Electronic Preprint for Journal of Information Processing Vol.25

Regular Paper

Concurrent Program Logic for Relaxed Memory Consistency Models with Dependencies across Loop Iterations

Tatsuya Abe[1,a)] Toshiyuki Maeda[1,2,b)]

Received: May 10, 2016, Accepted: August 4, 2016

Abstract: Relaxed memory consistency models specify effects of executions of statements among threads, which may or may not be reordered. Such reorderings may cross loop iterations. To the best of our knowledge, however, there exists no concurrent program logic which explicitly handles the reorderings across loop iterations. This paper provides concurrent program logic for relaxed memory consistency models that can represent, for example, total store ordering, partial store ordering, relaxed memory ordering, and acquire and release consistency. There are two novel aspects to our approach. First, we translate a concurrent program into a family of directed acyclic graphs with finite nodes and transitive edges called program graphs according to a memory consistency model that we adopt. These represent dependencies among statements which represent reorderings of not only statements but also visibility of their effects. Second, we introduce auxiliary variables that temporarily buffer the effects of write operations on shared memory, and explicitly describe the reflections of the buffered effects to shared memory. Specifically, we define a small-step operational semantics for the program graphs with the introduced auxiliary variables, then define sound and relatively complete logic to the semantics.

Keywords: memory consistency model, shared memory, concurrent program verification, dependencies across loop iterations, graph representation, partial correctness, relative completeness

1. Introduction

A memory consistency model is a formal definition of the behavior of shared memory that is simultaneously accessed by multiple threads. *Relaxed* memory consistency models [6] allow the shared memory to behave differently than that for one sequential thread. Today, relaxed memory consistency models are widely adopted by programming languages (e.g., Refs. [13], [22], [23], [24], [29], [34]) and CPU architectures (e.g., Refs. [20], [21], [38], [39]). This is because recent CPUs tend to have a number of cores, and large computing systems consist of a large number of computing nodes. Thus, it is difficult to adhere to non-relaxed memory consistency models without severe performance degradation.

One big problem with relaxed memory consistency models is that programming is very error prone because, from the programmer's point of view, the shared memory behaves unexpectedly. For example, in a certain relaxed model, the effects of memory operations performed by one thread can be observed by another thread in a different order. To address this problem, program verification can be used to detect bugs caused by unexpected behavior in the relaxed memory consistency models. Recently, program verification for relaxed memory consistency models has been extensively studied [1], [2], [3], [5], [7], [8], [9], [11], [14], [25], [35], [40], [49].

[1] STAIR Lab., Chiba Institute of Technology, Narashino, Chiba 275–0016, Japan
[2] RIKEN AICS, Kobe, Hyogo 650–0047, Japan
[a)] abet@stair.center
[b)] tosh@stair.center

```
while (r0 == 0) {
 r0 = z;
 r1 = y;
 x = r1 + 1;
 fence;
 y = r1 + 1;
}
```

```
r2 = x;
r3 = y;
z = 1;
```

where all the variables are initialized to `0`.

Fig. 1 A concurrent program with dependencies across loop iterations.

One approach of program verification for ensuring safety of programs is *program logic*. Program logic is based on *proofs*, which ensure that a given property holds on *any* execution of programs. Recently, concurrent program logics with relaxed memory consistency models are seen [17], [33], [44], [45].

However, it is not easy to construct concurrent program logic that can handle programs with loops under relaxed memory consistency models since *dependencies* among statements which represent reorderings of not only statements but also visibility of their effects may cross loop iterations.

For example, let us consider the concurrent program in **Fig. 1** consisting of two threads. The left-side thread writes the same value to `x` and `y` on a shared memory. The values are incremented at each iteration. The right-side thread reads values from `x` and `y`.

Under relaxed memory consistency models, effects of stores to a shared memory may be reordered. For example, `r2==2 &&`

r3==0 may occur under Partial Store Ordering (PSO) [6] since an effect of a write operation to y can be overtaken by those of write operations to x at the later iterations. To the best of our knowledge, there exists no concurrent program logic that is sound and relatively complete to such semantics.

On the other hand, under Total Store Ordering (TSO) [6], r2==2 && r3==0 never occurs since TSO *prohibits* effects of write operations to x to overtake that of a write operation to y. Thus, dependencies/independencies between effects across loop iterations depend on memory consistency models that we adopt. Since memory consistency models refer to effects that occur in low-level, we cannot handle effects across loop iterations through descriptions of loops in high-level. The authors learned its significance by an experience of detecting a similar bug in a numeric computing program of climate simulation [36], and started to construct verification theories that are parameterized by memory consistency models in consideration of dependency/independency between effects across loop iterations [4]. To the best of our knowledge, there exists no concurrent program logic which explicitly describes dependencies across loop iterations under relaxed memory consistency models.

This paper provides concurrent program logic for standard relaxed memory consistency models that can represent, for example, TSO, PSO, Relaxed Memory Ordering (RMO) [6], and Acquire and Release consistency (AR) [20]. There are two novel aspects to our approach. First, we translate a concurrent program into a family of directed acyclic graphs with finite nodes and transitive edges called *program graphs* according to a memory consistency model that we adopt. These represent the dependencies among statements, which may or may not be reordered even across loop iterations. Second, we introduce auxiliary variables that temporarily buffer the effects of write operations on shared memory, and explicitly handle the reflections of the buffered effects to shared memory. Specifically, we define a small-step operational semantics for the program graphs with the introduced auxiliary variables, then define a sound and relatively complete logic to the semantics.

The rest of this paper is organized as follows. Section 2 introduces program graphs, and memory consistency models. Section 3 defines the operational semantics for the program graphs, and Section 4 explains our concurrent program logic, and its relation to the operational semantics. Section 5 slightly extends the operational semantics and the concurrent program logic, then shows proofs of soundness and relative-completeness of the extended proof (and soundness of the original one). Section 6 shows example derivations on the concurrent program in Fig. 1 and its variants. Section 7 discusses related work, and Section 8 concludes the paper and discusses ideas for future work.

2. Representations of Programs and Memory Consistency Models

In this section, we formally define our target concurrent programs, representations of programs with reordering structures, representations of memory consistency models, and present some related definitions.

2.1 Programs

Similar to the conventional Hoare logic (e.g., [19]), sequential programs are defined as sequences of statements. Parentheses are often omitted, and operators are assumed to be left associative. Let r denote thread-local variables (which cannot be accessed by other threads), $x, y, \ldots$ denote shared variables, and e denotes thread-local expressions (thread-local variables, constant values *val*, arithmetic operations, and so on). A sequential program can then be defined as:

$$\mathtt{S}^i ::= \mathtt{SK}^i \mid \mathtt{MV}^i\ r\ e \mid \mathtt{LD}^i\ r\ x \mid \mathtt{ST}^i\ x\ e \mid \mathtt{FN}^i \mid \mathtt{IF}^i\ \varphi ?\mathtt{S}^i : \mathtt{S}^i \mid \mathtt{WL}^i\ \varphi ?\mathtt{S}^i \mid \mathtt{S}^i ; \mathtt{S}^i$$

$$\varphi ::= e = e \mid e \leq e \mid \neg\varphi \mid \varphi \supset \varphi \mid \forall r.\varphi \ .$$

In the above definition, the superscript i represents (an identifier of) the thread on which the associated statement will be executed. In the rest of this paper, this superscript is often omitted when the context renders it obvious. The SK statement denotes an ordinary no-effect statement (SKip). As in conventional Hoare logic, MV $r\,e$ denotes an ordinary variable substitution (MoVe). The load and store statements denote read and write operations, respectively, for shared variables (LoaD and STore). The effect of the store statement issued by one thread may not be observed by the other threads until the FN statement is issued. The FN statement ensures that other threads can observe the effect of store statements (FeNce). The IF and WL statements denote ordinary conditional branches and iterations, respectively, where we adopt ternary conditional operators (IF-then-else-end and WhiLe-do-end). Finally, S; S denotes a sequential composition of statements.

We write $\varphi \vee \psi$ and $\varphi \wedge \psi$, as $(\neg\varphi) \supset \psi$ and $\neg((\neg\varphi) \vee (\neg\psi))$, respectively. In the following, we assume that $\neg$, $\wedge$, $\vee$, and $\supset$ are stronger with respect to their connective powers. In addition to the above definition, $\top$ is defined as a tautology $\forall r.\, r = r$.

A concurrent program with N threads is defined as the composition of sequential programs by parallel connectives $\|$ as follows:

$$\mathtt{P} ::= \mathtt{S}^0 \parallel \mathtt{S}^1 \parallel \ldots \parallel \mathtt{S}^{N-1} \ .$$

In this paper, we assume that the number of threads N is fixed during program execution.

2.2 Program Graphs

As mentioned in Section 1, instead of directly handling the concurrent programs defined above, we translate them to a family of *program graphs* according to a memory consistency model. These are directed acyclic graphs with finite nodes and transitive edges that represent dependencies among statements. Specifically, the program graph G consists of nodes (called *commands*) and edges, where the nodes are defined as follows:

$$C^i ::= \mathtt{MV}^i\ r\ e \mid \mathtt{LD}^i\ r\ x \mid \mathtt{ST}^i\ x\ e \mid \mathtt{RF}^i\ x \mid \mathtt{FN}^i \mid \varphi^i \ .$$

As shown in the definition, we introduce $\mathtt{RF}^i\ x$, which explicitly denotes the effect of $\mathtt{ST}^i\ x\,e$ on a shared variable x (ReFlect). That is, the effect is not observed until $\mathtt{RF}^i\,x$ is executed. Also, note that IF and WL statements do not exist, and guards φ^i are introduced. IF and WL statements are *finitely* unfolded. That is, a program is represented by a (possibly infinite) family of program graphs.

Proving correctness of a program via its program graphs is discussed in Section 4. Nodes that share a common command are assumed to be identified by appropriate tags although this paper does not explicitly handle them.

In the generated program graphs, edges connect one node to another if the command represented by the latter node depends on that of the former node under a given memory consistency model. Let us consider the following three examples under TSO. The sequential composition $\mathtt{ST}^0\ x\ r_0; \mathtt{LD}^0\ r_1\ x$ is translated to the program graph $(\mathtt{ST}^0\, x r_0 \rightarrow \mathtt{LD}^0 r_1\, x) \sqcup \mathtt{RF}^0\, x$, which has an edge between nodes denoting the commands, because the two commands access the same shared variable x (and the effect of `store` is explicitly handled by `reflect`). Although the sequential composition $\mathtt{LD}^0\ r_2\ x; \mathtt{ST}^0\ x\ r_3$ does not appear to have such a dependency, this is translated to the program graph $\mathtt{LD}^0 r_2\, x \rightarrow \mathtt{ST}^0 x r_3 \rightarrow \mathtt{RF}^0 x$, which has an edge between nodes denoting the commands, because reordering of the commands makes a dependency that does not originally exist if no edge exist between the nodes. In contrast, the sequential composition $\mathtt{LD}^0\ r_2\ x; \mathtt{ST}^0\ y\ r_3$ is translated to the program graph $\mathtt{LD}^0\ r_2\ x \sqcup (\mathtt{ST}^0\ y\ r_3 \rightarrow \mathtt{RF}^0\ y)$ (that is, no edges between LD and ST), because there is no dependency where $G_0 \sqcup G_1$ denotes the disjoint union of G_0 and G_1 in graph theory.

We denote the nodes, edges, root nodes, and leaf nodes of a program graph G as $N(G)$, $E(G)$, $R(G)$, and $L(G)$, respectively. The root nodes are defined as those that do not appear as the destination of any edge, and the leaf nodes are defined as nodes that do not appear as the source of any edge. $G_0 \rightarrow G_1$ is defined by $N(G_0 \rightarrow G_1) = N(G_0) \cup N(G_1)$ and $E(G_0 \rightarrow G_1) = E(G_0) \cup E(G_1) \cup (L(G_0) \times N(G_1))$. In the rest of this paper, we often treat a program graph like a set of nodes when its edges are obvious from the context. For example, $G \setminus \{C\}$ denotes a full sub-graph of G that does not have a node C. Moreover, we denote a graph that holds only one node $\{C\}$ as C. Edges derived from transitivity are often omitted for readability.

2.3 Memory Consistency Models: Translations

We represent a memory consistency model as a translation from programs into families of program graphs. In this section, we give four example translations that represent TSO, PSO, RMO [6], and AR [20].

We first decompose a parallel composition into a set of sequential programs. Next, we translate each sequential program into a program graph. Finally, we create a program graph by taking a disjoint union of the program graphs.

The translation of sequential programs consists of two steps. The first step translation defined in **Fig. 2** is to regard sequential compositions of statements as a sequence of commands, unfold IF and WL statements finitely, and transform the program to sequences (of the finite length) of commands. Unfolding $\mathtt{IF}\varphi?\mathtt{S}_0{:}\mathtt{S}_1$ that has no nested IF and WL statement generates two sequences of commands according to $\mathtt{S}_0$ and $\mathtt{S}_1$. Similarly, for example, unfolding $\mathtt{WL}\ \varphi?\mathtt{S}_0$ generates infinite sequences (of the finite length) of commands. Each sequence corresponds to the number of executions of iterations of the loop (as discussed in Section 4).

The second step translation g shown in **Fig. 3** is to create nodes from the sequences of commands generated by the first step and edges between the nodes that have dependencies where $\overrightarrow{C}$ means a sequence of commands. The dependency relation $C^i \triangleright C_0^i$ between C^i and C_0^i is defined in **Tables 1**, **2**, **3** and **4**.

Table 1 represents a dependency relation of TSO where $\mathrm{fv}(e)$ means variables that occur in e. Formulas in cells in Table 1 mean assumptions that make $C^i \triangleright C_0^i$ true. For example, in Table 1, $\top$ for $\mathtt{LD}\ r'\ x' \triangleright \mathtt{ST}\ x\ e$ means that $\mathtt{LD}\ r'\ x'$ unconditionally depends on $\mathtt{ST}\ x\ e$. On the other hand, $\mathtt{RF}\ x'$ never depends on $\mathtt{LD}\ r\ x$ (blank cells mean contradiction). Although we do not explain each cell in detail, Table 1 represents key characteristics of TSO, that is, two loads follow the program order, two stores also follow the program order, no store overtakes any load, and a load

$$
\begin{aligned}
f(\mathtt{SK}^i) &= \varnothing & f(\mathtt{MV}^i\ r\ e) &= \{\mathtt{MV}^i\ r\ e\} \\
f(\mathtt{LD}^i\ r\ x) &= \{\mathtt{LD}^i\ r\ x\} & f(\mathtt{ST}^i\ x\ e) &= \{\mathtt{ST}^i\ x\ e\} \\
f(\mathtt{FN}^i) &= \{\mathtt{FN}^i\} & f(\mathtt{S}_0; \mathtt{S}_1) &= \{\, S_0 \cdot S_1 \mid S_0 \in f(\mathtt{S}_0), S_1 \in f(\mathtt{S}_1) \,\} \\
f(\mathtt{IF}^i\ \varphi?\mathtt{S}_0{:}\mathtt{S}_1) &= \{\, \varphi \cdot S_0 \mid S_0 \in f(\mathtt{S}_0) \,\} \cup \{\, \neg\varphi \cdot S_1 \mid S_1 \in f(\mathtt{S}_1) \,\} \\
f(\mathtt{WL}^i\ \varphi?\mathtt{S}_0) &= \bigcup \{\, \{\, S_0^n \cdot \neg\varphi \mid S_0^n \in Itr^n(f, \mathtt{S}_0) \,\} \mid 0 \le n \,\} \\
\text{where } Itr^0(f, \mathtt{S}) &= \varnothing \\
Itr^n(f, \mathtt{S}) &= \{\, S^{n-1} \cdot \varphi \cdot S^n \mid S^{n-1} \in Itr^{n-1}(f, \mathtt{S}), S^n \in f(\mathtt{S}) \,\} \quad (1 \le n) \\
(C_{0,0}; \ldots; C_{0,m-1}) \cdot (C_{1,0}; \ldots; C_{1,n-1}) &= C_{0,0}; \ldots; C_{0,m-1}; C_{1,0}; \ldots; C_{1,n-1}\ .
\end{aligned}
$$

Fig. 2 A transformation from the program to sequences of commands.

Table 1 A dependency relation $C \triangleright C_0^i$ that represents TSO.

		C_0^i					
		$\mathtt{MV}\ r\ e$	$\mathtt{LD}\ r\ x$	$\mathtt{ST}\ x\ e$	$\mathtt{RF}\ x$	FN	φ
C^i	$\mathtt{MV}\ r'\ e'$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
	$\mathtt{LD}\ r'\ x'$	$r' \in \mathrm{fv}(e)$ or $r' = r$	$\top$	$\top$	$\top$	$\top$	$\top$
	$\mathtt{ST}\ x'\ e'$	$r \in \mathrm{fv}(e')$	$r \in \mathrm{fv}(e')$ or $x' = x$	$\top$	$\top$	$\top$	$\top$
	$\mathtt{RF}\ x'$				$\top$	$\top$	
	FN	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
	φ'	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 2 The PSO dependency relation $C \triangleright C_0^i$.

		C_0^i					
		$\mathtt{MV}\ r\ e$	$\mathtt{LD}\ r\ x$	$\mathtt{ST}\ x\ e$	$\mathtt{RF}\ x$	FN	φ
C^i	$\mathtt{MV}\ r'\ e'$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
	$\mathtt{LD}\ r'\ x'$	$r' \in \mathrm{fv}(e)$ or $r' = r$	$\top$	$\top$	$\top$	$\top$	$\top$
	$\mathtt{ST}\ x'\ e'$	$r \in \mathrm{fv}(e')$	$r \in \mathrm{fv}(e')$ or $x' = x$	$\top$	$\top$	$\top$	$\top$
	$\mathtt{RF}\ x'$				$\boxed{x' = x}$	$\top$	
	FN	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
	φ'	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

$$
N(g(C^i)) = \begin{cases} \{\mathtt{ST}^i\ x\ e, \mathtt{RF}^i\ x\} & \text{if } C^i = \mathtt{ST}^i\ x\ e \\ \{C^i\} & \text{o.w.} \end{cases}
\qquad
N(g(\overrightarrow{C^i}; C_0^i)) = N(g(\overrightarrow{C^i})) \cup N(g(C_0^i))
$$

$$
E(g(C^i)) = \varnothing
\qquad
E(g(\overrightarrow{C^i}; C_0^i)) = \begin{cases} E(g(\overrightarrow{C^i})) \cup \{\, \langle C^i, C_1^i \rangle \mid C^i \triangleright C_1^i, C^i \in N(g(\overrightarrow{C^i})), C_1^i \in N(g(C_0^i)) \,\} \cup \{\langle C_0^i, \mathtt{RF}^i\ x \rangle\} & \text{if } C_0^i = \mathtt{ST}^i\ x\ e \\ E(g(\overrightarrow{C^i})) \cup \{\, \langle C^i, C_0^i \rangle \mid C^i \triangleright C_0^i, C^i \in N(g(\overrightarrow{C^i})) \,\} & \text{o.w.} \end{cases}
$$

Fig. 3 Graph creations with memory consistency models.

Table 3 The RMO dependency relation $C \triangleright C_0^i$.

		C_0^i					
		MV $r\ e$	LD $r\ x$	ST $x\ e$	RF x	FN	φ
C^i	MV $r'\ e'$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
	LD $r'\ x'$	$r' \in \mathrm{fv}(e)$ or $r' = r$	$\boxed{r' = r}$	$\boxed{r' \in \mathrm{fv}(e) \text{ or } x' = x}$	$\top$	$\top$	$\top$
	ST $x'\ e'$	$r \in \mathrm{fv}(e')$	$r \in \mathrm{fv}(e')$ or $x' = x$	$\top$	$\top$	$\top$	$\top$
	RF x'				$\boxed{x' = x}$	$\top$	
	FN	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
	φ'	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 4 A dependency relation $C \triangleright C_0^i$ that represents AR.

		C_0^i		
		$\langle$LD $r\ x, A\rangle$	$\langle$ST $x\ e, A\rangle$, rel $\notin A$	$\langle$ST $x\ e, A\rangle$, rel $\in A$
C^i	$\langle$LD $r'\ x', A'\rangle$, acq $\notin A'$	$r' = r$	$\top$	$\top$
	$\langle$LD $r'\ x', A'\rangle$, acq $\in A'$	$\top$	$\top$	$\top$
	$\langle$ST $x'\ e', A'\rangle$	$r \in \mathrm{fv}(e')$ or $x' = x$	$x' = x$	$\top$

may overtake stores. Please note that neither ST $x'\ e' \triangleright$ LD $r\ x$ nor RF $x' \triangleright$ LD $r\ x$ is $\top$ in Table 1. Under the so-called Sequential Consistency (SC) [28], it should be $\top$.

Table 2 represents a dependency relation of PSO. The unique difference between PSO and TSO is that PSO does not preserve orders of stores and reflects. This is achieved by relaxing the assumption of RF $x' \triangleright$ RF x from $\top$ to $x' = x$ (emphasized by the rectangle in Table 2).

Table 3 represents a dependency relation of RMO. The differences between RMO and PSO are that RMO does not preserve an order of loads, and load and store. These are achieved by strengthening the assumption of LD $r'\ x' \triangleright$ LD $r\ x$ from $\top$ to $r' = r$, and LD $r'\ x' \triangleright$ ST $x\ e$ from $\top$ to $r' \in \mathrm{fv}(e)$ or $x' = x$ (emphasized by the rectangles in Table 3), respectively.

Finally, let us give a translation to represent AR, which is adopted by C11/C++11 [22], [24]. Under AR, no LD statement can overtake a LD statement with attribute acquire (acq), and no ST can delay a ST with attribute release (rel).

We modify the definition of statements as follows:

$$\mathtt{S}^i ::= \ldots \mid \langle \mathtt{LD}^i\ r\ x, A\rangle \mid \langle \mathtt{ST}^i\ x\ e, A\rangle$$

where A is a set of attributes, acq, rel, etc. Statement $\langle \mathtt{LD}^i\ r\ x, A\rangle$ where acq $\in A$ means $\mathtt{LD}^i\ r\ x$ that no load and store statement overtake. Statement $\langle \mathtt{ST}^i\ x\ e, A\rangle$ where rel $\in A$ means $\mathtt{ST}^i\ x\ e$ that cannot overtake any load and store.

A dependency relation is defined in Table 4 where we omit cells except those for loads and stores. Attributes are used only to construct program graphs, and not used on semantics of program graphs as described in Section 3 and its logic as described in Section 4.

3. Operational Semantics

In this section, we define a small-step operational semantics for the program graphs defined in Section 2.2. Specifically, the semantics is defined as a standard state transition system, where a *state* (written as st) is represented as a pair of substitutions $\langle \sigma, \Sigma\rangle$. The first element of the pair σ gives the value of thread-local

$$\frac{\mathtt{MV}^i\ r\ e \in R(G)}{\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle G \setminus \{\mathtt{MV}^i\ r\ e\}, \langle \sigma[r := \langle\!| e |\!\rangle_\sigma], \Sigma\rangle\rangle}\ \text{(O-MV)}$$

$$\frac{\mathtt{LD}^i\ r\ x \in R(G)}{\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle G \setminus \{\mathtt{LD}^i\ r\ x\}, \langle \sigma[r := \sigma[\Sigma^i]x], \Sigma\rangle\rangle}\ \text{(O-LD)}$$

$$\frac{\mathtt{ST}^i\ x\ e \in R(G)}{\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle G \setminus \{\mathtt{ST}^i\ x\ e\}, \langle \sigma, \Sigma[i := \Sigma^i[x := \langle\!| e |\!\rangle_\sigma]]\rangle\rangle}\ \text{(O-ST)}$$

$$\frac{\mathtt{RF}^i\ x \in R(G)}{\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle G \setminus \{\mathtt{RF}^i\ x\}, \langle \sigma[x := \sigma[\Sigma^i]x], \Sigma[i := \Sigma^i[x := \mathtt{udf}]]\rangle\rangle}\ \text{(O-RF)}$$

$$\frac{\mathtt{FN}^i \in R(G)}{\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle G \setminus \{\mathtt{FN}^i\}, \langle \sigma, \Sigma\rangle\rangle}\ \text{(O-FN)}$$

$$\frac{\varphi \in R(G) \qquad \sigma \vDash \varphi}{\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle G \setminus \{\varphi\}, \langle \sigma, \Sigma\rangle\rangle}\ \text{(O-GD)}$$

Fig. 4 Our operational semantics.

$$\sigma \vDash e_1 = e_2 \Leftrightarrow \langle\!| e_1 |\!\rangle_\sigma = \langle\!| e_2 |\!\rangle_\sigma \qquad \sigma \vDash e_1 \le e_2 \Leftrightarrow \langle\!| e_1 |\!\rangle_\sigma \le \langle\!| e_2 |\!\rangle_\sigma$$

$$\sigma \vDash \neg\varphi \Leftrightarrow \sigma \nvDash \varphi \qquad \sigma \vDash \varphi \supset \varphi' \Leftrightarrow \sigma \vDash \varphi \text{ implies } \sigma \vDash \varphi'$$

$$\sigma \vDash \forall r.\, \varphi(r) \Leftrightarrow \sigma \vDash \varphi(v) \text{ for any } v\ .$$

Fig. 5 Satisfiabilities of formulas.

variables and shared variables. The second element Σ represents buffers that temporarily buffer the effects of store operations to shared variables. We assume that the set of value contains a special constant value udf to represent uninitialized or invalidated buffers. We define the following three operations for substitution functions:

$$f[v' := val]v = \begin{cases} val & \text{if } v = v' \\ fv & \text{o.w.} \end{cases} \qquad \sigma[\Sigma^i]x = \begin{cases} \Sigma^i x & \text{if } \Sigma^i x \neq \mathtt{udf} \\ \sigma x & \text{o.w.} \end{cases}$$

$$\Sigma[i' := g]i = \begin{cases} g & \text{if } i = i' \\ \Sigma^i & \text{o.w.} \end{cases}$$

where we write Σi as Σ^i, f ranges over both σ and Σ^i, and g is a function from shared variables to values. Specifically, $f[v' := val]$ and $\sigma[\Sigma^i]$ represent updated values via substitutions, and $\Sigma[i' := g]$ represents updated substitutions.

Figure 4 shows the rules of the operational semantics where $\langle\!| e |\!\rangle_\sigma$ denotes the valuation of an expression e as follows:

$$\langle\!| val |\!\rangle_\sigma = val \qquad \langle\!| r |\!\rangle_\sigma = \sigma r \qquad \langle\!| x |\!\rangle_\sigma = \sigma x$$

$$\langle\!| e_1 + e_2 |\!\rangle_\sigma = \langle\!| e_1 |\!\rangle_\sigma + \langle\!| e_2 |\!\rangle_\sigma \qquad \ldots$$

and $\sigma \vDash \varphi$ means satisfiability of φ on σ in the standard manner as defined in **Fig. 5**.

A pair of a program graph and a state is called a *configuration*. Each rule is represented by one-step reduction between configurations $\langle G, st\rangle \xrightarrow{c} \langle G', st'\rangle$, which indicates that a command makes $\langle G, st\rangle$ to transit to $\langle G', st'\rangle$.

Specifically, O-MV evaluates e and updates σ. Rule O-LD evaluates x on Σ^i, if $\Sigma^i x$ is defined, and on σ otherwise, and updates σ. O-ST evaluates e and updates Σ^i (not σ); i.e., the rule indicates that the effect of the store operation is buffered in Σ^i. Rule O-RF denotes the reflection from a store buffer to shared memory. Rule O-FN does not change the state. The command FN is only used to represent ordering constraints among the other statements. Rule O-GD handles guard nodes φ by simply asserting that φ is satisfied under state σ. If σ is not satisfied, we say

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset \lceil \mathtt{MV}^i\ r\ e \rceil^U_W \supset guar \quad \models pre \supset [e/r]post}{\{pre, rely\}\ \mathtt{MV}^i\ r\ e\ \{guar, post\}}\ \text{(L-MV)}$$

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset \mathrm{Inv}\ U \supset \mathrm{Inv}\ W \supset guar \quad \models pre \supset post}{\{pre, rely\}\ \varnothing\ \{guar, post\}}\ \text{(L-EM)}$$

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset \lceil \mathtt{LD}^i\ r\ x \rceil^U_W \supset guar \quad \models pre \supset [x^i/r]post}{\{pre, rely\}\ \mathtt{LD}^i\ r\ x\ \{guar, post\}}\ \text{(L-LD)}$$

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset \lceil \mathtt{ST}^i\ x\ e \rceil^U_W \supset guar \quad \models pre \supset [e/x^i]post}{\{pre, rely\}\ \mathtt{ST}^i\ x\ e\ \{guar, post\}}\ \text{(L-ST)}$$

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset \lceil \mathtt{RF}^i\ x \rceil^U_W \supset guar \quad \models pre \supset ((\neg\, \mathtt{df}(x^i) \wedge post) \vee (\mathtt{df}(x^i) \wedge \bigvee \{(\overrightarrow{\mathtt{df}(x^j)} \wedge \neg\, \overrightarrow{\mathtt{df}(x^k)} \wedge [x^i/x, \overrightarrow{x^i/x^k}]post) \mid \overrightarrow{x^j} \cup \overrightarrow{x^k} = \mathrm{fv}(post) \cap (B(x) \setminus \{x^i\})\}))}{\{pre, rely\}\ \mathtt{RF}^i\ x\ \{guar, post\}}\ \text{(L-RF)}$$

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset post}{\{pre, rely\}\ \mathtt{FN}^i\ \{guar, post\}}\ \text{(L-FN)}$$

$$\frac{\models pre \perp rely \quad \models post \perp rely \quad \models pre \supset \varphi \supset post}{\{pre, rely\}\ \varphi\ \{guar, post\}}\ \text{(L-GD)}$$

$$\frac{\forall C \in L(G).\ \{pre_C, rely\}\ G \setminus \{C\}\ \{guar, \Phi_C\} \quad \{\Phi_C, rely\}\ C\ \{guar, post\}}{\{\bigwedge\{pre_C \mid C \in L(G)\}, rely\}\ G\ \{guar, post\}}\ \text{(L-LN)}$$

$$\frac{\{pre, rely\}\ G_0\ \{guar, \Phi\} \quad \{\Phi, rely\}\ G_1\ \{guar, post\}}{\{pre, rely\}\ G_0 \rightarrow G_1\ \{guar, post\}}\ \text{(L-SQ)}$$

$$\frac{\{pre_0, rely_0\}\ G_0\ \{guar_0, post_0\} \quad \{pre_1, rely_1\}\ G_1\ \{guar_1, post_1\} \quad \models rely \vee guar_0 \supset rely_1 \quad \models rely \vee guar_1 \supset rely_0 \quad \models guar_0 \vee guar_1 \supset guar}{\{pre_0 \wedge pre_1, rely\}\ G_0 \sqcup G_1\ \{guar, post_0 \wedge post_1\}}\ \text{(L-PR)}$$

$$\frac{\{pre_0, rely_0\}\ G\ \{guar_0, post_0\} \quad \models pre \supset pre_0 \quad \models rely \supset rely_0 \quad \models guar_0 \supset guar \quad \models post_0 \supset post}{\{pre, rely\}\ G\ \{guar, post\}}\ \text{(L-WK)}$$

Fig. 6 Our concurrent program logic.

that G gets *stuck*.

The operational semantics is nondeterministic because all rules have an assumption $C \in R(G)$, and $R(G)$ may consist of more than one element. This nondeterminacy in the choice of one of the root nodes enables the operational semantics to simulate various relaxed memory consistency models by choosing different translation approaches.

4. Concurrent Program Logic

In this section, we define our concurrent program logic. Our assertion language is defined as follows:

$$\Phi ::= E = E \mid E \leq E \mid \mathtt{df}(x^i) \mid \mathtt{df}(\underline{x^i}) \mid \neg\, \Phi \mid \Phi \supset \Phi \mid \forall v.\, \Phi$$

$$v ::= r \mid x \mid x^i \mid \underline{r} \mid \underline{x} \mid \underline{x^i}$$

where E represents a pseudo-expression denoting thread-local variables r, shared variables x, buffered variables x^i, next thread-local variables $\underline{r}$, next shared variables $\underline{x}$, next buffered variables $\underline{x^i}$, constant values *val*, arithmetic operations, and so on. The buffered variable x^i represents the value written to the shared variable x by the ST on a thread with identifier i, but not yet reflected. In addition, $\mathtt{df}(x^i)$ indicates that x^i is defined; i.e., there is a pending ST for x on thread i. We define $[e/x^i]\mathtt{df}(x^i)$ as $\top$. The next variable $\underline{v}$ represents the value of v on a state to which the current state transits under the operational semantics. We call v a *current* variable.

Figure 6 shows the judgment rules. They are defined following the styles of Stølen and Xu's proof systems [42], [47], [48]. We assume that rely/guarantee-conditions are reflexive and transitive. Each judgment has form $\{pre, rely\}\ G\ \{guar, post\}$ (where *pre* and *post* have no next variable), which states that, if the program graph G is computed under the pre-condition *pre* and rely-condition *rely* (which the other threads guarantee) according to the operational semantics of Section 3, then the guarantee-condition *guar* (on which the other threads rely) holds (for any possible nondeterministic computation), as usual in the conventional rely-guarantee systems. Furthermore, if the computation terminates and does not get stuck, then the post-condition *post* also holds. In the rest of paper, we denote $\models$ $\{pre, rely\}G\{guar, post\}$ if $\{pre, rely\}G\{guar, post\}$ can be derived from the judgment rules of Fig. 6.

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models E_0 = E_1 \Leftrightarrow [\![E_0]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = [\![E_1]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle}$$

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models E_0 \leq E_1 \Leftrightarrow [\![E_0]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} \leq [\![E_1]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle}$$

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \mathtt{df}(x^i) \Leftrightarrow \Sigma^i x \neq \mathtt{udf}$$

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \mathtt{df}(\underline{x^i}) \Leftrightarrow \Sigma'^i x \neq \mathtt{udf}$$

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \neg\, \Phi \Leftrightarrow \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \not\models \Phi$$

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \Phi \supset \Phi' \Leftrightarrow \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \Phi \text{ implies } \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \Phi'$$

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \forall v.\, \Phi(v) \Leftrightarrow \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \Phi(v') \text{ for any } v' \text{ where}$$

$$[\![val]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = val \qquad [\![r]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma r \qquad [\![x]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma x$$

$$[\![x^i]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma[\Sigma^i]x \qquad [\![\underline{x}]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma' x \qquad [\![\underline{x^i}]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma'[\Sigma'^i]x$$

$$[\![E_1 + E_2]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = [\![E_1]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} + [\![E_2]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} \qquad \ldots$$

Fig. 7 The interpretation of the assertion language.

More specifically, rule L-MV handles the substitution of thread-local variables with expressions. This is the same as in conventional rely-guarantee proof systems. $[e/v]$ represents a substitution of v with e. We define $\models \Phi$ as $\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \Phi$ for any $\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle$, where $\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \models \Phi$ is defined as an extension (for buffered variables) of that in a standard manner of rely-guarantee system as shown in **Fig. 7**. In the following, we often write $\langle\sigma,\Sigma\rangle \models \Phi$ when Φ has no next variable. The first and second assumptions mean that *pre* and *post* are *stable under rely*, respectively, that another thread guarantees where we state that Φ is *stable under* Ψ (written as $\Phi \perp \Psi$) as $\Phi(\overrightarrow{v}) \wedge \Psi(\overrightarrow{v}, \overrightarrow{\underline{v}}) \supset \Phi(\overrightarrow{\underline{v}})$. The third assumption means that *pre* must be a sufficient condition that implies *guar* under an invariant condition about variables before and after an execution of C (written as $\lceil C \rceil^U_W$) where U and W are finite sets of current non-buffered and buffered variables that occur in *guar*, respectively. A formula $\lceil \mathtt{MV}^i\ r\ e \rceil^U_W$ is defined as $\underline{r} = e \wedge \bigwedge \mathrm{Inv}\ U \setminus \{r\} \wedge \bigwedge \mathrm{Inv}\ W$, which means that the value of $\underline{r}$ becomes equal to the evaluation of e while values of variables in $U \setminus \{r\}$ and W are invariant where

$$\mathrm{Inv}\ U = \{\, \underline{u} = u \mid u \in U \,\}$$

$$\mathrm{Inv}\ W = \{\, \underline{w} = w \wedge (\mathtt{df}(\underline{w}) \supset \mathtt{df}(w)) \mid w \in W \,\}$$

Its formal definition is shown in **Fig. 8** where $B(x)$ is $\{\, x^j \mid 0 \leq$

$$\lceil \mathtt{MV}^i\ r\ e \rceil^U_W \equiv \underline{r} = e \wedge \bigwedge \mathrm{Inv}\ U \setminus \{r\} \wedge \bigwedge \mathrm{Inv}\ W$$
$$\lceil \mathtt{LD}^i\ r\ x \rceil^U_W \equiv \underline{r} = x^i \wedge \bigwedge \mathrm{Inv}\ U \setminus \{r\} \wedge \bigwedge \mathrm{Inv}\ W$$
$$\lceil \mathtt{ST}^i\ x\ e \rceil^U_W \equiv \underline{x^i} = e \wedge \bigwedge \mathrm{Inv}\ U \wedge \bigwedge \mathrm{Inv}\ W \setminus \{x^i\}$$
$$\lceil \mathtt{RF}^i\ x \rceil^U_W \equiv ((\mathtt{df}(x^i) \wedge \neg\, \mathtt{df}(\underline{x^i}) \wedge \underline{x} = x^i) \vee (\neg\, \mathtt{df}(x^i) \wedge \neg\, \mathtt{df}(\underline{x^i}) \wedge \underline{x} = x)) \wedge \bigwedge \mathrm{Inv}\ U \setminus \{x\} \wedge \bigwedge \mathrm{Inv}\ W \setminus B(x) \wedge \bigwedge \{\, (\mathtt{df}(x^j) \supset x^j = \underline{x^j}) \wedge (\mathtt{df}(\underline{x^j}) \supset \mathtt{df}(x^j)) \mid 0 \le j < N, i \ne j \,\}$$

Fig. 8 Invariants.

$$A(pre, rely) = \left\{ c \;\middle|\; \begin{array}{l} St(c,0) \vDash pre, \text{ and} \\ St(c,i), St(c,i+1) \vDash rely \text{ for any } Cfg(c,i) \xrightarrow{e} Cfg(c,i+1) \end{array} \right\}$$
$$C(guar, post) = \left\{ c \;\middle|\; \begin{array}{l} St(c,i), St(c,i+1) \vDash guar \text{ for any } Cfg(c,i) \xrightarrow{c} Cfg(c,i+1), \text{ and} \\ |c| < \omega \text{ and } Prg(c,|c|-1) = \varnothing \text{ imply } St(c,|c|-1) \vDash post \end{array} \right\}$$

Fig. 9 Computations under pre/rely-conditions satisfies guarantee/post-conditions.

$j < N\}$. The fourth assumption means that *pre* must be a sufficient condition that implies *post* with respect to the substitution.

Rule L-EM of Fig. 6 states that the empty graph does not affect anything. Rule L-LD handles substitutions of thread-local variables with shared variables. Please note that r is substituted with the auxiliary buffer variables x^i, instead of the shared variables x. Rule L-ST handles the substitution of shared variables with expressions. Please note that, as in L-LD, this rule considers the buffered variable x^i instead of the shared variable x. Rule L-RF handles reflections from a store buffer to shared memory. Threads js and ks have and do not have pending stores in their own buffers, respectively. Any sequence of x^ks are simultaneously replaced with x^i since x^ks are undefined; i.e., thread k observes the effect of the store of thread i if there is no pending store for x on thread k where $\overrightarrow{X}$ denotes multiple Xs and $\mathrm{fv}(\Phi)$ represents a set of free variables in Φ in the standard manner. Any sequence of x^js are not replaced with x^i; i.e., thread j observes a pending store in its own buffer. Although a pre-conditions appears to become huge by combinations of buffered variables, we can often construct a derivation in which there exists buffered variables x^is only in a pre-condition of $\mathtt{RF}^i\ x$ by using L-PR appropriately. Please refer to an example verification in Section 6. Rule L-FN handles `FN` and does nothing like L-EM, which means that `FN` is only used as a landmark in linking nodes which have dependency. Rule L-GD handles a guard φ. It asserts that *pre* and φ implies *post*. Rule L-WK is the so-called consequence rule. Rule L-SQ handles $G_0 \to G_1$, which is considered as a sequential composition of program graphs since all nodes in G_0 must be executed before any command of G_1 is executed. L-PR handles program graphs that have G_0 and G_1 where there is no edge between G_0 and G_1. This corresponds to a rule of parallel compositions of programs in a standard rely-guarantee system. The third assumption means that G_1's rely condition $rely_1$ must be guaranteed by the global rely-condition *rely* or G_0's guarantee-condition $guar_0$. The fourth assumption is similar. The fifth assumption means that *guar* must be guaranteed by either $guar_0$ or $guar_1$. L-LN handles the nondeterministic choice of one of the leaf nodes of G. This rule focuses on *every* leaf node (denoted as C) in G one by one, and checks exhaustively whether $\{pre, rely\}\ G \setminus \{C\}\ \{guar, \Phi\}$ and $\{\Phi, rely\}\ C\ \{guar, post\}$ hold. It can be considered that this rule handles all the *linearizations* [18] of commands.

Validity

We define *computations* of program graphs, and give *validity* for judgments. We define the set of computations $Cmp(G)$ of G as a finite or infinite sequence c of configurations where adjacent configurations are related by $\xrightarrow{c}$ defined in Section 3 or $\xrightarrow{e}$, which is defined by $\langle G, st\rangle \xrightarrow{e} \langle G, st'\rangle$ for any st and st'. $Cfg(c,i)$, $Prg(c,i)$, and $St(c,i)$ as the i-th configuration, program graph, and state of c, respectively. The program graph $Prg(c,0)$ is G. $|c|$ denotes its length, which is the smallest limit ordinal ω if c is an infinite sequence. Let c' be a computation that satisfies $St(c', |c'| - 1) = St(c, 0)$. We define $c' \cdot c$ as a concatenation of c' and c. We define $\vDash \{pre, rely\}\ G\ \{guar, post\}$ as $Cmp(G) \cap A(pre, rely) \subseteq C(guar, post)$, which means that any computation under pre/rely-conditions satisfies guarantee/post-conditions as shown in **Fig. 9**. This kind of validity is called *partial correctness* [46].

Program Verification via Program Graphs

The verification of a program is carried out by deriving judgments with the rules defined in Fig. 6 for its program graphs that consist of *finitely* unfolded loops. Careful readers may notice that the logic described in this section handles a single program graph, but the translation in Section 2.3 may generate an infinite family of program graphs. Formally speaking, an infinite family of derivations for the generated program graphs correspond to the proof of the original program. That is, we have to construct a derivation for each generated program graph. Although it seems difficult to construct an infinite family of derivations, it is not always impossible. For example, we show an inductive construction of derivations in Section 6. Careful readers may also wonder why we unfold loops unlike the conventional program logics [19], [26]. The reason is that our work explicitly handles dependencies/independencies across loop iterations, and unfolding loops is the most straightforward way to achieve this. To the best of our knowledge, there is no concurrent program logic which handles dependencies/independencies across loop iterations. Careful readers may also wonder if a family of program graphs contains an *inadmissible* behavior of a program. For example, the program in Fig. 1 generates a program graph which has $\neg\, r_0 = 0$ as a root node, but the condition is never satisfied because r_0 is initialized to 0. Therefore, the program graph can be considered to denote an inadmissible behavior of the program. However, this is not a problem because the program graph gets stuck according to the operational semantics 3, and a derivation of a judgment for a program graph that gets stuck does not imply that the behavior of the program is valid by the definition of validity of judgments as described. In addition, careful readers may also wonder if a program graph cannot capture a behavior of a program that is represented by an *infinitely* unfolded loop. However, that is not a problem because a non-terminating program is out of the scope of this work that considers the so-called *partial correctness* [46].

5. Soundness and Relative Completeness

In this section, we show our concurrent program logic is sound to the operational semantics defined in Fig. 4. We also show soundness and *relative completeness* [46] between a slight extension of our concurrent program logic and its corresponding operational semantics with respect to *histories* of computations introduced in Refs. [47], [48].

Most of the soundness and relative completeness proofs in this paper follow those in Xu's PhD thesis and its journal version [47], [48]. Differences in some definitions, propositions, lemmas, and theorems are derived from that our semantics and logic handle *not* programs themselves *but* program graphs. Interesting differences are that our proof for completeness is simpler than the proof in Refs. [47], [48], and our completeness theorem claims a conclusion including *L-SQ and L-PR-freeness*.

To handle arbitrary program graphs, we added the L-LN rule in Section 4, which is unnecessary to handle programs only. The rule is so troublesome to require a huge number of judgments as assumptions and make constructions of derivations tedious. Instead of it, the L-LN rule is so powerful to make a proof for completeness theorem drastically simple, while the proof for the completeness theorem in Refs. [47], [48] appears to be complicated since constructing judgments as assumptions of the L-PR rule (from a valid judgment) is hard. Actually, L-SQ and L-PR-freeness in Theorem 2 shows that L-LN contains L-SQ and L-PR in provability.

In order to prove the relative completeness, we slightly extend the operational semantics of Section 3 and the concurrent program logic of Section 4 with a notion of a *history*, which informally holds a sequence of pairs of executed assignments and states, following the style of Refs. [47], [48]. History variables allow us to take snapshots of computations at arbitrary time, and formally enriches assertion languages. As high expressibility of assertion languages is a point of proving completeness of program logics such as Hoare logic [46], introducing history variables is a typical method to enrich assertion languages in concurrent program logics. Formally, we assume that the set of values contain histories, which consist of sequences of the form $\langle C_0, \overrightarrow{val_0}\rangle \cdots \langle C_{n-1}, \overrightarrow{val_{n-1}}\rangle$ where C_j is a command or a symbol *Env* (meaning an environment), and introduce special variables (called *history variables*). Given a program graph, let $\overrightarrow{v}$ be the sequence of its current variables and their buffered variables. We extend the definition of commands of assignments `MV` $r\ e$, `LD` $r\ x$, `ST` $x\ e$, and `RF` x so that they can have additional assignments of histories to history variables. Formally, C is converted to $\texttt{atomic}\{C, h_0 := h_0 \cdot \langle C, \overrightarrow{v}\rangle, \ldots, h_{n-1} := h_{n-1} \cdot \langle C, \overrightarrow{v}\rangle\}$ where $h_0, \ldots, h_{n-1}$ are history variables, and $\texttt{atomic}\{C, C'\}$ means that C and C' are *atomically* (without being disturbed by the other threads) executed.

We also extend the operational semantics so that history variables are updated appropriately. Please note that the effect of assignments to h are not saved at Σ but immediately reflected to σ, i.e., history variables are shared and *unbuffered*. We also extend the concurrent program logic so that, on each judgment rule for an assignment C, h in a post-condition is updated to $h \cdot \langle C, \overrightarrow{v}\rangle$, and $\underline{h} = h \cdot \langle C, \overrightarrow{v}\rangle$ is added to $\lceil C\rceil^U_W$ as a conjunction. We treat h as a non-buffered variable, i.e., h can belong to U in $\lceil C\rceil^U_W$. Furthermore, we add the so-called *auxiliary variables rule* [41], [42], [48] as follows:

$$\frac{\begin{array}{l}\{pre \wedge pre_0, rely \wedge rely_0\}\ G_0\ \{guar, post\} \\ \vDash \exists \underline{\overrightarrow{v_0}}.\, rely_0((\overrightarrow{v}, \overrightarrow{v_0}), (\overrightarrow{v}, \underline{\overrightarrow{v_0}})) \quad \vDash \exists \overrightarrow{v_0}.\, pre_0(\overrightarrow{v}, \overrightarrow{v_0}) \\ \overrightarrow{v_0} \cap (\mathrm{fv}(pre) \cup \mathrm{fv}(rely) \cup \mathrm{fv}(guar) \cup \mathrm{fv}(post)) = \varnothing\end{array}}{\{pre, rely\}\ (G_0)_{\overrightarrow{v_0}}\ \{guar, post\}}\ \text{(L-AX)}$$

where $(G)_{\overrightarrow{v_0}}$ is defined as the program graph that coincides with G except that $C \in N(G)$ is removed if

- C is an assignment whose left value belongs to $\overrightarrow{v_0}$,
- no variable in $\overrightarrow{v_0}$ occurs in assignments whose left values do not belong to $\overrightarrow{v_0}$, and
- no variable in $\overrightarrow{v_0}$ freely occurs in guards.

Let c be $\langle G_0, \langle \sigma_0, \Sigma_0\rangle\rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{i-1}} \langle G_i, \langle \sigma_i, \Sigma_i\rangle\rangle \xrightarrow{\delta_i} \cdots$ where δ_i is c or e for any $0 \le i$. We write $tr(c, i)$ as δ_i. Given $c \in Cmp(G_0 \sqcup G_1)$, $c_0 \in Cmp(G_0)$, and $c_1 \in Cmp(G_1)$, a ternary relation $c = c_0 \parallel c_1$ is defined if $|c| = |c_0| = |c_1|$ and

(1) $St(c, i) = St(c_0, i) = St(c_1, i)$,
(2) $tr(c, i) = \mathrm{c}$ implies either of $tr(c_0, i) = \mathrm{c}$ or $tr(c_1, i) = \mathrm{c}$ holds,
(3) $tr(c, i) = \mathrm{e}$ implies $tr(c_0, i) = \mathrm{e}$ and $tr(c_1, i) = \mathrm{e}$ hold, and
(4) $Prg(c, i) = Prg(c_0, i) \sqcup Prg(c_1, i)$

for $0 \le i < |c|$. We write $\mathrm{prefix}(c, i)$ and $\mathrm{postfix}(c, i)$ as the prefix of c with length $i + 1$ and the sequence that is derived from c by removing $\mathrm{prefix}(c, i - 1)$, respectively.

Proposition 1. *$Cmp(G_0 \sqcup G_1) = \{\, c_0 \parallel c_1 \mid c_0 \in Cmp(G_0), c_1 \in Cmp(G_1)\,\}$.*

Lemma 1. *Assume $\vdash \{pre_0 \wedge pre_1, rely\}\ G_0 \sqcup G_1\ \{guar, post_0 \wedge post_1\}$ by L-PR, $Cmp(G_0) \cap A(pre_0, rely_0) \subseteq C(guar_0, post_0)$, $Cmp(G_1) \cap A(pre_1, rely_1) \subseteq C(guar_1, post_1)$, $\vDash rely \vee guar_0 \supset rely_1$, $\vDash rely \vee guar_1 \supset rely_0$, $\vDash guar_0 \vee guar_1 \supset guar$, and $c \in Cmp(G_0 \sqcup G_1) \cap A(pre_0 \wedge pre_1, rely)$. In addition, we take $c_0 \in Cmp(G_0)$ and $c_1 \in Cmp(G_1)$ such that $c = c_0 \parallel c_1$ by Prop. 1.*

(1) $St(c, i), St(c, i + 1) \vDash guar_0$ and $St(c, i), St(c, i + 1) \vDash guar_1$ hold for any $Cfg(c_0, i) \xrightarrow{\mathrm{c}} Cfg(c_0, i + 1)$ and $Cfg(c_1, i) \xrightarrow{\mathrm{c}} Cfg(c_1, i + 1)$, respectively.

(2) $St(c, i), St(c, i + 1) \vDash rely \vee guar_1$ and $St(c, i), St(c, i + 1) \vDash rely \vee guar_0$ hold for any $Cfg(c_0, i) \xrightarrow{\mathrm{e}} Cfg(c_0, i + 1)$ and $Cfg(c_1, i) \xrightarrow{\mathrm{e}} Cfg(c_1, i + 1)$, respectively.

(3) $St(c, i), St(c, i + 1) \vDash guar$ for any $Cfg(c, i) \xrightarrow{\mathrm{c}} Cfg(c, i + 1)$ holds.

(4) Assume $|c| < \omega$ and $Prg(c, |c| - 1) = \varnothing$. Then, $St(c, |c| - 1) \vDash post_0 \wedge post_1$ holds.

Proof. 1. Let us consider the former case. Without loss of generality, we can assume that $St(c, i), St(c, i + 1) \nvDash guar_0$ where $St(c, j), St(c, j + 1) \vDash guar_0$ and $St(c, j), St(c, j + 1) \vDash guar_1$ for any $0 \le j < i$.

By the definition, there exists $Cfg(c, k) \xrightarrow{\mathrm{e}} Cfg(c, k + 1)$ or $Cfg(c_1, k) \xrightarrow{\mathrm{c}} Cfg(c_1, k + 1)$ corresponding to $Cfg(c_0, k) \xrightarrow{\mathrm{e}} Cfg(c_0, k + 1)$ for any $0 \le k \le i$. Therefore, $St(c, k), St(c, k + 1) \vDash rely \vee guar_1$ holds. By $\vDash rely \vee guar_1 \supset rely_0$, $\mathrm{prefix}(c_0, i + 1) \in A(pre_0, rely_0)$ holds. Since $Cmp(G_0) \cap A(pre_0, rely_0) \subseteq C(guar_0, post_0)$ holds, in particular, $St(c, i), St(c, i + 1) \vDash guar_0$ holds. This contradicts $St(c, i), St(c, i+1) \nvDash guar_0$. The latter case is similar.

2. Immediate from the definition of $c = c_0 \parallel c_1$ and 1.
3. Immediate from 1 and $\vDash guar_0 \vee guar_1 \supset guar$.
4. By 2, $\vDash rely \vee guar_0 \supset rely_1$, and $\vDash rely \vee guar_1 \supset rely_0$, $c_0 \in A(pre_0, rely_0)$ and $c_1 \in A(pre_1, rely_1)$ hold.

By $Cmp(G_0) \cap A(pre_0, rely_0) \subseteq C(guar_0, post_0)$ and $Cmp(G_1) \cap A(pre_1, rely_1) \subseteq C(guar_1, post_1)$, $St(c, |c|) \vDash post_0$ and $St(c, |c| - 1) \vDash post_1$ hold. Therefore, $St(c, |c|-1) \vDash post_0 \wedge post_1$ holds. □

Theorem 1. *The extended concurrent program logic (and the original one) is sound. That is,* $\vdash \{pre, rely\}\ G\ \{guar, post\}$ *implies* $\vDash \{pre, rely\}\ G\ \{guar, post\}$.

Proof. By induction on derivation and case analysis of the last inference rule.

First, assume L-MV. Let $c \in Cmp(\{\mathtt{MV}^i\ r\ e\}) \cap A(pre, rely)$. By O-MV, there exist $\sigma_0, \Sigma_0, \ldots$ such that $\langle \sigma_{n+1}, \Sigma_{n+1} \rangle = \langle \sigma_n[r := \langle\!| e |\!\rangle_{\sigma_n}], \Sigma_n \rangle$,

$$c = \langle \{\mathtt{MV}^i\ r\ e\}, \langle \sigma_0, \Sigma_0 \rangle \rangle \xrightarrow{\mathrm{e}}{}^{*} \langle \{\mathtt{MV}^i\ r\ e\}, \langle \sigma_n, \Sigma_n \rangle \rangle \xrightarrow{\mathrm{c}} \langle \varnothing, \langle \sigma_{n+1}, \Sigma_{n+1} \rangle \rangle \xrightarrow{\mathrm{e}} \cdots ,$$

$\langle \sigma_0, \Sigma_0 \rangle \vDash pre$, and $\langle \sigma_j, \Sigma_j \rangle, \langle \sigma_{j+1}, \Sigma_{j+1} \rangle \vDash rely$ for any $0 \le j < n$.

By $\vDash pre \perp rely$, $\langle \sigma_n, \Sigma_n \rangle \vDash pre$. By the definition, $\langle \sigma_n, \Sigma_n \rangle, \langle \sigma_n[r := \langle\!| e |\!\rangle_{\sigma_n}], \Sigma_n \rangle \vDash \lceil \mathtt{MV}^i\ r\ e \rceil^U_W$. By $\vDash pre \supset \lceil \mathtt{MV}^i\ r\ e \rceil^U_W \supset guar$, $\langle \sigma_n, \Sigma_n \rangle, \langle \sigma_n[r := \langle\!| e |\!\rangle_{\sigma_n}], \Sigma_n \rangle \vDash guar$, that is, $\langle \sigma_n, \Sigma_n \rangle, \langle \sigma_{n+1}, \Sigma_{n+1} \rangle \vDash guar$.

In addition, assume $|c| < \omega$. By $\vDash pre \supset [e/r]post$, $\langle \sigma_n, \Sigma_n \rangle \vDash [e/r]post$. By the definition, $\langle \sigma_n[r := \langle\!| e |\!\rangle_{\sigma_n}], \Sigma_n \rangle \vDash post$, that is, $\langle \sigma_{n+1}, \Sigma_{n+1} \rangle \vDash post$. By $\vDash post \perp rely$, $\langle \sigma_{|c|-1}, \Sigma_{|c|-1} \rangle \vDash post$.

The case that G is $\{\,\mathtt{atomic}\{C, h_0 := h_0 \cdot \langle C, \overrightarrow{v} \rangle, \ldots, h_{n-1} := h_{n-1} \cdot \langle C, \overrightarrow{v} \rangle\}\}$ is similar where C is $\mathtt{MV}^i\ r\ e$.

Cases of L-EM, L-LD, L-ST, L-RF, and L-FN are similar.

Second, assume L-WK. Let $c \in Cmp(G) \cap A(pre, rely)$. By $\vDash pre \supset pre_0$ and $\vDash rely \supset rely_0$, $c \in Cmp(G) \cap A(pre_0, rely_0)$ holds. By induction hypothesis, $c \in C(guar_0, post_0)$ holds. By $\vDash guar_0 \supset guar$ and $\vDash post_0 \supset post$, $c \in C(guar, post)$ holds.

Third, assume L-GD. Let $c \in Cmp(\{\varphi^i\}) \cap A(pre, rely)$. There exist $\sigma_0, \Sigma_0, \ldots$ such that $\langle \sigma_{n+1}, \Sigma_{n+1} \rangle = \langle \sigma_n, \Sigma_n \rangle$,

$$c = \langle \varphi^i, \langle \sigma_0, \Sigma_0 \rangle \rangle \xrightarrow{\mathrm{e}}{}^{*} \langle \varphi^i, \langle \sigma_n, \Sigma_n \rangle \rangle \xrightarrow{\mathrm{c}} \langle \varnothing, \langle \sigma_{n+1}, \Sigma_{n+1} \rangle \rangle \xrightarrow{\mathrm{e}} \cdots ,$$

$\sigma_n \vDash \varphi$, $\langle \sigma_0, \Sigma_0 \rangle \vDash pre$, and $\langle \sigma_j, \Sigma_j \rangle, \langle \sigma_{j+1}, \Sigma_{j+1} \rangle \vDash rely$ for any $0 \le j < n$.

By the reflexivity of $guar$, $\langle \sigma_n, \Sigma_n \rangle, \langle \sigma_n, \Sigma_n \rangle \vDash guar$ holds, that is, $\langle \sigma_n, \Sigma_n \rangle, \langle \sigma_{n+1}, \Sigma_{n+1} \rangle \vDash guar$ holds.

In addition, assume $|c| < \omega$. By $\vDash pre \perp rely$, $\langle \sigma_n, \Sigma_n \rangle \vDash pre$. By $\vDash pre \supset \varphi \supset post$ and $\sigma_n \vDash \varphi$, $\langle \sigma_n, \Sigma_n \rangle \vDash post$ holds. That is, $\langle \sigma_{n+1}, \Sigma_{n+1} \rangle \vDash post$ holds. By $\vDash post \perp rely$, $\langle \sigma_{|c|-1}, \Sigma_{|c|-1} \rangle \vDash post$ holds.

The case that G is $\{\,\mathtt{atomic}\{C, h_0 := h_0 \cdot \langle C, \overrightarrow{v} \rangle, \ldots, h_{n-1} := h_{n-1} \cdot \langle C, \overrightarrow{v} \rangle\}\}$ is similar where C is φ^i.

Fourth, assume L-SQ. Let $c \in Cmp(G_0 \to G_1) \cap A(pre, rely)$. There exist $st_0, \delta_0, \ldots$ such that

$$c = \langle G_0 \to G_1, st_0 \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle G_1, st_n \rangle \xrightarrow{\delta_n} \cdots ,$$

$st_0 \vDash pre$, and $st_j, st_{j+1} \vDash rely$ for any $0 \le j < n$.

Let c' and c'' be $\langle G_0, st_0 \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle \varnothing, st_n \rangle$ and postfix(c, n), respectively. Obviously, $c' \in Cmp(G_0) \cap A(pre, rely)$ holds. By induction hypothesis, $c' \in C(guar, \Phi)$ holds. By the definition, $\langle \sigma_n, \Sigma_n \rangle \vDash \Phi$ holds. Therefore, $c'' \in Cmp(G_1) \cap A(\Phi, rely)$ holds. By induction hypothesis, $c'' \in C(guar, post)$ holds. Therefore, $c \in C(guar, post)$ holds.

Fifth, assume L-PR. By Lem. 1.3 and 1.4.

Sixth, assume L-LN. Let $c \in Cmp(G) \cap A(pre, rely)$. There exist $st_0, \delta_0, \ldots$ such that

$$c = \langle G, st_0 \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle \{C\}, st_n \rangle \xrightarrow{\delta_n} \cdots ,$$

$st_0 \vDash \bigwedge \{ pre_C \mid C \in L(G) \}$, and $st_j, st_{j+1} \vDash rely$ for any $0 \le j < n$.

Let c' and c'' be $\langle G \setminus \{C\}, st_0 \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle \varnothing, st_n \rangle$ and postfix(c, n), respectively. Obviously, $c' \in Cmp(G \setminus \{C\}) \cap A(pre_C, rely)$ holds. By induction hypothesis, $c' \in C(guar, \Phi_C)$ holds. By the definition, $\langle \sigma_n, \Sigma_n \rangle \vDash \Phi_C$ holds. Therefore, $c'' \in Cmp(C) \cap A(\Phi_C, rely)$ holds. By induction hypothesis, $c'' \in C(guar, post)$ holds. Therefore, $c \in C(guar, post)$ holds.

Finally, assume L-AX. Let $c \in Cmp(G) \cap A(pre, rely)$. There exist $\sigma_0, \Sigma_0, \delta_0, \ldots$ such that

$$c = \langle (G)_{\overrightarrow{v_0}}, \langle \sigma_0, \Sigma_0 \rangle \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle G_n, \langle \sigma_n, \Sigma_n \rangle \rangle \xrightarrow{\delta_n} \cdots ,$$

$\langle \sigma_0, \Sigma_0 \rangle \vDash pre$, and $\langle \sigma_j, \Sigma_j \rangle, \langle \sigma_{j+1}, \Sigma_{j+1} \rangle \vDash rely$ for any $0 \le j < n$. Since $\vDash \exists \overrightarrow{v_0}.\, pre_0(\overrightarrow{v}, \overrightarrow{v_0})$, $\vDash \exists \overrightarrow{v_0}.\, rely_0((\overrightarrow{v}, \overrightarrow{v_0}), (\underline{\overrightarrow{v}}, \underline{\overrightarrow{v_0}}))$, and $\overrightarrow{v_0} \cap (\mathrm{fv}(pre) \cup \mathrm{fv}(rely) \cup \mathrm{fv}(guar) \cup \mathrm{fv}(post)) = \varnothing$, there exist $G'_0, \sigma'_0, \Sigma'_0, \ldots$ such that

$$c' = \langle G'_0, \langle \sigma'_0, \Sigma'_0 \rangle \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle G'_n, \langle \sigma'_n, \Sigma'_n \rangle \rangle \xrightarrow{\delta_n} \cdots ,$$

and $G'_0 = G$, $(G'_n)_{\overrightarrow{v_0}} = G_n$, $\sigma'_j v_1 = \sigma_j v_1$, $\Sigma'_j v_1 = \Sigma_j v_1$, $\langle \sigma'_0, \Sigma'_0 \rangle \vDash pre \wedge pre_0$, and $\langle \sigma'_j, \Sigma'_j \rangle, \langle \sigma'_{j+1}, \Sigma'_{j+1} \rangle \vDash rely \wedge rely_0$ for any $v_1 \notin \overrightarrow{v_0}$ and $0 \le j < n$. Therefore, $c' \in Cmp(G) \cap A(pre \wedge pre_0, rely \wedge rely_0)$ holds. By induction hypothesis, $c' \in C(guar, post)$ holds. Therefore, $c \in C(guar, post)$ holds. □

Next, we show relative completeness. We define G^h, which has an additional assignment to h for each assignment. We also define $(pre(\overrightarrow{v})^h)(\overrightarrow{v}, h) = pre(\overrightarrow{v}) \wedge h = \varepsilon$, and $(rely(\overrightarrow{v}, \underline{\overrightarrow{v}})^h)(\overrightarrow{v}, h, \underline{\overrightarrow{v}}, \underline{h}) = (rely(\overrightarrow{v}, \underline{\overrightarrow{v}}) \wedge h = \underline{h} \cdot (Env, \overrightarrow{v})) \vee (\overrightarrow{v} = \underline{\overrightarrow{v}} \wedge h = \underline{h})$ where ε denotes the sequence with length 0.

We define that Φ characterizes G_0, G_n, pre, and $rely$ if $st_n \vDash \Phi$ holds if and only if $\langle G_0, st_0 \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle G_n, st_n \rangle \in A(pre, rely)$ holds for some st_j and δ_j $(0 \le j < n)$. We say an assertion language is *expressive* if, for any G_0, G_n, pre, and $rely$, there exists Φ that characterizes ${G_0}^h$, ${G_n}^h$, pre^h, and $rely^h$.

Proposition 2. *Assume that h does not appears in G, pre, rely, guar, and post. Then,* $\vDash \{pre, rely\}\ G\ \{guar, post\}$ *if and only if* $\vDash \{pre^h, rely^h\}\ G^h\ \{guar, post\}$.

For any pre, $rely$, $post$, we define

$$pre^{rely}(\overrightarrow{v}) \equiv \exists \overrightarrow{v_0}.\, pre(\overrightarrow{v_0}) \wedge rely(\overrightarrow{v_0}, \overrightarrow{v})$$
$$post_{rely}(\overrightarrow{v}) \equiv post(\overrightarrow{v}) \wedge \forall \overrightarrow{v_0}.\, rely(\overrightarrow{v}, \overrightarrow{v_0}) \supset post(\overrightarrow{v_0}) .$$

Lemma 2. *(1)* $\vDash pre \supset pre^{rely}$.
(2) $\vDash post_{rely} \supset post$.
(3) $\vDash pre^{rely} \perp rely$.
(4) $\vDash post_{rely} \perp rely$
(5) $\vDash \{pre, rely\}\ G\ \{guar, post\}$ *iff* $\vDash \{pre^{rely}, rely\}\ G\ \{guar, post\}$.

(6) $\vDash \{pre, rely\}G\{guar, post\}$ *iff* $\vDash \{pre^{rely}, rely\}G\{guar, post_{rely}\}$.

Proof. 1. By reflexivity of *rely*.
2. By the definition of $post_{rely}$.
3. By transitivity of *rely*.
4. By the definition of $post_{rely}$.
5. By 1, 2, and L-WK, the if-part holds. Let us show the only-if-part. Let $c \in Cmp(G) \cap A(pre^{rely}, rely)$. Since $St(c, 0) \vDash pre^{rely}$, there exists st such that $c' \cdot c \in Cmp(G) \cap A(pre, rely)$ where c' is $\langle G, st\rangle \xrightarrow{e} \langle G, St(c, 0)\rangle$. By the assumption, $c' \cdot c \in C(guar, post)$. Therefore, $c \in C(guar, post)$.
6. In addition to the assumptions of the proof of 5, assume $|c| < \omega$. Let c'' be $\langle \varnothing, St(c' \cdot c, |c' \cdot c| - 1)\rangle \xrightarrow{e} \langle \varnothing, st''\rangle$ where $St(c' \cdot c, |c' \cdot c| - 1), st'' \vDash rely$. Therefore, $St(c \cdot c'', |c \cdot c''| - 1) \vDash post$. By the definition of $post_{rely}$, $c \in C(guar, post_{rely})$. □

Theorem 2. *The extended concurrent program logic is complete. Assume that the assertion language is expressive, and G is terminating and does not get stuck. Then,* $\vDash \{pre, rely\}\ G\ \{guar, post\}$ *implies that there exists an L-SQ and L-PR-free derivation of* $\vdash \{pre, rely\}\ G\ \{guar, post\}$.

Proof. By induction on cardinality of G.

First, assume $G = \{\mathtt{MV}^i\, re\}$. Let $\langle \sigma, \Sigma\rangle, st' \vDash pre^{rely} \wedge \lceil \mathtt{MV}^i\, re \rceil^{\mathrm{fv}(e)}_{\varnothing}$. Let c be $\langle G, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle \varnothing, \langle \sigma[r := \langle\!| e |\!\rangle_\sigma], \Sigma\rangle\rangle \in A(pre^{rely}, rely)$. By $\langle \sigma, \Sigma\rangle, st' \vDash \lceil \mathtt{MV}^i\ r\ e \rceil^{\mathrm{fv}(e)}_{\varnothing}$ and reflexivity of *rely*, we can define $c' \in A(pre^{rely}, rely)$ as $c \cdot \langle \varnothing, \langle \sigma[r := \langle\!| e |\!\rangle_\sigma], \Sigma\rangle\rangle \xrightarrow{e} \langle \varnothing, st'\rangle$. By Lem. 2, $c' \in C(guar, post_{rely})$ holds. Therefore, $\langle \sigma, \Sigma\rangle, st' \vDash guar$ holds.

In addition, by $c \in A(pre^{rely}, rely) \subseteq C(guar, post_{rely})$, $\langle \sigma[r := \langle\!| e |\!\rangle_\sigma], \Sigma\rangle \vDash post_{rely}$ holds. By the definition, $\langle \sigma, \Sigma\rangle \vDash [e/r](post_{rely})$ holds.

Therefore, $\vDash \{pre, rely\}\ G\ \{guar, post\}$ holds by L-MV, Lem. 2, and L-WK.

The case that G is $\{\,\mathtt{atomic}\{C, h_0 := h_0 \cdot \langle C, \overrightarrow{v}\rangle, \ldots, h_{n-1} := h_{n-1} \cdot \langle C, \overrightarrow{v}\rangle\}\,\}$ is similar where C is $\mathtt{MV}^i\ r\ e$.

Cases that G is $\varnothing$ or a program graph consisting of one node are similar.

Next, assume $G = \{\varphi^i\}$. Let $\langle \sigma, \Sigma\rangle, st' \vDash pre^{rely}$. Let c be $\langle \varphi^i, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{c} \langle \varnothing, \langle \sigma, \Sigma\rangle\rangle \in A(pre^{rely}, rely)$. By the definition, $\sigma \vDash \varphi$ holds. By reflexivity of *rely*, we can define $c' \in A(pre^{rely}, rely)$ as $c \cdot \langle \varnothing, \langle \sigma, \Sigma\rangle\rangle \xrightarrow{e} \langle \varnothing, st'\rangle$. By Lem. 2, $c' \in C(guar, post_{rely})$ holds. Therefore, $\langle \sigma, \Sigma\rangle, st' \vDash guar$ holds.

In addition, by $c \in A(pre^{rely}, rely) \subseteq C(guar, post_{rely})$, $\langle \sigma, \Sigma\rangle \vDash post_{rely}$ holds.

Therefore, $\vDash \{pre, rely\}\ G\ \{guar, post\}$ holds by L-GD, Lem. 2, and L-WK.

The case that G is $\{\,\mathtt{atomic}\{C, h_0 := h_0 \cdot \langle C, \overrightarrow{v}\rangle, \ldots, h_{n-1} := h_{n-1} \cdot \langle C, \overrightarrow{v}\rangle\}\,\}$ is similar where C is φ^i.

Finally, assume the other case. Let $C \in L(G)$. Let h be a history variable that does not appear G, *pre*, *rely*, *guar*, and *post*. By Prop. 2, $\vDash \{pre^h, rely^h\}\ G^h\ \{guar, post\}$ holds. Since the assertion language is expressive, there exists Φ_C that characterizes G^h, $\{C\}^h$, pre^h, and $rely^h$. Let $c \in Cmp(G^h \setminus \{C\}^h) \cap A(pre^h, rely^h)$ be $\langle G^h \setminus \{C\}^h, st_0\rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle \varnothing, st_n\rangle$. We define $c' \in Cmp(G^h) \cap A(pre^h, rely^h)$ as $\langle G^h, st_0\rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle \{C\}^h, st_n\rangle$. Since Φ_C characterizes G^h, $\{C\}^h$, pre^h, and $rely^h$, $st_n \vDash \Phi_C$ holds. Therefore, $c \in C(guar, \Phi_C)$ holds.

Also, let $c'' \in Cmp(\{C\}^h) \cap A(\Phi_C, rely^h)$. By the definition, $c \cdot c'' \in Cmp(G^h) \cap A(pre^h, rely^h)$ holds. Therefore, $c \cdot c'' \in C(guar, post)$ holds. By the definition, $c'' \in C(guar, post)$ holds.

We derive $\vdash \{pre^h, rely^h\}\ G^h \setminus \{C\}^h\ \{guar, \Phi_C\}$ and $\vdash \{\Phi_C, rely^h\}\ \{C\}^h\ \{guar, post\}$ from induction hypotheses. Since we take $C \in L(G)$ arbitrarily, $\vdash \{pre^h, rely^h\}\ G^h\ \{guar, post\}$ holds by L-LN. By L-AX, $\vdash \{pre, rely\}\ G\ \{guar, post\}$ holds. □

6. Example Derivations

In this section, we give example derivations in the logic in Section 4.

First, we give a simple derivation for a trivial program in **Fig. 10** to show that detecting an invariant is a point.

If a precondition is x is non-negative and the buffers are empty, then r_2 is non-negative when the program finishes, since x is increasing by the writer thread, where we assume that $0 \le r_1$, for simplicity. We can easily construct a derivation that ensures this. The program graph G_0 of the first (writer) thread of the program under PSO is shown in **Fig. 11** where a rectangle means a loop iteration. A judgment is $\{I, rely^0\}\ G'_0\ \{guar^0, I\}$ where G'_0 is *any* subgraph of G_0, and

$$I \equiv 0 \le x \wedge 0 \le x^0 \wedge 0 \le r_1$$
$$rely^0 \equiv guar^1 \equiv \mathrm{Inv}\{x, r_0, r_1\} \wedge \mathrm{Inv}\{x^0\}$$
$$rely^1 \equiv guar^0 \equiv \mathtt{df}(\underline{x^1}) \supset \mathtt{df}(x^1) \wedge 0 \le x \wedge \mathrm{Inv}\{r_2\}\ .$$

Since a judgment $\{0 \le x^1, rely^1\}\ G_1\ \{guar^1, 0 \le r_2\}$ is derived where G_1 is a program graph of the reader thread of the program, $\{I \wedge 0 \le x^1, rely^0 \wedge rely^1\}\, G_0 \sqcup G_1\, \{guar^0 \vee guar^1, 0 \le r_2\}$ is derivable. Thus, to detect an invariant I is a key point in constructing a derivation.

At last, we demonstrate verification of the program introduced in Section 1 by using the concurrent program logic in Section 4. Let us prove that $r_2 \le r_3 + 1$ holds when the program terminates with an appropriate pre-condition under TSO.

We denote a family of program graphs generated by the TSO translation (in Section 2) as $D_n \sqcup G_1$ where D_n is derived by unfolding `WL` loop on the left-side thread at n times, and G_1 is $\mathtt{LD}^1\ r_2\ x \to \mathtt{LD}^1\ r_3\ y \to \mathtt{ST}^1\ z\ 1 \to \mathtt{RF}^1\ z$ on the right-side thread. For example, **Fig. 12** shows D_2 where the two rectangles denote the two iterations. As shown in Fig. 12, there exists a dependency

```
while (r0==0) {
 r0 = z;
 r1 = x;
 x = r1 + 1;
}
```

```
r2 = x;
z = 1;
```

Fig. 10 An example program.

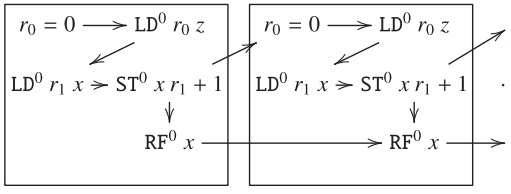


Fig. 11 A part of the program graph of the example program.

(denoted by an edge) across the two loop iterations except the edge from the exit to the entry of the loop.

Let us consider another view of the program graph, which enables us to use L-SQ rule. The program graph in **Fig. 13** also denotes a unit of D_n, where it does not exactly match an iteration in the original program. Formally, G_0 is defined as follows:

$$G_0 \equiv \mathtt{FN}^0 \to \mathtt{ST}^0\ y\ r_1 + 1 \to (G_0' \sqcup \{\mathtt{RF}^0\ y\}) \to \mathtt{RF}^0\ x$$
$$G_0' \equiv r_0 = 0 \to \mathtt{LD}^0\ r_0\ z \to \mathtt{LD}^0\ r_1\ y \to \mathtt{ST}^0\ x\ r_1 + 1\ .$$

By the definition, we can represent D_n as sequential compositions of program graphs:

$$D_0 \equiv \neg\, r_0 = 0$$
$$D_n \equiv G_0' \to \mathtt{RF}^0\ x \to {G_0}^{n-1} \to \mathtt{FN}^0 \to \mathtt{ST}^0\ y\ r_1 + 1 \to (\{\neg\, r_0 = 0\} \sqcup \{\mathtt{RF}^0\ y\}) \qquad (1 \le n)$$

where $G^0 = \varnothing$ and $G^n = G \to G^{n-1}$ $(1 \le n)$. We prove that G_0 has an invariant I at its entry and exit, as shown in **Fig. 14** where

$$I \equiv x = x^0 = r_1 + 1 \wedge y = y^0 = r_1$$
$$pre_0 \equiv x^0 = r_1 + 1 \wedge y = y^0 = r_1 \qquad pre_1 \equiv y \le y^0$$
$$post_1 \equiv y = y^0 \qquad post_2 \equiv x^0 = r_1 + 1 \wedge y^0 = r_1 \wedge y \le y^0$$

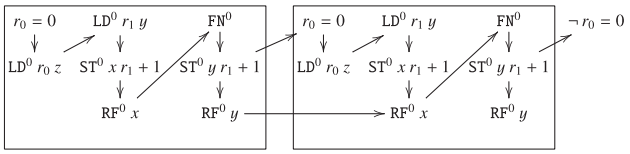


Fig. 12 A program graph D_2 under TSO.

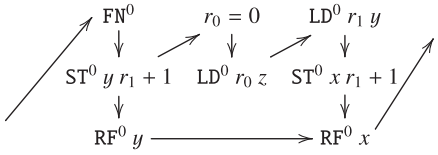


Fig. 13 The program graph D_n by a different viewpoint.

$$pre_2 \equiv post_3 \equiv x = x^0 = y^0 = r_1 \wedge y \le y^0$$
$$pre_3 \equiv post_4 \equiv x = x^0 = y^0 \wedge y \le y^0$$
$$rely^0 \equiv \mathrm{Inv}\{x, y, r_1\} \wedge \mathrm{Inv}\{x^0, y^0\}$$
$$guar^0 \equiv (\mathtt{df}(\underline{x}^1) \supset \mathtt{df}(x^1)) \wedge (\mathtt{df}(\underline{y}^1) \supset \mathtt{df}(y^1)) \wedge \mathrm{Inv}\{r_2, r_3\} \wedge y \le \underline{y} \wedge (x \le y + 1 \supset \underline{x} \le y + 1)$$

Figure 15 shows a full derivation where

$$pre_5 \equiv post_5 \equiv post_6 \equiv r_2 \le r_3 + 1$$
$$pre_6 \equiv post_7 \equiv \neg\, \mathtt{df}(y^1) \wedge r_2 \le y^1 + 1$$
$$pre_7 \equiv \neg\, \mathtt{df}(x^1) \wedge \neg\, \mathtt{df}(y^1) \wedge x \le y + 1$$
$$rely^1 \equiv (\mathtt{df}(\underline{x}^1) \supset \mathtt{df}(x^1)) \wedge (\mathtt{df}(\underline{y}^1) \supset \mathtt{df}(y^1)) \wedge \mathrm{Inv}\{r_2, r_3\} \wedge y \le \underline{y} \wedge (x \le y + 1 \supset \underline{x} \le y + 1)$$
$$guar^1 \equiv \mathrm{Inv}\{x, y, r_1\} \wedge \mathrm{Inv}\{x^0, y^0\}\ .$$

Since D_0 gets stuck under a pre-condition $r_0 = 0$ and a rely-condition $r_0 = \underline{r_0}$, all admissible behaviors of the program have been verified.

On the other hand, under PSO we can construct no similar derivation since there exists no edge from $\mathtt{RF}^0\ y$ to $\mathtt{RF}^0\ x$. Formally, there exists no derivation by soundness of logic described in Section 4 and existence of the computation as follows:

$$\begin{aligned}
&\langle D_2' \sqcup G_1, \langle \sigma, \Sigma\rangle\rangle \\
&\xrightarrow{\mathrm{c}}{}^{*}\ \langle (\mathtt{RF}^0\ y \to \mathtt{FN}^0 \to \mathtt{ST}^0\ y\ r_1 + 1 \to (\{\neg\, r_0 = 0\} \sqcup \{\mathtt{RF}^0\ y\})) \\
&\qquad \sqcup G_1, \langle \sigma[r_1 := 1, x := 2], \Sigma[0 := \Sigma^0[y := 1]]\rangle\rangle \\
&\xrightarrow{\mathrm{c}}{}^{*}\ \langle (\mathtt{RF}^0\ y \to \mathtt{FN}^0 \to \mathtt{ST}^0\ y\ r_1 + 1 \to (\{\neg\, r_0 = 0\} \sqcup \{\mathtt{RF}^0\ y\})), \\
&\qquad \langle \sigma[r_1 := 1, r_2 := 2, x := 2, z := 1], \Sigma[0 := \Sigma^0[y := 1]]\rangle\rangle \\
&\xrightarrow{\mathrm{c}}{}^{*}\ \langle \varnothing, \langle \sigma[r_1 := 1, r_2 := 2, x := 2, y := 2, z := 1], \Sigma\rangle\rangle
\end{aligned}$$

where D_1' and D_2' are the program graphs derived from D_1 and D_2 by removing the edges between $\mathtt{RF}^0\ y$ and $\mathtt{RF}^0\ x$, respectively, R^* is defined as the reflexive and transitive closure of a relation R, and σ and Σ are constant functions to 0 and $\mathtt{udf}$, respectively.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{pre_3, rely^0\}\ r_0 = 0\ \{guar^0, post_4\}}}{\{pre_3, rely^0\}\ \mathtt{LD}^0\ r_0\ z\ \{guar^0, post_4\}}}{\{pre_3, rely^0\}\ \mathtt{LD}^0\ r_1\ y\ \{guar^0, post_3\}}}{\{I, rely^0\}\ \mathtt{FN}^0\ \{guar^0, I\}}\qquad \dfrac{\dfrac{}{\{pre_2, rely^0\}\ \mathtt{ST}^0\ x\ r_1 + 1\ \{guar^0, post_2\}}}{\{pre_3, rely^0\}\ G_0'\ \{guar^0, post_2\}}\qquad \dfrac{}{\{pre_1, rely^0\}\ \mathtt{RF}^0\ y\ \{guar^0, post_1\}}}{\{I, rely^0\}\ \mathtt{ST}^0\ y\ r_1 + 1\ \{guar^0, post_4\}}}{\{I, rely^0\}\ \mathtt{FN}^0 \to \mathtt{ST}^0\ y\ r_1 + 1\ \{guar^0, post_4\}}\qquad \dfrac{}{\{pre_3, rely^0\}\ G_0' \sqcup \{\mathtt{RF}^0\ y\}\ \{guar^0, post_2 \wedge post_1\}}}{\{I, rely^0\}\ G_0 \setminus \{\mathtt{RF}^0\ x\}\ \{guar^0, post_2 \wedge post_1\}}\qquad \dfrac{}{\{pre_0, rely^0\}\ \mathtt{RF}^0\ x\ \{guar^0, I\}}}{\{I, rely^0\}\ G_0\ \{guar^0, I\}}$$

Fig. 14 A part of a derivation for G_0.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{pre_3 \wedge y = y^0, rely^0\}\ G_0'\ \{guar_0, post_2 \wedge y = y^0\}}}{\{pre_0, rely^0\}\ \mathtt{RF}^0\ x\ \{guar^0, I\}}}{\{I, rely^0\}\ G_0^{n-1}\ \{guar^0, I\}}}{\{I, rely^0\}\ \mathtt{FN}^0\ \{guar^0, I\}}\qquad \dfrac{}{\{pre_1, rely^0\}\ \neg\, r_0 = 0\ \{guar^0, pre_1\}}}{\{I, rely^0\}\ \mathtt{ST}^0\ y\ r_1 + 1\ \{guar^0, post_4\}}\qquad \dfrac{}{\{pre_1, rely^0\}\ \mathtt{RF}^0\ y\ \{guar^0, post_1\}}}{\{pre_3 \wedge y = y^0, rely^0\}\ D_n \setminus (\neg\, r_0 = 0 \sqcup \{\mathtt{RF}^0\ y\})\ \{guar^0, post_4\}}\qquad \dfrac{\dfrac{\dfrac{}{\{pre_7, rely^1\}\ \mathtt{LD}^1\ r_2\ x\ \{guar^1, post_7\}}}{\{pre_6, rely^1\}\ \mathtt{LD}^1\ r_3\ y\ \{guar^1, post_6\}}}{\{pre_5, rely^1\}\ \mathtt{ST}^1\ z\ 1\ \{guar^1, post_5\}}\qquad \dfrac{}{\{pre_5, rely^1\}\ \mathtt{RF}^1\ z\ \{guar^1, post_5\}}}{\{pre_7, rely^1\}\ G_1\ \{guar^1, post_5\}}}{\{pre_3 \wedge y = y^0, rely^0\}\ D_n\ \{guar^0, post_4 \wedge post_1\}}\qquad \dfrac{}{\{pre_3 \wedge y = y^0 \wedge pre_5, \mathrm{Inv}\ U \wedge \mathrm{Inv}\ W\}\ D_n \sqcup G_1\ \{\top, post_4 \wedge post_1 \wedge post_5\}}$$

Fig. 15 A derivation for the program graph by the TSO translation.

7. Related Work

There exist some papers about concurrent program logics for relaxed memory consistency models. However, most of them are specific to memory consistency models. Therefore, they cannot be essentially compared with this work, whose goal is to handle various memory consistency models.

Ridge [33] gave a rely-guarantee system for the x86-TSO semantics [32], and demonstrated differences of behaviors of Simpson's four slot algorithm [37] under SC and x86-TSO. However, the annotated pseudo-code in Ref. [33] has barriers and memory fences at exits of loop iterations, that is, each iteration has dependencies on the other iterations. It is unclear whether his system can handle the program with dependencies across loop iterations introduced in Section 1. Also, the system is specific to x86-TSO, and he did not prove its completeness, whereas our work expresses multiple relaxed memory consistency models as translations into program graphs, and proves relative completeness.

Ferreira et al. [17] introduced a relation $\preceq$ between commands in their paper (corresponding to statements in our paper) called *subsumptions*, and represented memory models as binary relations between statements. By extending judgments of the conventional separation logic [12], [30], they developed concurrent separation logic for relaxed memory models with invariants satisfied by the binary relations. However, since the subsumption relation $\preceq$ is *congruent*, for example, $c_0 \preceq c_1$ implies $\mathtt{WL}\ \varphi?c_0 \preceq \mathtt{WL}\ \varphi?c_1$, their concurrent separation logic looks to handle loops roughly, and does not seem to directly verify the program with dependencies across loop iterations introduced in Section 1. In addition, they did not show its completeness.

Vafeiadis et al. [44], [45] gave concurrent separation logics for restricted C11/C++11 memory models [22], [24], and showed some example derivations that include loops. However, all the loops consist of one expression, one conditional branch with one atomic load operation, or have *compare-and-swap* at their exits. Therefore, similar to [33], it is unclear how to handle the program with dependencies across loop iterations introduced in Section 1. Similarly, since no program in Lahav et al.'s paper, which performed Owicki-Gries reasoning for weak memory models, contains multiple statements except SK [27], it is never clear how to handle dependencies across loop iterations. In addition, they did not show their completeness.

This paper provides a concurrent program logic following Stølen and Xu's proof systems [42], [47], [48], which do not deal with relaxed memory consistency models. In other words, their systems deal with the strictest memory consistency model only.

8. Conclusion and Future Work

This paper provides concurrent program logic for relaxed memory consistency models that can represent standard memory consistency models and handle dependencies across loop iterations. We introduced graph representations of programs called program graphs, gave a small-step operational semantics for them, and formalized relaxed memory consistency models as translations to graphs. We keep our concurrent program logic theoretically simple, and not only sound but also relatively complete to the semantics.

There are four future directions for this work. The first is to support memory hierarchy. We assumed one-layer store buffers for simplicity of presentation. Therefore, our logic cannot currently verify programs that show different orders of effects to different threads, for example, *Independent-Reads-Independent-Write* [10]. The authors' previous works on model checking [2], [3], [5] handled such memory consistency models by introducing a notion of *visibility of effects on each thread*. We think that verification of the program is possible by introducing multilayered store buffers in operational semantics and auxiliary variables that represent variable on the buffers in concurrent program logic. The second is to support relaxed memory consistency models (e.g., UPC memory model [43] and C11/C++11 memory models [22], [24]) that do not assume *global time*. The authors' previous works on model checking [2], [3], [5] handled such memory consistency models by introducing a notion of *speculative behavior on each thread*. We think that the operational semantics and concurrent program logic can be extended in a similar way. The third is to enhance our approach to support pointers, arrays, functions, dynamic creation and termination of threads, and so forth, so that our approach can be applied to real-world programs like OpenMP [31]. We consider that our formulation seems compatible with some existing works (e.g., separation logic [12], [30], deny-guarantee [16], and concurrent views framework [15]). The fourth is to develop a software to construct/check derivations. As we have seen in Section 6, it is not easy to construct a derivation, on the contrary, to check whether the derivation follows the inference rules in our logic or not.

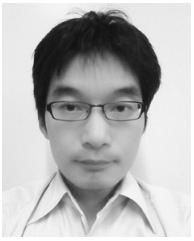
Finally, program graphs proposed in this paper also seem to be used as representations of programs in other fields, for example, representations of programs in model checking that is parameterized by memory consistency models, intermediate representations in compilers that are parameterized by memory consistency models, etc.

Acknowledgments Most of the soundness and relative completeness proofs in this paper follow those in Qiwen Xu's PhD thesis and its journal version [47], [48]. The authors would like to thank him for answering some questions respectfully. The authors also thank the anonymous reviewer for several comments to improve the final version of the paper. This work was supported by JSPS KAKENHI Grant Numbers 25871113 and 16K21335.

References

[1] Abe, T. and Maeda, T.: Model Checking with User-Definable Memory Consistency Models, *Proc. PGAS, short paper*, pp.225–230 (2013).
[2] Abe, T. and Maeda, T.: A General Model Checking Framework for Various Memory Consistency Models, *Proc. HIPS*, pp.332–341 (2014).
[3] Abe, T. and Maeda, T.: Optimization of a General Model Checking Framework for Various Memory Consistency Models, *Proc. PGAS* (2014).
[4] Abe, T. and Maeda, T.: Towards a Unified Verification Theory for Various Memory Consistency Models, *Proc. LOLA* (2015).
[5] Abe, T. and Maeda, T.: A General Model Checking Framework for Various Memory Consistency Models, *International Journal on Software Tools for Technology Transfer* (2016). To appear.
[6] Adve, S. and Gharachorloo, K.: Shared memory consistency models: A tutorial, *Computer*, Vol.29, No.12, pp.66–76 (1996).
[7] Alglave, J., Kroening, D., Nimal, V. and Tautschnig, M.: Software

Verification for Weak Memory via Program Transformation, *Proc. ESOP*, LNCS, No.7792, pp.512–532 (2013).
[8] Alglave, J., Maranget, L., Sarkar, S. and Sewell, P.: Fences in weak memory models, *Proc. CAV*, LNCS, pp.258–272 (2010).
[9] Atig, M.F., Bouajjani, A., Burckhardt, S. and Musuvathi, M.: On the verification problem for weak memory models, *Proc. POPL*, pp.7–18 (2010).
[10] Boehm, H.-J. and Adve, S.V.: Foundations of the C++ Concurrency Memory Model, *Proc. PLDI*, pp.68–78 (2008).
[11] Boudol, G. and Petri, G.: Relaxed memory models: An operational approach, *Proc. POPL*, pp.392–403 (2009).
[12] Brookes, S.: A Semantics for Concurrent Separation Logic, *Theoretical Comp. Sci.*, Vol.375, No.1–3, pp.227–270 (2007).
[13] Cray Inc: *Chapel Language Specification Version 0.98* (2015).
[14] Dan, A., Meshman, Y., Vechev, M. and Yahav, E.: Predicate Abstraction for Relaxed Memory Models, *Proc. SAS*, pp.84–104 (2013).
[15] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. and Yang, H.: Views: Compositional reasoning for concurrent programs, *ACM SIGPLAN Notices*, Vol.48, No.1, pp.287–300 (2013).
[16] Dodds, M., Feng, X., Parkinson, M. and Vafeiadis, V.: Deny-Guarantee Reasoning, *Proc. ESOP*, pp.363–377 (2009).
[17] Ferreira, R., Feng, X. and Shao, Z.: Parameterized Memory Models and Concurrent Separation Logic, *Proc. ESOP*, pp.267–286 (2010).
[18] Herlihy, M.P. and Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects, *TOPLAS*, pp.463–492 (1990).
[19] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming, *Comm. ACM*, Vol.12, No.10, pp.576–580, 583 (1969).
[20] Intel Corp.: *A Formal Specification of Intel Itanium Processor Family Memory Ordering* (2002).
[21] Intel Corp.: *Intel 64 and IA-32 Architectures Software Developer's Manual* (2016).
[22] ISO/IEC 14882:2011: *Programming Language C++* (2011).
[23] ISO/IEC 1539-1:2010: *Information technology – Programming languages – Fortran* (2010).
[24] ISO/IEC 9899:2011: *Programming Language C* (2011).
[25] Jagadeesan, R., Pitcher, C. and Riely, J.: Generative Operational Semantics for Relaxed Memory Models, *Proc. ESOP*, pp.307–326 (2010).
[26] Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference, PhD Thesis, Oxford University (1981).
[27] Lahav, O. and Vafeiadis, V.: Owicki-Gries Reasoning for Weak Memory Models, *Proc. ICALP*, pp.311–323 (2015).
[28] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
[29] Manson, J., Pugh, W. and Adve, S.V.: The Java memory model, *Proc. POPL*, pp.378–391 (2005).
[30] O'Hearn, P.W.: Resources, concurrency, and local reasoning, *Theor. Comput. Sci.*, Vol.375, No.1-3, pp.271–307 (2007).
[31] OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 4.0* (2013).
[32] Owens, S., Sarkar, S. and Sewell, P.: A Better x86 Memory Model: x86-TSO, *Proc. TPHOLs*, LNCS, Vol.5674, pp.391–407 (2009).
[33] Ridge, T.: A Rely-Guarantee Proof System for x86-TSO, *Proc. VSTTE*, pp.55–70 (2010).
[34] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: *X10 Language Specification Version 2.5.3* (2015).
[35] Saraswat, V., Jagadeesan, R., Michael, M. and von Praun, C.: A Theory of Memory Models, *Proc. PPoPP*, pp.161–172 (2007).
[36] SCALE: Scalable Computing for Advanced Library and Environment. available from ⟨http://scale.aics.riken.jp/⟩.
[37] Simpson, H.R.: Four-slot fully asynchronous communication mechanism, *IEE Proc. Part E Computers and Digital Techniques*, pp.17–30 (1990).
[38] SPARC International, Inc.: *The SPARC Architecture Manual, Version 8* (1991).
[39] SPARC International, Inc.: *The SPARC Architecture Manual, Version 9* (1994).
[40] Steinke, R.C. and Nutt, G.J.: A unified theory of shared memory consistency, *J. ACM*, Vol.51, No.5, pp.800–849 (2004).
[41] Stirling, C.: A Generalization of Owicki–Gries's Hoare Logic for a Concurrent While Language, *Theoretical Computer Science*, Vol.58, No.1–3, pp.347–359 (1988).
[42] Stølen, K.: Development of Parallel Programs on Shared Data-structures, Technical Report UMCS-91-1-1, Department of Computer Science, University of Manchester (1991).
[43] The UPC Consortium: *UPC Language Specifications Version 1.3* (2013).
[44] Turon, A., Vafeiadis, V. and Dreyer, D.: GPS: Navigating weak memory with ghosts, protocols, and separation, *Proc. OOPSLA*, pp.691–707 (2014).
[45] Vafeiadis, V. and Narayan, C.: Relaxed separation logic: A program logic for C11 concurrency, *Proc. OOPSLA*, pp.867–884 (2013).
[46] Winskel, G.: *The formal semantics of programming languages*, MIT Press (1993).
[47] Xu, Q.: A Theory of State-based Parallel Programming, PhD Thesis, Oxford University Computing Laboratory (1992).
[48] Xu, Q., de Roever, W.P. and He, J.: The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs, *Formal Asp. Comput.*, Vol.9, No.2, pp.149–174 (1997).
[49] Yang, Y., Gopalakrishnan, G. and Lindstrom, G.: UMM: An operational memory model specification framework with integrated model checking capability, *Concurr. Comput.: Pract. Exper.*, Vol.17, No.5-6, pp.465–487 (2005).



Tatsuya Abe was born in 1979. He received his B.Sc. and Ph.D. degrees from Kyoto University and the University of Tokyo in 2002 and 2007, respectively. He worked for National Institute of Advanced Industrial Science and Technology, Kyoto University, and RIKEN. He is currently a senior research scientist at STAIR Lab, Chiba Institute of Technology. His research interests include programming languages, program verification, concurrency, and distributed computation. He is a member of the IPSJ and ACM.

Toshiyuki Maeda was born in 1977, and received his bachelor's degree in Science, master's degree in Information Science and Technology, and Ph.D. in Information Science and Technology from University of Tokyo in 2000, 2002, and 2006, respectively. From 2006 to 2012, he was a Research Associate at Graduate School of Information Science and Technology, University of Tokyo. In April 2012, he joined RIKEN AICS (Advanced Institute for Computational Science) as the Team Leader of HPC Usability Research Team (he is the Part-Time Team Leader from 2016). In April 2016, he joined Chiba Institute of Technology as a principal research scientist of STAIR Lab. His research interests include programming languages, systems software, virtualization/container technologies, HPC usability, and software for research/development of artificial intelligence.