A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations

Hongjin Liang   Xinyu Feng   Ming Fu

School of Computer Science and Technology
University of Science and Technology of China

lhj1018@mail.ustc.edu.cn   xyfeng@ustc.edu.cn   fuming@ustc.edu.cn

Abstract

Verifying program transformations usually requires proving that the resulting program (the target) refines or is equivalent to the original one (the source). However, the refinement relation between individual sequential threads cannot be preserved in general with the presence of parallel compositions, due to instruction reordering and the different granularities of atomic operations at the source and the target. On the other hand, the refinement relation defined based on fully abstract semantics of concurrent programs assumes arbitrary parallel environments, which is too strong and cannot be satisfied by many well-known transformations.

In this paper, we propose a Rely-Guarantee-based Simulation (RGSim) to verify concurrent program transformations. The relation is parametrized with constraints of the environments that the source and the target programs may compose with. It considers the interference between threads and their environments, thus is less permissive than relations over sequential programs. It is compositional *w.r.t.* parallel compositions as long as the constraints are satisfied. Also, RGSim does not require semantics preservation under all environments, and can incorporate the assumptions about environments made by specific program transformations in the form of rely/guarantee conditions. We use RGSim to reason about optimizations and prove atomicity of concurrent objects. We also propose a general garbage collector verification framework based on RGSim, and verify the Boehm *et al.* concurrent mark-sweep GC.

1. Introduction

Many verification problems can be reduced to verifying program transformations, *i.e.*, proving the target program of the transformation has no more observable behaviors than the source. Below we give some typical examples in concurrent settings:

- *Correctness of compilation and optimizations of concurrent programs*. In this most natural program transformation verification problem, every compilation phase does a program transformation $\mathbf{T}$, which needs to preserve the semantics of the inputs.

- *Atomicity of concurrent objects*. A concurrent object or library provides a set of methods that allow clients to manipulate the shared data structure with abstract atomic behaviors [13]. Their correctness can be reduced to the correctness of the transformation from abstract atomic operations to concrete and executable programs in a concurrent context.

- *Verifying implementations of software transactional memory (STM)*. Many languages supporting STM provide a high-level atomic block **atomic**$\{\mathbb{C}\}$, so that programmers can assume the atomicity of the execution of $\mathbb{C}$. Atomic blocks are implemented using some STM protocol (*e.g.*, TL2 [10]) that allows very fine-grained interleavings. Verifying that the fine-grained

program respects the semantics of atomic blocks gives us the correctness of the STM implementation.

- *Correctness of concurrent garbage collectors (GCs)*. High-level garbage-collected languages (*e.g.*, Java) allow programmers to work at an abstract level without knowledge of the underlying GC algorithm. However, the concrete and executable low-level program involves interactions between the mutators and the collector. If we view the GC implementation as a transformation from high-level mutators to low-level ones with a concrete GC thread, the GC safety can be reduced naturally to the semantics preservation of the transformation.

To verify the correctness of a program transformation $\mathbf{T}$, we follow the standard approach [17] and define a refinement relation $\sqsubseteq$ between the target and the source programs, which says the target has no more observable behaviors than the source. Then we can formalize the correctness of the transformation as follows:

$$\mathsf{Correct}(\mathbf{T}) \;\triangleq\; \forall C, \mathbb{C}.\; C = \mathbf{T}(\mathbb{C}) \implies C \sqsubseteq \mathbb{C}. \qquad (1.1)$$

That is, for any source program $\mathbb{C}$ acceptable by $\mathbf{T}$, $\mathbf{T}(\mathbb{C})$ is a refinement of $\mathbb{C}$. When the source and the target are shared-state concurrent programs, the refinement $\sqsubseteq$ needs to satisfy the following requirements to support effective proof of $\mathsf{Correct}(\mathbf{T})$:

- Since the target $\mathbf{T}(\mathbb{C})$ may be in a different language from the source, the refinement should be general and independent of the language details.

- To verify fine-grained implementations of abstract operations, the refinement should support different views of program states and different granularities of state accesses at the source and the target levels.

- When $\mathbf{T}$ is syntax-directed (and it is usually the case for parallel compositions, *i.e.*, $\mathbf{T}(\mathbb{C} \parallel \mathbb{C}') = \mathbf{T}(\mathbb{C}) \parallel \mathbf{T}(\mathbb{C}')$), a *compositional* refinement is of particular importance for modular verification of $\mathbf{T}$.

However, existing refinement (or equivalence) relations cannot satisfy all these requirements at the same time. Contextual equivalence, the canonical notion for comparing program behaviors, fails to handle different languages since the contexts of the source and the target will be different. Simulations and logical relations have been used to verify compilation [4, 14, 17, 18], but they are usually designed for sequential programs (except [18, 22], which we will discuss in Section 8). Since the refinement or equivalence relation between sequential threads cannot be preserved in general with parallel compositions, we cannot simply adapt existing work on sequential programs to verify transformations of concurrent programs. Refinement relations based on fully abstract semantics of concurrent programs are compositional, but they assume arbitrary

program contexts, which is too strong for many practical transformations. We will explain the challenges in detail in Section 2.

In this paper, we propose a <u>R</u>ely-<u>G</u>uarantee-based <u>Sim</u>ulation (RGSim) for compositional verification of concurrent transformations. By addressing the above problems, we make the following contributions:

- RGSim parametrizes the simulation between concurrent programs with rely/guarantee conditions [15], which specify the interactions between the programs and their environments. This makes the corresponding refinement relation compositional *w.r.t.* parallel compositions, allowing us to decompose refinement proofs for multi-threaded programs into proofs for individual threads. On the other hand, the rely/guarantee conditions can incorporate the assumptions about environments made by specific program transformations, so RGSim can be applied to verify many practical transformations.

- Based on the simulation technique, RGSim focuses on comparing externally observable behaviors (*e.g.*, I/O events) only, which gives us considerable leeway in the implementations of related programs. The relation is mostly independent of the language details. It can be used to relate programs in different languages with different views of program states and different granularities of atomic state accesses.

- RGSim makes relational reasoning about optimizations possible in parallel contexts, We present a set of relational reasoning rules to characterize and justify common optimizations in a concurrent setting, including hoisting loop invariants, strength reduction and induction variable elimination, dead code elimination, redundancy introduction, *etc.*.

- RGSim gives us a refinement-based proof method to verify fine-grained implementations of abstract algorithms and concurrent objects. We successfully apply RGSim to verify concurrent counters, the concurrent GCD algorithm, Treiber's non-blocking stack and the lock-coupling list.

- We reduce the problem of verifying concurrent garbage collectors to verifying transformations, and present a general GC verification framework, which combines unary rely-guarantee-based verification [15] with relational proofs based on RGSim.

- We verify the Boehm *et al.* concurrent garbage collection algorithm [7] using our framework. As far as we know, it's the first time to formally prove the correctness of this algorithm.

In the rest of this paper, we first analyze the challenges for compositional verification of concurrent program transformations, and explain our approach informally in Section 2. Then we give the basic technical settings in Section 3 and present the formal definition of RGSim in Section 4. We show the use of RGSim to reason about optimizations in Section 5, verify atomicity of concurrent objects in Section 6, and prove the correctness of concurrent GCs in Section 7. Finally we discuss related work and conclude in Section 8.

2. Challenges and Our Approach

The major challenge we face is to have a compositional refinement relation $\sqsubseteq$ between concurrent programs, *i.e.*, we should be able to know $\mathbf{T}(\mathbb{C}_1) \parallel \mathbf{T}(\mathbb{C}_2) \sqsubseteq \mathbb{C}_1 \parallel \mathbb{C}_2$ if we have $\mathbf{T}(\mathbb{C}_1) \sqsubseteq \mathbb{C}_1$ and $\mathbf{T}(\mathbb{C}_2) \sqsubseteq \mathbb{C}_2$.

2.1 Sequential Refinement Loses Parallel Compositionality

Observable behaviors of sequential imperative programs usually refer to their control effects (*e.g.*, termination and exceptions) and final program states. However, refinement relations defined correspondingly cannot be preserved after parallel compositions. It has been a well-known fact in the compiler community that sound

```
local r1;              local r2;
x  := 1;               y  := 1;
r1 := y;        ||     r2 := x;
if (r1 = 0) then       if (r2 = 0) then
  critical region         critical region
```

(a) Dekker's Mutual Exclusion Algorithm

```
           x := x+1;  ||  x := x+1;
```
                        vs.
```
    local r1;              local r2;
    r1 := x;        ||     r2 := x;
    x  := r1 + 1;          x  := r2 + 1;
```

(b) Different Granularities of Atomic Operations

Figure 1. Equivalence Lost after Parallel Composition

optimizations for sequential programs may change the behaviors of multi-threaded programs [5]. The Dekker's algorithm shown in Figure 1(a) has been widely used to demonstrate the problem. Reordering the first two statements of the thread on the left preserves its sequential behaviors, but the whole program can no longer ensure exclusive access to the critical region.

In addition to instruction reordering, the different granularities of atomic operations between the source and the target programs can also break the compositionality of program equivalence in a concurrent setting. In Figure 1(b), the target program at the bottom behaves differently from the source at the top (assuming each statement is executed atomically), although the individual threads at the target and the source have the same behaviors.

2.2 Assuming Arbitrary Environments is Too Strong

The problem with the refinement for sequential programs is that it does not consider the effects of threads' intermediate state accesses on their parallel environments. People have given fully abstract semantics to concurrent programs (*e.g.*, [1, 8]). The semantics of a program is modeled as a set of execution traces. Each trace is an interleaving of state transitions made by the program itself and *arbitrary* transitions made by the environment. Then the refinement between programs can be defined as the subset relation between the corresponding trace sets. Since it considers all possible environments, the refinement relation has very nice compositionality, but unfortunately is too strong to formulate the correctness of many well-known transformations, including the four classes of transformations mentioned before:

- Many concurrent languages (*e.g.*, C++ [6]) do not give semantics to programs with data races (like the examples shown in Figure 1). Therefore the compilers only need to guarantee the semantics preservation of data-race-free programs.

- When we prove that a fine-grained implementation of a concurrent object is a refinement of an abstract atomic operation, we can assume that all accesses to the object in the context of the target program use the same set of primitives.

- Usually the implementation of STM (*e.g.*, TL2 [10]) ensures the atomicity of a transaction **atomic**$\{\mathbb{C}\}$ only when there are no data races. Therefore, the correctness of the transformation from high-level atomic blocks to fine-grained concurrent code assumes data-race-freedom in the source.

- Many garbage-collected languages are type-safe and prohibit operations such as pointer arithmetics. Therefore the garbage collector could make corresponding assumptions about the mutators that run in parallel.

In all these cases, the transformations of individual threads are allowed to make various assumptions about the environments. They do not have to ensure semantics preservation within all contexts.

2.3 Languages at Source and Target May Be Different

The use of different languages at the source and the target levels makes the formulation of the transformation correctness more difficult. If the source and the target languages have different views of program states and different atomic primitives, we cannot directly compare the state transitions made by the source and the target programs. This is another reason that makes the aforementioned subset relation between sets of program traces in fully abstract semantics infeasible. For the same reason, many existing techniques for proving refinement or equivalence of programs in the same language cannot be applied either.

2.4 Different Observers Make Different Observations

Concurrency introduces tensions between two kinds of observers: the outside observers as human beings and the parallel program contexts. Outside observers do not care about the implementation details of the source and the target programs. For them, intermediate state accesses (such as memory reads and writes) are silent steps (unobservable), and only external events (such as I/O operations) are observable. On the other hand, state accesses have effects on the parallel program contexts, and are not silent to them.

If the refinement relation relates externally observable event traces only, it cannot have parallel compositionality, as we explained in Section 2.1. On the other hand, relating all state accesses of programs is too strong. Any reordering of state accesses or change of atomicity would fail the refinement.

2.5 Our Approach

In this paper we propose a Rely-Guarantee-based Simulation (RGSim) $\preceq$ between the target and the source programs. It establishes a weak simulation, ensuring that for every externally observable event made by the target program there is a corresponding one in the source. We choose to view intermediate state accesses as silent steps, thus we can relate programs with different implementation details. This also makes our simulation independent of language details.

To support parallel compositionality, our relation takes into account explicitly the expected interference between threads and their parallel environments. Inspired by the rely-guarantee (R-G) verification method [15], the relation $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ is now parametrized with the interference specified using rely/guarantee conditions. It talks about not only the target C and the source $\mathbb{C}$, but also the interference $\mathcal{R}$ and $\mathcal{G}$ between C and its target-level environment, and $\mathbb{R}$ and $\mathbb{G}$ between $\mathbb{C}$ and its environment at the source level. Here the rely condition $\mathcal{R}$ (or $\mathbb{R}$) specifies the permitted state transitions that the environment may have, and $\mathcal{G}$ (or $\mathbb{G}$) specifies the possible transitions made by the thread itself.

Informally, $(C, \mathcal{R}, \mathcal{G}) \preceq (\mathbb{C}, \mathbb{R}, \mathbb{G})$ says the executions of C under the environment $\mathcal{R}$ do not exhibit more observable behaviors than the executions of $\mathbb{C}$ under the environment $\mathbb{R}$, and the state transitions of C and $\mathbb{C}$ satisfy $\mathcal{G}$ and $\mathbb{G}$ respectively. RGSim is now compositional, as long as the threads are composed with well-behaved environments only. The parallel compositionality lemma is in the following form. If we know $(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq (\mathbb{C}_1, \mathbb{R}_1, \mathbb{G}_1)$ and $(C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq (\mathbb{C}_2, \mathbb{R}_2, \mathbb{G}_2)$, and also the interference constraints are satisfied, *i.e.*, $\mathcal{G}_2 \subseteq \mathcal{R}_1$, $\mathcal{G}_1 \subseteq \mathcal{R}_2$, $\mathbb{G}_2 \subseteq \mathbb{R}_1$ and $\mathbb{G}_1 \subseteq \mathbb{R}_2$, we could get

$$(C_1 \,\|\, C_2, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq (\mathbb{C}_1 \,\|\, \mathbb{C}_2, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2).$$

The compositionality of RGSim gives us a proof theory for concurrent program transformations.

$$(Events) \quad e \quad ::= \quad \ldots \qquad (Labels) \;\; o \;\; ::= \;\; e \mid \tau$$

(a) Events and Transition Labels

$$
\begin{aligned}
(LState) & \quad \sigma \quad ::= \; \ldots \\
(LExpr) & \quad E \; \in \; LState \to Int_\perp \\
(LBExp) & \quad B \; \in \; LState \to \{\mathbf{true}, \mathbf{false}\}_\perp \\
(LInstr) & \quad c \; \in \; LState \rightharpoonup \mathcal{P}((Labels \times LState) \cup \{\mathbf{abort}\}) \\
(LStmt) & \quad C \; ::= \; \mathbf{skip} \mid c \mid C_1; C_2 \mid \mathbf{if}\,(B)\,C_1\,\mathbf{else}\,C_2 \\
& \qquad \mid \; \mathbf{while}\,(B)\,C \mid C_1 \| C_2 \\
(LStep) & \quad \longrightarrow_L \; \in \; LStmt/\{\mathbf{skip}\} \times LState \\
& \qquad\qquad \to \mathcal{P}((Labels \times LStmt \times LState) \cup \{\mathbf{abort}\})
\end{aligned}
$$

(b) The Low-Level Language

$$
\begin{aligned}
(HState) & \quad \Sigma \quad ::= \; \ldots \\
(HExpr) & \quad \mathbb{E} \; \in \; HState \to Int_\perp \\
(HBExp) & \quad \mathbb{B} \; \in \; HState \to \{\mathbf{true}, \mathbf{false}\}_\perp \\
(HInstr) & \quad \mathsf{c} \; \in \; HState \rightharpoonup \mathcal{P}((Labels \times HState) \cup \{\mathbf{abort}\}) \\
(HStmt) & \quad \mathbb{C} \; ::= \; \mathbf{skip} \mid \mathsf{c} \mid \mathbb{C}_1; ; \mathbb{C}_2 \mid \mathbf{if}\,\mathbb{B}\,\mathbf{then}\,\mathbb{C}_1\,\mathbf{else}\,\mathbb{C}_2 \\
& \qquad \mid \; \mathbf{while}\,\mathbb{B}\,\mathbf{do}\,\mathbb{C} \mid \mathbb{C}_1 \,\|\!|\, \mathbb{C}_2 \\
(HStep) & \quad \longrightarrow_H \; \in \; HStmt/\{\mathbf{skip}\} \times HState \\
& \qquad\qquad \to \mathcal{P}((Labels \times HStmt \times HState) \cup \{\mathbf{abort}\})
\end{aligned}
$$

(c) The High-Level Language

Figure 2. Generic Languages at Target and Source Levels

Also different from fully abstract semantics for threads, which assumes arbitrary behaviors of environments, RGSim allows us to instantiate the interference $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$ differently for different assumptions about environments, therefore it can be used to verify the aforementioned four classes of transformations. For instance, if we want to prove that a transformation preserves the behaviors of data-race-free programs, we can specify the data-race-freedom in $\mathbb{R}$ and $\mathbb{G}$. Then we are no longer concerned with the examples in Figure 1, both of which have data races.

3. Basic Technical Settings

In this section, we present the source and the target programming languages. Then we define a basic refinement $\sqsubseteq$, which relates the externally observable event traces of programs.

3.1 The Languages

Following standard simulation techniques, we model the semantics of target and source programs as labeled transition systems. Before showing the languages, we first define events and labels in Figure 2(a). We leave the set of events unspecified here. It can be instantiated by program verifiers, depending on their interest (*e.g.*, input/output events). A label that will be associated with a state transition is either an event or τ, which means the corresponding transition does not generate any event (*i.e.*, a silent step).

The target language, which we also call the low-level language, is shown in Figure 2(b). We abstract away the forms of states, expressions and primitive instructions in the language. An instruction is a partial function from states to sets of label and state pairs, describing the state transitions and the events it generates. Unsafe executions lead to **abort**. Note that the semantics of an instruction could be non-deterministic. Moreover, it might be undefined on some states, making it possible to model blocking operations such as acquiring a lock.

$$\frac{(\tau, \Sigma') \in \mathrm{c}\, \Sigma}{(\mathrm{c}, \Sigma) \longrightarrow (\textbf{skip}, \Sigma')} \qquad \frac{(e, \Sigma') \in \mathrm{c}\, \Sigma}{(\mathrm{c}, \Sigma) \overset{e}{\longrightarrow} (\textbf{skip}, \Sigma')} \qquad \frac{\textbf{abort} \in \mathrm{c}\, \Sigma}{(\mathrm{c}, \Sigma) \longrightarrow \textbf{abort}} \qquad \frac{\Sigma \notin dom(\mathrm{c})}{(\mathrm{c}, \Sigma) \longrightarrow (\mathrm{c}, \Sigma)}$$

$$\frac{}{(\textbf{skip} \| \textbf{skip}, \Sigma) \longrightarrow (\textbf{skip}, \Sigma)} \qquad \frac{(\mathbb{C}_1, \Sigma) \longrightarrow (\mathbb{C}_1', \Sigma')}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \longrightarrow (\mathbb{C}_1' \| \mathbb{C}_2, \Sigma')} \qquad \frac{(\mathbb{C}_2, \Sigma) \longrightarrow (\mathbb{C}_2', \Sigma')}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \longrightarrow (\mathbb{C}_1 \| \mathbb{C}_2', \Sigma')}$$

$$\frac{(\mathbb{C}_1, \Sigma) \overset{e}{\longrightarrow} (\mathbb{C}_1', \Sigma')}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \overset{e}{\longrightarrow} (\mathbb{C}_1' \| \mathbb{C}_2, \Sigma')} \qquad \frac{(\mathbb{C}_2, \Sigma) \overset{e}{\longrightarrow} (\mathbb{C}_2', \Sigma')}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \overset{e}{\longrightarrow} (\mathbb{C}_1 \| \mathbb{C}_2', \Sigma')} \qquad \frac{(\mathbb{C}_1, \Sigma) \longrightarrow \textbf{abort} \ \ \text{or} \ \ (\mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}{(\mathbb{C}_1 \| \mathbb{C}_2, \Sigma) \longrightarrow \textbf{abort}}$$

Figure 3. Operational Semantics of the High-Level Language

Statements are either primitive instructions or compositions of them. **skip** is a special statement used as a flag to show the end of executions. A single-step execution of statements is modeled as a labeled transition $\_ \longrightarrow_L \_$, which maps a program configuration (a pair of statement and state) to sets of label and configuration pairs. It is undefined when the initial statement is **skip**. The step aborts if an unsafe instruction is executed.

The high-level language (source language) is defined similarly in Figure 2(c), but it is important to note that its states and primitive instructions may be different from those in the low-level language. The compound statements are almost the same as their low-level counterparts. $\mathbb{C}_1;;\mathbb{C}_2$ and $\mathbb{C}_1 \| \mathbb{C}_2$ are sequential and parallel compositions of $\mathbb{C}_1$ and $\mathbb{C}_2$ respectively. Note that we choose to use the same set of compound statements in the two languages for simplicity only. This is not required by our simulation relation, which only needs both languages to have parallel compositions.

Figure 3 shows part of the definition of $\_ \overset{\_}{\longrightarrow}_H \_$, which gives the high-level operational semantics of statements. We often omit the subscript H (or L) in $\_ \overset{\_}{\longrightarrow}_H \_$ (or $\_ \overset{\_}{\longrightarrow}_L \_$). We put the label on top of the arrow in our notation, and omit it when it is τ. The semantics is mostly standard. We only show the rules for primitive instructions and parallel compositions here. Note that when a primitive instruction c is blocked at state Σ (*i.e.*, $\Sigma \notin dom(\mathrm{c})$), we let the program configuration reduce to itself. Primitive instructions in the high-level and low-level languages are *atomic* in the interleaving semantics. Below we use $\_ \longrightarrow^* \_$ for zero or multiple-step transitions with no events generated, and $\_ \overset{e}{\longrightarrow}^* \_$ for multiple-step transitions with *only one* event e generated.

3.2 The Event Trace Refinement

Now we can formally define the refinement relation $\sqsubseteq$ that relates the set of externally observable event traces generated by the target and the source programs. A trace is a sequence of events e, and may end with a termination marker **done** or a fault marker **abort**.

$$(EvtTrace) \quad \mathcal{E} \quad ::= \quad \epsilon \ | \ \textbf{done} \ | \ \textbf{abort} \ | \ e :: \mathcal{E}$$

Definition 1 (Event Trace Set). $ETrSet_n(C, \sigma)$ represents a set of external event traces produced by C in n steps from the state σ:

- $ETrSet_0(C, \sigma) \triangleq \{\epsilon\}$;
- $ETrSet_{n+1}(C, \sigma) \triangleq$
  $\{\mathcal{E} \ | \ (C, \sigma) \longrightarrow (C', \sigma') \wedge \mathcal{E} \in ETrSet_n(C', \sigma')$
  $\quad \vee (C, \sigma) \overset{e}{\longrightarrow} (C', \sigma') \wedge \mathcal{E}' \in ETrSet_n(C', \sigma') \wedge \mathcal{E} = e :: \mathcal{E}'$
  $\quad \vee (C, \sigma) \longrightarrow \textbf{abort} \wedge \mathcal{E} = \textbf{abort}$
  $\quad \vee C = \textbf{skip} \wedge \mathcal{E} = \textbf{done}\}$.

We define $ETrSet(C, \sigma)$ as $\bigcup_n ETrSet_n(C, \sigma)$.

We overload the notation and use $ETrSet(\mathbb{C}, \Sigma)$ for the high-level language. Then we define an event trace refinement as the subset relation between event trace sets.

Definition 2 (Event Trace Refinement). We say (C, σ) is an *e-trace refinement* of $(\mathbb{C}, \Sigma)$, *i.e.*, $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$, if and only if

$$\langle \mathcal{R}, \mathbb{R} \rangle_\alpha \triangleq \{((\sigma, \sigma'), (\Sigma, \Sigma')) \ | \ (\sigma, \sigma') \in \mathcal{R} \wedge (\Sigma, \Sigma') \in \mathbb{R}^* \\ \wedge (\sigma, \Sigma) \in \alpha \wedge (\sigma', \Sigma') \in \alpha\}$$

$$\mathsf{InitRel}_\mathbf{T}(\zeta) \triangleq \forall \sigma, \Sigma. \ \sigma = \mathbf{T}(\Sigma) \implies (\sigma, \Sigma) \in \zeta$$

$$B \Leftrightarrow \mathbb{B} \triangleq \{(\sigma, \Sigma) \ | \ B\, \sigma = \mathbb{B}\, \Sigma\} \qquad B \wedge \mathbb{B} \triangleq \{(\sigma, \Sigma) \ | \ B\, \sigma \wedge \mathbb{B}\, \Sigma\}$$

$$\mathsf{Intuit}(\alpha) \triangleq \forall \sigma, \Sigma, \sigma', \Sigma'. \ (\sigma, \Sigma) \in \alpha \wedge \sigma \subseteq \sigma' \wedge \Sigma \subseteq \Sigma' \\ \implies (\sigma', \Sigma') \in \alpha$$

$$\eta \# \alpha \triangleq (\eta \cap \alpha) \subseteq (\eta \uplus \alpha)$$

$$\beta \circ \alpha \triangleq \{(\sigma, \Sigma) \ | \ \exists \theta. \ (\sigma, \theta) \in \alpha \wedge (\theta, \Sigma) \in \beta\}$$

$$\alpha \uplus \beta \triangleq \{(\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2) \ | \ (\sigma_1, \Sigma_1) \in \alpha \wedge (\sigma_2, \Sigma_2) \in \beta\}$$

$$\mathsf{Id} \triangleq \{(\sigma, \sigma) \ | \ \sigma \in LState\} \qquad \mathsf{True} \triangleq \{(\sigma, \sigma') \ | \ \sigma, \sigma' \in LState\}$$

Figure 4. Auxiliary Definitions for RGSim

$$ETrSet(C, \sigma) \subseteq ETrSet(\mathbb{C}, \Sigma).$$

The refinement is defined for program configurations instead of for code only because the initial states may affect the behaviors of programs. In this case, the transformation $\mathbf{T}$ should translate states as well as code. We overload the notation and use $\mathbf{T}(\Sigma)$ to represent state transformation, and use $C \sqsubseteq_\mathbf{T} \mathbb{C}$ for

$$\forall \sigma, \Sigma. \ \sigma = \mathbf{T}(\Sigma) \implies (C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma),$$

then $\mathsf{Correct}(\mathbf{T})$ defined in formula (1.1) can be reformulated as

$$\mathsf{Correct}(\mathbf{T}) \triangleq \forall C, \mathbb{C}. \ C = \mathbf{T}(\mathbb{C}) \implies C \sqsubseteq_\mathbf{T} \mathbb{C}. \qquad (3.1)$$

The e-trace refinement relation is defined directly over the externally observable behaviors of programs. It is intuitive, and also abstract in that it is independent of language details. However, as we explained before, it is not compositional *w.r.t.* parallel compositions. In the next section we propose RGSim, which can be viewed as a compositional proof technique that allows us to derive the simple e-trace refinement.

4. The RGSim Relation

4.1 The Definition

We define RGSim in Definition 3. The co-inductively defined relation $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha; \gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$ is a simulation between program configurations (C, σ) and $(\mathbb{C}, \Sigma)$. It is parametrized with the rely and guarantee conditions at the low level and the high level, which are binary relations over states:

$$\mathcal{R}, \mathcal{G} \in \mathcal{P}(LState \times LState), \qquad \mathbb{R}, \mathbb{G} \in \mathcal{P}(HState \times HState).$$

The simulation also takes two additional parameters α and γ, which are relations between the low-level and the high-level states:

$$\alpha, \gamma, \zeta \in \mathcal{P}(LState \times HState).$$

The definition below uses $\langle \mathcal{R}, \mathbb{R} \rangle_\alpha$, the α-*related transitions* of $\mathcal{R}$ and $\mathbb{R}$. As defined in Figure 4, $\langle \mathcal{R}, \mathbb{R} \rangle_\alpha$ puts together the

corresponding transitions of $\mathcal{R}$ and $\mathbb{R}$ that can be related by α. Here a step in $\mathcal{R}$ may correspond to zero or multiple steps of $\mathbb{R}$.

Definition 3 (RGSim). Whenever $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(\sigma, \Sigma) \in \alpha$ and the following are true:

1. if $(C, \sigma) \longrightarrow (C', \sigma')$, then $(\sigma, \sigma') \in \mathcal{G}$ and there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\mathbb{C}', \Sigma')$, $(\Sigma, \Sigma') \in \mathbb{G}^*$ and $(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$;

2. if $(C, \sigma) \xrightarrow{e} (C', \sigma')$, then $(\sigma, \sigma') \in \mathcal{G}$ and there exist $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \xrightarrow{e}^* (\mathbb{C}', \Sigma')$, $(\Sigma, \Sigma') \in \mathbb{G}^*$ and $(C', \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}', \Sigma', \mathbb{R}, \mathbb{G})$;

3. if $C = \textbf{skip}$, then there exists Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\textbf{skip}, \Sigma')$, $(\Sigma, \Sigma') \in \mathbb{G}^*$ and $(\sigma, \Sigma') \in \gamma$;

4. if $(C, \sigma) \longrightarrow \textbf{abort}$, then $(\mathbb{C}, \Sigma) \longrightarrow^* \textbf{abort}$;

5. if $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R} \rangle_\alpha$, then $(C, \sigma', \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma', \mathbb{R}, \mathbb{G})$.

Then, $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$ iff for all σ and Σ, if $(\sigma, \Sigma) \in \zeta$, then $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$.

Informally, $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$ says the low-level configuration (C, σ) is simulated by the high-level configuration $(\mathbb{C}, \Sigma)$ with behaviors $\mathcal{G}$ and $\mathbb{G}$ respectively, no matter how their environments $\mathcal{R}$ and $\mathbb{R}$ interfere with them. It requires the following hold for every execution of C:

- Starting from α-related states, each step of C corresponds to zero or multiple steps of $\mathbb{C}$, and the resulting states are α-related too. If an external event is produced in the step of C, the same event should be produced by $\mathbb{C}$.

- The α relation reflects the abstractions from the low-level machine model to the high-level one, and is preserved by related steps at the two levels (so it is an *invariant*). For instance, in optimizations where the target machine is the same as the source, the α relation usually requires that the shared states be identical, and the thread-local parts be arbitrarily related.

- The executions of C and $\mathbb{C}$ need to satisfy their guarantee conditions $\mathcal{G}$ and $\mathbb{G}$ respectively. The guarantees are abstractions of the programs' behaviors. As we will show later, they will serve as the rely conditions of the parallel environments at the time of parallel compositions.

- If C terminates, then $\mathbb{C}$ terminates as well, and the final states should be related by the *postcondition* γ. Here we assume $\gamma \subseteq \alpha$, *i.e.*, the postcondition is stronger than the invariant.

- C is not safe only if $\mathbb{C}$ is not safe either. This means the transformation should not make a safe high-level program unsafe at the low level.

- Whatever the low-level environment $\mathcal{R}$ and the high-level one $\mathbb{R}$ do, as long as the state transitions are α-related, they should not affect the simulation between C and $\mathbb{C}$.

Then based on the simulation, we hide the states by the *precondition* ζ and define the RGSim relation between programs only. It's easy to see that $\zeta \subseteq \alpha$ if $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, *i.e.*, the precondition needs to be stronger than the invariant.

RGSim is sound *w.r.t.* the e-trace refinement (Definition 2). That is, $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$ ensures that (C, σ) does not have more observable behaviors than $(\mathbb{C}, \Sigma)$.

Theorem 4 (Soundness). *If there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α and γ such that $(C, \sigma, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma} (\mathbb{C}, \Sigma, \mathbb{R}, \mathbb{G})$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.*

For program transformations, since the initial state for the target program is transformed from the initial state for the source, we use

$\mathsf{InitRel}_\mathbf{T}(\zeta)$ (defined in Figure 4) to say the transformation $\mathbf{T}$ over states ensures the binary precondition ζ.

Corollary 5. *If there exist $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$, $\mathbb{G}$, α, ζ and γ such that $\mathsf{InitRel}_\mathbf{T}(\zeta)$ and $(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})$, then $C \sqsubseteq_\mathbf{T} \mathbb{C}$.*

4.2 Compositionality Rules

RGSim is compositional *w.r.t.* various program constructs, including parallel compositions. We present the compositionality rules in Figure 5, which gives us a relational proof method for concurrent program transformations.

As in the R-G logic [15], we require that pre- and postconditions be *stable* under the interference from the environments. Here we introduce the concept of stability of a relation ζ *w.r.t.* the α-related transitions of $\mathcal{R}$ and $\mathbb{R}$.

Definition 6 (Stability). $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R} \rangle_\alpha)$ holds iff for all σ, σ', Σ and Σ', if $(\sigma, \Sigma) \in \zeta$ and $((\sigma, \sigma'), (\Sigma, \Sigma')) \in \langle \mathcal{R}, \mathbb{R} \rangle_\alpha$, then $(\sigma', \Sigma') \in \zeta$.

It says whenever ζ holds initially and $\mathcal{R}$ and $\mathbb{R}$ perform related actions, the result states still satisfy ζ. It's easy to see that α itself is stable *w.r.t.* any α-related transitions, *i.e.*, $\mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R} \rangle_\alpha)$. Another simple example is given below, where both environments could increment x and the unary stable assertion $\{x \geq 0\}$ is lifted to the relation ζ:

$$\zeta \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x}) \geq 0\} \quad \alpha \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$$
$$\mathcal{R} \triangleq \{(\sigma, \sigma') \mid \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \sigma(\mathtt{x}) + 1\}\}$$
$$\mathbb{R} \triangleq \{(\Sigma, \Sigma') \mid \Sigma' = \Sigma\{\mathtt{x} \rightsquigarrow \Sigma(\mathtt{x}) + 1\}\}$$

We can prove $\mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathbb{R} \rangle_\alpha)$. Stability is assumed to be an implicit side-condition at every proof rule. We also require implicitly that the relies and guarantees are closed over identity transitions, since stuttering steps will not affect observable event traces.

The rules SKIP, SEQ, IF and WHILE reveal a high degree of similarity to the corresponding inference rules in Hoare logic. In the SEQ rule, γ serves as the postcondition of C_1 and $\mathbb{C}_1$ and the precondition of C_2 and $\mathbb{C}_2$ at the same time. The IF rule requires the conditions of both sides to be evaluated to the same value under the precondition ζ. We give the definitions of the sets $B \Leftrightarrow \mathbb{B}$ and $B \wedge \mathbb{B}$ in Figure 4. The rule also requires the precondition ζ to be stronger than the invariant α. In the WHILE rule, the γ relation is viewed as a loop invariant preserved at the loop entry point.

The PAR rule shows parallel compositionality of RGSim. The interference constraints say that two threads can be composed in parallel only if one thread's guarantee implies the rely of the other. After parallel composition, they are expected to run in the common environment and their guaranteed behaviors contain each single thread's behaviors.

We also develop some other useful rules about RGSim. For example, the STREN-α rule allows us to replace the invariant α by a stronger invariant α'. We need to check that α' is indeed an invariant preserved by the related program steps, *i.e.*, $\mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G} \rangle_\alpha)$ holds. Symmetrically, the WEAKEN-α rule requires α to be preserved by environment steps related by the weaker invariant α'. As usual, the pre/post conditions, the relies and the guarantees can be strengthened or weakened by the CONSEQ rule.

The FRAME rule allows us to use local specifications. When verifying the simulation between C and $\mathbb{C}$, we need to only talk about the locally-used resource in α, ζ and γ, and the local relies and guarantees $\mathcal{R}$, $\mathcal{G}$, $\mathbb{R}$ and $\mathbb{G}$. Then the proof can be reused in contexts where some extra resource η is used, and the accesses of it respect the invariant β and $\mathcal{R}'$, $\mathcal{G}'$, $\mathbb{R}'$ and $\mathbb{G}'$. We give the auxiliary definitions in Figure 4. The disjoint union $\uplus$ between states is lifted to state pairs. An intuitionistic state relation is monotone *w.r.t.* the extension of states. The disjointness $\eta \# \alpha$ says that any state pair satisfying both η and α can be split into two disjoint state pairs

$$\frac{\zeta \subseteq \alpha}{(\mathbf{skip}, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta \ltimes \zeta} (\mathbf{skip}, \mathbb{R}, \mathsf{Id})} \; (\textsc{skip}) \qquad \frac{(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \quad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma \ltimes \eta} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})}{(C_1; C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \eta} (\mathbb{C}_1;;\mathbb{C}_2, \mathbb{R}, \mathbb{G})} \; (\textsc{seq})$$

$$\frac{\begin{array}{c}(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_1 \ltimes \gamma} (\mathbb{C}_1, \mathbb{R}, \mathbb{G}) \quad (C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_2 \ltimes \gamma} (\mathbb{C}_2, \mathbb{R}, \mathbb{G}) \\ \zeta \subseteq (B \Leftrightarrow \mathbb{B}) \quad \zeta_1 = (\zeta \cap (B \wedge \mathbb{B})) \quad \zeta_2 = (\zeta \cap (\neg B \wedge \neg \mathbb{B})) \quad \zeta \subseteq \alpha \end{array}}{(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbf{if}\ \mathbb{B}\ \mathbf{then}\ \mathbb{C}_1\ \mathbf{else}\ \mathbb{C}_2, \mathbb{R}, \mathbb{G})} \; (\textsc{if})$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma_1 \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \quad \gamma \subseteq (B \Leftrightarrow \mathbb{B}) \quad \gamma_1 = (\gamma \cap (B \wedge \mathbb{B})) \quad \gamma_2 = (\gamma \cap (\neg B \wedge \neg \mathbb{B}))}{(\mathbf{while}\ (B)\ C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\gamma \ltimes \gamma_2} (\mathbf{while}\ \mathbb{B}\ \mathbf{do}\ \mathbb{C}, \mathbb{R}, \mathbb{G})} \; (\textsc{while})$$

$$\frac{\begin{array}{c}(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta \ltimes \gamma_1} (\mathbb{C}_1, \mathbb{R}_1, \mathbb{G}_1) \quad (C_2, \mathcal{R}_2, \mathcal{G}_2) \preceq_{\alpha;\zeta \ltimes \gamma_2} (\mathbb{C}_2, \mathbb{R}_2, \mathbb{G}_2) \\ \mathcal{G}_1 \subseteq \mathcal{R}_2 \quad \mathcal{G}_2 \subseteq \mathcal{R}_1 \quad \mathbb{G}_1 \subseteq \mathbb{R}_2 \quad \mathbb{G}_2 \subseteq \mathbb{R}_1 \end{array}}{(C_1 \parallel C_2, \mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2) \preceq_{\alpha;\zeta \ltimes (\gamma_1 \cap \gamma_2)} (\mathbb{C}_1 \| \mathbb{C}_2, \mathbb{R}_1 \cap \mathbb{R}_2, \mathbb{G}_1 \cup \mathbb{G}_2)} \; (\textsc{par})$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ (\zeta \cup \gamma) \subseteq \alpha' \subseteq \alpha \quad \mathsf{Sta}(\alpha', \langle \mathcal{G}, \mathbb{G} \rangle_\alpha) \end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \; (\textsc{stren-}\alpha) \qquad \frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \alpha \subseteq \alpha' \quad \mathsf{Sta}(\alpha, \langle \mathcal{R}, \mathbb{R} \rangle_{\alpha'}) \end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha';\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \; (\textsc{weaken-}\alpha)$$

$$\frac{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \quad \zeta' \subseteq \zeta \quad \gamma \subseteq \gamma' \subseteq \alpha \quad \mathcal{R}' \subseteq \mathcal{R} \quad \mathbb{R}' \subseteq \mathbb{R} \quad \mathcal{G} \subseteq \mathcal{G}' \quad \mathbb{G} \subseteq \mathbb{G}'}{(C, \mathcal{R}', \mathcal{G}') \preceq_{\alpha;\zeta' \ltimes \gamma'} (\mathbb{C}, \mathbb{R}', \mathbb{G}')} \; (\textsc{conseq})$$

$$\frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \eta \subseteq \beta \quad \mathsf{Intuit}(\{\alpha, \zeta, \gamma, \beta, \eta, \mathcal{R}, \mathbb{R}, \mathcal{R}', \mathbb{R}'\}) \\ \eta \; \# \; \{\zeta, \gamma, \alpha\} \quad \mathsf{Sta}(\eta, \{\langle \mathcal{G}, \mathbb{G} \rangle_\alpha, \langle \mathcal{R}', \mathbb{R}' \rangle_\beta\}) \end{array}}{(C, \mathcal{R} \uplus \mathcal{R}', \mathcal{G} \uplus \mathcal{G}') \preceq_{\alpha \uplus \beta;(\zeta \uplus \eta) \ltimes (\gamma \uplus \eta)} (\mathbb{C}, \mathbb{R} \uplus \mathbb{R}', \mathbb{G} \uplus \mathbb{G}')} \; (\textsc{frame}) \qquad \frac{\begin{array}{c}(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (M, R_M, G_M) \\ (M, R_M, G_M) \preceq_{\beta;\delta \ltimes \eta} (\mathbb{C}, \mathbb{R}, \mathbb{G}) \\ \langle \mathcal{R}, \mathbb{R} \rangle_{\beta \circ \alpha} = \langle R_M, \mathbb{R} \rangle_\beta \circ \langle \mathcal{R}, R_M \rangle_\alpha \end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\beta \circ \alpha;(\delta \circ \zeta) \ltimes (\eta \circ \gamma)} (\mathbb{C}, \mathbb{R}, \mathbb{G})} \; (\textsc{trans})$$

Figure 5. Compositionality Rules for RGSim

---

satisfying η and α respectively. For example, let $\eta \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{y}) = \Sigma(\mathtt{y})\}$ and $\alpha \triangleq \{(\sigma, \Sigma) \mid \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}$, then both η and α are intuitionistic and $\eta \; \# \; \alpha$ holds. We also require η to be stable under interference from the programs (*i.e.*, the programs do not change the extra resource) and the extra environments. We use $\eta \; \# \; \{\zeta, \gamma, \alpha\}$ as a shorthand for $(\eta \; \# \; \zeta) \wedge (\eta \; \# \; \gamma) \wedge (\eta \; \# \; \alpha)$. Similar representations are used in this rule.

Finally, the transitivity rule TRANS allows us to verify a transformation by using an intermediate level as a bridge. The intermediate environment R_M should be chosen with caution so that the $(\beta \circ \alpha)$-related transitions can be decomposed into β-related and α-related transitions. Here $\circ$ defines the composition of two relations, as shown in Figure 4. Due to the space limit, all the soundness proofs are put in the technical report submitted with this paper.

Instantiations of relies and guarantees. We can derive the sequential refinement and the fully-abstract-semantics-based refinement by instantiating the rely conditions in RGSim. For example, the refinement (4.1) over closed programs assumes identity environments, making the interference constraints in the PAR rule unsatisfiable. This confirms the observation in Section 2.1 that the sequential refinement loses parallel compositionality.

$$(C, \mathsf{Id}, \mathsf{True}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathsf{Id}, \mathsf{True}) \qquad (4.1)$$

The refinement (4.2) assumes arbitrary environments, which makes the interference constraints in the PAR rule trivially true. But this requirement is too strong and usually cannot be satisfied in practice.

$$(C, \mathsf{True}, \mathsf{True}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbb{C}, \mathsf{True}, \mathsf{True}) \qquad (4.2)$$

Here we use Id and True (defined in Figure 4) for the sets of identity transitions and arbitrary transitions respectively, and overload the notations at the low level to the high level.

Example. Below we give a simple example to illustrate the use of RGSim and its parallel compositionality. $\mathbb{C}_1 \| \mathbb{C}_2$ is transformed to

$C_1 \parallel C_2$, using a lock $\mathtt{l}$ to synchronize the accesses of the shared variable $\mathtt{x}$. We aim to prove $C_1 \parallel C_2 \sqsubseteq_\mathbf{T} \mathbb{C}_1 \| \mathbb{C}_2$. That is, although $\mathtt{x}:=\mathtt{x}+2$ is implemented by two steps of incrementing $\mathtt{x}$ in C_2, the parallel observer C_1 will not print unexpected values. Here we view output events as externally observable behaviors.

```
print(x);   |||   x := x + 2;

                 ⇓

lock(l);          lock(l);
print(x);    ||   x := x+1;  x := x+1;
unlock(l);        unlock(l);
```

By the soundness and compositionality of RGSim, we only need to prove simulations over individual threads, providing appropriate relies and guarantees. We first define the invariant α, which only cares about the value of $\mathtt{x}$ when the lock is free.

$$\alpha \quad \triangleq \quad \{(\sigma, \Sigma) \mid \sigma(\mathtt{l}) = 0 \implies \sigma(\mathtt{x}) = \Sigma(\mathtt{x})\}.$$

We let the pre- and post-conditions be α as well.

The high-level threads can be executed in arbitrary environments with arbitrary guarantees: $\mathbb{R} = \mathbb{G} \triangleq \mathsf{True}$. The transformation uses the lock to protect every access of $\mathtt{x}$, thus the low-level relies and guarantees are not arbitrary:

$$\begin{aligned}\mathcal{R} &\triangleq \{(\sigma, \sigma') \mid \sigma(\mathtt{l}) = \mathtt{cid} \implies \sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \wedge \sigma(\mathtt{l}) = \sigma'(\mathtt{l})\}; \\ \mathcal{G} &\triangleq \{(\sigma, \sigma') \mid \sigma' = \sigma \; \vee \; \sigma(\mathtt{l}) = 0 \wedge \sigma' = \sigma\{\mathtt{l} \rightsquigarrow \mathtt{cid}\} \\ &\qquad \vee \; \sigma(\mathtt{l}) = \mathtt{cid} \wedge \sigma' = \sigma\{\mathtt{x} \rightsquigarrow \_\} \\ &\qquad \vee \; \sigma(\mathtt{l}) = \mathtt{cid} \wedge \sigma' = \sigma\{\mathtt{l} \rightsquigarrow 0\}\}.\end{aligned}$$

The low-level thread guarantees that it updates $\mathtt{x}$ only when the lock is acquired. Its environment cannot update $\mathtt{x}$ or $\mathtt{l}$ if the current thread holds the lock. Here $\mathtt{cid}$ is the identifier of the current thread. When acquired, the lock holds the id of the owner thread.

Following the definition, we can prove $(C_1, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha \ltimes \alpha} (\mathbb{C}_1, \mathbb{R}, \mathbb{G})$ and $(C_2, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha \ltimes \alpha} (\mathbb{C}_2, \mathbb{R}, \mathbb{G})$. By applying the

PAR rule and from soundness of RGSim (Corollary 5), we know $C_1 \| C_2 \sqsubseteq_{\mathbf{T}} \mathbb{C}_1 \| \mathbb{C}_2$ holds for any $\mathbf{T}$ that respects α.

Perhaps interestingly, if we omit the lock and unlock operations in C_1, then $C_1 \| C_2$ would have more observable behaviors than $\mathbb{C}_1 \| \mathbb{C}_2$. This does *not* show the unsoundness of our PAR rule (which is sound!). The reason is that we cannot have $(\mathtt{print(x)}, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\alpha \ltimes \alpha} (\mathtt{print(x)}, \mathbb{R}, \mathbb{G})$ with the current definitions of α, $\mathcal{R}$ and $\mathcal{G}$, even though the code of the two sides are syntactically identical.

More discussions. RGSim ensures that the target program preserves safety properties (including the partial correctness) of the source, but allows a terminating source program to be transformed to a target having infinite silent steps. In the above example, this allows the low-level programs to be blocked forever (*e.g.*, at the time when the lock is held but never released by some other thread). Proving the preservation of the termination behavior would require liveness proofs in a concurrent setting (*e.g.*, proving the absence of deadlock), which we leave as future work.

In the next three sections, we show more serious examples to demonstrate the applicability of RGSim.

5. Relational Reasoning about Optimizations

As a general correctness notion of concurrent program transformations, RGSim establishes a relational approach to justify compiler optimizations on concurrent programs. Below we adapt Benton's work [3] on sequential optimizations to the concurrent setting.

5.1 Optimization Rules

Usually optimizations depend on particular contexts, *e.g.*, the assignment $x := E$ can be eliminated only in the context that the value of x is never used after the assignment. In a shared-state concurrent setting, we should also consider the parallel context for an optimization. RGSim enables us to specify various sophisticated requirements for the parallel contexts by rely/guarantee conditions. Based on RGSim, we provide a set of inference rules to characterize and justify common optimizations (*e.g.*, dead code elimination) with information of both the sequential and the parallel contexts. Due to the space limit, we only present some interesting rules here and leave other rules in the technical report. Note in this section the target and the source programs are in the same language.

Sequential skip Law

$$\frac{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta \ltimes \gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}{(\mathbf{skip}; C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta \ltimes \gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}$$

$$\frac{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta \ltimes \gamma} (C_2, \mathcal{R}_2, \mathcal{G}_2)}{(C_1, \mathcal{R}_1, \mathcal{G}_1) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbf{skip}; C_2, \mathcal{R}_2, \mathcal{G}_2)}$$

Plus the variants with **skip** after the code C_1 or C_2. That is, **skip**s could be arbitrarily introduced and eliminated.

Common Branch

$$\frac{\begin{array}{c} \forall \sigma_1, \sigma_2. \, (\sigma_1, \sigma_2) \in \zeta \implies B \, \sigma_2 \neq \perp \\ (C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_1 \ltimes \gamma} (C_1, \mathcal{R}', \mathcal{G}') \quad \zeta_1 = (\zeta \cap (\mathbf{true} \wedge B)) \\ (C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta_2 \ltimes \gamma} (C_2, \mathcal{R}', \mathcal{G}') \quad \zeta_2 = (\zeta \cap (\mathbf{true} \wedge \neg B)) \end{array}}{(C, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbf{if} \, (B) \, C_1 \, \mathbf{else} \, C_2, \mathcal{R}', \mathcal{G}')}$$

This rule says that, when the if-condition can be evaluated and both branches can be optimized to the same code C, we can transform the whole if-statement to C without introducing new behaviors.

Dead While

$$\frac{\zeta = (\zeta \cap (\mathbf{true} \wedge \neg B)) \quad \zeta \subseteq \alpha \quad \mathsf{Sta}(\zeta, \langle \mathcal{R}, \mathcal{R}' \rangle_\alpha)}{(\mathbf{skip}, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta \ltimes \zeta} (\mathbf{while} \, (B)\{C\}, \mathcal{R}', \mathsf{Id})}$$

We can eliminate the loop, if the loop condition is **false** (no matter how the environments update the states) at the loop entry point.

Dead Code Elimination

$$\frac{(\mathbf{skip}, \mathsf{Id}, \mathsf{Id}) \preceq_{\alpha;\zeta \ltimes \gamma} (C, \mathsf{Id}, \mathcal{G}) \quad \mathsf{Sta}(\{\zeta, \gamma\}, \langle \mathcal{R}, \mathcal{R}' \rangle_\alpha)}{(\mathbf{skip}, \mathcal{R}, \mathsf{Id}) \preceq_{\alpha;\zeta \ltimes \gamma} (C, \mathcal{R}', \mathcal{G})}$$

Intuitively $(\mathbf{skip}, \mathsf{Id}, \mathsf{Id}) \preceq_{\alpha;\zeta \ltimes \gamma} (C, \mathsf{Id}, \mathcal{G})$ says that the code C can be eliminated in a sequential context where the initial and the final states satisfy ζ and γ respectively. If both ζ and γ are stable *w.r.t.* the interference from the environments $\mathcal{R}$ and $\mathcal{R}'$, then the code C can be eliminated in such a parallel context as well.

Redundancy Introduction

$$\frac{(c, \mathsf{Id}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbf{skip}, \mathsf{Id}, \mathsf{Id}) \quad \mathsf{Sta}(\{\zeta, \gamma\}, \langle \mathcal{R}, \mathcal{R}' \rangle_\alpha)}{(c, \mathcal{R}, \mathcal{G}) \preceq_{\alpha;\zeta \ltimes \gamma} (\mathbf{skip}, \mathcal{R}', \mathsf{Id})}$$

Similar to the dead-code-elimination rule, we can also lift sequential redundant code introduction to the concurrent setting, as long as the pre/post conditions are stable *w.r.t.* the environments. Note that here c is a single instruction, because we should consider the interference from the environments at every intermediate state when introducing a sequence of redundant instructions.

5.2 An Example of Invariant Hoisting

With these rules, we can prove the correctness of many traditional compiler optimizations performed on concurrent programs in appropriate contexts. Here we only give a small example of hoisting loop invariants. More examples (*e.g.*, strength reduction and induction variable elimination) can be found in the technical report.

Target Code (C_1)		Source Code (C)
`local t;`		`local t;`
`t := x + 1;`		`while(i < n) {`
`while(i < n) {`	$\Leftarrow$	`  t := x + 1;`
`  i := i + t;`		`  i := i + t;`
`}`		`}`

When we do not care about the final value of `t`, it's not difficult to prove that the optimized code C_1 preserves the sequential behaviors of the source C [3]. But in a concurrent setting, safely hoisting the invariant code `t:=x+1` also requires that the environment should not update `x` nor `t`.

$$\mathcal{R} \triangleq \{(\sigma, \sigma') \mid \sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \wedge \sigma(\mathtt{t}) = \sigma'(\mathtt{t})\}.$$

The guarantee of the program can be specified as arbitrary transitions. Since we only care about the values of `i`, `n` and `x`, the invariant relation α can be defined as:

$$\alpha \triangleq \{(\sigma_1, \sigma) \mid \sigma_1(\mathtt{i}) = \sigma(\mathtt{i}) \wedge \sigma_1(\mathtt{n}) = \sigma(\mathtt{n}) \wedge \sigma_1(\mathtt{x}) = \sigma(\mathtt{x})\}.$$

We do not need special pre/post conditions, thus the correctness of the optimization is formalized as follows:

$$(C_1, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha \ltimes \alpha} (C, \mathcal{R}, \mathsf{True}). \tag{5.1}$$

We can prove (5.1) directly by the RGSim definition and the operational semantics of the code. Below we give a more convenient proof using the optimization rules and the compositionality rules.

We first prove the following by the dead-code-elimination and redundancy-introduction rules:

$$(\mathtt{t:=x+1}, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\alpha \ltimes \gamma} (\mathbf{skip}, \mathcal{R}, \mathsf{True});$$
$$(\mathbf{skip}, \mathcal{R}, \mathsf{True}) \preceq_{\alpha;\gamma \ltimes \eta} (\mathtt{t:=x+1}, \mathcal{R}, \mathsf{True}),$$

where γ and η specify the states at the specific program points:

$$\gamma \triangleq \alpha \cap \{(\sigma_1, \sigma) \mid \sigma_1(\mathtt{t}) = \sigma_1(\mathtt{x}) + 1\};$$
$$\eta \triangleq \gamma \cap \{(\sigma_1, \sigma) \mid \sigma(\mathtt{t}) = \sigma(\mathtt{x}) + 1\}.$$

After adding **skip**s to C_1 and C to make them the same "shape", we can prove the simulation by the compositionality rules. Finally, we remove all the **skip**s and conclude (5.1), *i.e.*, the correctness of the optimization in appropriate contexts. Since the relies only prohibit updates of x and t, we can execute C_1 and C concurrently with other threads which update i and n or read x, still ensuring semantics preservation.

6. Proving Atomicity of Concurrent Objects

A concurrent object provides a set of methods, which can be called in parallel by clients as the only way to access the object. RGSim gives us a refinement-based proof method to verify the atomicity of implementations of the object: we can define abstract atomic operations in a high-level language as specifications, and prove the concrete fine-grained implementations refine the corresponding atomic operations when executed in appropriate environments. For instance, in Figure 6(a) we define two atomic set operations, ADD(e) and RMV(e). Figure 6(b) gives a concrete implementation of the set object using a lock-coupling list. The algorithm was verified in RGSep [25, 26]. Here we show how to use our RGSim to verify its atomicity by proving the low-level methods refine the corresponding abstract operations.

We first take the generic languages in Figure 3, and instantiate the high-level program states below.

$$
\begin{array}{llll}
(HMem) & M_s, M_l & \in & (Loc \cup PVar) \rightharpoonup HVal \\
(HThrds) & \Pi & \in & ThrdID \rightarrow HMem \\
(HState) & \Sigma & \in & HThrds \times HMem
\end{array}
$$

The state consists of shared memory M_s (where the object resides) and a thread pool Π, which is a mapping from thread identifiers (t $\in$ *ThrdID*) to their memory M_l. The low-level state σ is defined similarly. We use m_s, m_l and π to represent the low-level shared memory, thread-local memory and the thread pool respectively.

To allow ownership transfer between the shared memory and thread-local memory, we use ***atom***$\{\mathbb{C}\}_{\mathbb{A}}$ (or $\langle C \rangle_{\mathcal{A}}$ at the low level) to convert the shared memory to the local memory and execute $\mathbb{C}$ (or C) atomically. Following RGSep [26], an abstract transition $\mathbb{A} \in \mathcal{P}(HMem \times HMem)$ (or $\mathcal{A} \in \mathcal{P}(LMem \times LMem)$) is used to specify the effects of the atomic operation over the shared memory, which allows us to split the resulting state back to shared and local. We omit the annotations $\mathbb{A}$ and $\mathcal{A}$ in Figure 6, which are the same as the corresponding guarantees in Figure 7, as we will explain below.

In Figure 6, the abstract set is implemented by an ordered singly-linked list pointed to by a shared variable Head, with two sentinel nodes at the two ends of the list containing the values MIN_VAL and MAX_VAL respectively. Each list node is associated with a lock. Traversing the list uses "hand-over-hand" locking: the lock on one node is not released until its successor is locked. add(e) inserts a new node with value e in the appropriate position while holding the lock of its predecessor. rmv(e) redirects the predecessor's pointer while both the node to be removed and its predecessor are locked.

We define the α relation, the guarantees and the relies in Figure 7. The predicate $m_s \models \text{list}(x, A)$ represents a singly-linked list in the shared memory m_s at the location x, whose values form the sequence A. Then the mapping shared_map between the low-level and the high-level shared memory is defined by only concerning about the value sequence on the list: the concrete list should be sorted and its elements constitute the abstract set. For a thread t's local memory of the two levels, we require that the values of e are the same and enough local space is provided for add(e) and rmv(e). Then α relates the shared memory by shared_map and the local memory of each thread t by local_map.

```
ADD(e) :                    RMV(e) :
0   atom {                  0   atom {
       S := S ∪ {e};               S := S − {e};
    }                              }
```

(a) An Abstract Set

```
add(e) :                    rmv(e) :
    local x,y,z,u;              local x,y,z,v;
0   <x := Head;>            0   <x := Head;>
1   lock(x);               1   lock(x);
2   <z := x.next;>         2   <y := x.next;>
3   <u := z.data;>         3   <v := y.data;>
4   while (u < e) {        4   while (v < e) {
5      lock(z);            5      lock(y);
6      unlock(x);          6      unlock(x);
7      x := z;             7      x := y;
8      <z := x.next;>      8      <y := x.next;>
9      <u := z.data;>      9      <v := y.data;>
    }                          }
10  if (u != e) {         10  if (v = e) {
11     y := new();        11     lock(y);
12     y.lock := 0;       12     <z := y.next;>
13     y.data := e;       13     <x.next := z;>
14     y.next := z;       14     unlock(x);
15     <x.next := y;>     15     free(y);
    }                         } else {
16  unlock(x);           16     unlock(x);
                             }
```

(b) The Lock-Coupling List-Based Set

Figure 6. The Set Object

The atomic actions of the algorithm are specified by $\mathcal{G}_{\text{lock}}$, $\mathcal{G}_{\text{unlock}}$, $\mathcal{G}_{\text{add}}$, $\mathcal{G}_{\text{rmv}}$ and $\mathcal{G}_{\text{local}}$ respectively, which are all parameterized with a thread identifier t. For example, $\mathcal{G}_{\text{rmv}}(t)$ says that when holding the locks of the node y and its predecessor x, we can transfer the node y from the shared memory to the thread's local memory. This corresponds to the action performed by the code of line 13 in rmv(e). Every thread t is executed in the environment that any other thread t' can only perform those five actions, as defined in $\mathcal{R}(t)$. Similarly, the high-level $\mathbb{G}(t)$ and $\mathbb{R}(t)$ are defined according to the abstract ADD(e) and RMV(e).

We can prove that for any thread t, the following hold:

$$
\begin{array}{l}
(\text{t.add(e)}, \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha;\alpha\ltimes\alpha} (\text{t.ADD(e)}, \mathbb{R}(t), \mathbb{G}(t)) ; \\
(\text{t.rmv(e)}, \mathcal{R}(t), \mathcal{G}(t)) \preceq_{\alpha;\alpha\ltimes\alpha} (\text{t.RMV(e)}, \mathbb{R}(t), \mathbb{G}(t)) .
\end{array}
$$

The proof is done operationally based on the definition of RGSim. Due to the space limit, detailed proofs are given in the technical report submitted with this paper.

By the compositionality and the soundness of RGSim, we know that the fine-grained operations (under the parallel environment $\mathcal{R}$) are simulated by the corresponding atomic operations (under the high-level environment $\mathbb{R}$), while $\mathcal{R}$ and $\mathbb{R}$ say all accesses to the set must be done through the add and remove operations. This gives us the atomicity of the concurrent implementation of the set object.

More examples. In the companion technical report, we also show the use of RGSim to prove the atomicity of other fine-grained algorithms, including the non-blocking concurrent counter [24], Treiber's stack algorithm [23], and a concurrent GCD algorithm (calculating greatest common divisors).

7. Verifying Concurrent Garbage Collectors

In this section, we explain in detail how to reduce the problem of verifying concurrent garbage collectors to transformation verification, and use RGSim to develop a general GC verification frame-

$$m_s \models \mathsf{list}(x, A) \quad\triangleq\quad (m_s = \phi \wedge x = \mathbf{null} \wedge A = \epsilon) \vee (\exists m_s'.\exists v.\exists y.\exists A'. \ m_s = m_s' \uplus \{x \rightsquigarrow (\_, v, y)\} \wedge A = v :: A' \wedge m_s' \models \mathsf{list}(y, A'))$$

$$\mathsf{shared\_map}(m_s, M_s) \quad\triangleq\quad \exists m_s'.\exists A.\exists x. \ m_s = m_s' \uplus \{\mathtt{Head} \rightsquigarrow x\} \wedge (m_s' \models \mathsf{list}(x, \mathtt{MIN\_VAL} :: A :: \mathtt{MAX\_VAL})) \wedge \mathsf{sorted}(A) \wedge (\mathsf{elems}(A) = M_s(\mathtt{S}))$$

$$\mathsf{local\_map}(m_l, M_l) \quad\triangleq\quad m_l(\mathbf{e}) = M_l(\mathbf{e}) \wedge \exists m_l'. \ m_l = m_l' \uplus \{\mathtt{x} \rightsquigarrow \_, \mathtt{y} \rightsquigarrow \_, \mathtt{z} \rightsquigarrow \_, \mathtt{u} \rightsquigarrow \_, \mathtt{v} \rightsquigarrow \_\}$$

$$\alpha \quad\triangleq\quad \{((\pi, m_s), (\Pi, M_s)) \mid \mathsf{shared\_map}(m_s, M_s) \wedge \forall \mathtt{t} \in dom(\Pi). \ \mathsf{local\_map}(\pi(\mathtt{t}), \Pi(\mathtt{t}))\}$$

$$\mathcal{G}_{\mathrm{lock}}(\mathtt{t}) \quad\triangleq\quad \{((\pi, m_s), (\pi, m_s')) \mid \exists x, v, y. \ m_s(x) = (0, v, y) \wedge m_s' = m_s\{x \rightsquigarrow (\mathtt{t}, v, y)\}\}$$

$$\mathcal{G}_{\mathrm{unlock}}(\mathtt{t}) \quad\triangleq\quad \{((\pi, m_s), (\pi, m_s')) \mid \exists x, v, y. \ m_s(x) = (\mathtt{t}, v, y) \wedge m_s' = m_s\{x \rightsquigarrow (0, v, y)\}\}$$

$$\mathcal{G}_{\mathrm{add}}(\mathtt{t}) \quad\triangleq\quad \{((\pi \uplus \{\mathtt{t} \rightsquigarrow m_l\}, m_s), (\pi \uplus \{\mathtt{t} \rightsquigarrow m_l'\}, m_s'))$$
$$\mid \exists x, y, z, u, v, w. \ m_s(x) = (\mathtt{t}, u, z) \wedge m_s(z) = (\_, w, \_)$$
$$\wedge \ m_s' = m_s\{x \rightsquigarrow (\mathtt{t}, u, y)\} \uplus \{y \rightsquigarrow (0, v, z)\} \wedge (m_l' \uplus \{y \rightsquigarrow (0, v, z)\} = m_l) \wedge u < v < w\}$$

$$\mathcal{G}_{\mathrm{rmv}}(\mathtt{t}) \quad\triangleq\quad \{((\pi \uplus \{\mathtt{t} \rightsquigarrow m_l\}, m_s), (\pi \uplus \{\mathtt{t} \rightsquigarrow m_l'\}, m_s'))$$
$$\mid \exists x, y, z, u, v. \ m_s(x) = (\mathtt{t}, u, y) \wedge m_s(y) = (\mathtt{t}, v, z)$$
$$\wedge \ m_s' \uplus \{y \rightsquigarrow (\mathtt{t}, v, z)\} = m_s\{x \rightsquigarrow (\mathtt{t}, u, z)\} \wedge m_l' = m_l \uplus \{y \rightsquigarrow (\mathtt{t}, v, z)\} \wedge v < \mathtt{MAX\_VAL}\}$$

$$\mathcal{G}_{\mathrm{local}}(\mathtt{t}) \quad\triangleq\quad \{((\pi \uplus \{\mathtt{t} \rightsquigarrow m_l\}, m_s), (\pi \uplus \{\mathtt{t} \rightsquigarrow m_l'\}, m_s)) \mid \pi \in (ThrdID \rightarrow LMem) \wedge m_l, m_l', m_s \in LMem\}$$

$$\mathcal{G}(\mathtt{t}) \quad\triangleq\quad \mathcal{G}_{\mathrm{lock}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{unlock}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{add}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{rmv}}(\mathtt{t}) \cup \mathcal{G}_{\mathrm{local}}(\mathtt{t}) \qquad\qquad \mathcal{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathcal{G}(\mathtt{t}')$$

$$\mathbb{G}_{\mathrm{add}}(\mathtt{t}) \quad\triangleq\quad \{((\Pi \uplus \{\mathtt{t} \rightsquigarrow M_l\}, M_s), (\Pi \uplus \{\mathtt{t} \rightsquigarrow M_l'\}, M_s')) \mid \exists e. \ M_s' = M_s\{\mathtt{S} \rightsquigarrow M_s(\mathtt{S}) \cup \{e\}\}\}$$

$$\mathbb{G}_{\mathrm{rmv}}(\mathtt{t}) \quad\triangleq\quad \{((\Pi \uplus \{\mathtt{t} \rightsquigarrow M_l\}, M_s), (\Pi \uplus \{\mathtt{t} \rightsquigarrow M_l'\}, M_s')) \mid \exists e. \ M_s' = M_s\{\mathtt{S} \rightsquigarrow M_s(\mathtt{S}) - \{e\}\}\}$$

$$\mathbb{G}_{\mathrm{local}}(\mathtt{t}) \quad\triangleq\quad \{((\Pi \uplus \{\mathtt{t} \rightsquigarrow M_l\}, M_s), (\Pi \uplus \{\mathtt{t} \rightsquigarrow M_l'\}, M_s)) \mid \Pi \in (ThrdID \rightarrow HMem) \wedge M_l, M_l', M_s \in HMem\}$$

$$\mathbb{G}(\mathtt{t}) \quad\triangleq\quad \mathbb{G}_{\mathrm{add}}(\mathtt{t}) \cup \mathbb{G}_{\mathrm{rmv}}(\mathtt{t}) \cup \mathbb{G}_{\mathrm{local}}(\mathtt{t}) \qquad\qquad \mathbb{R}(\mathtt{t}) \triangleq \bigcup_{\mathtt{t}' \neq \mathtt{t}} \mathbb{G}(\mathtt{t}')$$

Figure 7. Useful Definitions for the Lock-Coupling List

work. We apply the framework to prove the correctness of the Boehm *et al.* concurrent GC algorithm [7].

7.1 Correctness of Concurrent GCs

A concurrent GC is executed by a dedicate thread and performs the collection work in parallel with user threads (mutators), which access the shared heap via read, write and allocation operations. To ensure that the GC and the mutators share a coherent view of the heap, the heap operations from mutators may be instrumented with extra operations, which provide an interaction mechanism to allow arbitrary mutators to cooperate with the GC. These instrumented heap operations are called barriers (*e.g.*, read barriers, write barriers and allocation barriers).

The GC thread and the barriers constitute a concurrent garbage collecting system, which provides a higher-level user-friendly programming model for garbage-collected languages (*e.g.*, Java). In this high-level model, programmers feel they access the heap using regular memory operations, and are freed from manually disposing objects that are no longer in use. They do not need to consider the implementation details of the GC and the existence of barriers.

We could verify the GC system by using a Hoare-style logic to prove that the GC thread and the barriers satisfy their specifications. However, we say this is an indirect approach because it is unclear if the specified correct behaviors would indeed make the mutators happy and generate the abstract view for high-level programmers. Usually this part is examined by experts and then trusted.

Here we propose a more direct approach. We view a concurrent garbage collecting system as a transformation $\mathbf{T}$ from a high-level garbage-collected language to a low-level language. A standard atomic memory operation at the source level is transformed into the corresponding barrier code at the target level. In the source level, we assume there is an *abstract GC thread* that magically turns unreachable objects into reusable memory. The abstract collector *AbsGC* is transformed into the concrete GC code C_{gc} running concurrently with the target mutators. That is,

$$\mathbf{T}(\mathtt{t}_{\mathrm{gc}}.AbsGC \, \| \, \mathtt{t}_1.\mathbb{C}_1 \, \| \ldots \| \, \mathtt{t}_n.\mathbb{C}_n) \triangleq$$
$$\mathtt{t}_{\mathrm{gc}}.C_{\mathrm{gc}} \, \| \, \mathtt{t}_1.\mathbf{T}(\mathbb{C}_1) \, \| \ldots \| \, \mathtt{t}_n.\mathbf{T}(\mathbb{C}_n) \,,$$

where $\mathbf{T}(\mathbb{C})$ simply translates some memory access instructions in $\mathbb{C}$ into the corresponding barriers, and leaves the rest unchanged.

Then we reduce the correctness of the concurrent garbage collecting system to $\mathsf{Correct}(\mathbf{T})$, saying that any mutator program will not have unexpected behaviors when executed using this system.

7.2 A General Framework

We can use RGSim to prove $\mathsf{Correct}(\mathbf{T})$. By its compositionality, we can decompose the refinement proofs into proofs for the GC thread and each mutator thread.

Verifying the GC. The semantics of the abstract GC thread can be defined by a binary state predicate AbsGCStep:

$$\frac{(\Sigma, \Sigma') \in \mathsf{AbsGCStep}}{(\mathtt{t}_{\mathrm{gc}}.AbsGC, \Sigma) \longrightarrow (\mathtt{t}_{\mathrm{gc}}.AbsGC, \Sigma')}$$

That is, the abstract GC thread always makes AbsGCStep to change the high-level state. We can choose different AbsGCStep for different GCs, but usually AbsGCStep guarantees not modifying reachable objects in the heap.

Thus for the GC thread, we need to show that C_{gc} is simulated by *AbsGC* when executed in their environments. This can be reduced to unary Rely-Guarantee reasoning of C_{gc} by proving $\mathcal{R}_{\mathrm{gc}}; \mathcal{G}_{\mathrm{gc}} \vdash \{p_{\mathrm{gc}}\} C_{\mathrm{gc}} \{q_{\mathrm{gc}}\}$ in a standard Rely-Guarantee logic with proper $\mathcal{R}_{\mathrm{gc}}$, $\mathcal{G}_{\mathrm{gc}}$, p_{gc} and q_{gc}, as long as $\mathcal{G}_{\mathrm{gc}}$ is a concrete representation of AbsGCStep. The judgment says given an initial state satisfying the precondition p_{gc}, if the environment's behaviors satisfy $\mathcal{R}_{\mathrm{gc}}$, then each step of C_{gc} satisfies $\mathcal{G}_{\mathrm{gc}}$, and the postcondition q_{gc} holds at the end if C_{gc} terminates. In general, the collector never terminates, thus we can let q_{gc} be **false**. $\mathcal{G}_{\mathrm{gc}}$ and p_{gc} should be provided by the verifier, where p_{gc} is general enough that can be satisfied by any possible low-level initial state. $\mathcal{R}_{\mathrm{gc}}$ encodes the possible behaviors of mutators, which can be derived, as we will show below.

Verifying mutators. For the mutator thread, since $\mathbf{T}$ is syntax-directed on $\mathbb{C}$, we can reduce the refinement problem for arbitrary mutators to the refinement on each primitive instruction only, by the compositionality of RGSim. The proof needs proper rely/guarantee conditions. Let $\mathbb{G}(\mathtt{t.c})$ and $\mathcal{G}(\mathbf{T}(\mathtt{t.c}))$ denote the guarantees of the source instruction c and the target code $\mathbf{T}(\mathtt{c})$ respectively. Then we can define the general guarantees for a mutator thread t:

$$\begin{aligned} \mathbb{G}(\mathtt{t}) &\triangleq \bigcup_{\mathtt{c}} \mathbb{G}(\mathtt{t.c}) \\ \mathcal{G}(\mathtt{t}) &\triangleq \bigcup_{\mathtt{c}} \mathcal{G}(\mathtt{t}.\mathbf{T}(\mathtt{c})) \end{aligned} \qquad (7.1)$$

Its relies should include all the possible guarantees made by other threads, and GC's abstract and concrete behaviors respectively:

$$\begin{aligned} \mathbb{R}(t) &\triangleq \mathsf{AbsGCStep} \cup \bigcup_{t' \neq t} \mathbb{G}(t') \\ \mathcal{R}(t) &\triangleq \mathcal{G}_{gc} \cup \bigcup_{t' \neq t} \mathcal{G}(t') \end{aligned} \qquad (7.2)$$

The $\mathcal{R}_{gc}$ used for GC verification can now be defined below:

$$\mathcal{R}_{gc} \triangleq \bigcup_t \mathcal{G}(t) \qquad (7.3)$$

The refinement proof also needs definitions of binary α, ζ and γ relations. The invariant α relates the low-level and the high-level states and needs to be preserved by each low-level step. In general, a high-level state Σ can be mapped to a low-level state σ by giving a concrete local store for the GC thread, adding additional structures in the heap (to record information for collection), renaming heap cells (for copying GCs), *etc.*. For each mutator thread t, the relations $\zeta(t)$ and $\gamma(t)$ need to hold at the beginning and the end of each basic transformation unit (every high-level primitive instruction in this case) respectively. We let $\gamma(t)$ be the same as $\zeta(t)$ to support sequential compositions. We require $\mathsf{InitRel}_\mathbf{T}(\zeta(t))$ (see Figure 4), *i.e.*, $\zeta(t)$ holds over the initial states. In addition, the target and the source boolean expressions should be evaluated to the same value under related states, as required in the IF and WHILE rules in Figure 5.

$$\mathsf{Good}_\mathbf{T}(\zeta(t)) \triangleq \mathsf{InitRel}_\mathbf{T}(\zeta(t)) \wedge \forall \mathbb{B}.\, \zeta(t) \subseteq (\mathbf{T}(\mathbb{B}) \Leftrightarrow \mathbb{B}) \quad (7.4)$$

Theorem 7 (Verifying Concurrent Garbage Collecting Systems).
If there exist $\mathcal{R}_{gc}$, $\mathcal{G}_{gc}$, p_{gc}, $\mathcal{R}(t)$, $\mathbb{R}(t)$, $\zeta(t)$ and α such that (7.1), (7.2), (7.3), (7.4) and the following hold:

1. *(Verification of the GC code)*
   $\mathcal{R}_{gc}; \mathcal{G}_{gc} \vdash \{p_{gc}\} C_{gc} \{\mathbf{false}\};$
2. *(Correctness of $\mathbf{T}$ on mutator instructions)*
   $\forall c.\, (t.\mathbf{T}(c), \mathcal{R}(t), \mathcal{G}(t.\mathbf{T}(c))) \preceq_{\alpha;\zeta(t) \ltimes \zeta(t)} (t.c, \mathbb{R}(t), \mathbb{G}(t.c));$
3. *(Side Conditions)*
   $\mathcal{G}_{gc} \circ \alpha^{-1} \subseteq \alpha^{-1} \circ (\mathsf{AbsGCStep})^*;$
   $\forall \sigma, \Sigma.\, \sigma = \mathbf{T}(\Sigma) \implies p_{gc}\, \sigma;$

then $\mathsf{Correct}(\mathbf{T})$.

That is, to verify a concurrent garbage collecting system, we need to do the following:

- Define the α and $\zeta(t)$ relations, and prove the correctness of $\mathbf{T}$ on high-level primitive instructions. Since $\mathbf{T}$ preserves the syntax on most instructions, it's often immediate to prove the target instructions are simulated by their sources. But for instructions that are transformed to barriers, we need to verify the barriers that they implement both the source instructions (by RGSim) and the interaction mechanism (shown in their guarantees).

- Find some proper $\mathcal{G}_{gc}$ and p_{gc}, and verify the GC code by R-G reasoning. We require the GC's guarantee $\mathcal{G}_{gc}$ should not contain more behaviors than $\mathsf{AbsGCStep}$ (the first side condition), and C_{gc} can start its execution from any state σ transformed from a high-level one (the second side condition).

7.3 Application: Boehm *et al.* Concurrent GC Algorithm

We illustrate the applications of the framework (Theorem 7) by proving the correctness of a mostly-concurrent mark-sweep garbage collector proposed by Boehm *et al.* [7]. Variants of the algorithm have been used in practice (*e.g.*, by IBM [2]). Due to the space limit, we only describe the proof sketch here. Details are presented in the companion technical report.

Overview of the GC algorithm. The top-level code of the GC thread is shown in Figure 8. In each collection cycle, after an initialization process, the GC enters the concurrent mark-phase

```
{wfstate}
0 Collection() {
1   local mstk: Seq(Int);
    Loop Invariant: {wfstate * (own_np(mstk) ∧ mstk = ε)}
2   while (true) {
3     Initialize();
      {(wfstate ∧ reach_inv) * (own_np(mstk) ∧ mstk = ε)}
4     Trace();
      {(wfstate ∧ reach_inv) * (own_np(mstk) ∧ mstk = ε)}
5     CleanCard();
      {(wfstate ∧ reach_inv) * (own_np(mstk) ∧ mstk = ε)}
      atomic{
6       ScanRoot();
        {∃X.(wfstate ∧ reach_rtnw_stk(X) ∧ stk_black(X))
          *(own_np(mstk) ∧ mstk = X)}
7       CleanCard();
      }
      {(wfstate ∧ reach_black) * (own_np(mstk) ∧ mstk = ε)}
8     Sweep();
    }
  }
{false}
```

Figure 8. Outline of the GC Code and Proof Sketch

```
update(x.id, E) { // id ∈ {pt1, ..., ptm}
  atomic{ x.id := E;  aux := x; }
  atomic{ x.dirty := 1;  aux := 0; }
}
```

Figure 9. The Write Barrier for Boehm *et al.* GC

(line 4) and traces the objects reachable from the *roots* (*i.e.*, the mutators' local pointer variables that may contain references to the heap objects). A *mark stack* (mstk) is used to do a depth-first tracing. During the tracing, the connectivity between objects might be changed by the mutators, thus a write barrier is required to notify the collector of those modified objects by dirtying the objects' tags (called cards). When the tracing is done, the GC suspends all the mutators and re-traces from the dirty objects that have been marked (called *card-cleaning*, line 6 and 7). The stop-the-world phase is implemented by **atomic**$\{C\}$. Finally, all the reachable objects are ensured marked and the GC performs the concurrent sweep-phase (line 8), in which unmarked objects are reclaimed. Usually in practice, there is also a concurrent card-cleaning phase (line 5) before the stop-the-world card-cleaning to reduce the pause time. The full GC code C_{gc} is given in the technical report.

The write barrier is shown in Figure 9, where the dirty field is set after modifying the object's pointer field. Here we use a write-only auxiliary variable aux for each mutator thread to record the current object that the mutator is updating. The GC does not use read barriers nor allocation barriers.

We first present the high-level and low-level program state models in Figure 10. The behaviors of the high-level abstract GC thread are defined as follows:

$$\mathsf{AbsGCStep} \triangleq \{((\Pi, H), (\Pi, H')) \mid \forall l.\, \mathsf{reachable}(l)(\Pi, H) \implies H(l) = H'(l)\},$$

saying that, the mutator stores and the reachable objects in the heap are remained unmodified. Here $\mathsf{reachable}(l)(\Pi, H)$ means the object at the location l is reachable in H from the roots in Π.

The transformation. The transformation $\mathbf{T}$ is defined as follows. For *code*, the high-level abstract GC thread is transformed to the GC thread shown in Figure 8. Each instruction $x.\mathsf{id} := \mathbb{E}$ in mutators is transformed to the write barrier, where id is a pointer

$$(HStore)\ \mathsf{S} \in PVar \rightharpoonup HVal \qquad (HHeap)\ \mathsf{H} \in Loc \rightharpoonup HObj$$

$$(HThrds)\ \Pi \in MutID \rightharpoonup HStore \qquad (HState)\ \Sigma \in HThrds \times HHeap$$

$$(LStore)\ s \in PVar \rightharpoonup LVal \times \{0,1\} \qquad (LHeap)\ h \in [1..M] \rightharpoonup LObj$$

$$(LThrds)\ \pi \in ThrdID \rightharpoonup LStore \qquad (LState)\ \sigma \in LThrds \times LHeap$$

Figure 10. High-Level and Low-Level State Models

field of x. Other instructions and the program structures of mutators are unchanged.

The following transformations are made over initial states.

- First we require the high-level initial state to be *well-formed*:

  $$\mathsf{wfstate}(\Pi, \mathsf{H}) \quad \triangleq \quad \forall l.\ \mathsf{reachable}(l)(\Pi, \mathsf{H}) \Longrightarrow l \in dom(\mathsf{H})\,.$$

  That is, reachable locations cannot be dangling pointers.

- High-level locations are transformed to integers by a bijective function $\mathsf{Loc2Int} : Loc \leftrightarrow [0..M]$ satisfying $\mathsf{Loc2Int}(\mathbf{nil}) = 0$.

- Variables are transformed to the low level using an extra bit to preserve the high-level type information (0 for non-pointers and 1 for pointers).

- High-level objects are transformed to the low level by adding the `color` and `dirty` fields with initial values WHITE and 0 respectively. Other addresses in the low-level heap domain $[1..M]$ are filled out using unallocated objects whose `color`s are BLUE and all the other fields are initialized by 0. Here we use BLACK and WHITE for marked and unmarked objects respectively, and BLUE for unallocated memory.

- The concrete GC thread is given an initial store.

The formal definition of $\mathbf{T}$ is included in the technical report.

To prove $\mathsf{Correct}(\mathbf{T})$ in our framework, we apply Theorem 7, prove the refinement between low-level and high-level mutators, and verify the GC code using a unary Rely-Guarantee-based logic.

Refinement proofs for mutator instructions. We first define the α and $\zeta(\mathsf{t})$ relations.

$$\alpha \triangleq \{((\pi \uplus \{\mathsf{t}_{gc} \rightsquigarrow \_\}, h), (\Pi, \mathsf{H}))\ |$$
$$\forall \mathsf{t} \in dom(\Pi).\ \mathsf{store\_map}(\pi(\mathsf{t}), \Pi(\mathsf{t}))$$
$$\wedge\ \mathsf{heap\_map}(h, \mathsf{H}) \wedge \mathsf{wfstate}(\Pi, \mathsf{H})\}\,.$$

In α, the relation between low-level and high-level stores and heaps are enforced by $\mathsf{store\_map}$ and $\mathsf{heap\_map}$ respectively. Their definitions reflect the state transformations we describe above, ignoring the values of those high-level-invisible structures. It also requires the well-formedness of high-level states.

For each mutator thread t, the $\zeta(\mathsf{t})$ relation enforced at the beginning and the end of each transformation unit (each high-level instruction) is stronger than α. It requires that the value of the auxiliary variable `aux` (see Figure 9) be a null pointer (0^{p}):

$$\zeta(\mathsf{t}) \triangleq \alpha \cap \{((\pi, h), (\Pi, \mathsf{H}))\ |\ \pi(\mathsf{t})(\mathsf{aux}) = 0^{\mathsf{p}}\}\,.$$

The refinement between the write barrier at the low level and the pointer update instruction at the high level is formulated as:

$$(\mathsf{t}.\mathsf{update}(x.id, E), \mathcal{R}(\mathsf{t}), \mathcal{G}^{\mathsf{t}}_{\mathsf{write\_barrier}}) \preceq_{\alpha;\zeta(\mathsf{t}) \ltimes \zeta(\mathsf{t})}$$
$$(\mathsf{t}.(x.id := \mathbb{E}), \mathbb{R}(\mathsf{t}), \mathbb{G}^{\mathsf{t}}_{\mathsf{write\_pt}})\,,$$

where $\mathcal{G}^{\mathsf{t}}_{\mathsf{write\_barrier}}$ and $\mathbb{G}^{\mathsf{t}}_{\mathsf{write\_pt}}$ are the guarantees of the two-step write barrier and the high-level atomic write operation respectively. Since the transformation of other high-level instructions is identity, the corresponding refinement proofs are simple.

Rely-Guarantee reasoning of the GC code. The unary program logic we use to verify the GC thread is a standard Rely-Guarantee

logic adapted to the target language. We describe states using separation logic assertions, as shown below:

$$p, q ::= B\ |\ \mathsf{t.own}_{\mathsf{p}}(x)\ |\ \mathsf{t.own}_{\mathsf{np}}(x)\ |\ E_1.id \mapsto E_2\ |\ p * q\ |\ \dots$$

Following Parkinson *et al.* [20], we treat program variables as resource and use $\mathsf{t.own}_{\mathsf{p}}(x)$ and $\mathsf{t.own}_{\mathsf{np}}(x)$ for the thread t's ownerships of pointers and non-pointers respectively. We omit the thread identifiers if these predicates hold for the current thread.

We first give the precondition and the guarantee of the GC. The GC starts its executions from a low-level *well-formed* state, *i.e.*, $p_{gc} \triangleq \mathsf{wfstate}$. Just corresponding to the high-level $\mathsf{wfstate}$ definition, the low-level $\mathsf{wfstate}$ predicate says none of the reachable objects are BLUE. We define $\mathcal{G}_{gc}$ as follows:

$$\mathcal{G}_{gc} \triangleq \{((\pi \uplus \{\mathsf{t}_{gc} \rightsquigarrow s\}, h), (\pi \uplus \{\mathsf{t}_{gc} \rightsquigarrow s'\}, h'))$$
$$|\ \forall n.\ \mathsf{reachable}(n)(\pi, h)$$
$$\Longrightarrow \lfloor h(n) \rfloor = \lfloor h'(n) \rfloor$$
$$\wedge\ h(n).\mathsf{color} \neq \mathtt{BLUE} \wedge h'(n).\mathsf{color} \neq \mathtt{BLUE}\}\,.$$

The GC guarantees not modifying the mutator stores. For any mutator-reachable object, the GC does not update its fields coming from the high-level mutator, nor does it reclaim the object. Here $\lfloor \_ \rfloor$ lifts a low-level object to a new one that contains mutator data only.

The proof sketch is given in Figure 8. One of the key invariants used in the proof is $\mathsf{reach\_inv}$, which says any WHITE reachable object can either be traced from a root object in a path on which every object is WHITE, or be reachable from a BLACK object whose pointer field was updated and `dirty` bit was set to 1. Since the proof is done in the unary logic, the details here are orthogonal to our simulation-based proof (but it is RGSim that allows us to derive Theorem 7, which then links proofs in the unary-logic with relational proofs). We give the complete proofs in the companion technical report.

8. Related Work and Conclusion

There is a large body of work on refinements and verification of program transformations. Here we only focus on the work most closely related to the typical applications discussed in this paper.

Verifying compilation and optimizations of concurrent programs. Compiler verification for concurrent programming languages can date back to work by Wand [28] and Gladstein *et al.* [12], which is about functional languages using message-passing mechanisms. Recently, Lochbihler [18] presents a verified compiler for Java threads and prove semantics preservation by a weak bisimulation. He views every heap update as an observable move, thus does not allow the target and the source to have different granularities of atomic updates. To achieve parallel compositionality, he requires the relation to be preserved by any transitions of shared states, *i.e.*, the environment is assumed arbitrary. As we explained in Section 2, this is a too strong requirement in general for many transformations, including the examples in this paper.

Burckhardt *et al.* [9] present a proof method for verifying concurrent program transformations on relaxed memory models. The method relies on a compositional trace-based denotational semantics, where the values of shared variables are always considered arbitrary at any program point. In other words, they also assume arbitrary environments.

Following Leroy's CompCert project [17], Ševčík *et al.* [22] verify compilation from a C-like concurrent language to x86 by simulations. They focus on correctness of a particular compiler, and there are two phases in their compiler whose proofs are not compositional. Here we provide a general, compiler-independent, compositional proof technique to verify concurrent transformations.

We apply RGSim to justify concurrent optimizations, following Benton [3] who presents a declarative set of rules for sequential

optimizations. Also the proof rules of RGSim for sequential compositions, conditional statements and loops coincide with those in relational Hoare logic [3] and relational separation logic [29].

Proving linearizability or atomicity of concurrent objects. Filipović *et al.* [11] show linearizability can be characterized in terms of an observational refinement, where the latter is defined similarly to our Correct(**T**). There is no proof method given to verify the linearizability of fine-grained object implementations. Turon and Wand [24] propose a refinement-based proof method to verify concurrent objects. Their refinements are based on Brookes' fully abstract trace semantics [8], using rely conditions to specify permitted environments. They also prove the atomicity of a non-blocking counter and Treiber's stack algorithm. There both the fine-grained and the atomic versions of object operations are expressed in the same language. Both methods focus on verifying concurrent objects, which is only a special case in transformation verification. It is unclear if the methods can be applied to other transformations, such as concurrent GCs. We propose RGSim for general verification of concurrent program transformations. Verifying atomicity just shows one of its applications.

In his thesis [25], Vafeiadis proves linearizability of concurrent objects by introducing abstract objects and abstract atomic operations as auxiliary variables and code in RGSep logic. The refinement between the concrete implementation and the abstract version is implicitly embodied in the unary verification process, but is not spelled out in the meta-theory.

Verifying concurrent GCs. Vechev *et al.* [27] define transformations to generate concurrent GCs from an abstract collector. Afterwards, Pavlovic *et al.* [21] present refinements to derive concrete concurrent GCs from specifications. These methods focus on describing the behaviors of variants (or instantiations) of a correct abstract collector (or a specification) in a single framework, assuming all the mutator operations are atomic. By comparison, we provide a general correctness notion and a proof method for verifying concurrent GCs and the interactions with mutators (where the barriers could be fine-grained). Furthermore, the correctness of their transformations or refinements is expressed in a GC-oriented way (*e.g.*, the target GC should mark no less objects than the source), which cannot be used to justify other transformations.

Kapoor *et al.* [16] verify Dijkstra's GC using concurrent separation logic. To validate the GC specifications, they also verify a representative mutator in the same system. In contrast, we reduce the problem of verifying a concurrent GC to verifying a transformation, ensuring semantics preservation for *all* mutators. Our GC verification framework is inspired by McCreight *et al.* [19], who propose a framework for separate verification of stop-the-world and incremental GCs and their mutators, but their framework does not handle concurrency.

Conclusion and Future Work. We propose RGSim to verify concurrent program transformations. By describing explicitly the interference with environments, RGSim is compositional and can support many widely-used transformations. We have applied RGSim to reason about optimizations, prove atomicity of fine-grained concurrent algorithms and verify concurrent garbage collectors. In the future, we would like to further test its applicability with more applications, such as verifying STM implementations and compilers. It is also interesting to explore the possibility of building tools to automate the verification process.

References

[1] M. Abadi and G. Plotkin. A model of cooperative threads. In *Proc. POPL'09*, pages 29–40, 2009.

[2] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, 2005.

[3] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. POPL'04*, pages 14–25, 2004.

[4] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP'09*, pages 97–108, 2009.

[5] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI'05*, pages 261–268, 2005.

[6] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI'08*, pages 68–78, 2008.

[7] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proc. PLDI'91*, pages 157–164, 1991.

[8] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[9] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *Compiler Construction (CC'10)*, pages 104–123, 2010.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. DISC'06*, pages 194–208, 2006.

[11] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *Proc. ESOP'09*, 2009.

[12] D. S. Gladstein and M. Wand. Compiler correctness for concurrent languages. In *Proc. COORDINATION'96*, pages 231–248, 1996.

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

[14] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proc. POPL'11*, pages 133–146, 2011.

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

[16] K. Kapoor, K. Lodaya, and U. Reddy. Fine-grained concurrency with separation logic. Technical report, March 2010. `http://www.cs.bham.ac.uk/~udr/papers/gc.pdf`.

[17] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[18] A. Lochbihler. Verifying a compiler for java threads. In *Proc. ESOP'10*, pages 427–447, 2010.

[19] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI'07*, pages 468–479, 2007.

[20] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Proc. LICS'06*, pages 137–146, 2006.

[21] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In *Proc. MPC'10*, pages 353–376, 2010.

[22] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. POPL'11*, pages 43–54, 2011.

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

[24] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proc. POPL'11*, pages 247–258, 2011.

[25] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[26] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. CONCUR'07*, pages 256–271, 2007.

[27] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proc. PLDI'06*, pages 341–353, 2006.

[28] M. Wand. Compiler correctness for parallel languages. In *FPCA'95*, pages 120–134, 1995.

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