Compositional Verification of Termination-Preserving Refinement of Concurrent Programs

Hongjin Liang[†] Xinyu Feng[†] Zhong Shao[‡]

[†]University of Science and Technology of China [‡]Yale University

lhj1018@mail.ustc.edu.cn xyfeng@ustc.edu.cn zhong.shao@yale.edu

Abstract

Many verification problems can be reduced to refinement verification. However, existing work on verifying refinement of concurrent programs either fails to prove the preservation of termination, allowing a diverging program to trivially refine any programs, or is difficult to apply in compositional thread-local reasoning. In this paper, we first propose a new simulation technique, which establishes termination-preserving refinement and is a congruence with respect to parallel composition. We then give a proof theory for the simulation, which is the first Hoare-style concurrent program logic supporting termination-preserving refinement proofs. We show two key applications of our logic, i.e., verifying linearizability and lock-freedom *together* for fine-grained concurrent objects, and verifying *full* correctness of optimizations of concurrent algorithms.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Correctness proofs, Formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Theory, Verification

Keywords Concurrency, Refinement, Termination Preservation, Rely-Guarantee Reasoning, Simulation

1. Introduction

Verifying refinement between programs is the crux of many verification problems. For instance, reasoning about compilation or program transformations requires proving that every target program is a refinement of its source [9]. In concurrent settings, recent work [4, 12] shows that the correctness of concurrent data structures and libraries can be characterized via some forms of contextual refinements, i.e., every client that calls the concrete library methods should refine the client with some abstract atomic operations. Verification of concurrent garbage collectors [11] and OS kernels [18] can also be reduced to refinement verification.

Refinement from the source program S to the target T, written as $T \sqsubseteq S$, requires that T have no more observable behaviors than S. Usually observable behaviors include the traces of external events such as I/O operations and runtime errors. The question is, should termination of the source be preserved too by the target? If yes, how to verify such refinement?

Preservation of termination is an indispensable requirement in many refinement applications. For instance, compilation and optimizations are not allowed to transform a terminating source program to a diverging (non-terminating) target. Also, implementations of concurrent data structures are often expected to have progress guarantees (e.g., lock-freedom and wait-freedom) in addition to linearizability. The requirements are equivalent to some contextual refinements that preserve the termination of client programs [12].

Most existing approaches for verifying concurrent program refinement, including simulations (e.g., [11]), logical relations (e.g., [22]), and refinement logics (e.g., [21]), do not reason about the preservation of termination. As a result, a program that does an infinite loop without generating any external events, e.g. `while true do skip`, would trivially refine any source program (just like that it trivially satisfies partial correctness in Hoare logic). Certainly this kind of refinement is not acceptable in the applications mentioned above.

CompCert [9] addresses the problem by introducing a well-founded order in the simulation, but it works only for sequential programs. It is difficult to apply this idea to do thread-local verification of concurrent program refinement, which enables us to know $T_1 \parallel T_2 \sqsubseteq S_1 \parallel S_2$ by proving $T_1 \sqsubseteq S_1$ and $T_2 \sqsubseteq S_2$. In practice, the termination preservation in the refinement proofs of individual threads could be easily broken by the interference from their environments (i.e., other threads running in parallel). For instance, a method call of a lock-free data structure (e.g., Treiber stack) may never terminate when other threads call the methods and update the shared memory infinitely often. As we will explain in Sec. 2, the key challenge here is to effectively specify the environments' effects on the termination preservation of individual threads. As far as we know, no previous work can use "compositional" thread-local reasoning to verify termination-preserving refinement between (whole) concurrent programs.

In this paper, we first propose novel rely/guarantee conditions which can effectively specify the interference over the termination properties between a thread and its environment. Traditional rely/guarantee conditions [8] are binary relations of program states and they specify the state updates. We extend them with a boolean tag indicating whether a state update may let the thread or its environment make more moves.

With the help of our new rely/guarantee conditions, we then propose a new simulation RGSim-T, and a new Hoare-style program logic, both of which support compositional verification of termination-preserving refinement of concurrent programs. Our work is based on our previous compositional simulation RGSim [11] (which unfortunately cannot preserve termination), and is inspired by Hoffmann et al.'s program logic for lock-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright © 2014 ACM 978-1-4503-2886-9…$15.00.
http://dx.doi.org/10.1145/2603088.2603123

(a) source code S: `x++;`

```
1  local t;
2  t := x;
3  x := t + 1;
```

(b) target code T_b

```
1  local t, done := false;
2  while (! done) {
3    t := x;
4    done := cas(&x, t, t+1);
5  }
```

(c) target code T_c

Figure 1. Counters.

freedom [7] (which does not support refinement verification and has limitations on local reasoning, as we will explain in Sec. 7), but makes the following new contributions:

- We design a simulation, RGSim-T, to verify termination-preserving refinement of concurrent programs. As an extension of RGSim, it considers the interference between threads and the environments by taking our novel rely/guarantee conditions as parameters. RGSim-T is compositional. It allows us to thread-locally reason about the preservation of whole-program termination, but without enforcing the preservation of individual threads' termination, thus can be applied to many practical refinement applications.
- We propose the first program logic that supports compositional verification of termination-preserving refinement of concurrent programs. In addition to a set of compositionality (binary reasoning) rules, we also provide a set of unary rules (built upon the unary program logic LRG [3]) that can reason about conditional correspondence between the target and the source, a usual situation in concurrent refinement (see Sec. 2). The logic enables compositional verification of nested loops and supports programs with infinite nondeterminism. The soundness of the logic ensures RGSim-T between the target and the source, which implies the termination-preserving refinement.
- Our simulation and logic are general. They can be applied to verify linearizability and lock-freedom *together* for fine-grained concurrent objects, or to verify the full correctness of optimizations of concurrent programs, i.e., the optimized program preserves behaviors on both functionality and termination of the original one. We demonstrate the effectiveness of our logic by verifying linearizability and lock-freedom of Treiber stack [20], Michael-Scott queue [14] and DGLM queue [2], the full correctness of synchronous queue [16] and the equivalence between TTAS lock and TAS lock implementations [6].

It is important to note that our simulation and logic ensure that the target *preserves* the termination/divergence behaviors of the source. The target could diverge if the source diverges. Therefore our logic is not for verifying total correctness (i.e., partial correctness + termination). It is actually more powerful and general. We give more discussions on this point in Secs. 2.2 and 5.2.

In the rest of this paper, we first analyze the challenges and explain our approach informally in Sec. 2. Then we formulate termination-preserving refinement in Sec. 3. We present our new simulation RGSim-T in Sec. 4 and our new program logic in Sec. 5. We summarize examples that we verified in Sec. 6, and discuss the related work and conclude in Sec. 7.

2. Informal Development

Below we informally explain the challenges and our solutions in our design of the simulation and the logic respectively.

2.1 Simulation

Simulation is a standard technique for refinement verification. We start by showing a simple simulation for verifying *sequential* refinement and then discuss its problems in termination-preserving concurrent refinement verification.

$$\begin{tikzcd} S \arrow[r, dashed, "+"] \arrow[d, dash, "\preceq"'] & S' \arrow[d, dash, dashed, "\preceq"'] \\ T \arrow[r] & T' \end{tikzcd} \quad \text{or} \quad \begin{tikzcd} S \arrow[d, dash, "\preceq"'] \arrow[dr, dash, dashed, "\preceq"] & \\ T \arrow[r] & T' \end{tikzcd}$$

(with $|T'| < |T|$)

(a) $T \preceq S$

$$\begin{tikzcd} S_1 \arrow[rr, thick] \arrow[d, dash, "\preceq'"'] \arrow[dr, dash, dashed, "\preceq'"] \arrow[drr, phantom, "R"] & & S_2 \arrow[d, dash, dashed, "\preceq'"'] \arrow[dr, dash, dashed, "\preceq'"] & \\ T_1 \arrow[r] & T_2 \arrow[r, thick] & T_3 \arrow[r] & T_4 \end{tikzcd}$$

(with $|T_2| < |T_1|$ and $|T_4| < |T_3|$)

(b) $R \vdash T \preceq' S$

Figure 2. Simulation diagrams.

Fig. 1(a) shows the source code S that increments `x`. In a sequential setting, it can be implemented as T_b in Fig. 1(b). To show that T_b refines S, a natural way is to prove that they satisfy the (weak) simulation $\preceq$ in Fig. 2(a).

The simulation first establishes some consistency relation between the source and the target (note S and T here are whole program configurations consisting of both code and states). Then it requires that there is some correspondence between the execution of the target and the source so that the relation is always preserved. Every execution step of the target must either correspond to one or more steps of the source (the left part of Fig. 2(a)), or correspond to zero steps (the right part; Let's ignore the requirement of $|T'| < |T|$ for now).[1]

For our example in Fig. 1, the simulation requires that `x` at the target level have the same value with `x` in the source. We let line `2` at T_b correspond to zero steps of S, and line `3` correspond to the single step of S.

Such a simulation, however, has two problems for termination-preserving concurrent refinement verification. First, it does not require the target to preserve the termination of the source. Since a silent step at the target level may correspond to zero steps at the source (the right part of Fig. 2(a)), the target may execute such steps infinitely many times and never correspond to a step at the source. For instance, if we insert `while true do skip` before line `2` in T_b, the simulation still holds, but T_b diverges now. To address this problem, CompCert [9] introduces a metric $|T|$ over the target program configurations, which is equipped with a well-founded order $<$. If a target step corresponds to no moves of the source, the metric over the target programs should strictly decrease (i.e., the condition $|T'| < |T|$ in Fig. 2(a)). Since the well-founded order ensures that there are no infinite decreasing chains, execution of the target will finally correspond to at least one step at the source.

Second, it is not compositional w.r.t. parallel compositions. Though $T_b \preceq S$ holds, $(T_b \parallel T_b); \mathbf{print}(x) \preceq (S \parallel S); \mathbf{print}(x)$ does not hold since the left side may print out 1, which is impossible for the source on the right. The problem is that when we prove $T_b \preceq S$, T_b and S are viewed as closed programs and the interference from environments is ignored. To get the parallel compositionality, we follow the ideas in our previous work RGSim [11] and parameterize the simulation with the interference between the programs and their environments.

As shown in Fig. 2(b), the new simulation $\preceq'$ is parameterized with the environment interference R, i.e. the set of all possible transitions of the environments at the target and source levels. Here we use thin arrows for the transitions of the current thread at the source and the target levels (e.g., from T_1 to T_2 and from T_3 to T_4 in Fig. 2(b)), and thick arrows for the possible environment steps (e.g., from T_2 to T_3 and from S_1 to S_2 in the figure). We require the simulation $\preceq'$ to be preserved by R.

[1] Note here we only discuss silent steps (a.k.a. τ-steps) which produce no external events. The simulation also requires that every step with an external event at the target level must correspond to one step at the source with the same event plus zero or multiple τ-steps.

Then, to prove termination-preserving concurrent refinement, it seems natural to combine the two ideas and have a simulation parameterized with environment interference and a metric decreasing for target steps that correspond to no steps at the source. Therefore we require $|T_2| < |T_1|$ and $|T_4| < |T_3|$ in the case of Fig. 2(b). But *how would the environment steps change the metric?*

First attempt. Our first attempt to answer this question is to allow environment steps to arbitrarily change the metrics associated with the target program configurations. Therefore it is possible to have $|T_2| < |T_3|$ in Fig. 2(b).

The resulting simulation, however, is still not compositional w.r.t. parallel compositions. For instance, for the following two threads in the target program:

```
while(i==0) i--; || while(i==0) i++;
```

we can prove that this simulation holds between each of them and the source program `skip`, if we view `i` as local data used only at the target level. We could define the metric as 1 if `i` $= 0$ and 0 otherwise. For the left thread, it decreases the metric if it executes the loop body. The increment of `i` by its environment (the right thread) may change `i` back to 0, increasing the metric. This is allowed in our simulation. The case for the right thread is symmetric. However, if we view the parallel composition of the two threads as a whole program, it may not terminate, thus cannot be a termination-preserving refinement of `skip` || `skip`.

Second attempt. The first attempt is too permissive to have parallel compositionality, because we allow a thread to make more moves whenever its environment interferes with it. Thus our second attempt enforces the metric of a thread to decrease or stay unchanged under environment interference. For the case of Fig. 2(b), we require $|T_3| \leq |T_2|$ on environment steps.

This simulation is compositional, but it is too strong and cannot be satisfied by many useful refinements. For instance, T_c in Fig. 1(c) uses a compare-and-swap (`cas`) instruction to atomically update `x`. It is a correct lock-free implementation of S in concurrent settings, but the new simulation of our second attempt does not hold between T_c and S. If an environment step between lines 3 and 4 of T_c increments `x`, the `cas` at line 4 will return false and T_c needs to execute another round of loop. Therefore such an environment step increases the number of silent steps of T_c that correspond to no moves of S. However, our new simulation does not allow an environment step to increase the metric, so the simulation cannot be established.

Our solution. Our solution lies in the middle ground of the two failed attempts. We specify explicitly in the parameter R which environment steps may make the current thread move more (i.e., allow the thread's metric to increase in the simulation). Here we distinguish in R the steps that correspond to source level moves from those that do not. We allow the metric to be increased by the former (as in our first attempt), but not by the latter (which must decrease or preserve the metric, as in our second attempt).

This approach is based on the observation that the failure of `cas` in T_c of Fig. 1(c) must be caused by an environment step that successfully increments `x`, which corresponds to a step at the source level. Although the termination of the current thread T_c is delayed, the whole system consisting of both the current thread and the environment progresses by making a corresponding step at the source level. Therefore, the delay of the termination of the current thread should be acceptable, and we should allow such environment steps to increase the metric of the current thread.

In this paper, we follow the idea of rely/guarantee reasoning [8] and use the rely condition to specify environment steps. However, we extend the traditional rely conditions with an extra boolean tag indicating whether an environment step corresponds to a step at the source level. Our new simulation RGSim-T extends RGSim by incorporating the idea of metrics to achieve termination preservation. It is parameterized with the new rely (and guarantee) conditions so that we know how an environment step could affect the metric. The formal definition of RGSim-T is given in Sec. 4.

Relationships to lock-freedom, obstruction-freedom and wait-freedom. If the source program is just a single atomic operation (e.g. `x++`), our new simulation RGSim-T can be viewed as a proof technique for lock-freedom of the target, which ensures that there always *exists* some thread that will complete an operation at the source level in a finite number of steps. That is, the failure of a thread to finish its operation must be caused by the successful completion of source operations by its environment.

In fact, the simulations of our first and second attempts can be viewed as proof techniques for obstruction-freedom and wait-freedom respectively of concurrent objects. Obstruction-freedom ensures that every thread will complete its operation whenever it is executed in isolation (i.e., without interference from other threads). In the simulation of our first attempt, though a thread is allowed to not make progress under environment interference, it has to complete some source operations when its environments do not interfere. Wait-freedom ensures the completion of the operation of any thread. Correspondingly in the simulation of our second attempt, a thread has to make progress no matter what the environment does.

2.2 Program Logic

The compositionality of our new simulation RGSim-T allows us to decompose the refinement for large programs to refinements for small program units, therefore we could derive a set of syntactic Hoare-style rules for refinement verification, as we did for RGSim [11]. For instance, a sequential composition rule may be in the following form:

$$\frac{R \vdash \{P\}T_1 \preceq S_1\{P'\} \qquad R \vdash \{P'\}T_2 \preceq S_2\{Q\}}{R \vdash \{P\}T_1; T_2 \preceq S_1; S_2\{Q\}}$$

Here we use $R \vdash \{P\}T \preceq S\{Q\}$ to represent the corresponding syntactic judgment of RGSim-T. R denotes the environment interference. P, Q and P' are relational assertions that relate the program states at the target and the source levels. The rule says if we could establish refinements (in fact, RGSim-Ts) between T_1 and S_1, and between T_2 and S_2, we know $T_1; T_2$ refines $S_1; S_2$. We could give similar rules for parallel composition and other compositional commands.

However, in many cases the correspondence between program units at the target and the source levels cannot be determined statically. That is, just by looking at $T_1; T_2$ and $S_1; S_2$, we may not know statically that T_1 refines S_1 and T_2 refines S_2 and then apply the above sequential composition rule. To see the problem, we unfold the while-loop of T_c in Fig. 1 and get the following T_c':

```
1  local t, done;               4  while (!done) {
2  t := x;                      5    t := x;
3  done := cas(&x,t,t+1);       6    done := cas(&x,t,t+1);
                                7  }
```

Clearly T_c' refines S too. However, whether the `cas` instruction at line 3 fulfils the operation in S or not depends on whether the comparison succeeds in runtime. Thus we cannot apply the compositionality rules for RGSim-T to decompose the refinement about T_c'. We have to refer to the semantics of the simulation definition to prove the refinement, which would be rather ineffective for large scale programs. Similar issues also show up in our earlier work on RGSim [11], and in relational Hoare logic [1] and relational separation logic [25] if they are applied to concurrent settings.

To address this problem, we extend the assertion language to specify as auxiliary state the source code remaining to be refined.

In addition to the binary judgment $R \vdash \{P\}T \preceq S\{Q\}$, we introduce a unary judgment in the form of $R \vdash \{P \wedge \mathsf{arem}(S)\}T\{Q \wedge \mathsf{arem}(S')\}$ for refinements that cannot be decomposed. Here $\mathsf{arem}(S)$ means S is the remaining source to be refined by the target. Then $R, G \vdash \{P \wedge \mathsf{arem}(S)\}T\{Q \wedge \mathsf{arem}(\mathtt{skip})\}$ says that T refines S, since the postcondition shows at the end of the target T there are no remaining operations from S to be refined. We provide the following rule to derive the binary judgment from the unary one:

$$\frac{R \vdash \{P \wedge \mathsf{arem}(S)\}T\{Q \wedge \mathsf{arem}(\mathtt{skip})\}}{R \vdash \{P\}T \preceq S\{Q\}}$$

On the other hand, if the final remaining source is the same as the initial one, we know the execution steps of the target correspond to zero source steps. Then for the T'_c above, we can give pre- and post-conditions for line 3 as follows:

$\{\cdots \wedge \mathsf{arem}(S)\}$
`done := cas(&x, t, t+1)`
$\{\cdots \wedge (\mathtt{done} \wedge \mathsf{arem}(\mathtt{skip}) \vee \neg\mathtt{done} \wedge \mathsf{arem}(S))\}$

As the postcondition shows, whether the `cas` instruction refines S or not is now conditional upon the value of `done`. Thanks to the new assertions $\mathsf{arem}(S)$, we can reduce the relational and semantic refinement proofs to unary and syntactic Hoare-style reasoning.

The key to verifying the preservation of termination is the rule for while loops. One may first think of the total correctness rule for while loops in Hoare-style logics (e.g., [19]). However, preserving the termination does not necessarily mean that the code must terminate, and the total correctness rule would not be applicable in many cases. For example, the following T''_c and S' never terminate:

```
T''_c :                    | S' :
                           |
local t;                   |
while (true){              | while (true)
  t := x;                  |   x++;
  cas(&x, t, t+1);         |
}                          |
```

but $T''_c \preceq S'$ holds for our RGSim-T ($\preceq$) — Every iteration of T''_c either corresponds to a step of S', or is interfered by environment steps corresponding to source moves.

Inspired by Hoffmann et al.'s logic for lock-freedom [7], we introduce a counter n (i.e. the number of tokens assigned to the current thread) as a while-specific metric, which means the thread can only run the loop for no more than n rounds before it or its environment fulfils one or more source-level moves. The counter is treated as an auxiliary state, and decreases at the beginning of every round of loop (i.e., we pay one token for each iteration). If we reach a step in the loop body that corresponds to source moves, we could reset the counter to increase the number of tokens. Tokens could also increase under environment interference if the environment step corresponds to source moves. Correspondingly our WHILE rule is in the following form (we give a simplified version to demonstrate the idea here. The actual rule is given in Sec. 5):

$$\frac{P \wedge B \Rightarrow P' * \mathsf{wf}(1) \qquad R \vdash \{P'\}C\{P\}}{R \vdash \{P\}\textbf{while}\ (B)\ C\{P \wedge \neg B\}}$$

We use $\mathsf{wf}(1)$ to represent one token, and "$*$" for normal separating conjunction in separation logic. To verify the loop body C, we use the precondition P', which has one less token than P, showing that one token has been consumed to start this new round of loop. During the execution of C, the number of token could be increased if C itself or its environment steps correspond to source moves. As usual, the loop invariant P needs to be re-established at the end of C.

$$\begin{array}{lllllll}
(\mathit{Event}) & e & ::= \ \ldots & & (\mathit{Label}) & \iota & ::= \ e \mid \tau \\
(\mathit{Store}) & s, \mathbb{s} & \in \ \mathit{PVar} \rightharpoonup \mathit{Val} & & (\mathit{Heap}) & h, \mathbb{h} & \in \ \mathit{Addr} \rightharpoonup \mathit{Val} \\
(\mathit{State}) & \sigma, \Sigma & ::= \ (s, h) & & (\mathit{Instr}) & c, \mathbb{c} & ::= \ \ldots \\
(\mathit{Expr}) & E, \mathbb{E} & ::= \ x \mid n \mid E + E \mid \ldots & & & & \\
(\mathit{BExp}) & B, \mathbb{B} & ::= \ \textbf{true} \mid \textbf{false} \mid E = E \mid !B \mid \ldots & & & & \\
(\mathit{Stmt}) & C, \mathbb{C} & ::= \ \textbf{skip} \mid c \mid \langle C \rangle \mid C_1; C_2 \mid \textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2 & & & & \\
 & & \quad \mid \textbf{while}\ (B)\ C \mid C_1 \,\|\, C_2 & & & &
\end{array}$$

Figure 3. Generic language at target and source levels.

To prove that T''_c shown above preserves the termination of S', we set the initial number of tokens to 1. We use up the token at the first iteration, but could gain another token during the iteration (either by self moves or by environment steps) to pay for the next iteration. We can see that the above reasoning with tokens coincides with the direct refinement proof in our simulation RGSim-T. In fact, RGSim-T can serve as the meta-theory of our logic.

The use of tokens as an explicit metric for termination reasoning poses another challenge, which is to handle *infinite nondeterminism.* Consider the following target C.

C: `x := 0; while(i > 0) i--;`

Assume the environment R may arbitrarily update `i` when `x` is not 0, but does not change anything when `x` is 0. We hope to verify C refines `skip`. We can see that the loop in C must terminate (thus the refinement holds), and the number n of tokens must be no less than the value of `i` at the beginning of the loop. But we cannot decide the value of n before executing `x := 0`. This example cannot be verified if we have to predetermine and specify the metric for the while loops at the very beginning of the whole program.

To address this issue, we introduce the following hiding rule:

$$\frac{R \vdash \{p\}C\{q\}}{R \vdash \{\lfloor p \rfloor_{\mathsf{w}}\}C\{\lfloor q \rfloor_{\mathsf{w}}\}}$$

Here $\lfloor p \rfloor_{\mathsf{w}}$ discards all the knowledge about tokens in p. For the above example, we can hide the number of tokens after we verify the while loop. Then we do *not* need to specify the number of tokens in the precondition of the whole program. We formally present the set of logic rules in Sec. 5.

3. Formal Settings and Termination-Preserving Refinement

In this section, we define the termination-preserving refinement $\sqsubseteq$, which is the proof goal of our RGSim-T and logic.

3.1 The Language

Fig. 3 shows the programming language for both the source and the target levels. We model the program semantics as a labeled transition system. A label ι that will be associated with a state transition is either an event e or τ. The latter marks a silent step generating no events.

A state σ is a pair of a store and a heap. The store s is a finite partial mapping from program variables to values (e.g., integers and memory addresses) and a heap h maps memory addresses to values. Statements C are either primitive instructions or compositions of them. A single-step execution of statements is modeled as a labeled transition: $(C, \sigma) \xrightarrow{\iota} (C', \sigma')$. We abstract away the form of an instruction c. It may generate an external event (e.g., $\textbf{print}(E)$ generates an output event). It may be non-deterministic (e.g., $x := \textbf{nondet}$ assigns a random value to x). It may also be blocked at some states (e.g., requesting a lock). We assume primitive instructions are atomic in the semantics. We also provide an

$$\frac{(C,\sigma) \longrightarrow^{+} \textbf{abort}}{\mathit{ETr}(C,\sigma,\lightning)} \qquad \frac{(C,\sigma) \xrightarrow{e}{}^{+} (C',\sigma') \qquad \mathit{ETr}(C',\sigma',\mathcal{E})}{\mathit{ETr}(C,\sigma,e :: \mathcal{E})}$$

$$\frac{(C,\sigma) \longrightarrow^{*} (\textbf{skip},\sigma')}{\mathit{ETr}(C,\sigma,\Downarrow)} \qquad \frac{(C,\sigma) \longrightarrow^{+} (C',\sigma') \qquad \mathit{ETr}(C',\sigma',\mathcal{E})}{\mathit{ETr}(C,\sigma,\mathcal{E})}$$

Figure 4. Co-inductive definition of $\mathit{ETr}(C,\sigma,\mathcal{E})$.

$$\begin{array}{llrl}
(\mathit{RelAssn}) & P,Q,I & ::= & B \mid \mathsf{own}(x) \mid \mathsf{emp} \mid E \mapsto E \mid E \Mapsto E \\
 & & & \mid P * Q \mid P \vee Q \mid \llbracket p \rrbracket \mid \ldots \\
(\mathit{FullAssn}) & p,q & ::= & P \mid \mathsf{arem}(\mathbb{C}) \mid \mathsf{wf}(E) \mid \lfloor p \rfloor_{\mathsf{a}} \mid \lfloor p \rfloor_{\mathsf{w}} \\
 & & & \mid p * q \mid p \vee q \mid \ldots \\
(\mathit{RelAct}) & R,G & ::= & [P] \mid P \ltimes Q \mid P \propto Q \mid R * R \mid R^{+} \mid \ldots
\end{array}$$

Figure 5. Assertion language.

atomic block $\langle C \rangle$ to execute a piece of code C atomically. Then the generic language in Fig. 3 is expressive enough for the source and the target programs which may have different granularities of state accesses. Due to the space limit, the operational semantics and more details about the language are formally presented in TR [13].

Conventions. We usually write blackboard bold or capital letters ($\mathbb{s}$, $\mathbb{h}$, Σ, $\mathbb{c}$, $\mathbb{E}$, $\mathbb{B}$ and $\mathbb{C}$) for the notations at the source level to distinguish from the target-level ones (s, h, σ, c, E, B and C).

Below we use $\_ \longrightarrow^{*} \_$ for zero or multiple-step transitions with no events generated, $\_ \longrightarrow^{+} \_$ for multiple-step transitions without events, and $\_ \xrightarrow{e}{}^{+} \_$ for multiple-step transitions with *only one* event e generated.

3.2 Termination-Preserving Event Trace Refinement

Now we formally define the refinement relation $\sqsubseteq$ that relates the observable event traces generated by the source and the target programs. A trace $\mathcal{E}$ is a finite or infinite sequence of external events e, and may end with a termination marker $\Downarrow$ or an abortion marker $\lightning$. It is co-inductively defined as follows.

$$(\mathit{EvtTrace}) \quad \mathcal{E} \quad ::= \quad \Downarrow \mid \lightning \mid \epsilon \mid e :: \mathcal{E} \quad \text{(co-inductive)}$$

We use $\mathit{ETr}(C,\sigma,\mathcal{E})$ to say that the trace $\mathcal{E}$ is produced by executing C from the state σ. It is co-inductively defined in Fig. 4. Here **skip** plays the role of a flag showing the end of execution (the normal termination). Unsafe executions lead to **abort**. We know if C diverges at σ, then its trace $\mathcal{E}$ is either of infinite length or finite but does not end with $\Downarrow$ or $\lightning$. For instance, **while** (**true**) **skip** only produces an empty trace ϵ, and **while** (**true**) $\{$**print**$(1)\}$ only produces an infinite trace of output events.

Then we define a refinement $(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$, saying that every event trace generated by (C,σ) at the target level can be reproduced by $(\mathbb{C},\Sigma)$ at the source. Since we could distinguish traces of diverging executions from those of terminating executions, the refinement definition ensures that if (C,σ) diverges, so does $(\mathbb{C},\Sigma)$. Thus we know the target preserves the termination of the source.

Definition 1 (Termination-Preserving Refinement).
$(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$ iff $\forall \mathcal{E}.\, \mathit{ETr}(C,\sigma,\mathcal{E}) \Longrightarrow \mathit{ETr}(\mathbb{C},\Sigma,\mathcal{E})$.

4. RGSim-T: A Compositional Simulation with Termination Preservation

Below we propose RGSim-T, a new simulation as a compositional proof technique for the above termination-preserving refinement. As we explained in Sec. 2, the key to compositionality is to parameterize the simulation with the interferences between the programs and their environments. In this paper, we specify the interferences using rely/guarantee conditions [8], but extend them to also specify the effects on the termination preservation of individual threads.

Our simulation relation between C and $\mathbb{C}$ is in the form of $R, G, I \models \{P\}C \preceq \mathbb{C}\{Q\}$. It takes R, G, I, P and Q as parameters. R and G are rely and guarantee conditions specifying the interference between the current thread and its environment. The assertion I specifies the consistency relation between states at the target and the source levels, which needs to be preserved during the execution. P specifies the pair of initial states at the target and the source levels from which the simulation holds, and Q is about the pair of final states when the target and the source terminate. So before we give our definition of RGSim-T, we first introduce our assertion language.

4.1 Assertions and New Rely/Guarantee Conditions

We show the syntax of the basic assertion language in Fig. 5, including the state assertions P and Q, and our new rely/guarantee conditions R and G (let's first ignore the assertions p and q, which will be explained in Sec. 5).

The state assertions P and Q relate the program states σ and Σ at the target and source levels. They are separation logic assertions over a pair of states. We show their semantics in the top part of Fig. 6. For simplicity, we assume the program variables used in the target code are different from the ones in the source (e.g., we use x and X for target and source level variables respectively). B holds if it evaluates to true at the disjoint union of the target and the source stores s and $\mathbb{s}$. We treat program variables as resources [15] and use $\mathsf{own}(x)$ for the ownership of the program variable x. The assertion $E_1 \mapsto E_2$ specifies a singleton heap of the *target* level with E_2 stored at the address E_1 and requires that the stores contain variables used to evaluate E_1 and E_2. Its counterpart for source level heaps is represented as $E_1 \Mapsto E_2$, whose semantics is defined similarly. emp describes empty stores and heaps at both levels. Semantics of separating conjunction $P * Q$ is similar as in separation logic, except that it is now lifted to assertions over relational states (σ,Σ). The union of two disjoint relational states (σ_1,Σ_1) and (σ_2,Σ_2) is defined in the middle part of Fig. 6. We will define the assertion $\llbracket p \rrbracket$ in Sec. 5 (see Fig. 8), which ignores the additional information other than the relational states about p.

Our new rely/guarantee assertions R and G specify the transitions over the relational states (σ,Σ) and also the effects on termination preservation. Their semantics is defined in the bottom part of Fig. 6. Here we use $\mathcal{S}$ for the relational states. A model consists of the initial relational state $\mathcal{S}$, the resulting state $\mathcal{S}'$, and an effect bit b to record whether the target transitions correspond to some source steps and can affect the termination preservation of the current thread (for R) or other threads (for G).

We use $[P]$ for identity transitions with the relational states satisfying P. The action $P \ltimes Q$ says that the initial relational states satisfy P and the resulting states satisfy Q. For these two kinds of actions, we do not care whether there is any source step in the transition satisfying them (the effect bit b in their interpretations could either be **true** or **false**). We also introduce a new action $P \propto Q$ asserting that one or more steps are made at the source level (the effect bit b must be **true**). Following LRG [3], we introduce separating conjunction over actions to locally reason about shared state updates. $R_1 * R_2$ means that the actions R_1 and R_2 start from disjoint relational states and the resulting states are also disjoint. But here we also require consistency over the effect bits for the two disjoint state transitions. We use R^{+} for the transitive closure of R, where the effect bits in consecutive transitions are accumulated. The syntactic sugars Id, Emp and True represent arbitrary identity transitions, empty transitions and arbitrary transitions respectively.

Since we logically split states into local and shared parts as in LRG [3], we need a precise invariant I to fence actions over shared

$$
\begin{array}{ll}
((s,h),(\mathbb{s},\mathbb{h})) \models B & \text{iff } [\![B]\!]_{s\uplus\mathbb{s}} = \textbf{true} \\
((s,h),(\mathbb{s},\mathbb{h})) \models \mathsf{own}(x) & \text{iff } \mathit{dom}(s\uplus\mathbb{s}) = \{x\} \\
((s,h),(\mathbb{s},\mathbb{h})) \models E_1 \mapsto E_2 & \text{iff } h = \{[\![E_1]\!]_{s\uplus\mathbb{s}} \leadsto [\![E_2]\!]_{s\uplus\mathbb{s}}\} \\
((s,h),(\mathbb{s},\mathbb{h})) \models \mathsf{emp} & \text{iff } s = h = \mathbb{s} = \mathbb{h} = \emptyset
\end{array}
$$

$$f_1 \perp f_2 \text{ iff } (\mathit{dom}(f_1) \cap \mathit{dom}(f_2) = \emptyset) \qquad f_1 \uplus f_2 \stackrel{\text{def}}{=} f_1 \cup f_2\,, \text{ if } f_1 \perp f_2$$

$$(s_1,h_1)\perp(s_2,h_2) \text{ iff } (s_1 \perp s_2) \wedge (h_1 \perp h_2)$$

$$(s_1,h_1) \uplus (s_2,h_2) \stackrel{\text{def}}{=} (s_1 \cup s_2, h_1 \cup h_2)\,, \text{ if } (s_1,h_1)\perp(s_2,h_2)$$

$$(\sigma_1,\Sigma_1) \uplus (\sigma_2,\Sigma_2) \stackrel{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2)\,, \text{ if } \sigma_1 \perp \sigma_2 \text{ and } \Sigma_1 \perp \Sigma_2$$

$$\mathcal{S} ::= (\sigma, \Sigma)$$

$$(\mathcal{S},\mathcal{S}',b) \models [P] \text{ iff } (\mathcal{S} \models P) \wedge (\mathcal{S} = \mathcal{S}')$$

$$(\mathcal{S},\mathcal{S}',b) \models P \ltimes Q \text{ iff } (\mathcal{S} \models P) \wedge (\mathcal{S}' \models Q)$$

$$(\mathcal{S},\mathcal{S}',b) \models P \propto Q \text{ iff } (\mathcal{S} \models P) \wedge (\mathcal{S}' \models Q) \wedge (b = \textbf{true})$$

$$
\begin{array}{l}
(\mathcal{S},\mathcal{S}',b) \models R_1 * R_2 \text{ iff } \exists \mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_1',\mathcal{S}_2'.\,(\mathcal{S} = \mathcal{S}_1 \uplus \mathcal{S}_2) \wedge \\
\qquad (\mathcal{S}' = \mathcal{S}_1' \uplus \mathcal{S}_2') \wedge ((\mathcal{S}_1,\mathcal{S}_1',b) \models R_1) \wedge ((\mathcal{S}_2,\mathcal{S}_2',b) \models R_2)
\end{array}
$$

$$
\begin{array}{l}
(\mathcal{S},\mathcal{S}',b) \models R^+ \text{ iff } ((\mathcal{S},\mathcal{S}',b) \models R) \ \vee \ (\exists \mathcal{S}'',b',b''. \\
\qquad ((\mathcal{S},\mathcal{S}'',b') \models R) \wedge ((\mathcal{S}'',\mathcal{S}',b'') \models R^+) \wedge (b = b' \vee b''))
\end{array}
$$

$$\mathsf{Id} \stackrel{\text{def}}{=} [\textbf{true}] \qquad \mathsf{Emp} \stackrel{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp} \qquad \mathsf{True} \stackrel{\text{def}}{=} \textbf{true} \ltimes \textbf{true}$$

$$I \triangleright R \text{ iff } ([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I)$$

$$\mathsf{Sta}(P,R) \text{ iff } \forall \mathcal{S},\mathcal{S}',b.\,(\mathcal{S} \models P) \wedge ((\mathcal{S},\mathcal{S}',b) \models R) \Longrightarrow (\mathcal{S}' \models P)$$

Figure 6. Semantics of assertions (part I).

states, which is a state assertion like P and Q. We define the fence $I \triangleright R$ in a similar way as in our previous work [10] and LRG [3], which says that I precisely determines the boundaries of the states of the transitions in R (see Fig. 6). The formal definition of the precise requirement $\mathsf{Precise}(I)$ is given in TR [13], which follows its usual meaning as in separation logic but is now interpreted over relational states.

4.2 Definition of RGSim-T

Our simulation RGSim-T is parameterized over the rely/guarantee conditions R and G to specify the interferences between threads and their environments, and a precise invariant I to logically determine the boundaries of the shared states and fence R and G.

The simulation also takes a metric M, which was referred to as $|T|$ in our previous informal explanations in Sec. 2. We leave its type unspecified here, which can be instantiated by program verifiers, as long as it is equipped with a well-founded order $<$.

The formal definition below follows the intuition explained in Sec. 2. Readers who are interested only in the proof theory could skip this definition, which can be viewed as the meta-theory of our program logic presented in Sec. 4.3 and Sec. 5.

Definition 2 (RGSim-T). $R, G, I \models \{P\}C \preceq \mathbb{C}\{Q\}$ iff for all σ and Σ, if $(\sigma, \Sigma) \models P$, then there exists M such that $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$.

Here $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$ is the largest relation such that whenever $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, then $(\sigma, \Sigma) \models I * \textbf{true}$ and the following are true:

1. for any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', n, M', b, $\mathbb{C}'$ and Σ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$,
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^n (\mathbb{C}', \Sigma' \uplus \Sigma_F)$,
   (c) $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$,
   (d) $((\sigma, \Sigma), (\sigma', \Sigma'), b) \models G^+ * \mathsf{True}$, and
   (e) if $n = 0$, we need $M' < M$ and $b = \textbf{false}$, otherwise $b = \textbf{true}$.
2. for any e, C', σ'', σ_F and σ_F, if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', M', $\mathbb{C}'$ and Σ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$,
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \xrightarrow{e}{}^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$,
   (c) $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$, and
   (d) $((\sigma, \Sigma), (\sigma', \Sigma'), \textbf{true}) \models G^+ * \mathsf{True}$.
3. for any b, σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R^+ * \mathsf{Id}$, then there exists M' such that
   (a) $R, G, I \models (C, \sigma', M') \preceq_Q (\mathbb{C}, \Sigma')$, and
   (b) if $b = \textbf{false}$, we need $M' = M$.
4. if $C = \textbf{skip}$, then for any Σ_F such that $\Sigma \perp \Sigma_F$, there exist n and Σ' such that
   (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^n (\textbf{skip}, \Sigma' \uplus \Sigma_F)$,
   (b) $(\sigma, \Sigma') \models Q$,
   (c) if $n > 0$, then $((\sigma, \Sigma), (\sigma, \Sigma'), \textbf{true}) \models G^+ * \mathsf{True}$.
5. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \textbf{abort}$ and $\Sigma \perp \Sigma_F$, then $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \textbf{abort}$.

The simulation $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$ relates the executions of the target configuration (C, σ) (with its metric M) to the source $(\mathbb{C}, \Sigma)$, under the interferences with the environment specified by R and G. It first requires that the relational state (σ, Σ) satisfy $I * \textbf{true}$, I for the shared part and $\textbf{true}$ for the local part, establishing a consistency relation between the states at the two levels. For every silent step of (C, σ) (condition 1, let's first ignore the frame states σ_F and Σ_F which will be discussed later), the source could make n steps ($n \geq 0$) correspondingly (1(b)), and the simulation is preserved afterwards with a new metric M' (1(c)). Here we use $\_ \longrightarrow^n \_$ to represent n-step silent transitions. If $n = 0$ in 1(b) (i.e., the source does not move), the metric must decrease along the associated well-founded order ($M' < M$ in 1(e)), otherwise we do not have any restrictions over M'. We also require that the related steps at the two levels satisfy the guarantee condition $G^+ * \mathsf{True}$ (1(d)), the transitive closure G^+ for the shared part and True for the private. If the target step corresponds to *no* source moves ($n = 0$), we use **false** as the corresponding effect bit, otherwise the bit should be **true** (1(e)).

If a target step produces an event e, the requirements (condition 2) are similar to those in condition 1, except that we know for sure that target step corresponds to *one or more* source steps that produce the same e.

The simulation should be preserved after environment transitions satisfying $R^+ * \mathsf{Id}$, R^+ for the shared part and Id for the private (condition 3). If the corresponding effect bit of the environment transition is **true**, we know there are one or more source moves, therefore there are no restrictions over the metric M' for the resulting code (which could be larger than M). Otherwise, the metric should be unaffected under the environment interference (i.e., $M' = M$ in 3(b)).

If C terminates (condition 4), the corresponding $\mathbb{C}$ must also terminate and the resulting states satisfy the postcondition Q. Finally, if C is unsafe, then $\mathbb{C}$ must be unsafe too (condition 5).

Inspired by Vafeiadis [24], we directly embed the framing aspect of separation logic in Def. 2. At each condition, we introduce the frame states σ_F and Σ_F at the target and source levels to represent the remaining parts of the states owned by other threads in the system. The commands C and $\mathbb{C}$ must not change the frame states during their executions (see, e.g., conditions 1(a) and 1(b)). These σ_F and Σ_F quantifications in RGSim-T are crucial to admit the parallel compositionality and the frame rules (the B-FRAME rule in Fig. 7 and the FRAME rule in Fig. 9).

We then define $R, G, I \models \{P\}C \preceq \mathbb{C}\{Q\}$ by hiding the initial states via the precondition P and hiding the metric M.

$$\frac{R \vee G_2, G_1, I \vdash \{P_1 * P\}C_1 \preceq \mathbb{C}_1\{Q_1 * Q_1'\} \quad R \vee G_1, G_2, I \vdash \{P_2 * P\}C_2 \preceq \mathbb{C}_2\{Q_2 * Q_2'\} \quad P \vee Q_1' \vee Q_2' \Rightarrow I \quad I \rhd R}{R, G_1 \vee G_2, I \vdash \{P_1 * P_2 * P\}C_1 \,\|\, C_2 \preceq \mathbb{C}_1 \,\|\, \mathbb{C}_2\{Q_1 * Q_2 * (Q_1' \wedge Q_2')\}} \ (\textsc{b-par})$$

$$\frac{P \Rightarrow (B \Leftrightarrow \mathbb{B}) * I \quad R, G, I \vdash \{P \wedge B\}C \preceq \mathbb{C}\{P\}}{R, G, I \vdash \{P\}\textbf{while}\ (B)\ C \preceq \textbf{while}\ (\mathbb{B})\ \mathbb{C}\{P \wedge \neg B\}} \ (\textsc{b-while}) \qquad \frac{P \Rightarrow (E = \mathbb{E}) * I \quad \mathsf{Sta}(P, R * \mathsf{Id}) \quad I \rhd \{R, G\}}{R, G, I \vdash \{P\}\textbf{print}(E) \preceq \textbf{print}(\mathbb{E})\{P\}} \ (\textsc{b-prt})$$

$$\frac{R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\} \quad \mathsf{Sta}(P', R' * \mathsf{Id}) \quad I' \rhd \{R', G'\} \quad P' \Rightarrow I' * \textbf{true} \quad G^+ \Rightarrow G}{R * R', G * G', I * I' \vdash \{P * P'\}C \preceq \mathbb{C}\{Q * P'\}} \ (\textsc{b-frame})$$

Figure 7. Selected binary inference rules (compositionality of RGSim-T).

Adequacy. RGSim-T ensures the termination-preserving refinement by using the metric with a well-founded order. The proof of the following adequacy theorem is in TR [13].

Theorem 3 (Adequacy of RGSim-T). If there exist R, G, I, Q and a metric M (with a well-founded order $<$) such that $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.

4.3 Compositionality Rules

RGSim-T is compositional. We show some of the compositionality rules in Fig. 7. Here we use $R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}$ for the judgment to emphasize syntactic reasoning, whose semantics is RGSim-T (Def. 2). The rules can be viewed as the binary version of those in a traditional rely-guarantee-style logic (e.g., LRG [3] and RGSep [23]).

The B-PAR rule shows the compositionality w.r.t. parallel compositions. To verify $C_1 \parallel C_2$ is a refinement of $\mathbb{C}_1 \parallel \mathbb{C}_2$, we verify the refinement of each thread separately. The rely condition of each thread captures the interference from both the overall environment (R) and its sibling thread (G_1 or G_2). The related steps of $C_1 \parallel C_2$ and $\mathbb{C}_1 \parallel \mathbb{C}_2$ should satisfy either thread's guarantee. As in LRG [3], P_1 and P_2 specify the private (relational) states of the threads $C_1/\mathbb{C}_1$ and $C_2/\mathbb{C}_2$ respectively. The states P are shared by them. When both threads have terminated, their private states satisfy Q_1 and Q_2, and the shared states satisfy both Q_1' and Q_2'. We require that the shared states are well-formed (P, Q_1' and Q_2' imply I) and the overall environment transitions are fenced ($I \rhd R$).

The B-WHILE rule requires the boolean conditions of both sides to be evaluated to the same value. The resources needed to evaluate them should be available in the private part of P. The B-FRAME rule supports local reasoning. The frame P' may contain shared and private parts, so it should be stable w.r.t. $R' * \mathsf{Id}$ and imply $I' * \textbf{true}$, where I' is the fence for R' and G' (see Fig. 6 for the definitions of fences and stability). We also require G to be closed over transitivity. This rule is almost identical to the one in LRG [3]. Details are elided here.

We provide a few binary rules to reason about the basic program units when they are almost identical at both sides. For instance, the B-PRT rule relates a target print command to a source one, requiring that they always print out the same value. For more general refinement units, as we explained in Sec. 2, we reduce relational verification to unary reasoning (using the U2B rule in Fig. 9, which we will explain in the next section). Our TR [13] contains more rules and the full soundness proofs. The soundness theorem is shown below.

Theorem 4 (Soundness of Binary Rules).
If $R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}$, then $R, G, I \models \{P\}C \preceq \mathbb{C}\{Q\}$.

5. A Rely-Guarantee-Style Logic for Termination-Preserving Refinement

The binary inference rules in Fig. 7 allow us to decompose the refinement verification of large programs into the refinement units' verification. In this section, we explain the unary rules for verifying refinement units. All the binary and unary rules constitute our novel rely-guarantee-style logic for modular verification of termination-preserving refinement.

$$w \in \mathit{Nat} \qquad \mathbb{D} ::= \mathbb{C} \mid \bullet$$

$$\begin{array}{ll}
(\sigma, w, \mathbb{D}, \Sigma) \models P & \text{iff } (\sigma, \Sigma) \models P \\
(\sigma, w, \mathbb{D}, \Sigma) \models \mathsf{arem}(\mathbb{C}') & \text{iff } \mathbb{D} = \mathbb{C}' \\
((s, h), w, \mathbb{D}, \Sigma) \models \mathsf{wf}(E) & \text{iff } \exists n.\, ([\![E]\!]_s = n) \wedge (n \leq w) \\
(\sigma, w, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{a}} & \text{iff } \exists \mathbb{D}'.\, (\sigma, w, \mathbb{D}', \Sigma) \models p \\
(\sigma, w, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{w}} & \text{iff } \exists w'.\, (\sigma, w', \mathbb{D}, \Sigma) \models p \\
(\sigma, \Sigma) \models \lfloor\!\lfloor p \rfloor\!\rfloor & \text{iff } \exists w, \mathbb{D}.\, (\sigma, w, \mathbb{D}, \Sigma) \models p
\end{array}$$

$$\mathbb{D}_1 \perp \mathbb{D}_2 \text{ iff } (\mathbb{D}_1 = \bullet) \vee (\mathbb{D}_2 = \bullet) \qquad \mathbb{D}_1 \uplus \mathbb{D}_2 \stackrel{\text{def}}{=} \begin{cases} \mathbb{D}_2 & \text{if } \mathbb{D}_1 = \bullet \\ \mathbb{D}_1 & \text{if } \mathbb{D}_2 = \bullet \end{cases}$$

$$\begin{array}{l}
(\sigma_1, w_1, \mathbb{D}_1, \Sigma_1) \uplus (\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \stackrel{\text{def}}{=} \\
(\sigma_1 \uplus \sigma_2, w_1 + w_2, \mathbb{D}_1 \uplus \mathbb{D}_2, \Sigma_1 \uplus \Sigma_2)\,, \text{ if } \sigma_1 \perp \sigma_2, \mathbb{D}_1 \perp \mathbb{D}_2 \text{ and } \Sigma_1 \perp \Sigma_2
\end{array}$$

$$\begin{array}{l}
\mathsf{Sta}(p, R) \text{ iff } \forall \sigma, w, \mathbb{D}, \Sigma, \sigma', \Sigma', b. \\
\quad ((\sigma, w, \mathbb{D}, \Sigma) \models p) \wedge (((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R) \\
\quad \Longrightarrow \exists w'.\, (\sigma', w', \mathbb{D}, \Sigma') \models p \ \wedge \ (b = \textbf{false} \Longrightarrow w' = w)
\end{array}$$

$$\begin{array}{l}
p \Rrightarrow^0 q \text{ iff } p \Rightarrow q \\
p \Rrightarrow^+ q \text{ iff } \forall \sigma, w, \mathbb{D}, \Sigma, \Sigma_F.\, ((\sigma, w, \mathbb{D}, \Sigma) \models p) \wedge (\Sigma \perp \Sigma_F) \Longrightarrow \\
\ \exists w', \mathbb{C}', \Sigma'.\, (\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F) \wedge ((\sigma, w', \mathbb{C}', \Sigma') \models q)
\end{array}$$

Figure 8. Semantics of assertions (part II).

5.1 Assertions on Source Code and Number of Tokens

We first explain the new assertions p and q used in the unary rules that can specify the source code and metrics in addition to states. We define their syntax in Fig. 5, and their semantics in Fig. 8. A *full state assertion* p is interpreted on $(\sigma, w, \mathbb{D}, \Sigma)$. Here besides the states σ and Σ at the target and source levels, we introduce some auxiliary data w and $\mathbb{D}$. w is the number of tokens needed for loops (see Sec. 2). $\mathbb{D}$ is either some source code $\mathbb{C}$, or a special sign $\bullet$ serving as a unit for defining semantics of $p * q$ below.

In Fig. 8 we lift the relational assertion P as a full state assertion to specify the states. The new assertion $\mathsf{arem}(\mathbb{C})$ says that the remaining source code is $\mathbb{C}$ at the current program point. $\mathsf{wf}(E)$ states that the number of tokens at the current target code is *no less than* E. We can see $\mathsf{wf}(0)$ always holds, and for any n, $\mathsf{wf}(n+1)$ implies $\mathsf{wf}(n)$. We use $\lfloor p \rfloor_{\mathsf{a}}$ and $\lfloor p \rfloor_{\mathsf{w}}$ to ignore the descriptions in p about the source code and the number of tokens respectively. $\lfloor\!\lfloor p \rfloor\!\rfloor$ lifts p back to a relational state assertion.

Separating conjunction $p * q$ has the standard meaning as in separation logic, which says p and q hold over disjoint parts of $(\sigma, w, \mathbb{D}, \Sigma)$ respectively (the formal definition elided here). However, it is worth noting the definition of disjoint union over the quadruple states, which is shown in the middle part of Fig. 8. The disjoint union of the numbers of tokens w_1 and w_2 is simply the sum of them. The disjoint union of $\mathbb{D}_1$ and $\mathbb{D}_2$ is defined only if

$$\frac{R, G, I \vdash \{P \wedge \mathsf{arem}(\mathbb{C})\}C\{Q \wedge \mathsf{arem}(\mathbf{skip})\}}{R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}}\ (\textsc{u2b})$$

$$\frac{\begin{array}{c}\vdash_{\text{SL}} [p]C[q] \\ (\llbracket p \rrbracket \ltimes \llbracket q \rrbracket) \Rightarrow G * \mathsf{True} \quad I \triangleright G \quad p \vee q \Rightarrow I * \mathbf{true}\end{array}}{[I], G, I \vdash \{p\}\langle C\rangle\{q\}}\ (\textsc{atom}) \qquad \frac{\begin{array}{c}p \Rrightarrow^{a} p' \quad \vdash_{\text{SL}} [p']C[q'] \quad q' \Rrightarrow^{b} q \quad + \in \{a, b\} \\ (\llbracket p \rrbracket \propto \llbracket q \rrbracket) \Rightarrow G * \mathsf{True} \quad I \triangleright G \quad p \vee q \Rightarrow I * \mathbf{true}\end{array}}{[I], G, I \vdash \{p\}\langle C\rangle\{q\}}\ (\textsc{atom}^{+})$$

$$\frac{[I], G, I \vdash \{p\}\langle C\rangle\{q\} \quad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \quad I \triangleright R}{R, G, I \vdash \{p\}\langle C\rangle\{q\}}\ (\textsc{atom-r}) \qquad \frac{\begin{array}{c}p \Rightarrow (B = B) * I \\ p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp}) \quad R, G, I \vdash \{p'\}C\{p\}\end{array}}{R, G, I \vdash \{p\}\mathbf{while}\ (B)\ C\{p \wedge \neg B\}}\ (\textsc{while})$$

$$\frac{R, G, I \vdash \{p\}C\{q\}}{R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{w}}\}C\{\lfloor q \rfloor_{\mathsf{w}}\}}\ (\textsc{hide-w}) \qquad \frac{R, G, I \vdash \{p\}C\{q\} \quad \mathsf{Sta}(p', R' * \mathsf{Id}) \quad I' \triangleright \{R', G'\} \quad p' \Rightarrow I' * \mathbf{true} \quad G^{+} \Rightarrow G}{R * R', G * G', I * I' \vdash \{p * p'\}C\{q * p'\}}\ (\textsc{frame})$$

Figure 9. Selected unary inference rules.

```
local t;
{ $x = X \wedge \mathsf{arem}(S') \wedge \mathsf{wf}(1)$ }
while (true) {
  { $x = X \wedge \mathsf{arem}(S')$ }
  < t := x; >
  { $x = X = t \wedge \mathsf{arem}(S') \vee$
    $x = X \neq t \wedge \mathsf{arem}(S') \wedge \mathsf{wf}(1)$ }
  cas(&x, t, t+1);
  { $x = X \wedge \mathsf{arem}(S') \wedge \mathsf{wf}(1)$ }
}
```

```
// unfolding cas

< if (x = t)
    { $x = X = t \wedge \mathsf{arem}(S')$ }
    { $x = X = t \wedge \mathsf{arem}(X\texttt{++}; S')$ }
    x := t + 1;
    { $x = X = t + 1 \wedge \mathsf{arem}(S') \wedge \mathsf{wf}(1)$ }
>
```

(a) looping a counter: $I \overset{\text{def}}{=} (x = X)$ $\quad R = G \overset{\text{def}}{=} (I \propto I) \vee [I]$

```
local i := 100;
{ $i \geq 0 \wedge \mathsf{wf}(i) \wedge \mathsf{arem}(\mathbf{skip})$ }
while (i > 0) {
  { $i > 0 \wedge \mathsf{wf}(i\text{-}1) \wedge \mathsf{arem}(\mathbf{skip})$ }
  i--;
  { $i \geq 0 \wedge \mathsf{wf}(i) \wedge \mathsf{arem}(\mathbf{skip})$ }
}
```

(b) local termination:
$I \overset{\text{def}}{=} \mathsf{emp} \qquad R = G \overset{\text{def}}{=} \mathsf{Emp}$

Figure 10. Proofs for two small examples.

at least one of them is the special sign •, which has no knowledge about the remaining source code $\mathbb{C}$. Therefore we know the following holds (for any P and $\mathbb{C}$):

$$(P \wedge \mathsf{arem}(\mathbb{C}) \wedge \mathsf{wf}(1)) * (\mathsf{wf}(1) \wedge \mathsf{emp}) \ \Leftrightarrow\ (P \wedge \mathsf{arem}(\mathbb{C}) \wedge \mathsf{wf}(2))$$

One may think a more natural definition of the disjoint union is to require the two $\mathbb{D}$s be the same. But this would break the FRAME rule (see Fig. 9). For example, we can prove:

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{x = X \wedge \mathsf{arem}(X\texttt{++})\}\ x\texttt{++}\ \{x = X \wedge \mathsf{arem}(\mathbf{skip})\}$$

With the FRAME rule and the separating conjunction based on the alternative definition of disjoint union, we would derive the following:

$$\begin{array}{l}\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{(x = X \wedge \mathsf{arem}(X\texttt{++})) * \mathsf{arem}(X\texttt{++})\} \\ \qquad\qquad\qquad x\texttt{++}\ \{(x = X \wedge \mathsf{arem}(\mathbf{skip})) * \mathsf{arem}(X\texttt{++})\}\end{array}$$

which is reduced to an invalid judgment:

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{x = X \wedge \mathsf{arem}(X\texttt{++})\}\ x\texttt{++}\ \{\mathbf{false}\}$$

We require in $p * q$ that either p or q should *not* specify the source code, therefore in this example the precondition after applying the frame rule is invalid (thus the whole judgment is valid).

The stability of p w.r.t. an action R, defined at the bottom part of Fig. 8, specifies how the number of tokens of a program (specified by p) could change under R's interferences. As a simple example, for the following p, R_1 and R_2, $\mathsf{Sta}(p, R_1)$ holds while $\mathsf{Sta}(p, R_2)$ does not hold:

$$\begin{array}{rcl}p & \overset{\text{def}}{=} & (10 \mapsto 0 * 20 \Mapsto 0) \vee ((10 \mapsto 1 * 20 \Mapsto 0) \wedge \mathsf{wf}(1)) \\ R_1 & \overset{\text{def}}{=} & (10 \mapsto 0 * 20 \Mapsto 0) \propto (10 \mapsto 1 * 20 \Mapsto 0) \\ R_2 & \overset{\text{def}}{=} & (10 \mapsto 0 * 20 \Mapsto 0) \ltimes (10 \mapsto 1 * 20 \Mapsto 0)\end{array}$$

5.2 Unary Inference Rules

The judgment for unary reasoning is in the form of $R, G, I \vdash \{p\}C\{q\}$. We present some of the rules in Fig. 9.

The U2B rule, as explained in Sec. 2, turns unary proofs to binary ones. It says that if the remaining source code is $\mathbb{C}$ at the beginning of the target C, and it becomes **skip** at the end of C, then we know C is simulated by $\mathbb{C}$.

The ATOM rule allows us to reason sequentially about the target code in the atomic block. We use $\vdash_{\text{SL}} [p]C[q]$ to represent the total correctness of C in sequential separation logic. The corresponding rules are mostly standard and elided here. Note that C only accesses the target state σ, therefore in our sequential rules we require the source state Σ and the auxiliary data w and $\mathbb{D}$ in p should remain unchanged in q. We can lift C's total correctness to the concurrent setting as long as its overall transition over the shared states satisfies the guarantee G. Here we assume the environment is identity transitions. To allow general environment behaviors, we can apply the ATOM-R rule later, which requires that R be fenced by I and the pre- and post-conditions be stable w.r.t. R.

The ATOM^{+} rule is similar to the ATOM rule, except that it executes the source code simultaneously with the target atomic step. We use $p \Rrightarrow^{+} q$ for the multi-step executions from the source code specified by p to the code specified by q, which is defined in the bottom part of Fig. 8. We also write $p \Rrightarrow^{0} q$ for the usual implication $p \Rightarrow q$. Then, the ATOM^{+} rule says, we can execute the source code before or after the steps of C, as long as the overall transition (including the source steps *and* the target steps) with the effect bit **true** satisfies G for the shared parts.

The WHILE rule is the key to proving the preservation of termination. As we informally explained in Sec. 2, we should be able to decrease the number of tokens at the beginning of each loop iteration. And we should re-establish the invariant p between the states and the number of tokens at the end of each iteration. Below we give two examples, each of which shows a typical application of the WHILE rule.

Examples. The first example is the T_c'' and S' in Sec. 2. We show its proof in our logic in Fig. 10(a) (for simplicity, below we always assume the ownership of variables). We use `X` for the counter at the source, and the rely/guarantee conditions say that the counters at the two levels can be updated simultaneously with the effect bit **true**. The loop invariant above line 2 says that we should have at least one token to execute the loop. The loop body is verified with zero tokens, and should finally restore the invariant token number 1. The gaining of the token may be due to a successful `cas` at line 4 that corresponds to source steps, or caused by the environment interferences. More specifically, the assertion following line 3 says that we can gain a token if the counters have been updated. If the counters are not updated before the `cas` at line 4, the `cas` succeeds and we show the detailed proof at the right part of Fig. 10(a), in which we execute one iteration of the source code and gain a token (applying the ATOM$^+$ rule).

This example shows the most straightforward understanding of the WHILE rule: we pay a token at the beginning of an iteration and should be able to gain another token during the execution of the iteration. The next example is more subtle (though simpler). As shown in Fig. 10(b), it is a locally-terminating while loop (i.e., a loop that terminates regardless of environment interferences). We prove it refines **skip** under the environment Emp. The loop invariant above line 2 says that the number of tokens equals the value of `i`. If the loop condition (`i>0`) is satisfied, we pay one token. In the proof of the loop body, we do not (and are not able to) gain more tokens. Instead, the value of `i` will be decreased in the iteration, enabling us to restore the equality between the number of tokens and `i`.

Other rules and discussions. Another important rule is the HIDE-W rule in Fig. 9. It shows that tokens are just an auxiliary tool, which could be safely discarded (by using $\lfloor\_\rfloor_{\mathsf{w}}$) when the termination-preservation of a command C (say, a while loop) is already established. As we mentioned in Sec. 2, the HIDE-W rule is crucial to handle *infinite nondeterminism*. It is also important for local reasoning, so that when we verify a thread, we do not have to calculate and specify in the precondition the number of tokens needed by *all* the while loops. For nested loops, we could use the HIDE-W rule to hide the tokens needed by the inner loop, and use the FRAME rule to add back the tokens needed for the outer loop later when we compose the inner loop with other parts of the outer loop body.

The unary FRAME rule in Fig. 9 is similar to the binary one in Fig. 7. Other rules can be found in our TR [13], which are very similar to those in LRG [3], but we give different interpretations to assertions and actions.

The binary rules (in Fig. 7) and the unary rules (in Fig. 9) gives us a full proof theory for termination-preserving refinement. We want to remind the readers that the logic does not ensure termination of programs, therefore it is *not* a logic for total correctness. On the other hand, if we restrict the source code to **skip** (which always terminates), then our unary rules can be viewed as a proof theory for the *total* correctness of concurrent programs.

Also note that the use of a natural number w as the while-specific metric is to simplify the presentation only. It is easy to extend our work to support other types of the while-specific metrics for more complicated examples.

6. More Examples

We have seen a few small examples that illustrate the use of our logic, in particular, the WHILE rule. In this section, we discuss other examples that we have proved, which are summarized in Fig. 11. Their proofs are in TR [13].

Linearizability & Lock-Freedom	Counter and its variants
	Treiber stack [20]
	Michael-Scott lock-free queue [14]
	DGLM lock-free queue [2]
Non-Atomic Object Correctness	Synchronous queue [16]
Correctness of Optimized Algo (Equivalence)	Counter vs. its variants
	TAS lock vs. TTAS lock [6]

Figure 11. Verified examples using our logic.

Proving linearizability and lock-freedom together for concurrent objects. It has been shown [12] that the verification of linearizability and lock-freedom together can be reduced to verifying a contextual refinement that preserves the termination of any client programs. That is, for any client as the context $\mathscr{C}$, the termination-preserving refinement $\mathscr{C}[C] \sqsubseteq \mathscr{C}[\mathbb{C}]$ should hold. Here we use C for the concrete implementation of the object, and $\mathbb{C}$ for the corresponding abstract atomic operations. $\mathscr{C}[C]$ (or $\mathscr{C}[\mathbb{C}]$) denotes the whole program where the client accesses the object via method calls to C (or $\mathbb{C}$).

The compositionality rules of our logic (Fig. 7) allow us to verify the above contextual refinement by proving $R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}$. Then we apply the U2B rule and turn the relational verification to unary reasoning. As in a normal linearizability proof (e.g., [10, 23]), we need to find a single step of C (i.e., the linearization point) that corresponds to the atomic step of $\mathbb{C}$. Here we also have to prove lock-freedom: the failure to make progress (i.e., finish an abstract operation) of a thread must be caused by successful progress of its environment, which can be ensured by the WHILE rule (in Fig. 9) in our logic.

We have used the above approach to verify several linearizable and lock-free objects, including Treiber stack [20], Michael-Scott lock-free queue [14] and DGLM queue [2]. We can further extend the logic in this paper with the techniques [10] for verifying linearizability of algorithms with non-fixed linearization points, to support more sophisticated examples such as HSY elimination-based stack and Harris-Michael lock-free list.

Verifying concurrent objects whose abstract operations are* not *atomic. Sometimes we cannot define single atomic operations as the abstract specification of a concurrent object. For objects that implement synchronization between threads, we may have to explicitly take into account the interferences from other threads when defining the abstract behaviors of the current thread. For example, the synchronous queue [16] is a concurrent transfer channel in which each producer presenting an item must wait for a consumer to take this item, and vice versa. The corresponding abstract operations are no longer atomic. We used our logic to prove the contextual refinement between the concrete implementation (from [16], used in Java 6) and a more abstract synchronous queue. The refinement ensures that if a producer (or a consumer) is blocked at the concrete level, it must also be blocked at the source level.

Proving equivalence between optimized algorithms and original ones. We also use our logic to show variants of concurrent algorithms are correct optimizations of the original implementations. In this case, we show equivalence (in fact, contextual equivalence), i.e., refinements of both directions.

For instance, we proved the TTAS lock implementation is equivalent to the TAS lock implementation [6] for any client using the locks. The former tests the lock bit in a nested while loop until it appears to be free, and then uses the atomic `getAndSet` instruction to update the bit; while the latter directly tries `getAndSet` until success. The equivalence result between these two lock implementations shows that no client may observe their differences, including the differences on their termination behaviors (e.g., whether a client thread may acquire the lock). It gives us the full correctness of the TTAS lock. As an optimization of TAS lock, it preserves the behaviors on both functionality and termination of the latter.

7. Related Work and Conclusion

Hoffmann et al. [7] propose a program logic to verify lock-freedom of concurrent objects. They reason about termination quantitatively by introducing tokens, and model the environment's interference over the current thread's termination in terms of token transfer. The idea is simple and natural, but their logic has very limited support of local reasoning. One needs to know the total number of tokens needed by each thread (which may have multiple while loops) and the (fixed) number of threads, to calculate the number of tokens for a thread to lose or initially own. This requirement also disallows their logic to reason about programs with infinite nondeterminism. Here we allow a thread to set its effect bit in R/G without knowing the details of other threads; and other threads can determine by themselves how many tokens they gain. We also introduce the HIDE-W rule to hide the number of tokens and to support infinite nondeterminism. Another key difference is that our logic supports verification of refinement, which is not supported by their logic.

Gotsman et al. [5] propose program logic and tools to verify lock-freedom. Their approach is more heavyweight in that they need temporal assertions in the rely/guarantee conditions to specify interference between threads, and the rely/guarantee conditions need to be specified iteratively in multiple rounds to break circular reliance on progress. Moreover, their work relies on third-party tools to check termination of individual threads as closed sequential programs. Therefore they do not have a set of self-contained program logic rules and a coherent meta-theory as we do. Like Hoffmann et al. [7], they do not support refinement verification either.

As we explained in Sec. 1, none of recent work on general refinement verification of concurrent programs [11, 21, 22] and on verifying linearizability of concurrent objects [10, 23] (which can be viewed as a specialized refinement problem) preserves termination. Ševčík et al. equipped their simulation proofs for CompCertTSO [17] with a well-founded order, following the CompCert approach. Their approach is similar to our second attempt explained in Sec. 2, thus cannot be applied to prove lock-freedom of concurrent objects.

Conclusion and future work. We propose a new compositional simulation RGSim-T to verify termination-preserving refinement between concurrent programs. We also give a rely/guarantee program logic as a proof theory for the simulation. Our logic is the first to support compositional verification of termination-preserving refinement. The simulation and logic are general. They can be used to verify both correctness of optimizations (where the source may not necessarily terminate) and lock-freedom of concurrent objects. As future work, we would like to further extend them with the techniques of pending thread pools and speculations [10] to verify objects with non-fixed linearization points. We also hope to explore the possibility of building tools to automate the verification.

Acknowledgments

We thank anonymous referees for their suggestions and comments. This work is supported in part by China Scholarship Council, National Natural Science Foundation of China (NSFC) under Grant Nos. 61229201, 61379039 and 91318301, and the National Hi-Tech Research and Development Program of China (Grant No. 2012AA010901). It is also supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0915888 and 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

[2] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, pages 97–114, 2004.

[3] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[4] I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[5] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28, 2009.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[7] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative reasoning for proving lock-freedom. In *LICS*, pages 124–133, 2013.

[8] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[9] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43:363–446, December 2009.

[10] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[11] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, pages 455–468, 2012.

[12] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, pages 227–241, 2013.

[13] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs (extended version). Technical report, Univ. of Science and Technology of China, May 2014. `http://kyhcs.ustcsz.edu.cn/relconcur/rgsimt`.

[14] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[15] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, pages 137–146, 2006.

[16] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP*, pages 147–156, 2006.

[17] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[18] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *PLDI*, pages 471–482, 2013.

[19] K. Stølen. A method for the development of totally correct shared-state parallel programs. In *CONCUR*, pages 510–525, 1991.

[20] R. K. Treiber. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[21] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[22] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356, 2013.

[23] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, Computer Laboratory, 2008.

[24] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, pages 335–351, 2011.

[25] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375:308–334, 2007.