Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models

Tatsuya Abe(✉) and Toshiyuki Maeda

STAIR Lab, Chiba Institute of Technology,
2-17-1 Tsudanuma, Narashino, Chiba 275-0016, Japan
{abet,tosh}@stair.center

Abstract. Concurrent program logics are frameworks for constructing proofs, which ensure that concurrent programs work correctly. However, most conventional concurrent program logics do not consider the complexities of modern memory structures, and the proofs in the logics do not ensure that programs will work correctly. To the best of our knowledge, Independent Reads Independent Writes (IRIW), which is known to have non-intuitive behavior under relaxed memory consistency models, has not been fully studied under the context of concurrent program logics. One reason is the gap between theoretical memory consistency models that program logics can handle and the realistic memory consistency models adopted by actual computer architectures. In this paper, we propose observation variables and invariants that fill this gap, releasing us from the need to construct operational semantics and logic for each specific memory consistency model. We describe general operational semantics for relaxed memory consistency models, define concurrent program logic sound to the operational semantics, show that observation invariants can be formalized as axioms of the logic, and verify IRIW under an observation invariant. We also obtain a novel insight through constructing the logic. To define logic that is sound to the operational semantics, we dismiss shared variables in programs from assertion languages, and adopt variables observed by threads. This suggests that the so-called bird's-eye view of the whole computing system disturbs the soundness of the logic.

Keywords: Relaxed memory consistency model · Concurrent program logic · Rely/guarantee method · Observation · Independent Reads Independent Writes

1 Introduction

Memory structures are becoming increasingly complicated as computing systems continue to grow. This can be overwhelming when attempting to write programs that work on architectures consisting of complicated memory structures. Since conventional program verification considers architectures with simple memory structures, it struggles to deal with architectures that consist of complicated memory structures.

© Springer International Publishing AG 2016
A. Igarashi (Ed.): APLAS 2016, LNCS 10017, pp. 63–84, 2016.
DOI: 10.1007/978-3-319-47958-3_4

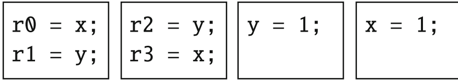

Fig. 1. Independent Reads Independent Writes

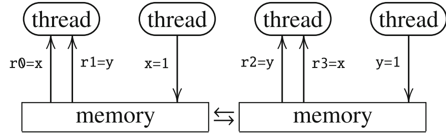

Fig. 2. Non-remote-write-atomic memories

To illustrate the problem, consider an example racy program. Readers may consider that racy programs should be prohibited as their behaviors are specified *undefined* in C++11 [12]. However, in lower-level programming (e.g., on virtual machine or computer architecture), racy programs are necessary to implement register algorithms, which provide mutual exclusion etc. Although such racy programs are not typically large, their non-intuitive behaviors make it difficult to verify them.

Figure 1 shows *Independent Reads Independent Writes (IRIW)* [4]. Variables x and y are shared, and variables r_0, r_1, r_2, and r_3 are thread-local. We assume that all variables are initialized to 0. IRIW consists of four threads. Two threads are *readers*, which read values from the shared variables x and y. The other two threads are *writers*. One writes 1 to x, and the other writes 1 to y. If the write to x is performed before the write to y, then $r_2 \leq r_3$ seems to hold, since $r_2 > r_3$ (i.e., $r_2 = 1$ and $r_3 = 0$) does not hold for the following reason:

1. $r_2 = 1$ implies $y = 1$, and
2. we assume that the write to x is performed before the write to y;
3. therefore, when x is read (to r_3), its value is 1.

Similarly, if the write to y is performed before the write to x, then $r_0 \leq r_1$ seems to hold. Therefore, it would appear that $r_0 \leq r_1 \vee r_2 \leq r_3$.

However, this is not always the case, because an architecture may realize a form of shared memory, as shown in Fig. 2. This means that the first reader and writer share the same physical memory, and the second reader and writer share another physical memory. Shared memory is realized by any mechanism for data transfer (denoted by $\leftrightarrows$) between the physical memories. The architecture that has mechanism for data transfer is sensitive to the so-called *remote-write-atomicity* [11] (also called *multi-copy-atomicity* in [24]). Remote-write-atomicity claims that if two thread write values to (possibly distinct) locations, then the other threads *must* observe the *same* order between the two write operations.

Here, let us assume that physical memories do not enjoy remote-write-atomicity, that is, effects on one memory *cannot* be immediately transferred to the other memory. Under this architecture, while the first reader *may* observe that the write to x is performed before the write to y, the second reader *may* observe that the write to y is performed before the write to x. Therefore, there is no guarantee that $r_0 \leq r_1 \vee r_2 \leq r_3$.

Thus, in modern program verification, we cannot ignore remote-write-atomicity. However, to the best of our knowledge, there exists no concurrent

program logic in which remote-write-atomicity can be switched on and off. One reason is the existence of a gap between theoretical memory consistency models, which concurrent program logics can handle, and realistic memory consistency models, which are those adopted by actual computer architectures. While theoretical memory consistency models (axioms written in assertion languages) in concurrent program logics describe relations between expressions, which are often the orders of values that are evaluated by expressions, realistic memory consistency models (which are often written in natural languages rather than formal languages) describe the orders of executions of statements on actual computer architectures. In this paper, we propose *observation variables and invariants* that fill this gap, thus releasing us from the need to construct operational semantics and logic for each specific memory consistency model. We define general operational semantics to consider cases in which threads own their memories, and construct concurrent program logic in which we give proofs that ensure certain properties hold when programs finish. We can control observation invariants as uniform *axioms* of the logic. This enables us to show that a property holds when a program finishes under an observation invariant, whereas the property does not hold when the program finishes without the observation invariant. In Sect. 9, we verify IRIW using this logic under an observation invariant induced by a realistic memory consistency model like SPARC-PSO [27].

To the best of our knowledge, the derivation shown in Sect. 9 is the first to ensure that a property holds in concurrent program logic that handles relaxed memory consistency models like SPARC-PSO, although the behavior of IRIW under more relaxed memory consistency models that refute the property has been discussed several times in the literature (e.g., [4,22,24,25,31]).

In constructing the concurrent program logic, we obtained a novel insight into the use of shared variables in an assertion language for operational semantics with relaxed memory consistency models. First, we extend an assertion language in the logic by introducing the additional variable x^i to denote x as observed by the i-th thread. The value of x^i is not necessarily the same as that of x. Next, we restrict the assertion language by dismissing shared variables in programs from assertion languages. This prohibits us from describing the value of x. By designing this assertion language, we can construct a concurrent program logic that is *sound* to operational semantics (explained in Sect. 4) with relaxed memory consistency models. This suggests that, in concurrent computation, the so-called bird's-eye view that overlooks the whole does not exist, and that each thread runs according to its own observations, and some (or all) threads sometimes reach a consensus.

The rest of this paper is organized as follows. Section 2 discusses related work, and Sect. 3 presents some definitions that are used throughout this paper. Section 4 gives operational semantics (based on the notion of state transition systems) for relaxed memory consistency models. Section 5 explains our concurrent program logic. Section 6 defines validity of judgments. Section 7 introduces the notion of observation invariants. Section 8 then presents our soundness theorem.

In Sect. 9, we provide example derivations for concurrent programs. Section 10 concludes the paper and discusses ideas for future work.

2 Related Work

Stølen [28] and Xu et al. [35,36] provided concurrent program logics based on rely/guarantee reasoning [13]. However, they did not consider relaxed memory consistency models containing observation invariants, that is, they handle *strict consistency*, the strictest memory consistency model. This paper handles relaxed memory consistency models. The memory consistency models with observation invariants in this paper are more relaxed than those obeying strict consistency.

Ridge [23] developed a concurrent program logic for x86-TSO [26] based on rely/guarantee reasoning by introducing *buffers*. However, buffers to *one* shared memory are known to be insufficient to cause the behavior of IRIW. This paper handles more relaxed memory consistency models than x86-TSO, as threads have their own memories to deal with observation invariants.

Ferreira et al. [7] introduced a concurrent separation logic that is parameterized by invariants, and explained the non-intuitive behavior of IRIW. Their motivation for constructing a parametric logic coincides with ours. However, their logic is based on *command subsumptions*, which describe the execution orders of statements. This is different from our notion of observations; their approach therefore has no direct connection to our logic, and gave no sufficient condition to ensure the correctness of IRIW. Any connection between their logic and ours remains an open question.

Vafeiadis et al. presented concurrent separation logics for restricted C++11 memory models [30,32]. The restricted C++11 semantics are so weak that the property for IRIW (shown in Sect. 1) does not hold without additional assumptions. However, unlike our approach, they did not handle programs that contain write operations to *distinct* locations, such as IRIW. In another paper [15], Lahav and Vafeiadis described an Owicki–Gries style logic and verified a program consisting of multiple reads and writes in the logic. This program is different from IRIW, as the reads/writes are from/to the same location. The essence of IRIW is to write to *distinct* locations x and y. Our paper proposes the notion of observation invariants, constructs a simple concurrent program logic, formalizes the axioms of our logic, and gives a formal proof for IRIW. This simplification provides the insight explained in Sect. 1.

The authors proposed a notion of program graphs, representations of programs with memory consistency models, gave operational semantics and construct program logic for them [1]. However, the semantics and logic cannot handle the non-intuitive behavior of IRIW since remote-write-atomicity is implicitly assumed.

There also exist verification methods that are different from those using concurrent program logics. Model checking based on exhaustive searches is a promising program verification method [2,10,14,17]. Given a program and an assertion, model checking is good at detecting execution traces that violate the assertions, but is less suitable for ensuring that the assertion holds.

Some reduction methods to *Sequential Consistency (SC)* via race-freedom of programs are well-known (e.g., [5,20,21]). However, verification of racy programs like concurrent copying protocols is one of the authors' concerns [3], and programs that have non-SC behaviors are our main targets.

Boudol et al. proposed an operational semantics approach to represent a relaxed memory consistency model [5,6]. They defined a process calculus, equipped with buffers that hold the effects of stores, and its operational semantics to handle the non-intuitive behavior of IRIW. He proved *Data Race Freedom (DRF)* guarantee theorem that DRF programs have the same behaviors as those under SC. However, IRIW is not DRF.

Owens et al. reported that x86-CC [25] allows the non-intuitive behavior of IRIW, and designed x86-TSO [22] that prohibits the behavior. He also extended DRF to *Triangular Race Freedom (TRF)* that TRF programs have the same behaviors under x86-TSO as those under SC [21]. Although IRIW is surely TRF, a slight modification of IRIW in which additional writes to distinct variables at the start on the reader threads are inserted is not TRF. Since the program has a non-SC behavior, verification of the program under SC cannot ensure correctness of the program under x86-TSO. Our verification method to use observation invariants is robust to such slight changes. In addition, our method is not specific to a certain memory consistency model like x86-TSO. An observation invariant introduced in Sect. 7 for IRIW is independent of the slight change, and we can construct a derivation for the program that is similar to the derivation for IRIW explained in Sect. 9.3.

3 Concurrent Programs

In this section, we formally define our target concurrent programs.

Similar to the conventional program logics (e.g., [8]), sequential programs are defined as sequences of statements. Let r denote the thread-local variables that cannot be accessed by other threads, $x, y, \ldots$ denote shared variables, and e denote thread-local expressions (thread-local variables, constant values val, arithmetic operations, and so on). A sequential program can then be defined as follows:

$$S^i ::= \mathtt{SK}^i \mid \mathtt{MV}^i\, r\, e \mid \mathtt{LD}^i\, r\, x \mid \mathtt{ST}^i\, x\, e \mid \mathtt{IF}^i\, \varphi ? S^i : S^i \mid \mathtt{WL}^i\, \varphi ? S^i \mid S^i ; S^i$$
$$\varphi ::= e = e \mid e \leq e \mid \neg\varphi \mid \varphi \supset \varphi \mid \forall r.\, \varphi.$$

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 is clear. The `SK` statement denotes an ordinary no-effect statement (SKip). As in conventional program logics, `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 immediately observed by the other threads. 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 $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \leftrightarrow \psi$, and $\exists\, r.\, \varphi$ as $(\neg\, \varphi) \supset \psi$, $\neg\, ((\neg\, \varphi) \vee (\neg\, \psi))$, $(\varphi \supset \psi) \wedge (\psi \supset \varphi)$, and $\neg\, \forall\, r.\, \neg\, \varphi$, 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 the tautology $\forall\, r.\, r = r$.

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

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

In this paper, the number of threads N is fixed during program execution. Parentheses are often omitted, and all operators except $\supset$ are assumed to be left associative.

4 Operational Semantics

In this section, we define small-step operational semantics for the programming language defined in Sect. 3. Specifically, the semantics is defined as a standard state transition system, where a *state* (written as *st*) is represented as a pair of $\langle \sigma, \Sigma \rangle$. The first element of the pair, σ, consists of the values of thread-local variables and shared variables that threads observe on registers and memories; formally, a function from thread identifiers and (thread-local and shared) variables to values. The second element, Σ, represents buffers that temporarily buffer the effects of store operations to shared variables. It may be worth noting that, in the present paper, buffers are not queues, that is, each buffer stores only the latest value written to its associated variable, for simplicity. This is because the present paper focuses on verifying IRIW, and storing the latest value suffices for the purpose. Replacing buffers with queues is straightforward and not shown in the present paper. $\Sigma^{i,j}$ refers to thread j's buffer for effects caused by thread i's statements. If $i \neq j$, then thread j cannot observe the effects that are buffered. If $i = j$, then thread j can observe the effects that are buffered.

Using Fig. 3, we now present an informal explanation of buffers and memories. Thread 0 executes statement $\mathtt{ST}^0\, x\, 1$, and buffers $\Sigma^{0,0}$ and $\Sigma^{0,1}$ are updated. Next, thread 1 executes statement $\mathtt{LD}^1\, r_1\, x$. If the effect of $\mathtt{ST}^0\, x\, 1$ has not yet reached σ^1, thread 1 cannot observe it, and reads the initialized value 0. If the effect of $\mathtt{ST}^0\, x\, 1$ has already reached σ^1, thread 1 reads 1. Finally, thread 0 executes statement $\mathtt{LD}^0\, r_0\, x$. Whether the effect of $\mathtt{ST}^0\, x\, 1$ has reached σ^1 or not, thread 0 can observe it. Therefore, thread 0 reads a value of 1. Updates from Σ to σ are performed without the need for statements.

Formally, in the following, a function from triples formed of two thread identifiers and shared variables to values is evaluated by *currying* all functions, for the convenience of *partial applications*. We assume that the set of values contains a special constant value udf to represent uninitialized or invalidated buffers. We often write σi, Σi, and $\Sigma i j$ as σ^i, Σ^i, and $\Sigma^{i,j}$, respectively, for readability. We

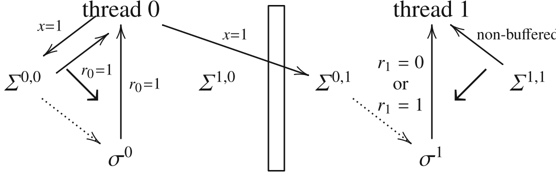

Fig. 3. Buffers and memories

$$f[b := c]a = \begin{cases} c & \text{if } a = b \\ fa & \text{otherwise} \end{cases}$$

$$\sigma^i[\Sigma^{i,i}]v = \begin{cases} \Sigma^{i,i}v & \text{if } \Sigma^{i,i}v \neq \texttt{udf} \\ \sigma^i v & \text{otherwise} \end{cases}$$

Fig. 4. Update functions

define the following two operations for update functions as in Fig. 4 where f ranges over each among σ, σ^i, Σ, Σ^i, and $\Sigma^{i,j}$.

Figure 5 shows the rules of the operational semantics, where $\langle\!| e |\!\rangle_{\sigma^i}$ denotes the valuation of an expression e as follows:

$$\langle\!| val |\!\rangle_{\sigma^i} = val \quad \langle\!| r |\!\rangle_{\sigma^i} = \sigma^i r \quad \langle\!| x |\!\rangle_{\sigma^i} = \sigma^i x \quad \langle\!| e_1 + e_2 |\!\rangle_{\sigma^i} = \langle\!| e_1 |\!\rangle_{\sigma^i} + \langle\!| e_2 |\!\rangle_{\sigma^i} \quad \ldots$$

and $\sigma^i \vDash \varphi$ denotes the satisfiability of φ on σ^i in the standard manner, which is defined as follows:

$$\sigma^i \vDash e_1 = e_2 \Leftrightarrow \langle\!| e_1 |\!\rangle_{\sigma^i} = \langle\!| e_2 |\!\rangle_{\sigma^i} \quad \sigma^i \vDash e_1 \leq e_2 \Leftrightarrow \langle\!| e_1 |\!\rangle_{\sigma^i} \leq \langle\!| e_2 |\!\rangle_{\sigma^i} \quad \sigma^i \vDash \neg\varphi \Leftrightarrow \sigma^i \nvDash \varphi$$
$$\sigma^i \vDash \varphi \supset \varphi' \Leftrightarrow \sigma^i \vDash \varphi \text{ implies } \sigma^i \vDash \varphi' \qquad \sigma^i \vDash \forall r.\, \varphi(r) \Leftrightarrow \sigma^i \vDash \varphi(v) \text{ for any } v.$$

$$\frac{}{\langle P, st\rangle \xrightarrow{\text{e}} \langle P, st'\rangle}\ \text{(O-ENV)} \qquad \frac{}{\langle \texttt{MV}^i\ r\ e, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}^i, \langle \sigma[i := \sigma^i[r := \langle\!| e |\!\rangle_{\sigma^i}]], \Sigma\rangle\rangle}\ \text{(O-MV)}$$

$$\frac{}{\langle \texttt{LD}^i\ r\ x, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}^i, \langle \sigma[i := \sigma^i[r := \sigma^i[\Sigma^{i,i}]x]], \Sigma\rangle\rangle}\ \text{(O-LD)}$$

$$\frac{}{\langle \texttt{ST}^i\ x\ e, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}^i, \langle \sigma, \Sigma[i := \Sigma^i[j := \Sigma^{i,j}[x := \langle\!| e |\!\rangle_{\sigma^i}] \mid 0 \leq j < N]]\rangle\rangle}\ \text{(O-ST)}$$

$$\frac{\sigma^i \vDash \varphi}{\langle \texttt{IF}^i\ \varphi?P{:}Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P, \langle\sigma,\Sigma\rangle\rangle}\ \text{(O-IT)} \qquad \frac{\sigma^i \nvDash \varphi}{\langle \texttt{IF}^i\ \varphi?P{:}Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle Q, \langle\sigma,\Sigma\rangle\rangle}\ \text{(O-IE)}$$

$$\frac{\sigma^i \vDash \varphi}{\langle \texttt{WL}^i\ \varphi?P, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P; \texttt{WL}^i\ \varphi?P, \langle\sigma,\Sigma\rangle\rangle}\ \text{(O-WT)} \qquad \frac{\sigma^i \nvDash \varphi}{\langle \texttt{WL}^i\ \varphi?P, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}^i, \langle\sigma,\Sigma\rangle\rangle}\ \text{(O-WE)}$$

$$\frac{\langle P, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}, \langle\sigma',\Sigma'\rangle\rangle}{\langle P; Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle Q, \langle\sigma',\Sigma'\rangle\rangle}\ \text{(O-SQ}_0\text{)} \qquad \frac{\langle P, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P', \langle\sigma',\Sigma'\rangle\rangle}{\langle P; Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P'; Q, \langle\sigma',\Sigma'\rangle\rangle}\ \text{(O-SQ}_1\text{)}$$

$$\frac{\langle P, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}, \langle\sigma',\Sigma'\rangle\rangle}{\langle P \parallel Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle Q, \langle\sigma',\Sigma'\rangle\rangle}\ \text{(O-PR}_0\text{)} \qquad \frac{\langle Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle \texttt{SK}, \langle\sigma',\Sigma'\rangle\rangle}{\langle P \parallel Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P, \langle\sigma',\Sigma'\rangle\rangle}\ \text{(O-PR}_1\text{)}$$

$$\frac{\langle P, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P', \langle\sigma',\Sigma'\rangle\rangle}{\langle P \parallel Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P' \parallel Q, \langle\sigma',\Sigma'\rangle\rangle}\ \text{(O-PR}_2\text{)} \qquad \frac{\langle Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle Q', \langle\sigma',\Sigma'\rangle\rangle}{\langle P \parallel Q, \langle\sigma,\Sigma\rangle\rangle \xrightarrow{\text{c}} \langle P \parallel Q', \langle\sigma',\Sigma'\rangle\rangle}\ \text{(O-PR}_3\text{)}$$

Fig. 5. Our operational semantics

A pair of a program and a state is called a *configuration*. Each rule is represented by a one-step transition between configurations $\langle P, st\rangle \xrightarrow{\delta} \langle P', st'\rangle$, which indicates that a statement causes $\langle P, st\rangle$ to transit to $\langle P', st'\rangle$, where δ is c or e.

Specifically, Rule O-ENV denotes a transition that requires no statement in P, which means that other threads are executing statements or memories are being updated from buffers. Although the rule was originally introduced to mean that other threads consume statements in conventional operational semantics for concurrent programming languages (with strict consistency) [35,36], the rule has the additional meaning here that memories are updated from buffers in constructing operational semantics for relaxed memory consistency models.

Readers unfamiliar with operational semantics for an imperative concurrent programming language (and its rely/guarantee reasoning) may consider $\xrightarrow{\text{e}}$ to be nonsense because $\xrightarrow{\text{e}}$ seems to allow any transitions. However, it is restricted to being admissible under a rely-condition by the notion of *validity* for judgments (called *rely/guarantee specifications*), as defined in Sect. 6. This is similar to the context of Hoare logic, where transitions consuming statements are defined to be large *at first*, and the transitions are restricted to be admissible ones under pre-conditions by the notion of validity for Hoare triples. This is one of the standard methods of defining operational semantics for an imperative concurrent programming language (e.g., as seen in [35,36]), and is not caused by handling relaxed memory consistency models. Here, in accordance with the standard, a transition $\xrightarrow{\text{e}}$ that consumes no statements is defined to be the product of states.

Rule O-MV evaluates e and updates σ with respect to r. Rule O-LD evaluates x on $\Sigma^{i,i}$, if $\Sigma^{i,i}x$ is defined (i.e., effects on x by statements on thread i itself are buffered), and on σ^i otherwise, and updates σ^i with respect to r. O-ST evaluates e and updates $\Sigma^{i,j}$ (not σ^i) with respect to x for any j; i.e., the rule indicates that the effect of the store operation is buffered in $\Sigma^{i,j}$. Rule O-IT handles a branch statement by asserting that φ is satisfied under state σ^i, and P is chosen. If σ^i is not satisfied, rule O-IE is applied and Q is chosen. Rule O-WT handles a loop statement by asserting that φ^i is satisfied under state σ, and an iteration is performed. If σ^i is not satisfied, rule O-WE is applied, and the program exits from the loop. Rules O-SQ and O-PR handle sequential and parallel compositions of programs, respectively.

5 Concurrent Program Logic

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

$$\Phi ::= E = E \mid E \leq E \mid \neg\, \Phi \mid \Phi \supset \Phi \mid \forall v.\, \Phi \qquad v ::= r \mid x^i \mid \underline{r} \mid \underline{x^i}$$

where E represents a pseudo-expression denoting thread-local variables r, observation variables x^i, next thread-local variables $\underline{r}$, next observation variables $\underline{x^i}$, constant values *val*, arithmetic operations, and so on. Our assertion language does not contain a shared variable x that occurs in programs. This means that *nobody* observes the whole system. This novelty is a key point of this paper. We

often write r as r^i when referring to r being a thread-local variable on the i-th thread. The observation variable x^i represents the value written to the shared variable x by ST on a thread with identifier i. The next variable $\underline{v}$ represents the value of v on a state to which the current state transits under the operational semantics.

Figure 6 shows the judgment rules. They are defined in the styles of Stølen and Xu's proof systems [28,35,36], which have two kinds of judgments. Each judgment of the form $\models \Phi$ refers to satisfiability in the first-order predicate logic with equations in a standard manner. Each judgment of the form $\{pre, rely\}P\{guar, post\}$ (where *pre* and *post* have no next variable) states that, if program P runs under pre-condition *pre* and rely-condition *rely* (which are guaranteed by the other threads as well as the *environments*, as explained in Sect. 6) according to the operational semantics of Sect. 4, then the guarantee-condition *guar* (on which the other threads rely) holds, as in conventional rely/guarantee systems. In the rest of this paper, we write $\vdash \{pre, rely\}\ P\ \{guar, post\}$ if $\{pre, rely\}\ P\ \{guar, post\}$ can be derived from the judgment rules of Fig. 6.

$$\frac{\begin{array}{l}\models pre \supset [e/r]post \\ \models pre \supset \lceil \mathtt{MV}^i\ r\ e \rceil^V \supset guar \\ \models pre \perp rely \qquad \models post \perp rely\end{array}}{\{pre, rely\}\ \mathtt{MV}^i\ r\ e\ \{guar, post\}}\ \text{(L-MV)} \qquad \frac{\begin{array}{l}\models pre \supset post \\ \models pre \supset \mathrm{I}(V) \supset guar \\ \models pre \perp rely \qquad \models post \perp rely\end{array}}{\{pre, rely\}\ \mathtt{SK}^i\ \{guar, post\}}\ \text{(L-SK)}$$

$$\frac{\begin{array}{l}\models pre \supset [x^i/r]post \\ \models pre \supset \lceil \mathtt{LD}^i\ r\ x \rceil^V \supset guar \\ \models pre \perp rely \qquad \models post \perp rely\end{array}}{\{pre, rely\}\ \mathtt{LD}^i\ r\ x\ \{guar, post\}}\ \text{(L-LD)} \qquad \frac{\begin{array}{l}\models pre \supset [e/x^i]post \\ \models pre \supset \lceil \mathtt{ST}^i\ x\ e \rceil^V \supset guar \\ \models pre \perp rely \qquad \models post \perp rely\end{array}}{\{pre, rely\}\ \mathtt{ST}^i\ x\ e\ \{guar, post\}}\ \text{(L-ST)}$$

$$\frac{\begin{array}{l}\models \{pre \wedge \varphi, rely\}\ S_0^i\ \{guar, post\} \\ \models \{pre \wedge \neg\varphi, rely\}\ S_1^i\ \{guar, post\} \\ \models pre \supset \mathrm{I}(V) \supset guar \qquad \models pre \perp rely\end{array}}{\{pre, rely\}\ \mathtt{IF}^i\ \varphi?S_0^i{:}S_1^i\ \{guar, post\}}\ \text{(L-IF)} \qquad \frac{\begin{array}{l}\models pre \supset \neg\varphi \supset post \\ \models \{pre \wedge \varphi, rely\}\ S^i\ \{guar, pre\} \\ \models pre \supset \mathrm{I}(V) \supset guar \\ \models pre \perp rely \qquad \models post \perp rely\end{array}}{\{pre, rely\}\ \mathtt{WL}^i\ \varphi?S^i\ \{guar, post\}}\ \text{(L-WL)}$$

$$\frac{\begin{array}{l}\{pre, rely\}\ S_0^i\ \{guar, \Phi\} \\ \{\Phi, rely\}\ S_1^i\ \{guar, post\}\end{array}}{\{pre, rely\}\ S_0^i; S_1^i\ \{guar, post\}}\ \text{(L-SQ)} \qquad \frac{\begin{array}{l}\{pre_0, rely_0\}\ P\ \{guar_0, post_0\} \\ \models pre \supset pre_0 \qquad \models rely \supset rely_0 \\ \models guar_0 \supset guar \qquad \models post_0 \supset post\end{array}}{\{pre, rely\}\ P\ \{guar, post\}}\ \text{(L-WK)}$$

$$\frac{\begin{array}{l}\{pre_0, rely_0\}\ P_0\ \{guar_0, post_0\} \qquad \{pre_1, rely_1\}\ P_1\ \{guar_1, post_1\} \\ \models rely \vee guar_0 \supset rely_1 \qquad \models rely \vee guar_1 \supset rely_0 \qquad \models guar_0 \vee guar_1 \supset guar\end{array}}{\{pre_0 \wedge pre_1, rely\}\ P_0 \parallel P_1\ \{guar, post_0 \wedge post_1\}}\ \text{(L-PR)}$$

Fig. 6. Our concurrent program logic

$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash E_0 = E_1 \Leftrightarrow [\![E_0]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = [\![E_1]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle}$$
$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash E_0 \le E_1 \Leftrightarrow [\![E_0]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} \le [\![E_1]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle}$$
$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \neg\,\Phi \Leftrightarrow \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \nvDash \Phi$$
$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \Phi \supset \Phi' \Leftrightarrow \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \Phi \text{ implies } \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \Phi'$$
$$\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \forall\, v.\,\Phi(v) \Leftrightarrow \langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \Phi(v') \text{ for any } v'$$

where $[\![val]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = val$ $\quad [\![r^i]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma^i r$

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

$[\![\underline{x}^i]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = \sigma'^i[\Sigma'^{i,i}]x$ $\quad [\![E_1 + E_2]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} = [\![E_1]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} + [\![E_2]\!]_{\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle} \dots$

Fig. 7. The interpretation of the assertion language

$$\lceil \mathtt{MV}^i\ r\ e \rceil^V \equiv \underline{r} = e \wedge \bigwedge \mathrm{I}(V \setminus \{r\}) \qquad \lceil \mathtt{LD}^i\ r\ x \rceil^V \equiv \underline{r} = x^i \wedge \bigwedge \mathrm{I}(V \setminus \{r\})$$
$$\lceil \mathtt{ST}^i\ x\ e \rceil^V \equiv \underline{x}^i = e \wedge \bigwedge \mathrm{I}(V \setminus \{\, x^j \mid 0 \le j < N \,\})$$

Fig. 8. Invariants about variables before and after assignments

More specifically, rule L-MV of Fig. 6 handles the substitution of thread-local variables with expressions. This is the same as in conventional rely/guarantee proof systems. $[e/v]$ represents the substitution of v with e. The first assumption means that *pre* must be a sufficient condition that implies *post* with respect to the substitution. We define $\vDash \Phi$ as $\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \Phi$ for any $\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle$, where $\langle\sigma,\Sigma\rangle,\langle\sigma',\Sigma'\rangle \vDash \Phi$ is defined in a similar manner to a conventional rely/guarantee system, as shown in Fig. 7. In the following, we often write $\langle\sigma,\Sigma\rangle \vDash \Phi$ when Φ has no next variable. The second assumption means that *pre* must be a sufficient condition that implies *guar* under an invariant about V before and after an execution of an assignment C (formally defined as $\lceil C \rceil^V$), where C is $\mathtt{MV}\ r\ e$, $\mathtt{LD}\ r\ x$, or $\mathtt{ST}\ x\ e$, and V is a finite set of non-next variables that occur in *guar*. A formula $\lceil \mathtt{MV}^i\ r\ e \rceil^V$ is defined as $\underline{r} = e \wedge \bigwedge \mathrm{I}(V \setminus \{r\})$, which means that the value of $\underline{r}$ is equal to the evaluation of e while the values of variables in $V \setminus \{r\}$ are assignment-invariant, where $\mathrm{I}(V)$ is $\{\, v = \underline{v} \mid v \in V \,\}$. Its formal definition is shown in Fig. 8. The third assumption means that *pre* and *post* are *stable under* the *rely* condition guaranteed by another thread, where we denote that Φ is *stable under* Ψ (written as $\Phi \perp \Psi$) as $\Phi(\boldsymbol{v}) \wedge \Psi(\boldsymbol{v}, \underline{\boldsymbol{v}}) \supset \Phi(\underline{\boldsymbol{v}})$, where $\boldsymbol{v}$ denotes a sequence of variables.

Rule L-SK states that an ordinary no-effect statement does not affect anything.

Rule L-LD handles the substitution of thread-local variables with shared variables. Note that r is substituted with the observation variables x^i, instead of the shared variables x. Rule L-ST handles the substitution of shared variables with expressions. Note that, as for L-LD, this rule considers the observation variable x^i instead of the shared variable x.

Rules L-IF and L-WL handle branch and loop statements, respectively. Careful readers may have noticed that Xu et al.'s papers [35,36] do not require

the third assumption, which is implicitly assumed because these logics adopt the restriction that rely/guarantee-conditions are *reflexive* (as in [19,29]). This restriction often makes it difficult to write down derivations. Therefore, in this paper, we do not adopt the restriction, following van Staden [33]. As suggested by Nieto in [19], reflexivity is used to ensure soundness. However, we do not adopt the reflexivity of rely/guarantee-conditions, but instead use the third assumption regarding L-IF and L-WL, which prohibits the so-called *stuttering transitions* [16], as explained in Sect. 6.

Rule L-SQ handles the sequential composition of programs. Rule L-WK is the so-called consequence rule. L-PR handles parallel compositions of programs in a standard rely/guarantee system. The third assumption means that P_1's rely-condition $rely_1$ must be guaranteed by the global rely-condition $rely$ or P_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 $guar_1$.

6 Validity for Judgments

We now define *computations* of programs, and *validity* for judgments.

We define the set of computations $Cmp(P)$ of P as a finite or infinite sequence c of configurations whose adjacent configurations are related by $\xrightarrow{\text{c}}$ or $\xrightarrow{\text{e}}$ defined in Sect. 4. We write $Cfg(c, i)$, $Prg(c, i)$, and $St(c, i)$ as the i-th configuration, program, and state of c, respectively. By definition, the program $Prg(c, 0)$ is P. As mentioned in Sect. 5, we do not assume that the rely/guarantee-conditions are reflexive. Therefore, our logic does not unconditionally ensure that the guarantee conditions hold on computations that contain $\langle P, st\rangle \xrightarrow{\text{e}} \langle P, st\rangle$, as Xu et al. noted in [36].

The length of a computation of c is denoted by $|c|$. If c is an infinite sequence, then $|c|$ is the smallest limit ordinal ω. 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 $\models \{pre, rely\} P \{guar, post\}$ as $Cmp(P) \cap A(pre, rely) \subseteq C(guar, post)$, which means that any computation under pre/rely-conditions satisfies guarantee/post-conditions, as shown in Fig. 9. Thus, this paper does not handle post-conditions in non-terminating computations. This kind of validity is called *partial correctness* [34].

Careful readers may have noticed that the second arguments of Σ and substitutions to $\Sigma^{i,j}$ $(i \neq j)$ at rule O-ST are redundant, as $\xrightarrow{\text{e}}$, which satisfies a rely-condition, is allowed at any time, and our assertion language cannot describe $\Sigma^{i,j}$ $(i \neq j)$. Strictly speaking, although technically unnecessary and redundant, we have adopted these arguments to explain admissible computations more intuitively. A computation that formally represents the non-intuitive behavior of IRIW *without* remote-write-atomicity in Sect. 9.3 may help readers understand how memories are updated by effects from buffers.

$$A(pre, rely) = \left\{ c \;\middle|\; \begin{array}{l} St(c,0) \models pre, \text{ and} \\ St(c,i), St(c,i+1) \models rely \text{ for any } Cfg(c,i) \xrightarrow{\text{e}} Cfg(c,i+1) \end{array} \right\}$$

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

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

7 Observation Invariant

In this section, we propose an *observation invariant*, which is an invariant written by observation variables. Formally, we define an observation invariant as a formula of the first-order predicate logic with the equations of Sect. 5.

We adopt observation invariants as axioms of the logic in Sect. 5. For example, let $x^0 = x^1$ be an observation invariant, which means that the value of x observed by thread 0 coincides with the value of x observed by thread 1. Adopting the observation invariant as an axiom means handling execution traces that always satisfy $\sigma^0[\Sigma^{0,0}]x = \sigma^1[\Sigma^{1,1}]x$.

Let us consider three examples of observation invariants. The program shown in Fig. 10 is called Dependence Cycle (DC). Although we intuitively think that either r_0 or r_1 has an initial value of 0, $r_0 = 1 \wedge r_1 = 1$ may not hold under a relaxed memory consistency model such as C++11 memory models. Memory consistency models for programming languages are often very relaxed in consideration of compiler optimization.

Our intuition that either r_0 or r_1 has an initial value is supported by *no speculation* regarding store statements on distinct threads, which is assumed under SPARC-PSO and similar architectures. For DC, this can be represented as $y^0 = 0 \supset y^1 = 0$, $x^1 \leq \underline{x^1} \supset x^0 \leq \underline{x^0}$, $x^1 = 0 \supset x^0 = 0$, and $y^0 \leq \underline{y^0} \supset y^1 \leq \underline{y^1}$ if the buffers are empty with respect to x and y when DC launches, and a rely-condition ensures no store operation to x and y. The first formula, $y^0 = 0 \supset y^1 = 0$, means that thread 1 observes $y = 0$ as long as thread 0 observes $y = 0$. This is because thread 0 is the only thread that has a store statement to y in DC. The second formula, $x^1 \leq \underline{x^1} \supset x^0 \leq \underline{x^0}$, means that thread 0 observes that x is *monotone* if thread 1 observes x is monotone. Thread 0 cannot observe x is not monotone, because thread 1 (which has a store statement to x and can see its own buffer) observes x is monotone. The third and fourth formulas are similar.

Next, let us consider an observation invariant for the One Reader One Writer (1R1W) program shown in Fig. 11, which consists of one reader thread and one writer thread. The reader thread in 1R1W has no store statement. Therefore, $\underline{y^1} \leq x^1 \supset \underline{y^0} \leq x^0$ is an observation invariant for 1R1W under x86-TSO [26]. This prohibits the reordering of effects of store statements, where we assume that the buffers are empty with respect to x and y when 1R1W launches, and a rely-condition ensures no store operations to x and y. Note that this is not an invariant under SPARC-PSO, which allows the effects of store statements

```
r0 = x;
y = 1;
```

```
r1 = y;
x = 1;
```

Fig. 10. Dependence cycle

```
r0 = y;
r1 = x;
```

```
x = 1;
y = 1;
```

Fig. 11. One reader one writer (1R1W)

to be reordered. The transfer of monotonicity $x^1 \le \underline{x}^1 \supset x^0 \le \underline{x}^0$ is also an observation invariant, even under SPARC-PSO, since the reader thread has no store statement to x.

Finally, let us consider an observation invariant for IRIW. Similar to DC, the transfer of monotonicity ($x^2 \le \underline{x}^2 \supset x^0 \le \underline{x}^0$, $x^2 \le \underline{x}^2 \supset x^1 \le \underline{x}^1$, $y^3 \le \underline{y}^3 \supset y^0 \le \underline{y}^0$, and $y^3 \le \underline{y}^3 \supset y^1 \le \underline{y}^1$) holds, because the reader threads in IRIW have no store statement. In addition, $\underline{x}^0 = \underline{x}^1$ and $\underline{y}^0 = \underline{y}^1$ are invariants under remote-write-atomicity (which is assumed under SPARC-PSO and similar architectures), as threads 0 and 1 can detect nothing in their own buffers, and share a common observation of a shared memory. Note that the invariants are properly weaker than the strict consistency assumed by conventional concurrent program logics [13,28,35,36], which forces the variable updates to be immediately observed by all threads, that is, $\underline{x}^0 = \underline{x}^1 = \underline{x}^2 = \underline{x}^3$ and $\underline{y}^0 = \underline{y}^1 = \underline{y}^2 = \underline{y}^3$.

8 Soundness

In this section, we present the soundness of the operational semantics defined in Fig. 5. In Sect. 5, we derived a concurrent program logic that is sound to the operational semantics defined in Fig. 5. However, the logic is actually insufficient to derive some valid judgments.

Auxiliary variables are known to enhance the provability of concurrent program logics [28]. Auxiliary variables are fresh variables that do not occur in the original programs, and are used only for the description of assertions. By using auxiliary variables in assertions, we can describe the progress of thread executions as rely/guarantee-conditions. In Sect. 9.3, we show a typical usage of auxiliary variables.

We extend our logic to contain the following inference rule (called *the auxiliary variables rule* [28,35]):

$$\frac{\begin{array}{ll}\{\mathit{pre} \wedge \mathit{pre}_0, \mathit{rely} \wedge \mathit{rely}_0\}\ P_0\ \{\mathit{guar}, \mathit{post}\} & \models \exists\, \underline{\boldsymbol{z}}.\ \mathit{rely}_0((\boldsymbol{v}, \boldsymbol{z}), (\underline{\boldsymbol{v}}, \underline{\boldsymbol{z}})) \\ \models \exists\, \boldsymbol{z}.\ \mathit{pre}_0(\boldsymbol{v}, \boldsymbol{z}) & \boldsymbol{z} \cap (\mathrm{fv}(\mathit{pre}) \cup \mathrm{fv}(\mathit{rely}) \cup \mathrm{fv}(\mathit{guar}) \cup \mathrm{fv}(\mathit{post})) = \varnothing\end{array}}{\{\mathit{pre}, \mathit{rely}\}\ (P_0)_{\boldsymbol{z}}\ \{\mathit{guar}, \mathit{post}\}}\ \text{(L-AX)}$$

where $\mathrm{fv}(\Phi)$ denotes free variables that occur in Φ in a standard manner, and $(P_0)_{\boldsymbol{z}}$ is defined as the program that coincides with P_0, except that an assignment A is removed if

– A is an assignment whose left value belongs to $\boldsymbol{z}$,

– no variable in $\boldsymbol{z}$ occurs in assignments whose left values do not belong to $\boldsymbol{z}$, and
– no variable in $\boldsymbol{z}$ freely occurs in conditional statements.

Let c be $\langle P_0, \langle \sigma_0, \Sigma_0 \rangle \rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{i-1}} \langle P_i, \langle \sigma_i, \Sigma_i \rangle \rangle \xrightarrow{\delta_i} \cdots$ for any $0 \leq i$. We write $tr(c, i)$ as δ_i. Given $c \in Cmp(P_0 \parallel P_1)$, $c_0 \in Cmp(P_0)$, and $c_1 \in Cmp(P_1)$, 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. $Prg(c, i) = Prg(c_0, i) \parallel Prg(c_1, i)$

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

Proposition 1. $Cmp(P_0 \parallel P_1) = \{ c_0 \parallel c_1 \mid c_0 \in Cmp(P_0), c_1 \in Cmp(P_1) \}$.

Lemma 2. *Assume* $\vdash \{pre_0 \wedge pre_1, rely\}\ P_0 \parallel P_1\ \{guar, post_0 \wedge post_1\}$ *by L-PR,* $Cmp(P_0) \cap A(pre_0, rely_0) \subseteq C(guar_0, post_0)$*,* $Cmp(P_1) \cap A(pre_1, rely_1) \subseteq C(guar_1, post_1)$*,* $\vDash rely \vee guar_0 \supset rely_1$*,* $\vDash rely \vee guar_1 \supset rely_0$*,* $\vDash guar_0 \vee guar_1 \supset guar$*, and* $c \in Cmp(P_0 \parallel P_1) \cap A(pre_0 \wedge pre_1, rely)$*. In addition, we take* $c_0 \in Cmp(P_0)$ *and* $c_1 \in Cmp(P_1)$ *such that* $c = c_0 \parallel c_1$ *by Proposition 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 $0 \leq j < i$.

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 $0 \leq k \leq i$. Therefore, $St(c, k), St(c, k + 1) \vDash rely \vee guar_1$ holds. By $\vDash rely \vee guar_1 \supset rely_0$, $c_0 i + 1 \in A(pre_0, rely_0)$ holds. Since $Cmp(P_0) \cap A(pre_0, rely_0) \subseteq C(guar_0, post_0)$ 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 $Cmp(P_0) \cap A(pre_0, rely_0) \subseteq C(guar_0, post_0)$ and $Cmp(P_1) \cap A(pre_1, rely_1) \subseteq C(guar_1, post_1)$, $St(c, |c|) \vDash post_0$ and $St(c, |c| - 1) \vDash post_1$ hold. Therefore, $St(c, |c| - 1) \vDash post_0 \wedge post_1$ holds. □

Theorem 3. $\vdash \{pre, rely\}\ P\ \{guar, post\}$ *implies* $\vDash \{pre, rely\}\ P\ \{guar, post\}$.

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

First, assume L-ST. Let $c \in Cmp(\mathtt{ST}^i\, x\, e) \cap A(pre, rely)$. By O-ST, there exist $\sigma_0, \Sigma_0, \ldots$ such that $\langle \sigma_{n+1}, \Sigma_{n+1}\rangle = \langle \sigma_n, \Sigma[i := \Sigma^i[j := \Sigma^{i,j}[x := (\!|e|\!)_{\sigma^i}] \mid 0 \leq j < N]]\rangle$,

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

$\langle \sigma_0, \Sigma_0\rangle \vDash pre$, and $\langle \sigma_j, \Sigma_j\rangle, \langle \sigma_{j+1}, \Sigma_{j+1}\rangle \vDash rely$ for any $0 \leq j < n$. By $\vDash pre \perp rely$, $\langle \sigma_n, \Sigma_n\rangle \vDash pre$. By the definition, $\langle \sigma_n, \Sigma_n\rangle, \langle \sigma_n, \Sigma[i := \Sigma^i[j := \Sigma^{i,j}[x := (\!|e|\!)_{\sigma^i}] \mid 0 \leq j < N]]\rangle \vDash \lceil \mathtt{ST}^i\ x\ e\rceil^V$. By $\vDash pre \supset \lceil \mathtt{ST}^i\ x\ e\rceil^V \supset guar$, $\langle \sigma_n, \Sigma_n\rangle, \langle \sigma_n, \Sigma[i := \Sigma^i[j := \Sigma^{i,j}[x := (\!|e|\!)_{\sigma^i}] \mid 0 \leq j < N]]\rangle \vDash guar$, that is, $\langle \sigma_n, \Sigma_n\rangle, \langle \sigma_{n+1}, \Sigma_{n+1}\rangle \vDash guar$. In addition, assume $|c| < \omega$. By $\vDash pre \supset [e/x^i]post$, $\langle \sigma_n, \Sigma_n\rangle \vDash [e/x^i]post$. By the definition, $\langle \sigma_n, \Sigma[i := \Sigma^i[j := \Sigma^{i,j}[x := (\!|e|\!)_{\sigma^i}] \mid 0 \leq j < N]]\rangle \vDash post$, that is, $\langle \sigma_{n+1}, \Sigma_{n+1}\rangle \vDash post$. By $\vDash post \perp rely$, $\langle \sigma_{|c|-1}, \Sigma_{|c|-1}\rangle \vDash post$.

Second, assume L-WL. Let $c \in Cmp(\mathtt{WL}^i\ \varphi ? S_0^i) \cap A(pre, rely)$, which consists of the following five segments:

- $\langle S^i, \langle \sigma_{k_n}, \Sigma_{k_n}\rangle\rangle \xrightarrow{\mathrm{e}}{}^* \langle S^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle$,
- $\langle S^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle \xrightarrow{\mathrm{c}} \langle S_0^i; S^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle$ where $\sigma_{k_0} \vDash \varphi$,
- $\langle S^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle \xrightarrow{\mathrm{c}} \langle \mathtt{SK}^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle \xrightarrow{\mathrm{e}} \cdots$ where $\sigma_{k_0} \nvDash \varphi$,
- $\langle S_0^i; S^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle \longrightarrow^* \langle S^i, \langle \sigma_{k_n}, \Sigma_{k_n}\rangle\rangle$.
- $\langle S_0^i; S^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle \longrightarrow \cdots$ which does not reach S^i.

where $\langle \sigma', \Sigma'\rangle, \langle \sigma'', \Sigma''\rangle \vDash rely$ for any $\langle S', \langle \sigma', \Sigma'\rangle\rangle \xrightarrow{\mathrm{e}} \langle S'', \langle \sigma'', \Sigma''\rangle\rangle$ in the five segments. By $\vDash pre \perp rely$, $\langle \sigma_{k_0}, \Sigma_{k_0}\rangle \vDash pre$. Let c' be $\langle S_0^i, \langle \sigma_{k_0}, \Sigma_{k_0}\rangle\rangle \longrightarrow^* \langle \mathtt{SK}^i, \langle \sigma_{k_n}, \Sigma_{k_n}\rangle\rangle$. By induction hypothesis, $\langle \sigma', \Sigma'\rangle, \langle \sigma'', \Sigma''\rangle \vDash guar$ holds for any $\langle S', \langle \sigma', \Sigma'\rangle\rangle \xrightarrow{\mathrm{c}} \langle S'', \langle \sigma'', \Sigma''\rangle\rangle$ in c' holds. The case that c does not reach S^i is similar. Therefore, since $\vDash pre \supset \mathrm{I}(V) \supset guar$ holds, $\langle \sigma', \Sigma'\rangle, \langle \sigma'', \Sigma''\rangle \vDash guar$ holds for any $\langle S', \langle \sigma', \Sigma'\rangle\rangle \xrightarrow{\mathrm{c}} \langle S'', \langle \sigma'', \Sigma''\rangle\rangle$ in c holds. In addition, assume $|c| < \omega$. By $\vDash pre \perp rely$ and induction hypothesis, $\langle \sigma_{k_0}, \Sigma_{k_0}\rangle \vDash pre$ holds. By $\vDash pre \supset \neg\varphi \supset post$ and $\vDash post \perp rely$, $St(c, |c|-1) \vDash post$.

Third, assume L-SQ. Let $c \in Cmp(P_0^i; P_1^i) \cap A(pre, rely)$. There exist $st_0, \delta_0, \ldots$ such that

$$c = \langle P_0^i; P_1^i, st_0\rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle P_1^i, st_n\rangle \xrightarrow{\delta_n} \cdots,$$

$st_0 \vDash pre$, and $st_j, st_{j+1} \vDash rely$ for any $0 \leq j < n$. Let c' and c'' be $\langle P_0^i, st_0\rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle \mathtt{SK}^i, st_n\rangle$ and $\mathrm{postfix}(c, n)$, respectively. Obviously, $c' \in Cmp(P_0^i) \cap A(pre, rely)$ holds. By induction hypothesis, $c' \in C(guar, \Phi)$ holds. By the definition, $\langle \sigma_n, \Sigma_n\rangle \vDash \Phi$ holds. Therefore, $c'' \in Cmp(P_1^i) \cap A(\Phi, rely)$ holds. By induction hypothesis, $c'' \in C(guar, post)$ holds. Therefore, $c \in C(guar, post)$ holds.

Fourth, assume L-PR. By Lemmas 2.3 and 2.4.

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

$$c = \langle (P)_z, \langle \sigma_0, \Sigma_0\rangle\rangle \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{n-1}} \langle P_n, \langle \sigma_n, \Sigma_n\rangle\rangle \xrightarrow{\delta_n} \cdots,$$

$$\dfrac{\dfrac{\{pre_0, rely^0\}\ \mathtt{LD}^0\ r_0\ x\ \{guar^0, post_0\} \quad \{pre_1, rely^0\}\ \mathtt{ST}^0\ y\ 1\ \{guar^0, post_1\}}{\{pre_0, rely^0\}\ \mathtt{LD}^0\ r_0\ x; \mathtt{ST}^0\ y\ 1\ \{guar^0, post_1\}} \quad \dfrac{\{pre_2, rely^0\}\ \mathtt{LD}^1\ r_1\ y\ \{guar^1, post_2\} \quad \{pre_3, rely^1\}\ \mathtt{ST}^1\ x\ 1\ \{guar^1, post_3\}}{\{pre_2, rely^1\}\ \mathtt{LD}^1\ r_1\ y; \mathtt{ST}^1\ x\ 1\ \{guar^1, post_3\}}}{\{pre_0 \land pre_2, rely^0 \land rely^1\}\ \mathtt{LD}^0\ r_0\ x; \mathtt{ST}^0\ y\ 1 \parallel \mathtt{LD}^1\ r_1\ y; \mathtt{ST}^1\ x\ 1\ \{guar^0 \lor guar^1, post_1 \land post_3\}}$$

Fig. 12. A essential part of a derivation for DC

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

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

and $P'_0 = P$, $(P'_n)_{\boldsymbol{z}} = P_n$, ${\sigma'}_j^i v = {\sigma_j}^i v$, ${\Sigma'}_j^{i,i'} v = \Sigma_j^{i,i'} v$, $\langle \sigma'_0, \Sigma'_0 \rangle \models pre \land pre_0$, and $\langle \sigma'_j, \Sigma'_j \rangle, \langle \sigma'_{j+1}, \Sigma'_{j+1} \rangle \models rely \land rely_0$ for any $v \notin \boldsymbol{z}$, $0 \leq i, i' < N$ and $0 \leq j < n$. Therefore, $c' \in Cmp(P) \cap A(pre \land pre_0, rely \land rely_0)$ holds. By induction hypothesis, $c' \in C(guar, post)$ holds. Therefore, $c \in C(guar, post)$ holds.

The other cases are similar and omitted due to space limitation. □

9 Examples

In this section, we verify several example racy programs.

9.1 Verification of DC

The first example program is DC, introduced in Sect. 7. The verification property, a judgment consisting of the post-condition $r_0 = 0 \lor r_1 = 0$ under appropriate pre/rely-conditions, is shown with a derivation for DC.

Figure 12 shows an essential part of a derivation for DC, where

$$\begin{aligned}
pre_0 &\equiv y^0 = 0 \land ((x^0 = 0 \land r_0 = 0) \lor (x^0 = 1 \land r_1 = 0)) \\
pre_1 \equiv post_0 &\equiv y^0 = 0 \land (r_0 = 0 \lor (x^0 = 1 \land r_1 = 0)) \\
post_1 &\equiv y^0 = 1 \land (r_0 = 0 \lor (x^0 = 1 \land r_1 = 0)) \\
rely^0 &\equiv (y^0 = 0 \lor r_0 = 1 \supset \underline{r_1} = 0) \land x^0 \leq \underline{x^0} \land \mathrm{I}\{y^0, r_1\} \land D \land \underline{D} \\
guar^0 &\equiv (x^0 = 0 \lor r_1 = 1 \supset \underline{r_0} = 0) \land y^0 \leq \underline{y^0} \land \mathrm{I}\{x^0, x^1, r_1\} \land D \land \underline{D} \\
pre_2 &\equiv x^1 = 0 \land ((y^1 = 0 \land r_1 = 0) \lor (y^1 = 1 \land r_0 = 0)) \\
pre_3 \equiv post_2 &\equiv x^1 = 0 \land (r_1 = 0 \lor (y^1 = 1 \land r_0 = 0)) \\
post_3 &\equiv x^1 = 1 \land (r_1 = 0 \lor (y^1 = 1 \land r_0 = 0)) \\
rely^1 &\equiv (x^1 = 0 \lor r_1 = 1 \supset \underline{r_0} = 0) \land y^1 \leq \underline{y^1} \land \mathrm{I}\{x^1, r_1\} \land D \land \underline{D} \\
guar^1 &\equiv (y^1 = 0 \lor r_0 = 1 \supset \underline{r_1} = 0) \land x^1 \leq \underline{x^1} \land \mathrm{I}\{y^0, y^1 r_1\} \land D \land \underline{D}
\end{aligned}$$

$$\dfrac{\dfrac{\{y^0 \le x^0, rely^0\}\ \mathtt{LD}^0\ r_0\ y\ \{guar^0, r_0 \le x^0\} \quad \{r_0 \le x^0, rely^0\}\ \mathtt{LD}^0\ r_1\ x\ \{guar^0, r_0 \le r_1\}}{\{y^0 \le x^0, rely^0\}\ 1\mathrm{R}\ \{guar^0, r_0 \le r_1\}} \quad \dfrac{\{x^1 = 0 \land y^1 = 0, rely^1\}\ \mathtt{ST}^1\ x\ 1\ \{guar^1, x^1 = 1\} \quad \{x^1 = 1, rely^1\}\ \mathtt{ST}^1\ y\ 1\ \{guar^1, \top\}}{\{x^1 = 0 \land y^1 = 0, rely^1\}\ 1\mathrm{W}\ \{guar^1, \top\}}}{\{y^0 \le x^0 \land x^1 = 0 \land y^1 = 0, rely\}\ 1\mathrm{R} \parallel 1\mathrm{W}\ \{guar, r_0 \le r_1 \land \top\}}$$

Fig. 13. An essential part of a derivation for 1R1W

and D, $\underline{D}$ are $\bigwedge\{ (x^i = 0 \lor x^i = 1) \land (y^i = 0 \lor y^i = 1) \mid 0 \le i < 4\}$ and $\bigwedge\{ (\underline{x^i} = 0 \lor \underline{x^i} = 1) \land (\underline{y^i} = 0 \lor \underline{y^i} = 1) \mid 0 \le i < 4\}$, respectively. Some assumptions regarding the inference rules are omitted when the context renders them obvious.

A key point is that $\models (rely^0 \land rely^1) \lor guar^0 \supset rely^1$ and $\models (rely^0 \land rely^1) \lor guar^1 \supset rely^0$ are derived from the observation invariants for DC, $y^0 = 0 \supset y^1 = 0$, $x^1 \le \underline{x^1} \supset x^0 \le \underline{x^0}$, $x^1 = 0 \supset x^0 = 0$, and $y^0 \le \underline{y^0} \supset y^1 \le \underline{y^1}$ introduced in Sect. 7 at the final inference by L-PR.

9.2 Verification of 1R1W

Let us consider a relaxed memory consistency model that prohibits the reordering of the effects of store statements. Therefore, we expect $r_0 \le r_1$ under an appropriate condition when the program in Fig. 11 finishes.

Figure 13 shows an essential part of a derivation for 1R1W, where

$$rely^0 \equiv \underline{y^0} \le x^0 \land x^0 \le \underline{x^0} \land \mathrm{I}\{r_0, r_1\} \quad guar^0 \equiv \mathrm{I}\{x^1, y^1\}$$
$$rely^1 \equiv \mathrm{I}\{x^1, y^1\} \quad guar^1 \equiv \underline{y^1} \le x^1 \land x^1 \le \underline{x^1} \land \mathrm{I}\{r_0, r_1\}$$

$1\mathrm{R} \equiv \mathtt{LD}^0\ r_0\ y; \mathtt{LD}^0\ r_1\ x$, $1\mathrm{W} \equiv \mathtt{ST}^1\ x\ 1; \mathtt{ST}^1\ y\ 1$, and some assumptions of the inference rules are omitted when the context renders them obvious.

A key point here is that $\models (rely^0 \land rely^1) \lor guar^0 \supset rely^1$ and $\models (rely^0 \land rely^1) \lor guar^1 \supset rely^0$ are derived from the observation invariants for 1R1W, $\underline{y^1} \le x^1 \supset \underline{y^0} \le x^0$ and $x^1 \le \underline{x^1} \supset x^0 \le \underline{x^0}$ introduced in Sect. 7 at the final inference by L-PR.

As explained in Sect. 7, under SPARC-PSO, $\models (rely^0 \land rely^1) \lor guar^1 \supset rely^0$ is not implied, since $y^1 \le x^1 \supset y^0 \le x^0$ is not an observation invariant.

9.3 Verification of IRIW

Finally, we demonstrate the verification of the program introduced in Sect. 1. The verification property is a judgment consisting of the post-condition $r_0 \le r_1 \lor r_2 \le r_3$ under appropriate pre/rely-conditions, although the judgment is formally shown as (2) in this section since the pre/rely-conditions require some notation.

First, note that the post-condition does not always hold without axioms for remote-write-atomicity. Actually, the following computation:

$$
\begin{array}{l}
\langle \mathtt{LD}^0\ r_0\ x; \mathtt{LD}^0\ r_1\ y \parallel \mathtt{LD}^1\ r_2\ y; \mathtt{LD}^1\ r_3\ x \parallel \mathtt{ST}^2\ x\ 1 \parallel \mathtt{ST}^3\ y\ 1, \langle \sigma, \Sigma \rangle \rangle \\
\xrightarrow{\mathrm{c}}{}^{*} \langle \mathtt{LD}^0\ r_0\ x; \mathtt{LD}^0\ r_1\ y \parallel \mathtt{LD}^1\ r_2\ y; \mathtt{LD}^1\ r_3\ x, \langle \sigma, \\
\quad \Sigma[x^{2,0} \mapsto 1, y^{3,0} \mapsto 1, x^{2,1} \mapsto 1, y^{3,1} \mapsto 1, x^{2,2} \mapsto 1, y^{3,2} \mapsto 1, x^{2,3} \mapsto 1, y^{3,3} \mapsto 1] \rangle \rangle \\
\xrightarrow{\mathrm{e}}{}^{*} \langle \mathtt{LD}^0\ r_0\ x; \mathtt{LD}^0\ r_1\ y \parallel \mathtt{LD}^1\ r_2\ y; \mathtt{LD}^1\ r_3\ x, \langle \sigma[x^0 \mapsto 1, y^1 \mapsto 1], \\
\quad \Sigma[y^{3,0} \mapsto 1, x^{2,1} \mapsto 1, x^{2,2} \mapsto 1, y^{3,2} \mapsto 1, x^{2,3} \mapsto 1, y^{3,3} \mapsto 1] \rangle \rangle \\
\xrightarrow{\mathrm{c}}{}^{*} \langle \mathtt{SK}, \langle \sigma[{r_0}^0 \mapsto 1, {r_1}^0 \mapsto 0, {r_2}^1 \mapsto 1, {r_3}^1 \mapsto 0, x^0 \mapsto 1, y^1 \mapsto 1], \\
\quad \Sigma[y^{3,0} \mapsto 1, x^{2,1} \mapsto 1, x^{2,2} \mapsto 1, y^{3,2} \mapsto 1, x^{2,3} \mapsto 1, y^{3,3} \mapsto 1] \rangle \rangle
\end{array}
$$

implies this fact, where we write the substitutions $[v^i \mapsto n]$ and $[v^{i,j} \mapsto n]$ as $[i := \sigma^i[v := n]]$ and $[i := \Sigma^i[j := \Sigma^{i,j}[v := n]]]$, respectively, for readability. Additionally, σ and Σ are constant functions to 0 and $\mathtt{udf}$, respectively. Note that we must confirm that $\xrightarrow{\mathrm{e}}$ satisfies the rely-condition of (2).

Thus, the post-condition does not always hold with no additional axiom. Let us show that the post-condition holds under appropriate pre/rely/guarantee-conditions with axioms for remote-write-atomicity. To construct a derivation, we add the auxiliary variables z_0 and z_1, as shown in Fig. 14.

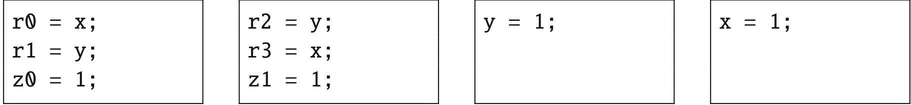

Fig. 14. IRIW with auxiliary variables

We construct a derivation on each thread. The following three judgments:

$$
\left\{ \begin{array}{c} z_0 = 0 \land (x^0 \leq y^0 \lor \\ (z_1 = 1 \supset r_2 \leq r_3)), \mathit{rely}^0 \end{array} \right\} \mathtt{LD}\ r_0\ x \left\{ \begin{array}{c} \mathit{guar}^0, z_0 = 0 \land r_0 \leq x^0 \land \\ ((x^0 \leq y^0 \land r_0 \leq y^0) \lor (z_1 = 1 \supset r_2 \leq r_3)) \end{array} \right\}
$$

$$
\left\{ \begin{array}{c} z_0 = 0 \land r_0 \leq x^0 \land \\ ((x^0 \leq y^0 \land r_0 \leq y^0) \lor \\ (z_1 = 1 \supset r_2 \leq r_3)), \mathit{rely}^0 \end{array} \right\} \mathtt{LD}\ r_1\ y \left\{ \begin{array}{c} \mathit{guar}^0, z_0 = 0 \land r_0 \leq x^0 \land r_1 \leq y^0 \land \\ ((x^0 \leq y^0 \land r_0 \leq r_1) \lor \\ (z_1 = 1 \supset r_2 \leq r_3)) \end{array} \right\}
$$

$$
\left\{ \begin{array}{c} z_0 = 0 \land r_0 \leq x^0 \land r_1 \leq y^0 \land \\ ((x^0 \leq y^0 \land r_0 \leq r_1) \lor \\ (z_1 = 1 \supset r_2 \leq r_3)), \mathit{rely}^0 \end{array} \right\} z_0 = 1 \left\{ \begin{array}{c} \mathit{guar}^0, z_0 = 1 \land r_0 \leq x^0 \land r_1 \leq y^0 \land \\ ((x^0 \leq y^0 \land r_0 \leq r_1) \lor \\ (z_1 = 1 \supset r_2 \leq r_3)) \end{array} \right\}
$$

are derived by L-LD and L-MV, where $\mathrm{M}(V)$ is $\bigwedge\{ v \leq \underline{v} \mid v \in V \}$, and

$$
\begin{array}{l}
\mathit{rely}^0 \equiv (\underline{y^0} < \underline{x^0} \lor (z_0 = 1 \land r_1 < r_0) \supset \underline{z_1} = 1 \supset \underline{r_2} \leq \underline{r_3}) \land \mathrm{M}\{x^0, y^0, z_1\} \land \mathrm{I}\{z_0, r_0, r_1\} \\
\mathit{guar}^0 \equiv (\underline{x^0} < \underline{y^0} \lor (z_1 = 1 \land r_3 < r_2) \supset \underline{z_0} = 1 \supset \underline{r_0} \leq \underline{r_1}) \land \\
\qquad \mathrm{I}\{z_1, x^1, x^2, x^3, y^1, y^2, y^3, r_2, r_3\}.
\end{array}
$$

$$\left\{\begin{matrix} z_0 = 0 \wedge (x^0 \leq y^0 \vee \\ (z_1 = 1 \supset r_2 \leq r_3)), rely^0 \end{matrix}\right\} \begin{matrix} \texttt{LD}\ r_0\ x; \texttt{LD}\ r_1\ y; \\ z_0 = 1 \end{matrix} \left\{\begin{matrix} guar^0, z_0 = 1 \wedge ((x^0 \leq y^0 \wedge r_0 \leq r_1) \vee \\ (z_1 = 1 \supset r_2 \leq r_3)) \end{matrix}\right\}$$

is derivable by L-SQ. Similarly, so is

$$\left\{\begin{matrix} z_1 = 0 \wedge (x^1 \leq y^1 \vee \\ (z_0 = 1 \supset r_0 \leq r_1)), rely^1 \end{matrix}\right\} \begin{matrix} \texttt{LD}\ r_2\ y; \texttt{LD}\ r_3\ x; \\ z_1 = 1 \end{matrix} \left\{\begin{matrix} guar^1, z_1 = 1 \wedge ((x^1 \leq y^1 \wedge r_2 \leq r_3) \vee \\ (z_0 = 1 \supset r_0 \leq r_1)) \end{matrix}\right\}$$

from symmetricity, where

$$\begin{aligned} rely^1 &\equiv (\underline{x^1} < \underline{y^1} \vee (z_1 = 1 \wedge r_3 < r_2) \supset \underline{z_0} = 1 \supset \underline{r_0} \leq \underline{r_1}) \wedge \mathrm{M}\{x^1, y^1, z_0\} \wedge \mathrm{I}\{z_1, r_2, r_3\} \\ guar^1 &\equiv (\underline{y^1} < \underline{x^1} \vee (z_0 = 1 \wedge r_1 < r_0) \supset \underline{z_1} = 1 \supset \underline{r_2} \leq \underline{r_3}) \wedge \\ &\quad \mathrm{I}\{z_0, x^0, x^2, x^3, y^0, y^2, y^3, r_0, r_1\}. \end{aligned}$$

Let D and $\underline{D}$ be $\bigwedge\{ (x^i = 0 \vee x^i = 1) \wedge (y^i = 0 \vee y^i = 1) \mid 0 \leq i < 4\}$ and $\bigwedge\{ (\underline{x^i} = 0 \vee \underline{x^i} = 1) \wedge (\underline{y^i} = 0 \vee \underline{y^i} = 1) \mid 0 \leq i < 4\}$, respectively. Note that $v < v' \wedge D \wedge \underline{D}$ means $v = 0 \wedge v' = 1$.

By L-ST, $\{D, rely^2\}\ \texttt{ST}\ y\ 1\ \{guar^2, \top\}$ and $\{D, rely^3\}\ \texttt{ST}\ x\ 1\ \{guar^3, \top\}$ are derivable, where

$$\begin{aligned} rely^2 &\equiv x^2 \leq \underline{x^2} \wedge D \wedge \underline{D} \qquad rely^3 \equiv y^3 \leq \underline{y^3} \wedge D \wedge \underline{D} \\ guar^2 &\equiv y^2 \leq \underline{y^2} \wedge y^3 \leq \underline{y^3} \wedge \mathrm{I}\{x^0, x^1, x^2, x^3, r_0, r_1, r_2, r_3\} \wedge D \wedge \underline{D} \\ guar^3 &\equiv x^2 \leq \underline{x^2} \wedge x^3 \leq \underline{x^3} \wedge \mathrm{I}\{y^0, y^1, y^2, y^3, r_0, r_1, r_2, r_3\} \wedge D \wedge \underline{D}. \end{aligned}$$

Let us construct separate derivations corresponding to Independent Reads (IR), Independent Writes (IW), and IRIW. To construct a derivation for IR, it is sufficient that

$$\models (rely^0 \wedge rely^1) \vee guar^0 \supset rely^1 \qquad \models (rely^0 \wedge rely^1) \vee guar^1 \supset rely^0 \tag{1}$$

is satisfied, as this implies

$$\left\{\begin{matrix} z_0 = 0 \wedge z_1 = 0 \wedge \\ pre_{01}, \\ rely^0 \wedge rely^1 \end{matrix}\right\} \begin{matrix} \texttt{LD}\ r_0\ x; \texttt{LD}\ r_1\ y; \\ z_0 = 1 \end{matrix} \Bigg\| \begin{matrix} \texttt{LD}\ r_2\ y; \texttt{LD}\ r_3\ x; \\ z_1 = 1 \end{matrix} \left\{\begin{matrix} guar^0 \vee guar^1, \\ z_0 = 1 \wedge z_1 = 1 \wedge \\ (r_0 \leq r_1 \vee r_2 \leq r_3) \end{matrix}\right\}$$

under L-PR and L-WK, where $pre_{01} \equiv x^0 = 0 \wedge y^0 = 0 \wedge x^1 = 0 \wedge y^1 = 0$. Therefore, we can deduce

$$\{pre_{01}, rely^{01}\}\ \texttt{LD}\ r_0\ x; \texttt{LD}\ r_1\ y \parallel \texttt{LD}\ r_2\ y; \texttt{LD}\ r_3\ x\ \{guar^0 \vee guar^1, r_0 \leq r_1 \vee r_2 \leq r_3\}$$

by L-AX and L-WK, where $rely^{01} \equiv \mathrm{M}\{x^0, y^0, x^1, y^1\} \wedge \mathrm{I}\{r_0, r_1, r_2, r_3\}$.

Similarly, to construct a derivation for IW, it is sufficient that

$$\models (rely^2 \wedge rely^3) \vee guar^2 \supset rely^3 \qquad \models (rely^2 \wedge rely^3) \vee guar^3 \supset rely^2$$

is satisfied, since this allows us to deduce that $\{D, rely^2 \wedge rely^3\}\ \{\texttt{ST}\ y\ 1 \parallel \texttt{ST}\ x\ 1, guar^2 \vee guar^3\}\top$.

We now construct the following derivation for IRIW

$$\{pre_{01} \wedge D, rely^{01} \wedge rely^{2} \wedge rely^{3}\} \text{ IRIW } \{\bigvee\{\, guar^{i} \mid 0 \leq i < 4\,\}, r_0 \leq r_1 \vee r_2 \leq r_3\} \tag{2}$$

it is sufficient that the following is satisfied:

$$\vDash guar^0 \vee guar^1 \supset rely^2 \wedge rely^3 \qquad \vDash guar^2 \vee guar^3 \supset rely^{01}. \tag{3}$$

Let us recall the observation invariants $\underline{x^0} = \underline{x^1}$ and $\underline{y^0} = \underline{y^1}$ under the remote-write-atomicity explained in Sect. 7. Obviously, the observation invariants imply (1). Additionally, the transfer of monotonicity implies (3). Thus, under remote-write-atomicity, which is more relaxed than strict consistency (and therefore under SPARC-PSO), IRIW is guaranteed to work correctly.

10 Conclusion and Future Work

This paper has proposed the notion of observation invariants to fill the gap between theoretical and realistic relaxed memory consistency models. We have derived general small-step operational semantics for relaxed memory consistency models, introduced additional variables x^i to denote a value of x observed by i in an assertion language, and stated a concurrent program logic that is sound with respect to the operational semantics. Our analysis suggests that the non-existence of shared variables without observations by threads in the assertion language ensures the soundness. We have successfully constructed a formal proof for the correctness of IRIW via the notion of observation invariants. To the best of our knowledge, the derivation in this paper is the first to verify IRIW in a logic that handles relaxed memory consistency models like SPARC-PSO.

There are four directions for future work. The first is to invent systematic construction of observation invariants and to find further applications of observation invariants. The observation invariants shown in this paper are given in ad-hoc ways. The example programs that are verified in this paper are small. Systematic construction of observation invariants will tame observation invariants for larger programs, provide further applications of observation invariants, and enable us to compare our method with existing methods. The second is to implement a theorem prover that can verify programs in the logic in this paper. Manual constructions of derivations, which are done in this paper, are tedious and error-prone. The third is to compare our logic with *doxastic logic* [18], which is based on the notion of *belief*. We introduced the additional variable x^i to denote x as observed by thread i, but this variable does not always coincide with x on physical memories. Therefore, x^i may be considered to be x as *believed* by thread i. The fourth is a mathematical formulation of our logic. Although mathematical formulations of rely/guarantee-reasoning have been stated in some studies (e.g., [9]), they assume that (program) shared variables are components in assertion languages (called a *cheat* in [9]). Since the insight provided in this paper dismisses shared variables from assertion languages, the assumption cannot be admissible, and a new mathematical formulation of our logic based on rely/guarantee-reasoning is significant.

Acknowledgments. Some definitions in this paper are inspired by Qiwen Xu's PhD thesis [35]. The authors would like to thank him for answering our questions respectfully. The authors also thank the anonymous reviewers for several comments to improve the paper. This work was supported by JSPS KAKENHI Grant Number 16K21335.

References

1. Abe, T., Maeda, T.: Concurrent program logic for relaxed memory consistency models with dependencies across loop iterations. J. Inf. Process. (2016, to appear)
2. Abe, T., Maeda, T.: A general model checking framework for various memory consistency models. Int. J. Softw. Tools Technol. Transferr (2016, to appear). doi:10.1007/s10009-016-0429-y
3. Abe, T., Ugawa, T., Maeda, T., Matsumoto, K.: Reducing state explosion for software model checking with relaxed memory consistencymodels. In: Proceedings of SETTA. LNCS, vol. 9984 (2016, to appear). doi:10.1007/978-3-319-47677-3_8
4. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of PLDI, pp. 68–78 (2008)
5. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Proceedings of POPL, pp. 392–403 (2009)
6. Boudol, G., Petri, G., Serpette, B.P.: Relaxed operational semantics of concurrent programming languages. In: Proceedings of EXPRESS/SOS, pp. 19–33 (2012)
7. Ferreira, R., Feng, X., Shao, Z.: Parameterized memory models and concurrent separation logic. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 267–286. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11957-6_15
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580, 583 (1969)
9. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebraic Program **80**(6), 266–296 (2011)
10. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003)
11. Intel Corporation: A Formal Specification of Intel Itanium Processor Family Memory Ordering (2002)
12. ISO, IEC 14882: 2011: Programming Language C++ (2011)
13. Jones, C.B.: Development methods for computer programs including a notion of interference. Ph.D. thesis, Oxford University (1981)
14. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). SIGARCH Comput. Archit. News **36**(5), 65–71 (2008)
15. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47666-6_25
16. Lamport, L.: The temporal logic of actions. ACM TOPLAS **16**(3), 872–923 (1994)
17. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16164-3_16
18. Meyer, J.J.C.: Modal epistemic and doxastic logic. In: Gabbay, D.M., Guenthner, F. (eds.) Handbook of Philosophical Logic, vol. 10, 2nd edn, pp. 1–38. Springer, Dordrecht (2004)

19. Nieto, L.P.: The rely-guarantee method in Isabelle/HOL. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 348–362. Springer, Heidelberg (2003). doi:10.1007/3-540-36575-3_24
20. Oracle Corporation: The Java Language Specification. Java SE 8 Edition (2015)
21. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14107-2_23
22. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03359-9_27
23. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15057-9_4
24. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of PLDI, pp. 175–186 (2011)
25. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: Proceedings of POPL, pp. 379–391 (2008)
26. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)
27. SPARC International Inc.: The SPARC Architecture Manual, Version 9 (1994)
28. Stølen, K.: Development of parallel programs on shared data-structures. Technical report UMCS-91-1-1, Department of Computer Science, University of Manchester (1991)
29. Tofan, B., Schellhorn, G., Bäumler, S., Reif, W.: Embedding rely-guarantee reasoning in temporal logic. Technical report, Institut für Informatik, Universität Augsburg (2010)
30. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: Navigating weak memory with ghosts, protocols, and separation. In: Proceedings of OOPSLA. 691–707(2014)
31. Vafeiadis, V.: Formal reasoning about the C11 weak memory model. In: Proceedings of CPP (2015)
32. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: Proceedings of OOPSLA, pp. 867–884 (2013)
33. Staden, S.: On rely-guarantee reasoning. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 30–49. Springer, Heidelberg (2015). doi:10.1007/978-3-319-19797-5_2
34. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)
35. Xu, Q.: A theory of state-based parallel programming. Ph.D. thesis, Oxford University Computing Laboratory (1992)
36. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997)