, the notation $h_1 \vee h_2$ stands for the disjunction between h_1 and h_2, and $h_1 \wedge h_2$ represents the conjunction between h_1 and h_2.

2.2 Semantic Model

In this subsection, we investigate the trace semantic model of TSO. We illustrate the behaviors of a process with a trace of snapshots, which records the sequence of actions.

A snapshot in a trace is expressed as a tuple in the form: $(id, cont, oflag, eflag)$.
Here:

- id is used to number the statements and it starts from 1 in each program. The reason for using this element here is to differentiate two same statements, which will be explained in detail later.
- $cont$ can be: 1) in the form of (var, val) indicating the data state of one variable at some point, or 2) a particular unary element fence.
- We use parameter $oflag$ to distinguish different kinds of operations, which is simply discussed in Table 1.

Table 1. Different Types of Operations Divided by $oflag$

$oflag$	Type
0	Fence instructions
1	Committing to write buffers
2	Propagating to the shared memory
3	Register writes

1) Considering global assignments, if $oflag$ is 1, the action committing to the write buffer is carried out, and the operation propagating to the whole memory is described when $oflag$ is equal to 2.

2) $oflag = 3$ denotes writing to local variables (i.e., register writes).

3) In addition, a special situation is that the operation is fence and the corresponding $oflag$ is 0.

- $eflag$ is applied to mark whether the operation is performed by the process itself or its environment. Once the process does the action on its own, it sets the variable to be 1. Otherwise, $eflag$ will be 0.

And the projection function π_i ($i \in \{1, 2, 3, 4\}$) is defined to get the i-th element of a snapshot.

$$\pi_1(id, cont, oflag, eflag) = id,$$
$$\pi_2(id, cont, oflag, eflag) = cont,$$
$$\pi_3(id, cont, oflag, eflag) = oflag,$$
$$\pi_4(id, cont, oflag, eflag) = eflag.$$

If var and val exist (in other words, $cont$ is not described as fence), we call the projection function π_i ($i \in \{1, 2\}$) to obtain the variable and value respectively.

$$\pi_1(\pi_2(id, cont, oflag, eflag)) = var,$$
$$\pi_2(\pi_2(id, cont, oflag, eflag)) = val.$$

We use $traces(P)$ to describe the set containing all possible behaviors of process P, and $sb(P)$ to record the original order of program statements, which facilitates the model of reorderings. When making sequential composition, $sb(P)$ is used to check whether the interleaved trace (explained in Subsection 2.3) is valid or not with the definition of program order and modification order. Each element of a sequence in the set sb has the form of $(id, cont)$. The meanings of id and $cont$ here are the same as above.

Now two examples are given to intuitively illustrate the structures of $traces$ and sb.

Example 1. Assume a is a local variable and x is a global variable.

$$1... \ a := x + 1;$$
$$2... \ a := x + 1.$$

Here, we may understand the importance of the use of id, because the two statements in $a := x + 1; a := x + 1$ are described in the same way $(a, r(x + 1))$ (i.e., $(a, (r(x) + 1))$). The values 1 and 2, which are prefixes

of the statements above and framed in the formulas below, have the ability to differentiate them. $r(x)$ is a function for reading variable x, and its detailed definition is given in Appendix A.

$$\begin{aligned} &traces(a := x+1; a := x+1) \\ &= \left\{\langle(\boxed{1}, (a, r(x+1)), 3, 1), (\boxed{2}, (a, r(x+1)), 3, 1)\rangle\right\}. \\ &sb(a := x+1; a := x+1) \\ &= \left\{\langle(\boxed{1}, (a, r(x+1))), (\boxed{2}, (a, r(x+1)))\rangle\right\}. \end{aligned}$$

Example 2. Assume variable x is global. Consider the program below.

$$\begin{aligned} &1...\ x := 1; \\ &2...\ x := 2. \end{aligned}$$

A global assignment may be split into different actions acting on the write buffer or memory in the sequences of *traces*. However, *sb* relates to the original statements merely.

$$\begin{aligned} &traces(x := 1; x := 2) \\ &= \left\{\begin{array}{l} \langle(\boxed{1}, (x, 1), 1, 1), (\boxed{1}, (x, 1), 2, 1), \\ (2, (x, 2), 1, 1), (2, (x, 2), 2, 1)\rangle, \\ \langle(\boxed{1}, (x, 1), 1, 1), (2, (x, 2), 1, 1), \\ (\boxed{1}, (x, 1), 2, 1), (2, (x, 2), 2, 1)\rangle \end{array}\right\}. \\ &sb(x := 1; x := 2) = \{\langle(1, (x, 1)), (2, (x, 2))\rangle\}. \end{aligned}$$

It is worth noting that the parameter *id* of the snapshot $(1, (x, 1), 1, 1)$ and that of $(1, (x, 1), 2, 1)$ are identical.

2.3 Trace Semantics

In this subsection, we present the trace semantics for each program P under the TSO memory model, where $traces(P)$ and $sb(P)$ are defined for each program P.

Local Assignment. The local variables are written to the private registers directly.

$$\begin{aligned} &traces(a := e) \\ &=_{\text{df}} \{s^\frown\langle(1, (a, r(e)), 3, 1)\rangle\}^{①}, \text{ where } \pi_4^*(s) \in 0^*. \\ &sb(a := e) =_{\text{df}} \{\langle(1, (a, r(e)))\rangle\}. \end{aligned}$$

The change of data state aiming at a local variable a is described by the third parameter 3.

Here, s denotes the trace produced by the environment and we use 0^* to allow the environment to carry out any number of operations. Each snapshot in s is in the form $(\_, \_, \_, 0)$, i.e., the $eflag$ element is 0. With this approach to include the environment's behaviors for a process, the process can get the contributions produced by its environment. As mentioned before, the projection function π_4 is to obtain the fourth element of a snapshot. Then, the notation $\pi_4^*(s)$ represents the repeated execution of the function π_4 on each snapshot in trace s. 0^* stands for the sequence containing any number of the integer 0.

Example 3. In this example, we focus on the program $a := x$ performed in core 1, where a is a local variable and x is a global variable.

Firstly, the environment (i.e., the program $x := 1$ in core 2) has updated the value of the variable x, through the actions labeled by (1) and (2) in Fig.2. The modified value is 1. For the program $a := x$, one trace is $\langle(1, (x, 1), 1, \boxed{0}), (1, (x, 1), 2, \boxed{0}), (1, (a, r(x)), 3, \boxed{1})\rangle$. For simplicity, we ignore other environment operations. The former two snapshots are contributed by the environment, and the last with $eflag$ being 1, is produced by the thread itself.

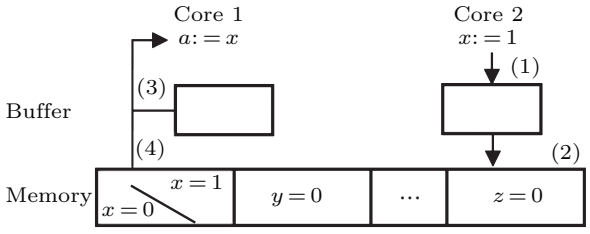

Fig.2. Analysis of local assignment.

Faced with some statements whose values are expressions instead of constants, we need to apply the read function r to get the dynamic values. Given a variable x, the detailed formalization and explanation of the read function $r(x)$ can be found in Appendix A. Further, for an expression e, we should carry out the read function for each variable in it. After getting the values of those variables, the value of expression e can be computed. For example, $r(x + y)$ is expressed as $r(x) + r(y)$ in our model.

Example 3: *Continuation.* Here, we start to discuss the read function r appearing in the trace $\langle(1, (x, 1), 1, 0), (1, (x, 1), 2, 0), (1, \boxed{(a, r(x))}, 3, 1)\rangle$.

The value of x, which is going to be assigned to the local variable a, can be obtained by the actions (3) and (4) in Fig.2. The buffer in core 1 is explored firstly,

①The notation $=_{\text{df}}$ refers to definitions, whereas $s^\frown t$ stands for the concatenation of traces s and t. Further, $s^\frown T = \{s^\frown t \mid t \in T\}$ and $S^\frown T =_{\text{df}} \{s^\frown t \mid s \in S \wedge t \in T\}$. It indicates that $s^\frown \phi = \phi$.

shown by action (3). Because such a snapshot whose var, $oflag$ and $eflag$ are x, 1 and 1 respectively does not exist in the above trace, nothing can be returned from the private buffer. Then the memory is searched which is exhibited by action (4), and value 1 is provided.

The definition for $traces(fence)$ and $sb(fence)$ is similar to that of local assignment:

$$\begin{aligned} &traces(fence)\\ &=_{\mathrm{df}}\{s^\frown\langle(1, fence, 0, 1)\rangle\},\ \text{where } \pi_4^*(s)\in 0^*.\\ &sb(fence) =_{\mathrm{df}} \{\langle(1, fence)\rangle\}. \end{aligned}$$

Global Assignment. The execution of a global assignment $x := e$ can be separated into two steps: writing into its own thread's write buffer and then propagating to the shared memory from the write buffer. And this order is fixed.

$$\begin{aligned} &traces(x := e)\\ &=_{\mathrm{df}}\{u^\frown\langle(\boxed{1}, (x, r(e)), 1, 1)\rangle^\frown v^\frown\langle(\boxed{1}, (x, r(e)), 2, 1)\rangle\}, \end{aligned}$$

where $\pi_4^*(u)\in 0^*$ and $\pi_4^*(v)\in 0^*$.

Note that committing and propagating the same memory write have the same value of parameter id shown in the framed area above.

Similar to local assignments, the environment can do any number of operations before each step of the global assignment, which is expressed by adding subtraces u and v in the above formula.

For the snapshot $(1, (x, r(e)), 1, 1)$, $x := e$ is the first statement in the program, and then the first element id has the value of 1. With the second one $(x, r(e))$, we can know the data state currently, which indicates that the value of x is $r(e)$ when performing $x := e$. The calculation of expression e can be completed via the read function r. The third one is used to describe the place (the write buffer or the main memory) where $x := e$ plays a role. Here, the value 1 tells that the program brings an effect to the thread's write buffer. Because the action we discuss is carried out by the process itself, we set the last element to be 1.

In addition we introduce sb to construct sequential composition in a better way. The function is to record the serial numbers and data states of the original program statements.

$$sb(x := e) =_{\mathrm{df}} \{\langle(1, (x, r(e)))\rangle\}.$$

Sequential Composition. In this subsection, we concentrate on investigating the trace semantics of sequential composition, and it mainly reflects the thread local reordering. The investigation is always separated into two steps: 1) interleaving two traces s and t to calculate all the sequences, where s and t are the traces of the two processes which make sequential composition, and 2) applying the functions named po and mo which give the description of the program order and the modification order to remove those invalid executions.

For the first step, to interleave two traces s and t, where updating the sequence number of statements (i.e., id) in snapshots is considered, we introduce a function $inleave(s, t, pc)$ below. Here pc records the new sequence number in the interleaved trace and it starts from 1. Thus compared with the traditional interleaving semantics, the one explained in this paper introduces the occurrence of pc. Further, $inleave(s, t, 1)$ is applied when we give the semantics for sequential composition.

Before giving the detailed formalization and explanation of the function $inleave$, we use example 4 below to provide intuitive understanding.

Example 4. Let variables x and y be both global, and a be a local variable. We consider one trace $s = \langle(1, (x, 1), 1, 1), (1, (x, 1), 2, 1)\rangle$ of the program $x := 1$ and another trace $t = \langle(1, (a, r(y)), 3, 1)\rangle$ of $a := y$. For simplicity, the environment operations are not exhibited here.

The analysis of one trace $s; t$ interleaved from s and t shown in Fig.3 is given as follows.

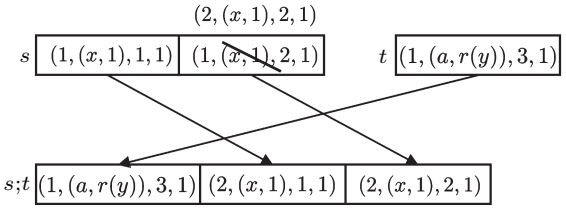

Fig.3. Illustration of function $inleave$.

1) The snapshot $(1, (a, r(y)), 3, 1)$ in the latter trace t is scheduled first. It is added as the head of trace $s; t$ merely.

2) The first snapshot $(1, (x, 1), 1, 1)$ in trace s is selected to be interleaved. The value of its $oflag$ is 1 and it is the snapshot of committing the write to location x to the write buffer. Because there already exists a snapshot which originates from another statement in the interleaved trace $s; t$, parameter id should be changed to 2. At the same time, id of the snapshot of the corresponding propagation action (i.e., the second snapshot $(1, (x, 1), 2, 1)$ in s) is also modified to 2. Finally we place the updated snapshot $(\boxed{2}, (x, 1), 1, 1)$ in the second of $s; t$ and continue the interleaving procedure.

3) We put the last snapshot in s (i.e., the updated snapshot $(\boxed{2}, (x, 1), 2, 1)$) in the third position of trace $s; t$.

Now, we give the formal definition of the function $inleave(s, t, pc)$. For the function $inleave(s, t, pc)$, the result of interleaving two empty traces is still empty, which is illustrated in case 1. For the two traces which need to be interleaved, if one of them is empty and the other is nonempty, the result follows the nonempty one, and case 2 describes it.

$$\begin{aligned} &\text{case 1} \quad inleave(\langle\rangle, \langle\rangle, pc) =_{\text{df}} \{\langle\rangle\}.\\ &\text{case 2} \quad inleave(s, \langle\rangle, pc) =_{\text{df}} \{s\},\\ &\qquad\quad inleave(\langle\rangle, t, pc) =_{\text{df}} \{t\}. \end{aligned}$$

For general scenarios, $inleave(s, t, pc)$ can engage in the first snapshot in s (described by the function $inleave_l$) or in the next trace t (illustrated by $inleave_r$), denoted by case 3.

$$\begin{aligned} &\text{case 3} \quad inleave(s, t, pc)\\ &\quad =_{\text{df}} \quad inleave_l(s, t, pc) \cup inleave_r(s, t, pc). \end{aligned}$$

Because the treatment of interleaving the two branches is similar, we illustrate the function $inleave_l(s, t, pc)$ for example.

When making sequential composition of two programs, what we focus on in the interleaving procedure is the operations contributed by the programs instead of their environment. Hence we skip those snapshots with $eflag$ being 0, which is formalized by (1). Here, notation $hd(s)$ stands for the first snapshot of trace s, and $tl(s)$ is applied to denote the result of removing the first snapshot in trace s.

$$\begin{aligned} &inleave_l(s, t, pc)\\ =_{\text{df}} \Bigg(&hd(s)^\frown inleave(tl(s), t, pc) && (1)\\ &\quad \lhd\ \pi_4(hd(s)) = 0\ \rhd\\ &\Bigg(\boxed{hd(s)[pc/\pi_1(hd(s))]}\ ^\frown inleave(\boxed{tl(s)[(pc, \_, 2, 1)/(\pi_1(hd(s)), \_, 2, 1)]}, t, pc + 1) && (2)\\ &\quad \lhd\ \pi_3(hd(s)) = 1\ \rhd\\ &\Bigg(\boxed{hd(s)}\ ^\frown inleave(tl(s), t, pc) && (3)\\ &\quad \lhd\ \pi_3(hd(s)) = 2\ \rhd\\ &\boxed{hd(s)[pc/\pi_1(hd(s))]}\ ^\frown inleave(tl(s), t, pc + 1) && (4) \Bigg) \Bigg) \Bigg). \end{aligned}$$

With regard to the operations performed by the thread itself, we are supposed to take three situations into our account. If the interleaving procedure starts from the first snapshot $hd(s)$ in trace s, whose operation is writing to the write buffer, there are mainly three steps modeled by (2). Above all, for the snapshot whose id is identical to that of $hd(s)$ in the remaining trace $tl(s)$, we change its id to the current program counter pc. It is formalized in the second boxed area in (2). Afterwards, the parameter id of $hd(s)$ is modified to pc, and then the updated $hd(s)$ is placed in the head, described by the first boxed area in (2). Finally, pc increases by 1 and the interleaving procedure continues.

Here, $e \lhd b \rhd f$ represents e if condition b is true; otherwise f. The notation $hd(s)[u/v]$ describes the replacement of v by u in snapshot $hd(s)$.

If the interleaving procedure is triggered by the snapshot whose $oflag$ is 2, we only need to put the snapshot in the head of the interleaving of traces s and t, since its id has been changed previously, denoted by (3). Otherwise, before placing the first snapshot, we should set its id to pc, and make pc add 1 subsequently, illustrated by (4) above.

Based on the above analysis, we can know that these situations modeled by (2) and (4) result in the difference between the traditional interleaving semantics and the one introduced here.

Now we use example 5 below to discuss the trace semantics of sequential composition with the application of the function $inleave$.

Example 5. Consider the sequential composition $x := 1; fence; a := x$, where

$$\begin{aligned} s &= traces(x := 1) = \{\langle (1, (x, 1), 1, 1),\ (1, (x, 1), 2, 1)\rangle\},\\ t &= traces(fence; a := x)\\ &= \{\langle (1, fence, 0, 1),\ (2, (a, r(x)), 3, 1)\rangle\}. \end{aligned}$$

Variable x is global and a is a local variable. For simplicity, we do not take the environment operations into our consideration here.

Step 1. We use the function $inleave$ to generate the

original traces.

$$
\begin{aligned}
&inleave(s,t,1)\\
&=\left\{\begin{array}{ll}
\langle(1,(x,1),1,1),\ (1,(x,1),2,1),\\
\ (2,fence,0,1),\ (3,(a,r(x)),3,1)\rangle, & (5)\\
\langle(1,(x,1),1,1),\ (2,fence,0,1),\\
\ (1,(x,1),2,1),\ (3,(a,r(x)),3,1)\rangle, & (6)\\
\langle(1,(x,1),1,1),\ (2,fence,0,1),\\
\ (3,(a,r(x)),3,1),\ (1,(x,1),2,1)\rangle, & (7)\\
\langle(1,fence,0,1),\ (2,(x,1),1,1),\\
\ (2,(x,1),2,1),\ (3,(a,r(x)),3,1)\rangle, & (8)\\
\langle(1,fence,0,1),\ (2,(x,1),1,1),\\
\ (3,(a,r(x)),3,1),\ (2,(x,1),2,1)\rangle, & (9)\\
\langle(1,fence,0,1),\ (2,(a,r(x)),3,1),\\
\ (3,(x,1),1,1),\ (3,(x,1),2,1)\rangle & (10)
\end{array}\right\}.
\end{aligned}
$$

Intuitively, we discover that cases (6), (7), (8), (9) and (10) are not in line with reality and should be removed. Then we complete it with the functions defined below.

Owing to the fact that the environment does not affect the interleaving process, the following functions only act on those snapshots with $eflag$ being 1. Now, we give the introduction to function po to illustrate the program order specified by a program. po can be achieved through selecting the quads (i.e., the snapshots in the trace) with the value of the third element being 0 or 1 or 3. Because we are going to compare the result with the sequence in the sb set, the first two elements of the quad are within our consideration. It says that the parameter of the function po is a trace, and its return value is the sequence of $(id, cont)$. The definition is provided in the following.

$$
\begin{aligned}
&po(\langle event\rangle \ ^\wedge\ tr)\\
&=_{\mathrm{df}}\left(\begin{array}{l}
\langle(\pi_1(event),\pi_2(event))\rangle \ ^\wedge\ po(tr)\\
\quad \lhd \left(\begin{array}{l}\pi_3(event)=0\ \vee\\ \pi_3(event)=1\ \vee\\ \pi_3(event)=3\end{array}\right)\wedge \pi_4(event)=1 \rhd\\
po(tr)
\end{array}\right),\\
&po(\langle\rangle)=_{\mathrm{df}}\langle\rangle.
\end{aligned}
$$

In addition to program order, we illustrate the program's modification order with function mo. This case is connected with the operations propagating to the whole memory and the special instruction fence.

$$
\begin{aligned}
&mo(\langle event\rangle \ ^\wedge\ tr)\\
&=_{\mathrm{df}}\left(\begin{array}{l}
\langle(\pi_1(event),\pi_2(event))\rangle \ ^\wedge\ mo(tr)\\
\quad \lhd \left(\begin{array}{l}\pi_3(event)=0\ \vee\\ \pi_3(event)=2\end{array}\right)\wedge \pi_4(event)=1 \rhd\\
mo(tr)
\end{array}\right),\\
&mo(\langle\rangle)=_{\mathrm{df}}\langle\rangle.
\end{aligned}
$$

Finally, based on the above formalizations, we give the trace semantics of the sequential composition $P;Q$. This process can be split into four steps below.

• Firstly, we need to choose one trace s contributed by program P and one trace t produced by Q. And then we interleave them.

• One execution order of statements in P is recorded by sb_1 and that in Q is sb_2. These parameters can be obtained through the definition of $c(P,Q)$. Then, we define notation sb_1+sb_2 to denote the execution order of $P;Q$.

• For the interleaved trace u, it is necessary to check whether the result of the function po acting on u is a subsequence of sb_1+sb_2. And it is the same to mo. When finishing these examinations formalized by $(s;t,sb)$ below, if trace u is invalid, it will be deleted. Here, function $subseq(u,v)$ indicates that u is a subsequence of v.

• If all the traces generated by P and Q are interleaved and examined by the above three steps, we can achieve the trace semantics of the sequential program $P;Q$.

$$
\begin{aligned}
&(s;t,sb)\\
&=_{\mathrm{df}}\left\{\begin{array}{l}u|subseq(po(u),sb)\wedge subseq(mo(u),sb)\\ \quad\wedge\ u\in inleave(s,t,1)\end{array}\right\},\\
&traces(P;Q)=_{\mathrm{df}}\bigcup_{c(P,Q)}(s;t,sb_1+sb_2),
\end{aligned}
$$

where

$$
\begin{aligned}
&c(P,Q)=_{\mathrm{df}}\left(\begin{array}{l}s\in traces(P)\wedge t\in traces(Q)\wedge\\ sb_1\in sb(P)\wedge sb_2\in sb(Q)\end{array}\right),\\
&sb_1+sb_2=_{\mathrm{df}}\ sb_1^\wedge sb_2',
\end{aligned}
$$

where

$$
\begin{aligned}
sb_2'=_{\mathrm{df}}\ &\forall(id,cont)\in sb_2\bullet\\
&sb_2[((len(sb_1)+id),cont)/(id,cont)].
\end{aligned}
$$

Here, $\bigcup$ stands for the union of the trace sets. The symbol "•" means "such that" [9].

Example 5: *Continuation.*

Step 2. We know that one execution sequence sb_1 of $sb(P)$ is $\langle(1,(x,1))\rangle$, and sb_2 of $sb(Q)$ is $\langle(1,fence),(2,(a,r(x)))\rangle$. After checking whether the return values of functions po and mo are both subsequences of sb_1+sb_2, we filter the results presented in step 1.

$$
\begin{aligned}
&sb_1+sb_2=\langle(1,(x,1)),(2,fence),(3,(a,r(x)))\rangle,\\
&mo((6))=\langle(2,fence),(1,(x,1))\rangle,
\end{aligned}
$$

$$mo((7)) = \langle(2, fence), (1, (x, 1))\rangle,$$
$$po((8)) = \langle(1, fence), (2, (x, 1)), (3, (a, r(x)))\rangle,$$
$$po((9)) = \langle(1, fence), (2, (x, 1)), (3, (a, r(x)))\rangle,$$
$$po((10)) = \langle(1, fence), (2, (a, r(x))), (3, (x, 1))\rangle.$$

Obviously, $mo((6))$ and $mo((7))$ are not the subsequences of $sb_1 + sb_2$, and $po((8))$, $po((9))$ and $po((10))$ are also not. Therefore the five cases are deleted. Thus the final trace is:

$$traces(x := 1; fence; a := x) = \left\{ \begin{array}{l} \langle(1, (x, 1), 1, 1),\ (1, (x, 1), 2, 1), \\ \ (2, fence, 0, 1),\ (3, (a, r(x)), 3, 1)\rangle \end{array} \right\}.$$

From the unique trace above, we can see that the effect of the write to location x must appear firstly, if statement $x := 1$ precedes the instruction fence.

In particular, the simplest composition without any reordering between a and P is formalized as below.

$$traces(a \to P) = {}_{\rm df}\ traces(a)^{\wedge} traces(P),\ \text{where } traces(a) =_{\rm df} \{\langle a\rangle\}.$$

Conditional. Taking the execution of Conditional into our consideration, it will behave the same as P if h is true. Otherwise, it acts like process Q.

$$traces(\text{if } h \text{ then } P \text{ else } Q) = {}_{\rm df}\ traces(P) \lhd h \rhd traces(Q).$$
$$sb(\text{if } h \text{ then } P \text{ else } Q) =_{\rm df} sb(P) \lhd h \rhd sb(Q).$$

The notation $e \lhd b \rhd f$ stands for e if b is true; otherwise f. If there is some variable x in the Boolean condition h, the read function will be applied to achieve the value of x.

Iteration. For "while h do P", we take the understanding as "if h then $(P;$ while h do $P)$ else Π". Based on the analysis of Conditional, $traces(\text{while } h \text{ do } P)$ and $sb(\text{while } h \text{ do } P)$ can be obtained by applying the least fixed point concept [18–20].

$$traces(\text{while } h \text{ do } P) =_{\rm df} \bigcup_{n=0}^{\infty} traces\{F^n(\text{STOP})\},$$
$$sb(\text{while } h \text{ do } P) =_{\rm df} \bigcup_{n=0}^{\infty} sb\{F^n(\text{STOP})\},$$

where, $F(X) =_{\rm df}$ if h then $(P; X)$ else Π,

$$F^0(X) =_{\rm df} X,$$
$$F^{n+1}(X) =_{\rm df} F(F^n(X)) = \underbrace{F(...(F}_{n \text{ times}}(F(X)))...),$$
$$traces(\Pi) =_{\rm df} \{\varepsilon\}, sb(\Pi) =_{\rm df} \{\varepsilon\},$$

and $traces(\text{STOP}) =_{\rm df} \{\}, sb(\text{STOP}) =_{\rm df} \{\}$.
Here, Π and STOP are applied to facilitate giving the definition of the trace semantics of Iteration.

Parallel Construct. Now, we explore the trace semantics of parallel construct. The traces of parallel construct are formed by the merging of snapshots performed by the two components.

Example 6. Consider the parallel program $P||Q$, where $P =_{\rm df} x := 1; a := y$, $Q =_{\rm df} b := x$, x and y are global variables, and a and b are local variables. $P||Q$ is activated with $x = y = 0$.

Although there are many executing cases, here we only take one scenario below into account. Assume program P is scheduled to execute first, and then P commits the write to x to the store buffer. Next, P and Q choose to read the values of variables y and x respectively. Finally, P moves the write from its own write buffer to the shared memory.

Fig.4 illustrates one trace of P (i.e., seq_1) and Q (i.e., seq_2) respectively, i.e., the above scenario. seq is one trace of $P||Q$, which is merged from P and Q.

Let us consider the first element for the above three traces. The first element of seq_1 indicates that the action is contributed by P itself. Thus $eflag$ of the first element in seq_2 is 0, representing that this action is Q's environment's contribution. No matter whether an action is done by program P or Q, it is contributed by the parallel program $P||Q$. Therefore, the parameter $eflag$ of the first element in seq is 1.

What we should pay attention to is that the read functions $r(y)$ and $r(x)$ are both executed at the point that the sequential composition just completes. The reason is that only before the parallel composition, each thread can classify the private information and the shared data. As shown in Fig.4, the values of $r(y)$ and $r(x)$ are both the initial value 0.

The sequence seq_1 of process P and seq_2 of process Q are said to be comparable, if

1) $\pi_i^*(seq_1) = \pi_i^*(seq_2)$, where $i \in \{1, 2, 3\}$: $i = 1$ denotes that the id sequences for the two traces are the same, and $i = 2$ represents that they are built from the same sequence of data states, and $i = 3$ indicates that the sequences of action types are identical;

2) none of their snapshots is made by both the components: $2 \notin \pi_4^*(seq_1) + \pi_4^*(seq_2)$.

Then, their trace merging is defined as below.

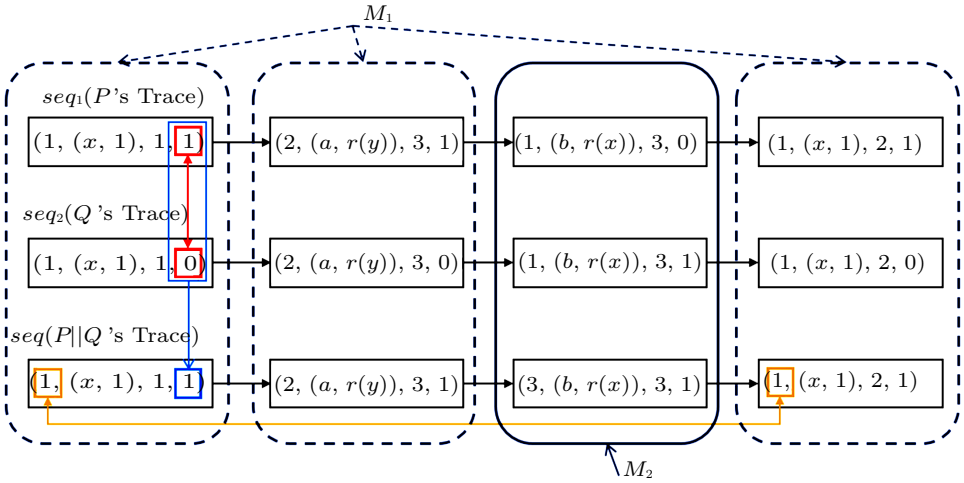

Fig.4. Illustration of merging.

$$
\begin{aligned}
&Merge(seq_1, seq_2, pc)\\
=_{\mathrm{df}}&\left(\begin{array}{l} M_3(seq_1, seq_2, pc)\\ \quad \triangleleft\ \pi_4(hd(seq_1)) + \pi_4(hd(seq_2)) = 0\ \triangleright \\ \left(\begin{array}{l} M_1(seq_1, seq_2, pc)\\ \quad \triangleleft\ \pi_4(hd(seq_1)) = 1\ \triangleright \\ M_2(seq_1, seq_2, pc)\end{array}\right)\end{array}\right),\\
&Merge(\langle\rangle, \langle\rangle, 1) =_{\mathrm{df}} \langle\rangle.
\end{aligned}
$$

Here, the notation $hd(seq_1)$ stands for the first snapshot of the sequence seq_1. The merged sequence seq is produced by the function $Merge$, with the initial value being $\langle\rangle$, and the notation $\langle\rangle$ denotes the empty sequence.

For facilitating explaining the above function, we make such an assumption that the parallel composition is in the form of $(P||Q)||R$ firstly. When merging one trace seq_1 of P and seq_2 of process Q, we take two situations into our consideration. One is that the snapshots $hd(seq_1)$ and $hd(seq_2)$ want to be merged and the formula $\pi_4(hd(seq_1)) + \pi_4(hd(seq_2)) = 0$ holds. It means that the operation corresponding to the snapshot $hd(seq_1)$ is performed by the environment (i.e., process R). Then the parameter $eflag$ of the snapshot merged from $hd(seq_1)$ and $hd(seq_2)$ is still 0. In other words, the merged snapshot, which is the same as $hd(seq_1)$, is added in seq.

$$
\begin{aligned}
&M_3(seq_1, seq_2, pc)\\
=_{\mathrm{df}}&\boxed{hd(seq_1)}\ ^\frown Merge(tl(seq_1), tl(seq_2), pc).
\end{aligned}
$$

The other situation is that the action is carried out by process $P||Q$, i.e., either by P or by Q. We introduce function M_1 to complete the procedure of merging the snapshots in seq_1 which are contributed by process P with the corresponding snapshots in seq_2. M_2 is defined similarly with the contribution produced by program Q. Thus we ignore the detailed explanation of function M_2 later. As shown in Fig.4, function M_1 acts on the transformations in the dotted boxes, while M_2 plays a role in the solid box.

There are two reasons for introducing the program counter pc. One is facilitating generating new id for the parallel process $P||Q$, depicted by seq in Fig.4. The other is to guarantee that the two snapshots split by a global assignment still have the same sequence number (i.e., id), which is described by the yellow line in Fig.4. The initial value of pc is 1, and the function $Merge(s, t, 1)$ is applied in constructing the trace semantics of parallel composition.

Now, we introduce the function M_1, whose definition is similar to that of $inleave_l$ in Subsection 2.3. For M_1, what we need to do is judging whether the parameter $oflag$ in the first snapshot $hd(seq_1)$ is 1 or not. If so, because this snapshot is related to committing a memory write, the other snapshot in the rest sequence $tl(seq_1)$, whose id and $eflag$ are $\pi_1(hd(seq_1))$ and 1 respectively, should update its id to the current program counter pc firstly. Its purpose is to keep consistent between the parameter id of committing a memory write and that of propagating the same write. This operation is modeled by the latter framed area in (11) below. Secondly, before attaching $hd(seq_1)$ to seq, its id should be modified to pc, and other parameters remain unchanged, which is described as the former framed area

in (11). Thirdly, pc adds 1 subsequently.

If the parameter $oflag$ of $hd(seq_1)$ is 2, we move $hd(seq_1)$ to the tail of sequence seq merely, since such snapshots have been updated in the above case. And it is formalized in (12). For other cases denoted by (13), the difference from the first case is that the cases discussed here do not have to make changes to the snapshots in $tl(seq_1)$.

$$
\begin{aligned}
&M_1(seq_1, seq_2, pc)\\
=_{\mathrm{df}}\ &\left(\begin{array}{ll}
\boxed{hd(seq_1)[pc/\pi_1(hd(seq_1))]} \ ^\frown Merge(\boxed{tl(seq_1)[(pc,\_,2,1)/(\pi_1(hd(seq_1)),\_,2,1)]}, tl(seq_2), pc+1) & (11)\\
\quad \triangleleft\ \pi_3(hd(seq_1)) = 1 \ \triangleright & \\
\left(\begin{array}{ll}
\boxed{hd(seq_1)} \ ^\frown Merge(tl(seq_1), tl(seq_2), pc) & (12)\\
\quad \triangleleft\ \pi_3(hd(s_1)) = 2 \ \triangleright & \\
\boxed{hd(seq_1)[pc/\pi_1(hd(seq_1))]} \ ^\frown Merge(tl(seq_1), tl(seq_2), pc+1) & (13)
\end{array}\right) &
\end{array}\right).
\end{aligned}
$$

An example is given below to provide an intuitive illustration of function $Merge$.

Example 7. Consider the three sequences in Fig.5. seq is $\langle\rangle$, and pc is 1 initially.

1) In the beginning, we would like to merge the first snapshot $(1,(x,1),1,1)$ by process P and $(1,(x,1),1,0)$ by another process Q. The merge function M_1 is triggered. We find that $\pi_3((1,(x,1),1,1)) = 1$ holds in trace seq_1, and the snapshots $(1,(x,1),2,1)$ in the remaining trace $\langle(2,(a,r(y)),3,1),(1,(b,r(x)),3,0),(1,(x,1),2,1)\rangle$ and $(1,(x,1),1,1)$ are contributed by the same thread, and parameter id of $(1,(x,1),2,1)$ is equal to that of $(1,(x,1),1,1)$. Then we update the element id of $(1,(x,1),2,1)$ to the current program counter pc (i.e., the value 1 presented in Fig.5). Afterwards, for snapshot $(1,(x,1),1,1)$, if its id is also changed to current pc, it will be added to the tail of seq. Consequently, the newly generated seq is $\langle(1,(x,1),1,1)\rangle$. Finally pc is updated to 2.

2) For snapshots $(2,(a,\ r(y)),\ 3,\ 1)$ and $(2,(a,r(y)),3,0)$, we still carry out the merge function M_1, and the snapshot $(2,(a,r(y)),3,1)$ is attached to seq. Now pc increases by 1, and is equal to 3.

3) The function M_2 is performed when meeting snapshot $(1,(b,r(x)),3,0)$ by process P and $(1,(b,r(x)),3,1)$ contributed by Q. seq is updated to $\langle(1,(x,1),1,1),(2,(a,r(y)),3,1),\ (3,(b,r(x)),3,1)\rangle$, and then pc is changed to 4.

4) The updated snapshot $(1,(x,1),2,1)$ contributed by P is moved to the tail of seq directly with the application of M_1. The program counter pc does not change through this step.

Next, we give the definition of the trace semantics of parallel composition. To facilitate merging, we concatenate sequence s contributed by P's environment with trace tr_1 in process P, and it is the same for Q, where $\pi_4^*(s) \in 0^*$ and $\pi_4^*(t) \in 0^*$.

$$
\begin{aligned}
&traces(P||Q)\\
=_{\mathrm{df}}\ &\left\{\begin{array}{l}
tr | tr_1 \in traces(P) \wedge tr_2 \in traces(Q) \wedge\\
\left(\begin{array}{l}
tr = Merge(tr_1^\frown s, tr_2, 1) \vee\\
tr = Merge(tr_1, tr_2^\frown t, 1)
\end{array}\right)
\end{array}\right\}.
\end{aligned}
$$

Now we introduce the function $generate$ to generate a sequence of sb from a trace of a program.

$$
\begin{aligned}
&generate(tr, pc)\\
=_{\mathrm{df}}\ &\left(\begin{array}{l}
generate(tl(tr), pc)\\
\quad \triangleleft\ \pi_4(hd(tr)) = 0 \vee \pi_3(hd(tr)) = 2 \ \triangleright\\
\langle(pc, \pi_2(hd(tr)))\rangle^\frown generate(tl(tr), pc+1)
\end{array}\right),\\
&generate(\langle\rangle, 1) =_{\mathrm{df}} \langle\rangle.
\end{aligned}
$$

Because $sb(P)$ is used to record the information of the statements in P, we need to filter out the elements

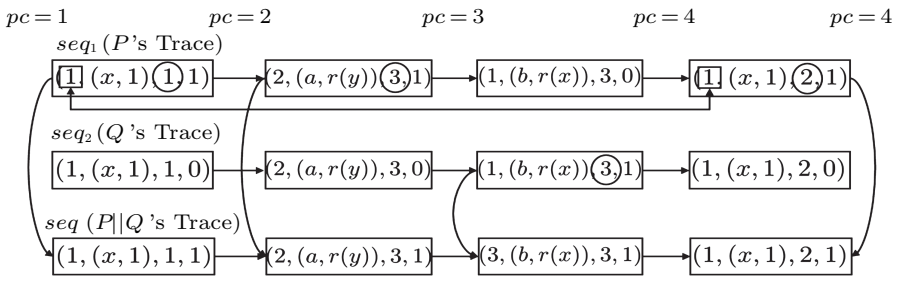

Fig.5. Illustration of function $Merge$.

with $eflag$ being 0 or $oflag$ being 2. Then the definition of $sb(P||Q)$ is given as below.

$$sb(P||Q)$$
$$=_{\mathrm{df}}\{s \mid tr \in traces(P||Q) \wedge s = generate(tr, 1)\}.$$

3 Algebraic Properties

Our work towards the formalization of the TSO memory model aims to deduce its interesting properties, which are usually expressed using algebraic laws in the form of equations. In this section, we explore algebraic laws for the TSO model, including a set of sequential and parallel expansion laws. Our approach is that every program can be presented in the head normal form of the guarded choice. Based on this, the linearizability of TSO can be supported.

3.1 Guarded Choice

A configuration in a program is supposed to be expressed in the form of $h\&(act, tid, idx)$.
Here:

- h is a Boolean condition;
- the element act can be a general action, such as writing to the write buffer or local assignments, a memory action or a special action fence;
- tid records the ID of the thread which performs the action;
- we use parameter idx to denote the location of an action. It can also distinguish whether the action is propagating to the shared memory or not. If the judgment is true, idx is equal to 2. Otherwise, it is 1. Example 8 below helps to illustrate the intuitive understanding of idx.

Example 8. Consider the three different statements in the following.

Here x is a global variable. The actions $\langle x = 1\rangle$ and $x = 1$ are split from $x := 1$. $\langle x = 1\rangle$ is committing the write to x to the write buffer and its index is $\langle 1\rangle$, while $x = 1$ is to move the same write to the main memory and then the index of this action is $\langle 2\rangle$, as shown in Fig.6. Action $a = 1$ is corresponding to statement $a := 1$, and a is a local variable, and then its index is $\langle 1\rangle$. The analysis of a fence instruction is similar. For the program which has more than one statement, the generation of idx of each action in the program is explained by the following law (seq–2) in Subsection 3.3.

The thread ID tid is to indicate which thread an action is due to for a parallel program. We assign λ to it for sequential process, whereas we use example 9 below to illustrate the concept of tid (in other words, the locality of a thread) in the parallel environment.

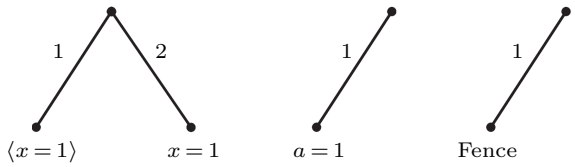

Fig.6. Illustration of parameter idx.

Example 9. Let $P = U||V$, and $U = A||B$.

The thread ID of process U is $\langle 1\rangle$, and that of V is $\langle 2\rangle$. Then processes A and B can be marked by $\langle 1\rangle^\frown\langle 1\rangle$ and $\langle 1\rangle^\frown\langle 2\rangle$ respectively. In the structure of P described in Fig.7, processes A, B and V are leaf processes, whereas process U is not.

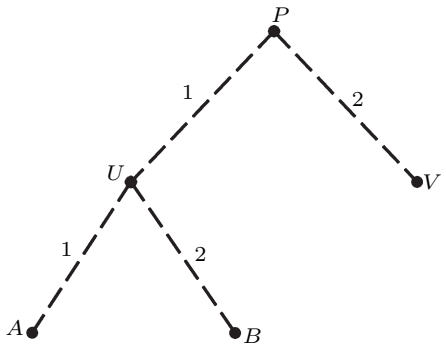

Fig.7. Structure of process P.

Note that we use $\langle 1, 1\rangle$ instead of $\langle 1\rangle^\frown\langle 1\rangle$ for simplicity. Further for any tid, we have $tid^\frown\lambda = tid$.

Now we introduce the concept of guarded choice for the TSO memory model expressed in the form of $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \hookrightarrow P_i'[q_i]\}$. It is able to model the execution of a program with a variety of reorderings. Here, $h_i\&(act_i, tid_i, idx_i) \hookrightarrow P_i'[q_i]$ is a guarded component, and if the Boolean condition h_i is satisfied, the subsequent program is $(act_i, tid_i, idx_i) \hookrightarrow P_i'[q_i]$. Here:

1) If act_i is the operation committing a memory write to the store buffer, q_i is in the form of $h_i'\&(act_i', tid_i, idx_i')$, where act_i' is the corresponding operation propagating the mentioned write to the main memory. For $h_i'\&(act_i', tid_i, idx_i')$, h_i' is true as well.

2) If act_i is a local assignment or fence instruction, then q_i is ε.

3) When meeting the condition judgment in "if" or "while" structure, h_i is used to describe the conditional guard and act_i is ε.

4) q_i may reorder with some configurations in P_i', and the precise definition of their relation is given in the following law (seq–3) in Subsection 3.3.

Every program can be represented in the form of the guarded choice. And it can have three types under the TSO model.

1) $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \looparrowright P_i'[(act_i', tid_i, idx_i')]\}$. The first type of guarded choice is composed of a set of buffer action components. If a Boolean condition is satisfied, the corresponding buffer action can be selected to execute.

2) $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \looparrowright P_i'\}$. The second type of guarded choice consists of a set of local assignment or fence or branching condition components. Any assignment or fence instruction or condition judgment can be activated when the corresponding Boolean condition is satisfied.

3) $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \looparrowright P_i'[(act_i', tid_i, idx_i')]\} [\!] [\!]_{j\in J}\{h_j\&(act_j, tid_j, idx_j) \looparrowright Q_j'\}$. The third type is formed by combining the first and the second types of guarded choice.

3.2 Head Normal Form

In this subsection, we assign every program P a normal form which is named as head normal form $HF(P)$.

1) For a local assignment, the remaining part after the first step expansion is empty. We use the notation E to denote the empty process.

$$HF(a := e) =_{\text{df}} [\!]\{\text{true}\&(a = e, \lambda, \langle 1\rangle) \looparrowright E\}.$$

2) The head normal form of fence is similar to that of local assignment.

$$HF(fence) =_{\text{df}} [\!]\{\text{true}\&(fence, \lambda, \langle 1\rangle) \looparrowright E\}.$$

3) Below is the analysis of global assignment. What is different from local assignment is that global assignment is not atomic. This assignment can be divided into two parts, and the first part is committing a memory write to the store buffer. The second part propagating the same write to the memory is left in the square brackets.

$$\begin{aligned} &HF(x := e)\\ =_{\text{df}} &[\!]\{\text{true}\&(\langle x = e\rangle, \lambda, \langle 1\rangle) \looparrowright E[(x = e, \lambda, \langle 2\rangle)]\}. \end{aligned}$$

4) For Conditional, the configurations $h\&(\varepsilon, \lambda, \text{null})$ and $\neg h\&(\varepsilon, \lambda, \text{null})$ are applied to generate the head normal form. They indicate that the evaluation h does not make any effect in the private registers or write buffers, or the shared memory. Moreover, we have to pay attention to the location in which the evaluation can carry out through the thread ID, and the thread ID may change according to the placement in different concurrent environments. Since the parameter idx in the Boolean condition is null, and the reorder between a Boolean condition and the instructions in a branch is not allowed under the TSO memory model, we use the symbol "$\rightarrow$" instead of "$\looparrowright$" here. We will explain these two symbols in detail in Subsection 3.3.

$$\begin{aligned} &HF(\text{if } h \text{ then } P \text{ else } Q)\\ =_{\text{df}} &[\!]\{h\&(\varepsilon, \lambda, \text{null}) \looparrowright P, \neg h\&(\varepsilon, \lambda, \text{null}) \looparrowright Q\}\\ = &[\!]\{h\&(\varepsilon, \lambda, \text{null}) \rightarrow P, \neg h\&(\varepsilon, \lambda, \text{null}) \rightarrow Q\}. \end{aligned}$$

5) Considering Iteration, its description is similar to that of Conditional, i.e., HF(4).

$$\begin{aligned} &HF(\text{while } h \text{ do } P)\\ =_{\text{df}} &[\!]\left\{\begin{array}{l} h\&(\varepsilon, \lambda, \text{null}) \looparrowright (P; \text{while } h \text{ do } P),\\ \neg h\&(\varepsilon, \lambda, \text{null}) \looparrowright E \end{array}\right\}\\ = &[\!]\left\{\begin{array}{l} h\&(\varepsilon, \lambda, \text{null}) \rightarrow (P; \text{while } h \text{ do } P),\\ \neg h\&(\varepsilon, \lambda, \text{null}) \rightarrow E \end{array}\right\}. \end{aligned}$$

Afterwards, the definition of the head normal form for sequential and parallel composition is obtained with the application of the corresponding expansion laws (please refer to the algebraic laws (seq) and (par) in Subsection 3.3).

3.3 Algebraic Laws

This subsection focuses on the algebraic laws including a set of sequential and parallel expansion laws.

Firstly, we take the sequential expansion laws into account. Law (guar–1) indicates that sequential composition distributes leftward over guarded choice. Because the subsequent program Q is not constrained by parameter i, it does not make a difference whether program Q is placed in the guarded choice or not.

(guar–1) $\quad [\!]_{i\in I}\{P_i\}; Q = [\!]_{i\in I}\{P_i; Q\}.$

Definition 1 is presented for facilitating illustrating the important feature "reordering" under the TSO memory model. When the subsequent program Q comes to make sequential composition, the memory operation q (if exists) is supposed to act on the whole program $P'; Q$. Because the definition is given recursively, the store-store order will not be broken with the application of some constraints given in the following laws.

Definition 1.

$$P'[q]; Q =_{\text{df}} (P'; Q)[q].$$

Then, the investigation of the remaining sequential laws can be divided into three steps: 1) extracting the

first configuration; 2) modifying the indices of the remainder (excluding the memory operation related to the extracted one); 3) completing the generation of all possible configuration sequences through the constrained interleaving between the memory actions and others. Those three procedures are performed recursively, and they are formalized by the laws (seq–1), (seq–2) and (seq–3) shown below.

(seq–1)

Let $P = \|_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \looparrowright P_i'[q_i]\}$.

Then $P;Q = \|_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \looparrowright (P_i';Q)[q_i]\}$.

Law (seq–1) can be regarded to be the extension of Definition 1 and Law (guar–1). For the program $P;Q$, after a configuration is extracted, configuration q_i of the corresponding memory action working for P_i' now can work for the sequential composition $P_i';Q$.

With regard to idx, as discussed in Subsection 3.1, the configuration of every single statement has the value of 1 or 2. When making sequential composition, once the first configuration is chosen, the corresponding memory action (if exists) remains unchanged and the indices of the following configurations are supposed to add a $\langle 1\rangle$ prefix. It is formalized by the law (seq–2).

(seq–2)
$$\begin{aligned} & h\&(act, tid, idx) \looparrowright P'[q] \\ = \ & h\&(act, tid, idx) \to (\langle 1\rangle^\frown P')[q], \\ & \text{where } (\langle 1\rangle^\frown E) = E,\ E[q] = q. \end{aligned}$$

The function $\langle 1\rangle^\frown P$ makes such an effect that the indices of all the configurations in P add the extra prefix $\langle 1\rangle$, which is formalized in the following.

$$\begin{aligned} \langle 1\rangle^\frown P =_{\text{df}} \ & \forall h\&(act, tid, idx) \in P \bullet \\ & P[h\&(act, tid, \langle 1\rangle^\frown idx)/h\&(act, tid, idx)], \end{aligned}$$

where $\langle 1\rangle^\frown \text{null} = \text{null}$, and "null" stands for idx for the configuration of a branching condition. "$\bullet$" means "such that" [9].

When presenting the sequential laws, there are mainly two operators involved here. One is the operator "$\looparrowright$", and it is applied to connect configurations with original indices. If the indices are updated, we use the operator "$\to$" to connect.

Example 10. Let $P =_{\text{df}} x := 1$ and $Q =_{\text{df}} y := 1$, where x and y are both global variables. With the application of the sequential laws (seq–1) and (seq–2), the normal form of $P;Q$ (i.e., $x := 1; y := 1$) is formalized as below.

$$\begin{aligned} HF(P) &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \looparrowright E[(x = 1, \lambda, \langle 2\rangle)]. \\ HF(P;Q) &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \looparrowright (E;Q)[(x = 1, \lambda, \langle 2\rangle)] \\ &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \looparrowright ((\langle y = 1\rangle, \lambda, \langle 1\rangle) \looparrowright E[(y = 1, \lambda, \langle 2\rangle)])[(x = 1, \lambda, \langle 2\rangle)] \\ &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to (\langle 1\rangle^\frown((\langle y = 1\rangle, \lambda, \langle 1\rangle) \looparrowright E[(y = 1, \lambda, \langle 2\rangle)]))[(x = 1, \lambda, \langle 2\rangle)] \\ &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to ((\langle y = 1\rangle, \lambda, \boxed{\langle 1, 1\rangle}) \looparrowright E[(y = 1, \lambda, \boxed{\langle 1, 2\rangle})])[(x = 1, \lambda, \langle 2\rangle)] \\ &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to ((\langle y = 1\rangle, \lambda, \boxed{\langle 1, 1\rangle}) \to (y = 1, \lambda, \boxed{\langle 1, 2\rangle}))[(x = 1, \lambda, \langle 2\rangle)]. \end{aligned}$$

As we can see in Fig.8, after sequential composition, the indices of the actions $\langle y = 1\rangle$ and $y = 1$ are transfered to $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$ respectively.

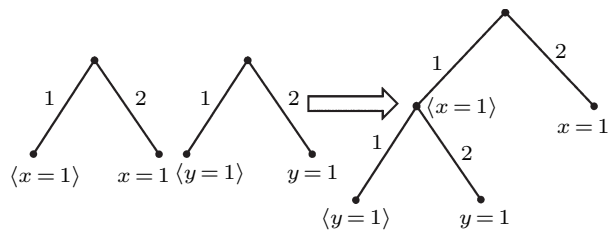

Fig.8. Combination of two statements.

Applying the law (seq–3), all the possible configuration sequences including a variety of reorderings can be achieved. We give the illustration of the constraints between two actions (one or more actions are related to the main memory). 1) The memory actions are supposed to be performed sequentially, since the write buffer conforms to the FIFO principle. 2) If a general action and a memory action have the same origin, they must carry out in order. 3) In addition, when meeting a fence instruction, the memory action belonging to the statement before fence should be executed in advance. Constraint 2 has been realized by the law (seq–2), and condition *cond* in the following gives the formalization of constraints 1 and 3.

$$\begin{aligned} \text{(seq–3)}\ \ (p \to P')[q] \ = \ & (q \to (p \to P')) \\ & \|(p \to P'[q]) \ \text{ if } \mathit{cond}. \end{aligned}$$

Here *cond* is defined as below.

$$cond =_{\mathrm{df}} \left(\begin{array}{l} \left(\begin{array}{l} \pi_3(p) \text{ does not end in 2} \wedge \\ \pi_1(p) \text{ is not a } fence \text{ instruction} \end{array} \right) \\ \vee \left(\begin{array}{l} (\pi_3(p) \text{ ends in 2 }) \wedge \\ ((len(\pi_3(p))) < (len(\pi_3(q)))) \end{array} \right) \end{array} \right),$$

where p and q are both configurations, and the notation $len(idx)$ records the length of the parameter idx.

(guar–2) $\quad a \to [\!]_{i\in I}\{b_i \to Q_i\} = [\!]_{i\in I}\{a \to b_i \to Q_i\}.$

Law (guar–2) represents that the prefix operator distributes leftward over guarded choice. Due to the fact that event a is not bound by parameter i, it can be placed inside the guarded choice. Also, a can be put outside the guarded choice. The prefix operator has been introduced in Subsection 2.3.

Assuming P' satisfies the property linearizability, obviously, $(p \to P')[q]$ also conforms to linearizability according to the law (guar–2).

Example 10: *Continuation.* For the subsequence $((\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle))[(x = 1, \lambda, \langle 2\rangle)]$, with the law (seq–3), there are only two cases. One is that the configuration $(\langle y = 1\rangle, \lambda, \langle 1, 1\rangle)$ is first scheduled. The other is that $(x = 1, \lambda, \langle 2\rangle)$ is selected firstly. The order between the configurations $(x = 1, \lambda, \langle 2\rangle)$ and $(y = 1, \lambda, \langle 1, 2\rangle)$ is fixed according to constraint 1. Hence all the configuration sequences of the sequential program $P; Q$ are exhibited as below.

$$\begin{aligned} HF(P;Q) &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to ((\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle))[(x = 1, \lambda, \langle 2\rangle)] \\ &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to \left(\begin{array}{l} ((\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle))[(x = 1, \lambda, \langle 2\rangle)] \\ [\!] \\ (x = 1, \lambda, \langle 2\rangle) \to (\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle) \end{array} \right) \\ &= (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to \left(\begin{array}{l} (\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (x = 1, \lambda, \langle 2\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle) \\ [\!] \\ (x = 1, \lambda, \langle 2\rangle) \to (\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle) \end{array} \right) \\ &= \left(\begin{array}{l} (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to (\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (x = 1, \lambda, \langle 2\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle) \\ [\!] \\ (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to (x = 1, \lambda, \langle 2\rangle) \to (\langle y = 1\rangle, \lambda, \langle 1, 1\rangle) \to (y = 1, \lambda, \langle 1, 2\rangle) \end{array} \right). \end{aligned}$$

Now, we continue to consider the parallel expansion law. Our parallel model is based on the configuration sequences produced by the above sequential expansion laws, and it can be explained as an interleaving model.

(par–1)

Let $P = [\!]_{i\in I}\{h_i \& (act_i, tid_i, idx_i) \to P'_i\}$,
$Q = [\!]_{j\in J}\{h_j \& (act_j, tid_j, idx_j) \to Q'_j\}$.

Then

$$\begin{aligned} P||Q = {} & [\!]_{i\in I}\{h_i \& (act_i, \langle 1\rangle^\wedge tid_i, idx_i) \to (P'_i||Q)\} \\ & [\!][\!]_{j\in J}\{h_j \& (act_j, \langle 2\rangle^\wedge tid_j, idx_j) \to (P||Q'_j)\}. \end{aligned}$$

If the configuration in the left branch (i.e., process P) is chosen, $\langle 1\rangle$ is attached to the head of the corresponding tid_i. When selecting the configuration in the right (i.e., process P), the prefix $\langle 2\rangle$ is added to the corresponding tid_j.

Example 11. Consider the parallel program $P||Q$, where $P =_{\mathrm{df}} x := 1; a := y$, $Q =_{\mathrm{df}} y := 1; b := x$, a and b are local, and x and y are global variables. Now we calculate the head normal form for $P||Q$ through the mentioned algebraic laws. For simplicity, we only exhibit one configuration sequence of $P||Q$ in Fig.9.

$$\begin{aligned} & HF(P||Q) \\ = {} & \left(\begin{array}{l} (\langle x = 1\rangle, \lambda, \langle 1\rangle) \leftarrowtail a := y[(x = 1, \lambda, \langle 2\rangle)] \\ || \\ (\langle y = 1\rangle, \lambda, \langle 1\rangle) \leftarrowtail b := x[(y = 1, \lambda, \langle 2\rangle)] \end{array} \right) \\ = {} & \left(\begin{array}{l} (\langle x = 1\rangle, \lambda, \langle 1\rangle) \leftarrowtail ((a = y, \lambda, \langle 1\rangle) \leftarrowtail E)[(x = 1, \lambda, \langle 2\rangle)] \\ || \\ (\langle y = 1\rangle, \lambda, \langle 1\rangle) \leftarrowtail ((b = x, \lambda, \langle 1\rangle) \leftarrowtail E)[(y = 1, \lambda, \langle 2\rangle)] \end{array} \right) \\ = {} & \left(\begin{array}{l} (\langle x = 1\rangle, \lambda, \langle 1\rangle) \to (\langle 1\rangle^\wedge((a = y, \lambda, \langle 1\rangle) \leftarrowtail E))[(x = 1, \lambda, \langle 2\rangle)] \\ || \\ (\langle y = 1\rangle, \lambda, \langle 1\rangle) \to (\langle 1\rangle^\wedge((b = x, \lambda, \langle 1\rangle) \leftarrowtail E))[(y = 1, \lambda, \langle 2\rangle)] \end{array} \right) \end{aligned}$$

$$= \begin{pmatrix} \Big((\langle x=1\rangle, \lambda, \boxed{\langle 1\rangle}) \to (a=y, \lambda, \boxed{\langle 1,1\rangle})[(x=1, \lambda, \boxed{\langle 2\rangle})] \Big) \\ \| \\ \Big((\langle y=1\rangle, \lambda, \boxed{\langle 1\rangle}) \to (b=x, \lambda, \boxed{\langle 1,1\rangle})[(y=1, \lambda, \boxed{\langle 2\rangle})] \Big) \end{pmatrix}$$

$$= \begin{pmatrix} (\langle x=1\rangle, \lambda, \langle 1\rangle) \to (a=y, \lambda, \langle 1,1\rangle) \to (x=1, \lambda, \langle 2\rangle) \\ [\!] \\ (\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1, \lambda, \langle 2\rangle) \to (a=y, \lambda, \langle 1,1\rangle) \end{pmatrix} \Big\| \begin{pmatrix} (\langle y=1\rangle, \lambda, \langle 1\rangle) \to (b=x, \lambda, \langle 1,1\rangle) \to (y=1, \lambda, \langle 2\rangle) \\ [\!] \\ (\langle y=1\rangle, \lambda, \langle 1\rangle) \to (y=1, \lambda, \langle 2\rangle) \to (b=x, \lambda, \langle 1,1\rangle) \end{pmatrix}.$$

To sum up, any finite program P can be expressed as below.

$$P = [\!]_{i\in I}\{c_1 \to c_2 \to ... \to c_n\},$$

where $c_j = (act_{ij}, tid_{ij}, idx_{ij})$ or $h_{ij}\&(\varepsilon, tid_{ij}, \text{null})$.

3.4 Deriving Trace Semantics from Algebraic Semantics

Now we consider how to derive trace semantics from algebraic semantics under the TSO memory model. Our approach only focuses on finite programs. Let $seq = c_1 \to c_2 \to ... \to c_n$ represent a configuration sequence of the given program P.

If seq is the configuration sequence of the single process P, its derived trace semantics can be achieved with the application of $Atraces(seq)$ whose definition is put in the following. The single process P means that it does not include any parallel composition. The parameter tid of each configuration in seq can be λ_0 contributed by P itself, or λ_i produced by the environment of P where $i > 0$.

With regard to one sequence seq of the parallel program P, for facilitating deriving the corresponding trace semantics from seq in the algebraic model, we define the function $M^+(seq)$ as below.

$M^+(seq)$ separates the derivation into three steps.

1) The configuration sequence of each single process is extracted. It is worth noting that in the parallel construct $(A||B)||V$, for the extracted sequence seq_A of the single process A, the thread ID of the configuration which is contributed by the outer environment process V in seq_A is λ_1, and that of the inner environment B is λ_2. Generally speaking, the environment processes with different levels have different values of tid. This step is formalized as the functions sep_1 and sep_2.

Example 12. Let us consider such a parallel composition $P = U||V$ and $U = A||B$, where $A =_{\text{df}} a := 1$, $B =_{\text{df}} b := 1$ and $V =_{\text{df}} c := 1$, which is illustrated in Fig.10. Variables a, b and c are local variables.

From the perspective of the single process A, action $a = 1$ corresponding to statement $a := 1$ running in process A is performed by itself. The action $b = 1$ is contributed by the inner environment process (i.e., process B), while the operation $c = 1$ is produced by the outer environment process (i.e., V).

Based on the above analysis, for a given sequence $seq = (b = 1, \langle 1,2\rangle, \langle 1\rangle) \to (a = 1, \langle 1,1\rangle, \langle 1\rangle) \to (c = 1, \langle 2\rangle, \langle 1\rangle)$ of the parallel construct P, the corresponding sequence for the single process A is $seq_A = (b = 1, \boxed{\lambda_2}, \langle 1\rangle) \to (a = 1, \boxed{\lambda_0}, \langle 1\rangle) \to (c = 1, \boxed{\lambda_1}, \langle 1\rangle)$. The procedure extracting seq_A from seq will be introduced later.

2) Then we achieve the derived trace semantics of each single process with the application of the definition of $Atraces$. It is formalized by (14).

3) The $Merge$ function introduced in Subsection 2.3 works on these derived traces, and produces the trace semantics of the parallel composition.

Here parameter c is applied to distinguish the environment with different levels, and it is 1 initially.

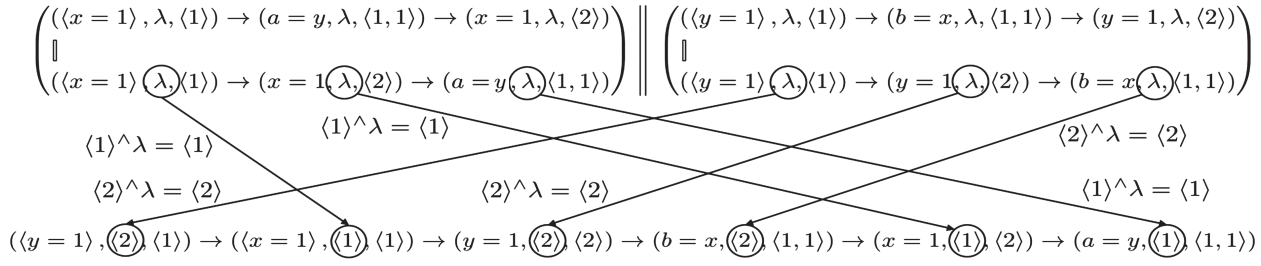

Fig.9. Application of law (par–1).

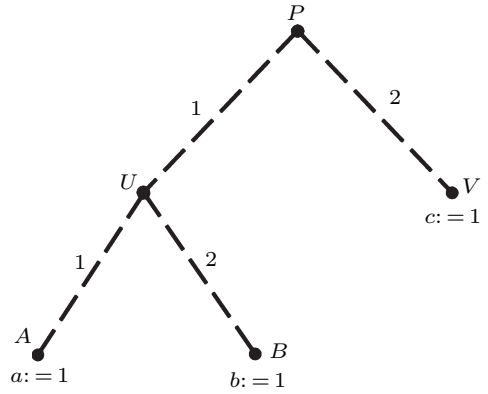

Fig.10. Introduction to process identity in extracted sequence.

$$M^+(seq,c) =_{\mathrm{df}} \left(\begin{array}{l} Atraces(seq) \qquad (14) \\ \quad \triangleleft\ \pi_2^*(seq) \in \lambda^*\ \triangleright \\ \left\{ \begin{array}{r} Merge(u,v) | u \in M^+(sep_1(seq,c), c+1) \wedge \\ v \in M^+(sep_2(seq,c), c+1) \end{array} \right\} \end{array} \right),$$

where $Atraces(seq) =_{\mathrm{df}} DAtraces(seq, \mathbb{1}, \Phi)$.

Now, we explain functions sep_1 and sep_2. The function sep_1 is applied to extract the sequence of the process in the left branch of operator "||". If the prefix of tid of one configuration is $\langle 1 \rangle$ (denoted by $\leqslant$), this configuration belongs to the left process, and then its tid is modified by removing the prefix, which is denoted by the framed area in (15). Note that $\langle 1 \rangle \backslash \langle 1 \rangle = \lambda_0$.

$$sep_1(seq,c) =_{\mathrm{df}} \left(\begin{array}{l} \boxed{(hd(seq) \backslash \langle 1 \rangle)} \ ^\frown sep_1(tl(seq)) \qquad (15) \\ \quad \triangleleft\ 1 \leqslant \pi_2(hd(seq))\ \triangleright \\ \left(\begin{array}{l} \boxed{(hd(seq)[\lambda_c/tid])} \ ^\frown sep_1(tl(seq)) \quad (16) \\ \quad \triangleleft\ 2 \leqslant \pi_2(hd(seq))\ \triangleright \\ \boxed{hd(seq)} \ ^\frown sep_1(tl(seq)) \end{array} \right) \end{array} \right),$$

$$sep_1(\varepsilon, c) =_{\mathrm{df}} \varepsilon.$$

Once the thread ID of a configuration has such a prefix whose value is 2, the configuration is contributed by the right (i.e., the environment of the left process), we use notation λ_c to take place of tid, shown in (16). Otherwise, for other cases, the thread ID tid remains unchanged.

Afterwards, removing the prefix $\langle 1 \rangle$ is formalized in the following.

$$h\&(act, \langle 1 \rangle ^\frown tid, idx) \backslash \langle 1 \rangle =_{\mathrm{df}} h\&(act, tid, idx).$$

The function sep_2 is performed from the perspective of the right side of the operator "||". Because the definition is similar to that of sep_1, we do not give its detailed explanation here.

$$sep_2(seq,c) =_{\mathrm{df}} \left(\begin{array}{l} \boxed{(hd(seq) \backslash \langle 2 \rangle)} \ ^\frown sep_2(tl(seq)) \\ \quad \triangleleft\ 2 \leqslant \pi_2(hd(seq))\ \triangleright \\ \left(\begin{array}{l} \boxed{(hd(seq)[\lambda_c/tid])} \ ^\frown sep_2(tl(seq)) \\ \quad \triangleleft\ 1 \leqslant \pi_2(hd(seq))\ \triangleright \\ \boxed{hd(seq)} \ ^\frown sep_2(tl(seq)) \end{array} \right) \end{array} \right),$$

$$sep_2(\varepsilon, c) =_{\mathrm{df}} \varepsilon.$$

Example 13. Let $P = U||V$, and $U = A||B$, where $A =_{\mathrm{df}} x := 1$, $B =_{\mathrm{df}} a := x$ and $V =_{\mathrm{df}} b := y$. Variables x and y are global, while a and b are local variables. Assume $seq = (\langle x = 1 \rangle, \langle 1,1 \rangle, \langle 1 \rangle) \to (a = x, \langle 1,2 \rangle, \langle 1 \rangle) \to (x = 1, \langle 1,1 \rangle, \langle 2 \rangle) \to (b = y, \langle 2 \rangle, \langle 1 \rangle)$ is a configuration sequence of process P.

$$\begin{aligned}
& sep_1(seq, \boxed{1}) = (\langle x = 1 \rangle, \langle 1 \rangle, \langle 1 \rangle) \to (a = x, \langle 2 \rangle, \langle 1 \rangle) \to (x = 1, \langle 1 \rangle, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle), \\
& sep_2(seq, \boxed{1}) = (\langle x = 1 \rangle, \lambda_1, \langle 1 \rangle) \to (a = x, \lambda_1, \langle 1 \rangle) \to (x = 1, \lambda_1, \langle 2 \rangle) \to (b = y, \lambda_0, \langle 1 \rangle), \\
& sep_1((((\langle x = 1 \rangle, \langle 1 \rangle, \langle 1 \rangle) \to (a = x, \langle 2 \rangle, \langle 1 \rangle) \to (x = 1, \langle 1 \rangle, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle)), \boxed{2}) \\
= {} & (\langle x = 1 \rangle, \lambda_0, \langle 1 \rangle) \to (a = x, \lambda_2, \langle 1 \rangle) \to (x = 1, \lambda_0, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle), \\
& sep_2((((\langle x = 1 \rangle, \langle 1 \rangle, \langle 1 \rangle) \to (a = x, \langle 2 \rangle, \langle 1 \rangle) \to (x = 1, \langle 1 \rangle, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle)), \boxed{2}) \\
= {} & (\langle x = 1 \rangle, \lambda_2, \langle 1 \rangle) \to (a = x, \lambda_0, \langle 1 \rangle) \to (x = 1, \lambda_2, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle), \\
& M^+(seq, \boxed{1}) \\
= {} & Merge(M^+((((\langle x = 1 \rangle, \langle 1 \rangle, \langle 1 \rangle) \to (a = x, \langle 2 \rangle, \langle 1 \rangle) \to (x = 1, \langle 1 \rangle, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle)), \boxed{2}), \\
& \qquad M^+((((\langle x = 1 \rangle, \lambda_1, \langle 1 \rangle) \to (a = x, \lambda_1, \langle 1 \rangle) \to (x = 1, \lambda_1, \langle 2 \rangle) \to (b = y, \lambda_0, \langle 1 \rangle)), \boxed{2})) \\
= {} & Merge(Merge(M^+((((\langle x = 1 \rangle, \lambda_0, \langle 1 \rangle) \to (a = x, \lambda_2, \langle 1 \rangle) \to (x = 1, \lambda_0, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle)), \boxed{3}), \\
& \qquad M^+((((\langle x = 1 \rangle, \lambda_2, \langle 1 \rangle) \to (a = x, \lambda_0, \langle 1 \rangle) \to (x = 1, \lambda_2, \langle 2 \rangle) \to (b = y, \lambda_1, \langle 1 \rangle))), \boxed{3}), \\
& \qquad M^+((((\langle x = 1 \rangle, \lambda_1, \langle 1 \rangle) \to (a = x, \lambda_1, \langle 1 \rangle) \to (x = 1, \lambda_1, \langle 2 \rangle) \to (b = y, \lambda_0, \langle 1 \rangle)), \boxed{2})).
\end{aligned}$$

The procedure of extracting the sequences of U and V is exhibited in Fig.11(a), and we show how to extract the sequences of A and B in Fig.11(b). And the formalization of the procedures is given above. The numbers framed in the formulas reflect the changes of program counter c.

After discussing how to extract sequence seq of each single process, we give the introduction to $Atraces(seq)$ presented in (14). The function $DAtraces(seq, \mathbb{1}, \Phi)$ is applied in defining the trace semantics (derived from algebraic semantics) $Atraces(seq)$ for seq. Now, we give the formalization and explanation of $DAtraces(seq, pc, tb)$ as below.

For configuration $hd(seq)$ in sequence seq, it is in the form of $h\&(act, tid, idx)$. If the Boolean condition h is not satisfied, the derived trace semantics of seq is ϕ, formalized in (20). The condition judgment is completed through the function $guard$. In addition, when h is true but the parameter act in $hd(seq)$ is ε, the trace semantics derived from $hd(seq)$ is $\{\varepsilon\}$. (19) gives its formal description.

$$
\begin{aligned}
&DAtraces(seq, pc, tb)\\
=_{\mathrm{df}}\ &\Bigg(\Bigg(\Bigg(\Bigg(\Big(\boxed{s^\frown\langle(\boxed{tb_i[len(\pi_3(hd(seq)))]}, cont, oflag, eflag)\rangle}\ ^\frown DAtraces(tl(seq), pc, tb) \quad (17)\\
&\quad \lhd\ last(\pi_3(hd(seq))) = 2\ \rhd\\
&\Big(\boxed{s^\frown\langle(\boxed{pc_i}, cont, oflag, eflag)\rangle}\ ^\frown DAtraces(tl(seq), \boxed{pc[(pc_i+1)/pc_i]}, \boxed{tb[tb_i+/tb_i]}\Big) \quad (18)\Bigg)\\
&\quad \lhd\ \pi_2(hd(seq)) = \lambda_i\ \rhd\Bigg)\\
&DAtraces(tl(seq), pc, tb)\Bigg)\\
&\quad \lhd\ \pi_1(hd(seq))! = \varepsilon\ \rhd\Bigg)\\
&DAtraces(tl(seq), pc, tb) \quad (19)\Bigg)\\
&\quad \lhd\ guard(hd(seq)) = \mathrm{true}\ \rhd\Bigg)\\
&\phi \quad (20)\Bigg),
\end{aligned}
$$

$DAtraces(\varepsilon, pc, tb) =_{\mathrm{df}} \{\varepsilon\}$,

where, $tb_i+ =_{\mathrm{df}} tb_i \Leftarrow (len(\pi_3(hd(seq))), pc_i)$.

Otherwise, the derived trace semantics of such a configuration $hd(seq)$ is always in the form of $\{s^\frown\langle(id, cont, oflag, eflag)\rangle \mid \pi_4^*(s) \in 0^*\}$, and constraint $\pi_4^*(s) \in 0^*$ indicates trace s is performed by the environment. The transformation of these three parameters $cont$, $oflag$ and $eflag$ is not complex, and it is listed in the following.

1) The element $cont$ can be obtained from the action act, and the moment when the read function $r(\_)$ works is that the recursion reaches the leaf processes. Then other processes can use the value gotten in the above read function directly.

2) We assign 0, 1, 2, 3 to variable $oflag$ once the action act is in the form of fence, $\langle x = e\rangle$, $x = e$ and

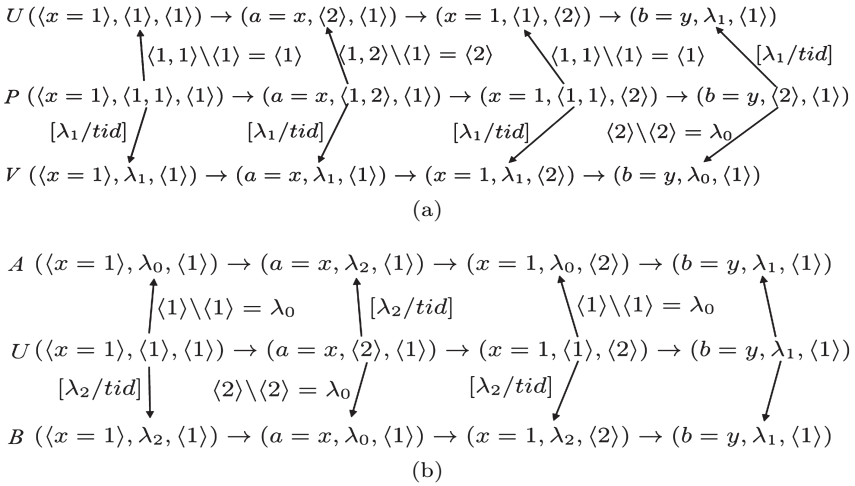

Fig.11. Extracting configuration sequence of each single process.

$a = e$ respectively.

3) If one configuration contains value λ_0, the corresponding snapshot in the derived trace semantics has such $eflag$ whose value is 1. Otherwise, the value of $eflag$ is 0.

In addition, we illustrate how to generate parameter id in the derived trace semantics. Given one sequence of the single process P, for the configurations with tid being λ_0, different snapshots derived from those configurations have different sequence numbers (i.e., parameter id). The numbers are increasing, but committing and propagating the same memory write have the same number. It is the same for the configurations contributed by each environment process.

Then for facilitating generating id in the trace semantic model, we introduce the auxiliary variable pc_i where $i \geqslant 0$, which is corresponding to the thread ID λ_i. Their relation is shown in Table 2. The initial value of the program counter pc_i is 1. Parameter pc used in function $DAtraces(seq, pc, tb)$ is the set of the above auxiliary variables and its initial value is $\mathbb{1}$.

When coming with a configuration $hd(seq)$, we first observe its tid to decide which program counter pc_i will be used in the following steps. Then we check whether idx of the above configuration ends in 2. If not, because the snapshots describing committing and propagating the same memory write have the same id, we need to generate the same id if two configurations have the same length of indices. Thus, we record the pair including the length of this configuration's idx and pc_i in the corresponding tb_i, which is modeled as tb_i+. Here, the initial value of tb_i is $\langle\rangle$, and operator $\Leftarrow$ is used to attach a pair to the tail of tb_i. Any tb_i is included in the tb set (i.e., one parameter in $DAtraces(seq, pc, tb)$), and tb is Φ initially. Then we make pc_i add 1, as described in the middle framed area in (18). Otherwise, we do not need to do any update.

Table 2. Corresponding Program Counter

Thread ID	Corresponding Program Counter
λ_0	pc_0
λ_1	pc_1
λ_2	pc_2
$\vdots$	$\vdots$
λ_n	pc_n

When all the preparation has been done, if the suffix of the configuration's idx is 2 with the function $last$, the parameter id of the derived snapshot is set to be $tb_i(len(\pi_3(hd(seq))))$. The framed area in (17) describes the update. Otherwise, it has the value pc_i, as modeled in the first framed area of (18).

Example 14. The full derivation of the trace semantics of $P =_{\text{df}} (x := 1||a := x)||b := y$ is formalized as below. Here, u_i where $i \in \{1, 2, 3, 4\}$ denotes the trace produced by the environment of the parallel composition P. In the following formalization, the numbers in bold show the changes of id during the execution of the function $Merge$.

$$
\begin{aligned}
M^+(seq, \boxed{1}) = Merge(Merge(M^+((((\langle x=1\rangle, \lambda_0, \langle 1\rangle) \to (a=x, \lambda_2, \langle 1\rangle) \to (x=1, \lambda_0, \langle 2\rangle) \to (b=y, \lambda_1, \langle 1\rangle)), \boxed{3}),\\
M^+((((\langle x=1\rangle, \lambda_2, \langle 1\rangle) \to (a=x, \lambda_0, \langle 1\rangle) \to (x=1, \lambda_2, \langle 2\rangle) \to (b=y, \lambda_1, \langle 1\rangle))), \boxed{3}),\\
M^+((((\langle x=1\rangle, \lambda_1, \langle 1\rangle) \to (a=x, \lambda_1, \langle 1\rangle) \to (x=1, \lambda_1, \langle 2\rangle) \to (b=y, \lambda_0, \langle 1\rangle)), \boxed{2}))\\
= \{Merge(Merge(u_1^\wedge\langle(1,(x,1),1,1)\rangle^\wedge u_2^\wedge\langle(1,(a,r(x)),3,0)\rangle^\wedge u_3^\wedge\langle(1,(x,1),2,1)\rangle^\wedge u_4^\wedge\langle(1,(b,r(y)),3,0)\rangle,\\
u_1^\wedge\langle(1,(x,1),1,0)\rangle^\wedge u_2^\wedge\langle(1,(a,r(x)),3,1)\rangle^\wedge u_3^\wedge\langle(1,(x,1),2,0)\rangle^\wedge u_4^\wedge\langle(1,(b,r(y)),3,0)\rangle),\\
u_1^\wedge\langle(1,(x,1),1,0)\rangle^\wedge u_2^\wedge\langle(2,(a,r(x)),3,0)\rangle^\wedge u_3^\wedge\langle(1,(x,1),2,0)\rangle^\wedge u_4^\wedge\langle(1,(b,r(y)),3,1)\rangle)\}\\
= \{Merge(u_1^\wedge\langle(1,(x,1),1,1)\rangle^\wedge u_2^\wedge\langle(\mathbf{2},(a,r(x)),3,1)\rangle^\wedge u_3^\wedge\langle(1,(x,1),2,1)\rangle^\wedge u_4^\wedge\langle(1,(b,r(y)),3,0)\rangle,\\
u_1^\wedge\langle(1,(x,1),1,0)\rangle^\wedge u_2^\wedge\langle(\mathbf{2},(a,r(x)),3,0)\rangle^\wedge u_3^\wedge\langle(1,(x,1),2,0)\rangle^\wedge u_4^\wedge\langle(1,(b,r(y)),3,1)\rangle)\}\\
= \{u_1^\wedge\langle(\mathbf{1},(x,1),1,1)\rangle^\wedge u_2^\wedge\langle(\mathbf{2},(a,r(x)),3,1)\rangle^\wedge u_3^\wedge\langle(\mathbf{1},(x,1),2,1)\rangle^\wedge u_4^\wedge\langle(\mathbf{3},(b,r(y)),3,1)\rangle\}.
\end{aligned}
$$

We should pay attention to that the read functions $r(x)$ and $r(y)$ get the concrete value before the $Merge$ function executes. It is to say that the values of them are both 0 in the trace semantics shown above.

In order to describe the intuitive understanding of the generation of id, we consider the generation of one sequence $(\langle x=1\rangle, \lambda_0, \langle 1\rangle) \to (a=x, \lambda_2, \langle 1\rangle) \to (x=1, \lambda_0, \langle 2\rangle) \to (b=y, \lambda_1, \langle 1\rangle)$ shown in Fig.12. Initially, the values of pc_0, pc_1 and pc_2 are all 1.

1) For the first configuration $(\langle x=1\rangle, \lambda_0, \langle 1\rangle)$, because of $\pi_3((\langle x=1\rangle, \lambda_0, \langle 1\rangle)) = \langle 1\rangle$, the value 1 of the program counter pc_0 is returned to the parameter id of

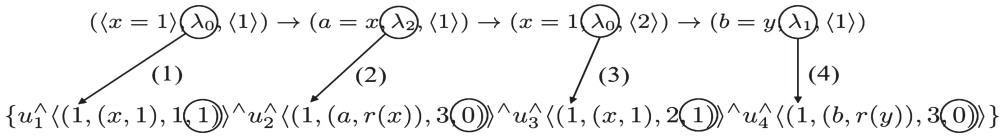

Fig.12. Generation of parameter *id*.

the snapshot generated from $(\langle x = 1\rangle, \lambda_0, \langle 1\rangle)$. It is described by the arrow (1) in Fig.12. Then the pair $(1, 1)$ is added in tb_0, and pc_0 is updated to 2.

2) The treatment of $(a = x, \lambda_2, \langle 1\rangle)$ is similar to that of $(\langle x = 1\rangle, \lambda_0, \langle 1\rangle)$. By using another program counter pc_2, the generated *id* is 1.

3) $\pi_3((\langle x = 1\rangle, \lambda_0, \langle 2\rangle)) = \langle 2\rangle$ requires us to search tb_0. The length of $\langle 2\rangle$ is 1; hence the corresponding value 1 is assigned to *id*.

4) The treatment of the configuration $(b = y, \lambda_1, \langle 1\rangle)$ is also similar to that of $(a = x, \lambda_2, \langle 1\rangle)$. The only difference is that the program counter pc_1 is used here, and then the value 1 is returned to *id*.

4 Related Work

Several effects have been made to model Total Store Order (TSO) formally. Abdulla *et al.* presented an efficient stateless model checking technique to check the programs under TSO and PSO memory models, whose basis is chronological traces [21]. Apart from the model checking technique, different program semantics is defined to describe the TSO model. Two provably-equivalent models of x86-TSO, including an intuitive operational model based on local write buffers and an axiomatic model were presented in [5]. Hóu *et al.* [22] gave the axiomatic TSO model and the operational TSO model on the top of the high-level ISA model and the low-level ISA model respectively. Khyzha and Gotsman [23] formalized the valid executions of the TSO memory model as graphs of memory access events subject to a set of validity axioms, inspired by the definition of C++ memory model [24]. Ridge introduced a Rely-Guarantee proof system for concurrent low-level assembly code and the weak x86-TSO memory model [25]. Process assertions included in the logic could refer to the local state of other processes. Kavanagh and Brookes used the concept of partially-ordered multiset (pomset) to give the denotational semantics, which could capture the behaviors permitted by SPARC TSO [26].

Unifying Theories of Programming (abbreviated as UTP) [9] developed by Hoare and He has been successfully applied in investigating the semantics and algebraic laws of a variety of programming languages. In this paper, we studied the denotational semantics of TSO, where our approach is based on UTP and the trace structure is applied. We also explored the algebraic laws of the TSO model, including a set of sequential and parallel laws. Based on them, every program could be converted into the head normal form of guarded choice. Then our approach can support the linearizability [16, 17] of TSO.

The UTP approach aims at proposing a convincing unified framework to combine and link denotational, algebraic and operational semantics. For the study of semantic linking, Hoare and He investigated the derivation of the operational semantics from the algebraic semantics [27]. According to the corresponding derivation strategy, the operational semantics of Communicating Sequential Processes (CSP) [20] could be derived from its algebra. The operational semantics of Guarded Command Language (GCL) was derived from its algebraic laws, using the derivation strategy called the step relation. Recently, Hoare and van Staden proposed a challenge research topic of the linking theory of semantics among algebra, denotations, transitions and deductions [28, 29]. Its starting point is from the algebraic semantics. Sheng *et al.* studied how algebraic semantics links with the corresponding operational semantics and denotational semantics for MDESL, where the mechanical approach is applied [30, 31]. This paper provides a derivation strategy for generating the trace semantics from the algebraic semantics under the TSO memory model.

5 Conclusions

TSO is a weak memory model which allows store-load reorder through each core's write buffer. This paper presented the trace semantics for the TSO memory model, acting in the denotational semantics style. Two functions *po* and *mo* depicting the program order and modification order were defined to remove all the invalid traces. The concept of guarded choice was introduced to express the execution of a program including a variety of reorderings, and then we investigated the algebraic laws for TSO, including a set of sequential

and parallel expansion laws. Further, we provided the derivation strategy to derive the trace semantics from its algebraic semantics. The method proposed in this paper can also be adapted to suit other weak memory models.

In the future, we will continue our work on the TSO memory model. We will explore the linking theories among the program semantics of TSO [32–34]. With some optimization for our semantic model, we will also consider the mechanization of the program semantics for TSO in proof assistant Coq/PVS [35, 36].

References

[1] Passos L B C, Pfitscher G H, Filho R T M. Performance evaluation of two parallel programming paradigms applied to the symplectic integrator running on COTS PC cluster. In *Proc. the 21st Int. Symp. Parallel and Distributed Processing*, Mar. 2007, pp.1-8. DOI: 10.1109/IPDPS.2007.370563.

[2] Adve S V, Gharachorloo K. Shared memory consistency models: A tutorial. *Computer*, 1996, 29(12): 66-76. DOI: 10.1109/2.546611.

[3] Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 1979, C-28(9): 690-691. DOI: 10.1109/TC.1979.1675439.

[4] Sorin D J, Hill M D, Wood D A. A Primer on Memory Consistency and Cache Coherence. Morgan & Claypool Publishers, 2011. DOI: 10.2200/S00346ED1V01Y201104CAC016.

[5] Owens S, Sarkar S, Sewell P. A better x86 memory model: x86-TSO. In *Proc. the 22nd Int. Conf. Theorem Proving in Higher Order Logics*, Aug. 2009, pp.391-407. DOI: 10.1007/978-3-642-03359-9_27.

[6] Dongol B, Derrick J, Smith G. Reasoning algebraically about refinement on TSO architectures. In *Proc. the 11th Int. Colloquium on Theoretical Aspects of Computing*, Sept. 2014, pp.151-168. DOI: 10.1007/978-3-319-10882-7_10.

[7] Kang J, Hur C K, Lahav O, Vafeiadis V, Dreyer D. A promising semantics for relaxed-memory concurrency. In *Proc. the 44th Symp. Principles of Programming Languages*, Jan. 2017, pp.175-189. DOI: 10.1145/3009837.3009850.

[8] Sewell P, Sarkar S, Owens S, Nardelli F Z, Myreen M O. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 2010, 53(7): 89-97. DOI: 10.1145/1785414.1785443.

[9] Hoare C A R, He J. Unifying Theories of Programming. Prentice Hall, 1998.

[10] Plotkin G D. A structural approach to operational semantics. Technical Report, Aarhus University, 1981. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid = 5155CF1E81938DF7CA5AEE78AA3CCD0D?doi=10.1.1.4.8 186-&rep=rep1&type=pdf, Sept. 2021.

[11] Stoy J E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1981.

[12] Hoare C A R, Hayes I J, He J, Morgan C C, Roscoe A W, Sanders J W, Sorensen I H, Spivey J M, Sufrin B A. Laws of programming. *Communications of the ACM*, 1987, 30(8): 672-686. DOI: 10.1145/27651.27653.

[13] Hoare C A R, He J, Sampaio A. Normal form approach to compiler design. *Acta Informatica*, 1993, 30(8): 701-739. DOI: 10.1007/BF01191809.

[14] Sampaio A. An Algebraic Approach to Compiler Design. World Scientific, 1997. DOI: 10.1142/2870.

[15] Kavanagh R, Brookes S. A denotational semantics for SPARC TSO. In *Proc. the 33rd Conf. Mathematical Foundations of Programming Semantics*, Jun. 2017, pp.223-239. DOI: 10.1016/j.entcs.2018.03.025.

[16] Travkin O, Wehrheim H. Handling TSO in mechanized linearizability proofs. In *Proc. the 10th Int. Haifa Verification Conference*, Nov. 2014, pp.132-147. DOI: 10.1007/978-3-319-13338-6_11.

[17] Winter K, Smith G, Derrick J. Observational models for linearizability checking on weak memory models. In *Proc. the 12th Int. Symp. Theoretical Aspects of Software Engineering*, Aug. 2018, pp.100-107. DOI: 10.1109/TASE.2018.00021.

[18] Tarski A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955, 5(2): 285-309. DOI: 10.2140/pjm.1955.5.285.

[19] Brookes S D. Full abstraction for a shared-variable parallel language. *Information and Computation*, 1996, 127(2): 145-163. DOI: 10.1006/inco.1996.0056.

[20] Hoare C A R. Communicating Sequential Processes. Prentice-Hall, 1985.

[21] Abdulla P A, Aronis S, Atig M F, Jonsson B, Leonardsson C, Sagonas K. Stateless model checking for TSO and PSO. *Acta Informatica*, 2017, 54(8): 789-818. DOI: 10.1007/s00236-016-0275-0.

[22] Hóu Z, Sanán D, Tiu A, Liu Y, Hoa K C, Dong J S. An Isabelle/HOL formalisation of the SPARC instruction set architecture and the TSO memory model. *Journal of Automated Reasoning*, 2021, 65(4): 569-598. DOI: 10.1007/s10817-020-09579-4.

[23] Khyzha A, Gotsman A. Compositional reasoning about concurrent libraries on the axiomatic TSO memory model. Technical Report, IMDEA Software Institute, 2012. https://pageperso.lis-lab.fr/~pierrealain.reynier/publis/movep12.pdf#page=116, Mar. 2021.

[24] Batty M J. The C11 and C++ 11 concurrency model [Ph.D. Thesis]. Wolfson College, University of Cambridge, Cambridge, 2015.

[25] Ridge T. A Rely-Guarantee proof system for x86-TSO. In *Proc. the 3rd Int. Conf. Verified Software: Theories, Tools, and Experiments*, Aug. 2010, pp.55-70. DOI: 10.1007/978-3-642-15057-9_4.

[26] Kavanagh R, Brookes S. A denotational account of C11-style memory. arXiv:1804.04214, 2018. https://arxiv.org/abs/1804.04214, Mar. 2021.

[27] He J, Hoare C A R. From algebra to operational semantics. *Information Processing Letters*, 1993, 45(2): 75-80. DOI: 10.1016/0020-0190(93)90219-Y.

[28] Hoare C A R, Van Staden S. The laws of programming unify process calculi. *Science of Computer Programming*, 2014, 85: 102-114. DOI: 10.1016/j.scico.2013.08.012.

[29] Hoare C A R. Laws of programming: The algebraic unification of theories of concurrency. In *Proc. the 25th Int. Conf. Concurrency Theory*, Sept. 2014, pp.1-6. DOI: 10.1007/978-3-662-44584-6_1.

[30] Sheng F, Zhu H, He J, Yang Z, Bowen J P. Theoretical and practical approaches to the denotational semantics for MDESL based on UTP. *Formal Aspects of Computing*, 2020, 32(2): 275-314. DOI: 10.1007/s00165-020-00513-4.

[31] Sheng F, Zhu H, He J, Yang Z, Bowen J P. Theoretical and practical aspects of linking operational and algebraic semantics for MDESL. *ACM Transactions on Software Engineering and Methodology*, 2019, 28(3): Article No. 14. DOI: 10.1145/3295699.

[32] Zhu H, Yang F, He J, Bowen J P, Sanders J W, Qin S. Linking operational semantics and algebraic semantics for a probabilistic timed shared-variable language. *The Journal of Logic and Algebraic Methods Program*, 2012, 81(1): 2-25. DOI: 10.1016/j.jlap.2011.06.003.

[33] Zhu H. Linking the semantics of a multithreaded discrete event simulation language [Ph.D. Thesis]. Institute for Computing Research, London South Bank University, 2005.

[34] Chen Y. Generic composition. *Formal Aspects of Computing*, 2002, 14(2): 108-122. DOI: 10.1007/s001650200031.

[35] Huet G, Kahn G, Paulin-Mohring C. The Coq proof assistant—A tutorial (Version v8.1). Technical Report, INRIA, 2005. https://coq.inria.fr/distrib/current/refman/, Mar. 2021.

[36] Owre S, Rushby J M, Shankar N. PVS: A prototype verification system. In *Proc. the 11th Int. Conf. Automated Deduction*, Jun. 1992, pp.748-752. DOI: 10.1007/3-540-55602-8_217.

Li-Li Xiao is currently a Ph.D. candidate in Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai. She received her B.S. degree in software engineering in Xiangtan University, Xiangtan, in 2017. Her research interests include process algebra and its applications, program analysis and verification, and weak memory.

Hui-Biao Zhu is currently a professor in East China Normal University, Shanghai. He earned his Ph.D. degree in formal methods from London South Bank University, London, in 2005. During these years, he has studied various semantics and their linking theories for Verilog, SystemC, web services and probability system. He was the Chinese PI of the Sino-Danish Basic Research Center IDEA4CPS.

Qi-Wen Xu received his B.S. degree in computer science from East China Normal University, Shanghai, in 1985, and his Ph.D. degree in computer science from Oxford University Computing Laboratory, Oxford, in 1992. He is currently an assistant professor in the University of Macau, Macau. His research interests include formal methods, concurrency and security.

Appendix

A. Read Function

Here we introduce the read function r in detail. Firstly, we check whether the variable read from is global or not, and use the functions g and l to obtain the value of a global or local variable respectively, because the treatment of them is not the same. We use *global* to represent the set of all the global variables.

$$\begin{aligned} & r(x, tr^\frown\langle event\rangle) \\ =_{\mathrm{df}}\ & g(x, tr^\frown\langle event\rangle) \lhd x \in global \rhd l(x, tr^\frown\langle event\rangle), \\ & r(x, \langle\rangle) =_{\mathrm{df}} g(x, \langle\rangle) \lhd x \in global \rhd l(x, \langle\rangle), \end{aligned}$$

where $tr^\frown\langle event\rangle$ represents the trace prefixing the read statement. The symbol $e \lhd h \rhd f$ stands for e if the condition judgment h is true; otherwise f. Note that, for simplicity we only use $r(x)$ in the snapshots in a trace.

For global variables, in the beginning we need to search the corresponding thread's write buffer. Once the value can be obtained, the procedure terminates immediately. If not, the shared memory will be explored for the final answer. Considering our trace model, both two search processes are carried out in the reverse order, and we will use the initial value 0 if nothing can be found in the memory.

$$\begin{aligned} & g(x, tr^\frown\langle event\rangle) \\ =_{\mathrm{df}}\ & \begin{pmatrix} m(x, tr^\frown\langle event\rangle) \\ \lhd \begin{pmatrix} w(x, tr^\frown\langle event\rangle) = \mathrm{null} \vee \\ cnt_1(x, tr^\frown\langle event\rangle) = \\ \qquad cnt_2(x, tr^\frown\langle event\rangle) \end{pmatrix} \rhd \\ w(x, tr^\frown\langle event\rangle) \end{pmatrix}, \\ & g(x, \langle\rangle) =_{\mathrm{df}} m(x, \langle\rangle). \end{aligned}$$

There are two situations when illustrating that nothing can be obtained from the store buffer. One is that for location x, the writes to it have not been committed to the buffer. The other is that the writes to x have all been propagated from the buffer to the main memory.

In our trace model, the latter situation indicates that the number of the snapshots which contain location x and target at the write buffer, and that targeting at the shared memory contributed by the same thread are the same. These numbers can be achieved with the application of functions cnt_1 and cnt_2 respectively. The definition of them is similar, and then we only present the formalization of function cnt_1.

$$\begin{aligned} &cnt_1(x, tr^{\wedge}\langle event\rangle)\\ &=_{\mathrm{df}} \begin{pmatrix} cnt_1(x,tr)+1\\ \lhd \begin{pmatrix} \mathrm{ASCII}(\pi_1(\pi_2(event))) = \mathrm{ASCII}(x)\\ \wedge \pi_3(event) = 1 \wedge \pi_4(event) = 1 \end{pmatrix} \rhd\\ cnt_1(x,tr) \end{pmatrix},\\ &cnt_1(x, \langle\rangle) =_{\mathrm{df}} 0. \end{aligned}$$

Function w is applied to get the value of a specific variable from write buffer.

$$\begin{aligned} &w(x, tr^{\wedge}\langle event\rangle)\\ &=_{\mathrm{df}} \begin{pmatrix} \begin{pmatrix} \pi_2(\pi_2(event))\\ \lhd \mathrm{ASCII}(\pi_1(\pi_2(event))) = \mathrm{ASCII}(x) \rhd \end{pmatrix}\\ w(x,tr)\\ \lhd\ \pi_3(event) = 1 \wedge \pi_4(event) = 1 \rhd\\ w(x,tr) \end{pmatrix},\\ &w(x, \langle\rangle) =_{\mathrm{df}} \mathrm{null}. \end{aligned}$$

We search the trace, and when meeting a snapshot, we check if its $oflag$ and $eflag$ are both 1, because each thread can only see its private write buffer. If the previous conditions are satisfied, we judge whether the variable contained in the snapshot is the one we want to read. If the judgment is true, we get the value of the variable and terminate the searching process. Otherwise, null will be assigned to this function. As known to all, ASCII is used to specify the binary numbers of common symbols.

Due to the fact that the memory is visible to all threads, what we concern about is whether the snapshot we check contains the variable x, denoted by the latter in the conjunction ($\wedge$) formula. The former states if this snapshot takes effect in the shared memory.

$$\begin{aligned} &m(x, tr^{\wedge}\langle event\rangle)\\ &=_{\mathrm{df}} \begin{pmatrix} \pi_2(\pi_2(event))\\ \lhd \begin{pmatrix} \pi_3(event) = 2 \wedge\\ \mathrm{ASCII}(\pi_1(\pi_2(event))) = \mathrm{ASCII}(x) \end{pmatrix} \rhd\\ m(x,tr) \end{pmatrix},\\ &m(x, \langle\rangle) =_{\mathrm{df}} 0. \end{aligned}$$

Example 15. We use the programs below to illustrate different scenarios we can get values.

1) *Reading from Write Buffer: Consider the Program Below.*

$$\begin{pmatrix} x := 1;\\ a := x;\\ b := y \end{pmatrix} \parallel \begin{pmatrix} y := 1;\\ c := y;\\ d := x \end{pmatrix}.$$

Assume the pre-existing trace is $\langle(1,(x,1),1,1)\rangle$ before performing $a := x$. Hence, when executing the read function relevant to x, we can get the value 1 from the buffer described by the third element 1.

2) *Reading From Memory: Consider the Parallel Program Below.*

$$\begin{pmatrix} x := 1;\\ a := y \end{pmatrix} \parallel \begin{pmatrix} y := 1;\\ b := x \end{pmatrix}.$$

Suppose that the program $x := 1; a := y$ is performed in core 1, while $y := 1; b := x$ is carried out in core 2. When we are going to read the value of y, the available trace of core 1 is $\langle(1,(x,1),1,1),\ (1,(x,1),2,1)\rangle$, or $\langle(1,(x,1),1,1)\rangle$. That is to say, nothing can be obtained from the write buffer because something that is related to y is absent. Then, we need to observe the states of the shared memory. If the snapshot $(1,(y,1),2,1)$, a part of the trace of the program in core 2, has been generated in the memory which is visible to both core 1 and core 2, variable y can get 1; otherwise it can get the initial value of 0.

Considering local variables, the model requires us to search the thread's register. If we do not find the value from the previous trace, the initial value 0 is used.

$$\begin{aligned} &l(x, tr^{\wedge}\langle event\rangle)\\ &=_{\mathrm{df}} \begin{pmatrix} \pi_2(\pi_2(event))\\ \lhd \begin{pmatrix} \mathrm{ASCII}(\pi_1(\pi_2(event))) = \mathrm{ASCII}(x) \wedge\\ \pi_3(event) = 3 \wedge \pi_4(event) = 1 \end{pmatrix} \rhd\\ l(x,tr) \end{pmatrix},\\ &l(x, \langle\rangle) =_{\mathrm{df}} 0. \end{aligned}$$

We can know that based on the read function produced by the read mechanism under the TSO memory model, the private information is not visible to others.