3-2020

Automatic verification of multi-threaded programs by inference of rely-guarantee specifications

Xuan-Bach LE

David SANAN

Jun SUN

Shang-Wei LIN

Automatic Verification of Multi-threaded Programs by Inference of Rely-Guarantee Specifications

Xuan-Bach Le[1], David Sanán[1], Sun Jun[2], Shang-Wei Lin[1]
[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2]School of Information Systems, Singapore Management University, Singapore
[1]{bach.le, sanan, shang-wei.lin}@ntu.edu.sg [2]junsun@smu.edu.sg

***Abstract*—Rely-Guarantee is a comprehensive technique that supports compositional reasoning for concurrent programs. However, specifications of the Rely condition — environment interference, and Guarantee condition — local transformation of thread state — are challenging to establish. Thus the construction of these conditions becomes bottleneck in automating the technique. To tackle the above problem, we propose a verification framework that, based on Rely-Guarantee principles, constructs the correctness proof of concurrent program through inferring suitable Rely-Guarantee conditions automatically. Our framework first constructs a Hoare-style sequential proof for each thread and then applies abstraction refinement to elevate these proofs into concurrent ones with appropriate Rely-Guarantee relations. Experiment results demonstrate that our approach is efficient in proving the correctness of concurrent programs.**

***Index Terms*—Rely-Guarantee, Concurrency, CEGAR**

I. Introduction

The past couple decades have witnessed a surge of interest in concurrent programs. Consequently, a fruitful number of concurrent verification frameworks, *e.g.* [4], [23], [6], [20], has emerged to counterbalance the security risks arising from concurrency bugs. Just like writing correct concurrent codes is practically difficult — as evidenced by several formal studies [14], [27], proving correctness of concurrent programs remains to be a profoundly challenging research topic.

The execution of multiple concurrent threads results in an exponential number of interleavings, causing the *state space explosion* problem that overwhelms the verification. To tackle this problem, one prominent solution is the Rely-Guarantee (R-G) methodology [23] that supports modular reasoning for concurrent programs. In this approach, each thread is associated with a pair of abstract conditions, one is Rely — environment interference, the other Guarantee — local transformation of thread state. These conditions help to scale up the verification by enabling the correctness proof of each thread to be constructed locally without referring to other threads. As a result, the R-G framework has been fruitfully adopted by many verification systems, *e.g.* [33], [11], [17].

To preserve the consistency of the logic, the local actions taken by each thread need to comply with the respective R-G conditions. Besides, each Guarantee must also imply the Relies of other threads. Due to these intertwined relationships, the automation of R-G framework is heavily constrained by the complex construction of the R-G conditions. There were several approaches proposed to resolve this bottleneck systematically, *e.g.* via reachability abstraction refinement using Horn clauses [17], or weak simulation among succinct automata that capture the program states [40]. However, we discovered through experiments that [17] suffered from the scalability issue, especially when verifying against programs with large loops. Similarly, the work in [40] also runs into the same problem due to the intractability nature of automata.

In this work, we propose a framework that automatically constructs a R-G proof using deductive verification, *ala* theorem proving, combined with CEGAR — CounterExample Guided Abstraction Refinement [7]. Our methodology starts with constructing a sequential proof for each thread, and establishes the R-G relations through iterative abstraction refinement, where spurious counterexamples that witness the inconsistency in R-G conditions are generated to refine the modular proofs. The procedure terminates when a valid R-G proof is established. Our approach features the use of abstract predicates to precisely encode the R-G conditions, as well as loop invariant to compactly capture the loop behaviors. Experiment results showed that our framework overall enjoys better scalibility as compared to [17].

A high-level overview of our approach is illustrated in Figure 1. It has two core components: Proof Generator (**PG**) that carries out the construction of R-G proof, and Consistency Verifier (**CV**) that generates counterexamples for refinement.

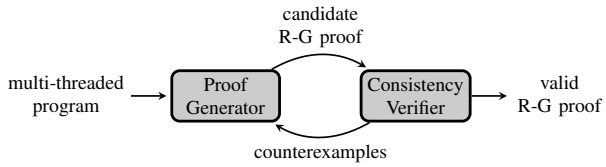


Fig. 1: Our framework for proving multi-threaded programs with generated Rely-Guarantee conditions.

To verify a multi-threaded program with asserted pre/post-conditions, **PG** first uses the precondition to construct, for each thread, an initial local Hoare-style proof and R-G relations. These key ingredients are combined to create the candidate R-G proof, which is checked by **CV** for global consistency. If this condition fails to satisfy, counterexamples representing the missing program states are generated in the form of abstract predicates. Both checking consistency and generating counterexamples are fully automated by SMT solver Z3 [10]

978-1-7281-8558-3/20/$31.00 ©2020 IEEE
DOI 10.1109/ICECCS51672.2020.00013

via appropriate transformation into SAT constraints. The next iteration begins with **PG** using the discovered counterexamples to refine the local proofs, and subsequently to construct the new R-G conditions. The refinement step terminates once the consistency is established, in which local postconditions are conjuncted together to verify the program's postcondition.

Our contributions are summarized as follows:

- We propose an automatic procedure for verifying concurrent programs using deductive inference based on R-G reasoning. Its main algorithms are addressed in §V-A and §VI-A. To the best of our knowledge, this is the first work to automate deductive proving based on R-G reasoning.
- We develop an inference system §V-B for the construction of local Hoare-style proof and Guarantee relation.
- We develop inference rules §VI-B to validate and repair the stability of the locally generated proofs with regards to the environment interference in Rely.
- We implement §VIII-A our procedure in a prototype[1] dubbed REGASOL and optimize it in REGASOL+. The prototypes are benchmarked and compared §VIII-B with other tools. Experiment results §VIII-C show that our approach is efficient in verifying concurrent programs.

The rest of this paper is organized as follows. §II gives preliminaries on R-G, §III provides a motivative example, §IV describes the semantics of the R-G methodology, §V and §VI discuss the main components of the framework, followed by the main soundness result in §VII. In §VIII, the implementation and optimization of the tool REGASOL are presented, followed by the discussion on evaluation. Finally, §IX reviews the related work and concludes the paper.

II. PRELIMINARIES

Here we briefly explain the underlying logic to help readers get familiar with the reasoning style deployed by our approach.

Abstract predicates are used to write assertions and R-G relations. Their full syntax is in Fig. 2 where *e* represents arithmetic expression, P first-order predicate, $\mathcal{B}$ binary relation.

$$\begin{array}{lll} e & ::= & \mathit{var} \mid \mathit{const} \mid e+e \mid e-e \mid e\times e \mid e \ \mathbf{div}\ e \mid e\ \%\ e \\ P & ::= & \top \mid \bot \mid e=e \mid e\leq e \mid \exists x.P \mid \forall x.P \mid \neg P \mid P\wedge P \mid P\vee P \\ \mathcal{B} & ::= & \emptyset \mid \{P \gg P\} \mid \mathcal{B}\cup\mathcal{B} \end{array}$$

Fig. 2: Syntax for abstract predicates.

Notice that $\mathcal{B}$ encodes R-G conditions. It is represented as an unordered set of *transitions* of the form $P_1 \gg P_2$, which indicates the change from a state satisfying P_1 to a state satisfying P_2. The formal semantics is fleshed out in §IV-B.

Hoare logic [19] is a formal system for establishing program correctness proofs. The central idea is the *Hoare triple* $\{P\}\mathtt{c}\{Q\}$ where P is the precondition, $\mathtt{c}$ the proving program, and Q the postcondition. Hoare logic is used in the form of deductive systems (*e.g.* [34]) consisting of rules for the construction of modular proofs. For example, proofs can be combined using the composition rule, where $\{P\}\mathtt{c}_1\{Q\}$ and $\{Q\}\mathtt{c}_2\{R\}$ deduce $\{P\}\mathtt{c}_1;\mathtt{c}_2\{R\}$. As a result, Hoare logic and its extensions [32], [4] are excellent choices for compositional reasoning as they have found many practical applications in verifying complex programs, *e.g.* [3], [21], [31], [35], and real-life systems, *e.g.* [24], [5], [1].

Rely-Guarantee [23] is a technique for compositionally proving the correctness of concurrent programs. The Rely $\mathcal{R}$ represents the abstraction of the environment transition, *i.e.* how the environment can change the thread's state, whereas the Guarantee $\mathcal{G}$ represents the abstraction of local state transition that is consistent with the thread execution. These conditions, when integrated into Hoare logic, can be pleasantly represented as:

$$\mathcal{R},\mathcal{G} \vdash \{P\}\mathtt{c}\{Q\}$$

which means that the proof $\{P\}\mathtt{c}\{Q\}$ satisfies the conditions $\mathcal{R},\mathcal{G}$. The full semantics of this notation are explained in detail in Section §IV-C; for now it is important to know that P and Q are necessarily *stable* under $\mathcal{R}$. Intuitively, this indicates that the environment does not take steps that makes P or Q invalid. Formally, a predicate P is stable under $\mathcal{R}$ if for program states s_1, s_2 such that the evaluation of P is true in s_1, and $(s_1, s_2) \in \mathcal{R}$, then the evaluation of P is also true in s_2.

Most importantly, what makes the R-G framework viable is the parallel rule below, as it explains how the proof of a concurrent program $\mathtt{c}_1 \| \mathtt{c}_2$ can be constructed from the proofs of its threads $\mathtt{c}_1$ and $\mathtt{c}_2$:

$$\frac{\begin{array}{c}\mathcal{R}\cup\mathcal{G}_2,\mathcal{G}_1 \vdash \{P_1\}\mathtt{c}_1\{Q_1\} \\ \mathcal{R}\cup\mathcal{G}_1,\mathcal{G}_2 \vdash \{P_2\}\mathtt{c}_2\{Q_2\}\end{array}}{\mathcal{R},\mathcal{G}_1\cup\mathcal{G}_2 \vdash \{P_1\wedge P_2\}\mathtt{c}_1\|\mathtt{c}_2\{Q_1\wedge Q_2\}}\ [\text{PAR}]$$

Without going into excessive detail, notice that $\mathcal{G}_1$ — the Guarantee of $\mathtt{c}_1$ — implies $\mathcal{R}\cup\mathcal{G}_1$ — the Rely of $\mathtt{c}_2$ — and vice versa, where $\mathcal{R}$ is the global Rely. In particular, $\mathcal{R}$ can be empty, in which the Relies of $\mathtt{c}_1$ and $\mathtt{c}_2$ are reduced to $\mathcal{G}_2$ and $\mathcal{G}_1$ respectively. These properties are necessary for the consistency of the R-G proof.

III. ILLUSTRATIVE EXAMPLE

The example in Figure 3 illustrates the proof construction for a concurrent program with a loop. It features the use of a loop invariant in our framework to scale up the verification task. The input program consists of two threads $T_1 \| T_2$ where T_1 is the loop:

$$\mathbf{while}\ x < 10\ \mathbf{do}\ \{\mathtt{x} := \mathtt{x} + 1;\}$$

and T_2 is the assignment:

$$\mathtt{x} := 20;$$

With precondition $x = 0$, the verification postcondition is $x = 20 \vee x = 21$. The outcome $x = 21$ occurs when T_2 is interleaved some time right before the assignment $\mathtt{x} := \mathtt{x} + 1$ of T_1. This results in x being updated to 20, followed by the increment of 1 to 21, in which the loop in T_1 terminates afterward. Any other scenario results in $x = 20$.

Furthermore, an initial sequential loop invariant $x \leq 10$ is provided for T_1. Although our framework by default does

[1] https://github.com/lexuanbach/ReGaSol-tool

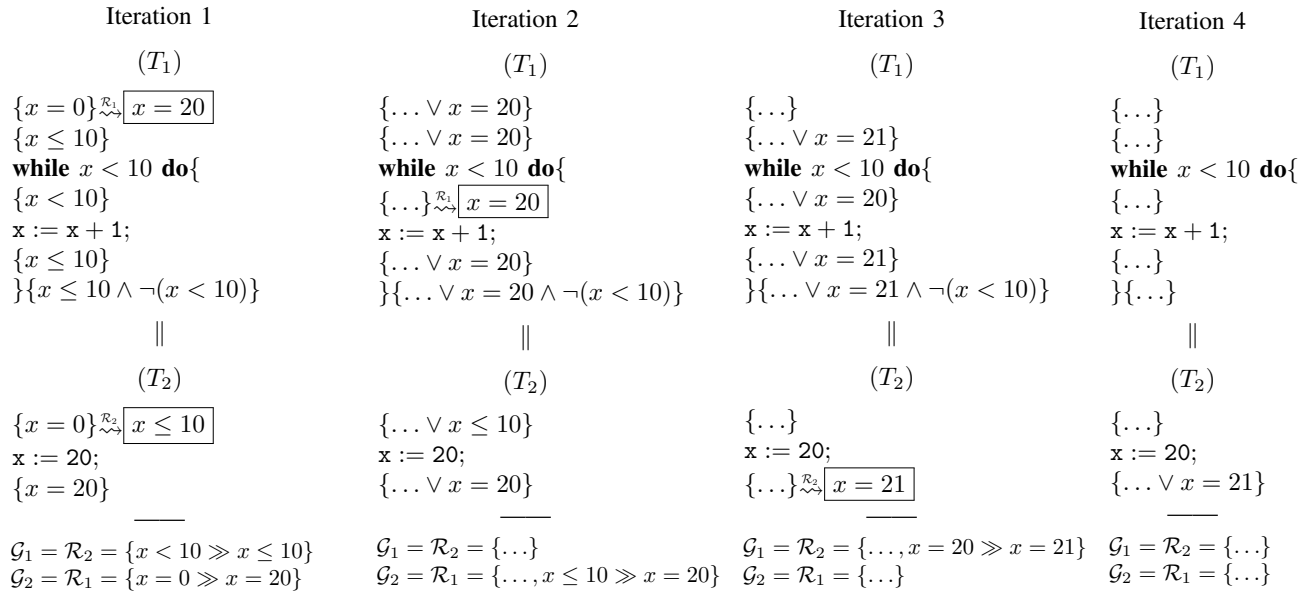


Fig. 3: Example of proof refinement via counterexamples.

not support the discovering of such invariant, its strength is to maneuver the initial sequential invariant to construct the counterpart concurrent invariant through iterative refinement. Moreover, such sequential invariant can be obtained by leveraging static analysis tools like [26] or [36].

In the first iteration, **PG** constructs a sequential proof for each thread using the precondition $x = 0$. As for T_1, the invariant $x \leq 10$ helps to establish both the loop precondition and the postcondition at the end of each iteration. The postcondition of T_1 is obtained by taking the conjunction of the invariant and the negation of the loop condition, which is equivalent to $x = 10$. As for T_2, the triple $\{x = 0\}\mathtt{x} := 20\{x = 20\}$ is derived. After that, the Guarantees are constructed from these proofs by considering all the transitions — pairs of pre/postconditions in Hoare triples — where the program state is modified. The Rely of T_1 (resp. T_2) is then derived from the Guarantee of T_2 (resp. T_1). This gives us:

$$\mathcal{G}_1 = \mathcal{R}_2 = \{x < 10 \gg x \leq 10\}$$
$$\mathcal{G}_2 = \mathcal{R}_1 = \{x = 0 \gg x = 20\}$$

The next step involves **CV** checking the consistency between each local proof and the respective Rely. Intuitively, this condition indicates that the local proof correctly reflects the environment interference in the Rely. Here **CV** finds that the assertion $x = 0$ in T_1 is *unstable* under $\mathcal{R}_1$, meaning that the assertion fails to contain the interference state $x = 20$ in $\mathcal{R}_1$. Technically, this check is achieved by verifying that the following stability formula, as being derived from the unstable assertion and the Rely $\mathcal{R}_1$, is invalid:

$$x_1 = 0 \rightarrow x_1 = 0 \rightarrow x_2 = 20 \rightarrow x_2 = 0$$

The counterexample $x = 20$, the image of the transition $x = 0 \gg x = 20$ in $\mathcal{R}_1$, is generated by **CV** to refine the proof of T_1. This counterexample represents the missing states to be included in the proof during the next refinement cycle. Likewise, **CV** detects in T_2 that the predicate $x = 0$ is unstable under $\mathcal{R}_2$, in which it generates the counterexample $x \leq 10$.

The second iteration begins with **PG** using counterexamples to refine proofs. Each counterexample is treated as precondition to construct a sub-proof to be disjunctively merged with the main proof. The intuition behind this sub-proof construction and merging is to introduce new states, as being caused by the environment interference, into the proof. This refinement is guided by a set of inference rules which are elaborated in §V-B. Furthermore, if the counterexample happens to be in the middle of the proof, then only the affected proof segment needs to be refined. More specifically, the refinement typically begins at the assertion associated with the counterexample. Yet one single exception is when the counterexample is in the loop body and causes the loop invariant to break. In this case, the refinement begins before the loop so that the loop invariant can be updated properly.

Back to our example, the counterexample $x = 20$ of T_1 is set as the precondition to construct the sub-proof, where it also becomes the loop invariant. As the main proof of T_1 being merged with the new sub-proof, the invariant of T_1 is updated to be the disjunction of the two invariants:

$$x \leq 10 \vee x = 20$$

Similarly, the counterexample $x \leq 10$ is used to refine the proof of T_2, where the sub-proof $\{x \leq 10\}\mathtt{x} := 20\{x = 20\}$ is constructed for merging. Besides, the R-G relations in both threads are extended to reflect the new transitions from sub-proofs, *i.e.* the transition $x \leq 10 \gg x = 20$ is added to both $\mathcal{G}_2$ and $\mathcal{R}_1$. This time, the predicate $x < 10$ in T_1 is unstable under $\mathcal{R}_1$, and is witnessed by the counterexample $x = 20$.

In the third iteration, the proof of T_1 is refined by the counterexample $x = 20$. The corresponding sub-proof is constructed with the new invariant $x = 21$, followed by the routine of merging sub-proof with main proof. Here the fresh invariant is combined with the old invariant of T_1 into:

$$x \leq 10 \vee x = 20 \vee x = 21 \tag{1}$$

As the new transition $x = 20 \gg x = 21$ being added to $\mathcal{G}_1$ and $\mathcal{R}_2$, the postcondition $x = 20$ of T_2 becomes unstable under $\mathcal{R}_2$. This results in the counterexample $x = 21$ being generated by **CV**.

The last refinement step takes place in T_2 during the fourth iteration, where the fresh counterexample $x = 21$ is disjunctively merged with the postcondition $x = 20$ into $x = 20 \vee x = 21$. In doing so, a valid R-G proof is established, in which the final R-G relations are constructed to be:

$$\mathcal{G}_1 = \mathcal{R}_2 = \{x < 10 \gg x \leq 10, x = 20 \gg x = 21\}$$
$$\mathcal{G}_2 = \mathcal{R}_1 = \{x = 0 \gg x = 20, x \leq 10 \gg x = 20\}$$

and the invariant in T_1 is transformed from the initial predicate $x < 10$ into its final form (1). Since the refinement process is completed, the postcondition of $T_1 \| T_2$ is obtained by conjuncting the postconditions of T_1 and T_2 together, which implies the verification condition $x = 20 \vee x = 21$ as desired.

IV. Language and semantics

We explain the semantics backbone of our approach, starting from the language to write concurrent programs §IV-A, followed by the introduction of semantics notations §IV-B, and finally the construction of R-G semantics §IV-C.

A. Programming language

$$\begin{array}{lcl} e & ::= & \mathit{var} \mid \mathit{const} \mid e + e \mid e - e \mid e \times e \mid e \ \mathbf{div}\ e \mid e \ \%\ e \\ b & ::= & \mathbf{true} \mid \mathbf{false} \mid e = e \mid e \leq e \mid !b \mid b \ \&\&\ b \mid b \mid\mid b \\ \mathtt{c} & ::= & \mathbf{skip} \mid e := e \mid \mathtt{c}\,;\mathtt{c} \mid \mathbf{if}\ b\ \mathbf{then}\ \mathtt{c}\ \mathbf{else}\ \mathtt{c} \mid \\ & & \mathbf{while}\ b\ \mathbf{do}\ \mathtt{c} \mid \mathbf{atomic}\ \mathtt{c} \mid \mathbf{await}\ b\ \mathbf{do}\ \mathtt{c} \mid \mathtt{c} \| \mathtt{c} \end{array}$$

Fig. 4: Syntax for writing concurrent programs.

Fig. 4 displays our lightweight language for writing concurrent programs. We use non-terminal symbol e and b to represent arithmetic and boolean expressions respectively. The program syntax in $\mathtt{c}$ includes standard sequential statements such as **skip**, assignment, composition, and conditional statement. More importantly, $\mathtt{c}$ also contains statements for writing concurrent code. In particular, $\mathbf{atomic}\ \mathtt{c}$ enables atomic execution of $\mathtt{c}$ without interleaving, $\mathbf{await}\ b\ \mathbf{do}\ \mathtt{c}$ preempts the program execution until the condition b holds then runs $\mathtt{c}$ atomically, and $\mathtt{c}_1 \| \mathtt{c}_2$ runs $\mathtt{c}_1$ and $\mathtt{c}_2$ concurrently.

B. Notation of the semantics

A program state s is a mapping from variables to values. The evaluation of a variable x in a state s is represented as $[\![x]\!]_s$, in which we override it as $[\![e]\!]_s$ and $[\![P]\!]_s$ for the evaluation of the expression e and the predicate P on the state s. Given a transition $P \gg Q$, we denote $\Theta(P \gg Q)$ the set of variables not used in $P \gg Q$, formally:

$$\forall x, v_1, v_2, s_1, s_2. v_1 \neq v_2 \rightarrow [\![x]\!]_{s_1} = v_1 \rightarrow [\![x]\!]_{s_2} = v_2 \rightarrow ([\![P]\!]_{s_1} = [\![P]\!]_{s_2} \wedge [\![Q]\!]_{s_1} = [\![Q]\!]_{s_2})$$

A pair of states (s_1, s_2) is in $P \gg Q$ if $[\![P]\!]_{s_1} \wedge [\![Q]\!]_{s_2}$, and $\forall x \in \Theta(P \gg Q). [\![x]\!]_{s_1} = [\![x]\!]_{s_2}$, that is variables belonging to $\Theta(P \gg Q)$ remain intact. In general, a binary relation $\mathcal{B} = \{P_1 \gg Q_1, \ldots, P_n \gg Q_n\}$ represents all pairs (s_1, s_2) where $s_1 = s_2$, or there exists some $P_i \gg Q_i \in \mathcal{B}$ such that $(s_1, s_2) \in P_i \gg Q_i$. The former condition implies that our relations contain the identity relation, which is a necessary technicality of the R-G framework.

Given a predicate P and a relation $\mathcal{R}$, we say P is *stable* under $\mathcal{R}$ — denoted by *stable*$(P, \mathcal{R})$ — if for every state s_1, s_2 such that $(s_1, s_2) \in \mathcal{R}$ and $[\![P]\!]_{s_1}$, we have $[\![P]\!]_{s_2}$. Also, a predicate C is a *counterexample* of P derived from $\mathcal{R}$ (or simply counterexample if there is no ambiguity) if the following criteria are met:

1) C is $\bot$ or of the form $\bigvee U$ where each U is retrieved from the image of some transition $R \gg U$ in $\mathcal{R}$, and
2) The disjunction $P \vee C$ is stable under $\mathcal{R}$.

Intuitively, the counterexample C represents the missing states of P as caused by the environment interference in $\mathcal{R}$.

C. Semantics

We briefly recall the standard R-G semantics [23]. A *configuration* (*config*) $\langle \mathtt{c}, s \rangle$ consists of a program $\mathtt{c}$ and its state s. A *config* is *final* if the program component is **skip**. The *sequential transition* $\rightsquigarrow$ is a binary relation between two *configs*. By contrast, the *concurrent transition* $\overset{\chi}{\underset{\mathcal{R}}{\rightsquigarrow}}$, in which $\mathcal{R}$ is the Rely and χ the label, is the union of two sub-transitions:

1) the internal transition $\overset{\delta}{\underset{\mathcal{R}}{\rightsquigarrow}}$ that is analogous to the sequential transition, where the thread state is modified according to its internal code, and
2) the external transition $\overset{\epsilon}{\underset{\mathcal{R}}{\rightsquigarrow}}$ where $\langle \mathtt{c}, s \rangle \overset{\epsilon}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathtt{c}, s' \rangle$ means $(s, s') \in \mathcal{R}$, and the code $\mathtt{c}$ is untouched. This transition accounts for the state alteration caused by the environment interference.

The closures of the sequential transition and concurrent transition are denoted by $\overset{*}{\rightsquigarrow}$ and $\overset{*}{\underset{\mathcal{R}}{\rightsquigarrow}}$ respectively. A *program execution* is a sequence of consecutive configs that satisfy the transition relation, *e.g.* as with concurrent transition:

$$\langle \mathtt{c}_1, s_1 \rangle \overset{\chi}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathtt{c}_2, s_2 \rangle \overset{\chi}{\underset{\mathcal{R}}{\rightsquigarrow}} \ldots \overset{\chi}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathtt{c}_{n-1}, s_{n-1} \rangle \overset{\chi}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathtt{c}_n, s_n \rangle$$

or $\langle \mathtt{c}_1, s_1 \rangle \overset{*}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathtt{c}_n, s_n \rangle$ if intermediate configs are not important.

The sequential triple $\{P\}\mathtt{c}\{Q\}$ is defined by the formula:

$$\{P\}\mathtt{c}\{Q\} \triangleq \forall s, s'.\ [\![P]\!]_s \rightarrow \langle \mathtt{c}, s \rangle \overset{*}{\rightsquigarrow} \langle \mathbf{skip}, s' \rangle \rightarrow [\![Q]\!]_{s'}$$

This means that for every program execution starting from a config $\langle \mathtt{c}, s \rangle$ — in which $[\![P]\!]_s$ holds — and terminating in some final config $\langle \mathbf{skip}, s' \rangle$, it follows that $[\![Q]\!]_{s'}$ holds.

The semantics of Hoare triple with R-G relations is quite cumbersome. First, we need the notation $\mathcal{R}, \mathcal{G} \models_n \langle \mathtt{c}, s \rangle$ to indicate that the config $\langle \mathtt{c}, s \rangle$ satisfies the conditions $\mathcal{R}, \mathcal{G}$ for n steps. Inductively, $\mathcal{R}, \mathcal{G} \models_n \langle \mathtt{c}, s \rangle$ holds if $n = 0$; or $n > 0$

Algorithm 1: Proof construction and refinement by **PG**.

1 **Procedure** `construct()`
  Input: program $\mathtt{c_1} \| \ldots \| \mathtt{c_n}$, precondition P
  Output: sequential proofs with R-G conditions
2 **foreach** *thread* $i = 1 \ldots n$ **do**
3 use P to construct sequential proof $\mathbb{P}_i$ and $\mathcal{G}_i$
4 **foreach** *thread* $i = 1 \ldots n$ **do** $\mathcal{R}_i \leftarrow \bigcup_{\substack{j=1 \\ j \neq i}}^{n} \mathcal{G}_j$
5 **return** $\{(\mathbb{P}_i, \mathcal{G}_i, \mathcal{R}_i)\}_{i=1}^{n}$
6 **Procedure** `refine()`
  Input: current local proofs with R-G conditions $\{(\mathbb{P}_i, \mathcal{G}_i, \mathcal{R}_i)\}_{i=1}^{n}$, counterexamples $\{C_i\}_{i=1}^{n}$
  Output: refined local proofs and R-G conditions
7 **foreach** *thread* $i = 1 \ldots n$ **do**
8 **foreach** U_j *in* $C_i = \bigvee_{j=1}^{k_i} U_j$ **do**
9 use U_j to construct the sub-proof $\mathbb{P}_j$
10 merge $\mathbb{P}_j$ with $\mathbb{P}_i$ and update $\mathcal{G}_i$
11 **foreach** *thread* $i = 1 \ldots n$ **do** $\mathcal{R}_i \leftarrow \bigcup_{\substack{j=1 \\ j \neq i}}^{n} \mathcal{G}_j$
12 **return** $\{(\mathbb{P}_i, \mathcal{G}_i, \mathcal{R}_i)\}_{i=1}^{n}$

and for every $\mathtt{c}'$, s', χ s.t. $\langle \mathtt{c}, s \rangle \overset{\chi}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathtt{c}', s' \rangle$, the following conditions hold:

1) $\mathcal{R}, \mathcal{G} \models_{n-1} \langle \mathtt{c}', s' \rangle$, and
2) $(s, s') \in \mathcal{G}$ if $\chi = \sigma$, *i.e.* each internal step must satisfy the Guarantee relation.

With the new notation, the concurrent Hoare triple with R-G conditions $\mathcal{R}, \mathcal{G} \vdash \{P\}\mathtt{c}\{Q\}$ can be defined by the formula:

$$\mathcal{R}, \mathcal{G} \vdash \{P\}\mathtt{c}\{Q\} \triangleq \forall s, s'.\ [\![P]\!]_s \rightarrow \Big(\big(\forall n.\ \mathcal{R}, \mathcal{G} \models_n \langle \mathtt{c}, s \rangle\big) \wedge \big(\langle \mathtt{c}, s \rangle \overset{*}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathbf{skip}, s' \rangle \rightarrow [\![Q]\!]_s\big)\Big)$$

That is, for every execution starting from $\langle \mathtt{c}, s \rangle$ in which $[\![P]\!]_s$ holds, the condition $\mathcal{R}, \mathcal{G} \models_n \langle \mathtt{c}, s \rangle$ satisfies for every n, and if the execution terminates in $\langle \mathbf{skip}, s' \rangle$ then $[\![Q]\!]_s$ holds.

V. Proof Generator

Proof Generator (**PG**) — one of the two main components in our framework — applies a set of inference rules to construct and refine proofs. In this section, the key procedures in **PG** are first addressed in §V-A, followed by the discussion on the related inference rules in §V-B.

A. Proof construction and refinement

Algo. 1 presents the two key procedures `construct` and `refine` in **PG**. As suggested by its name, the procedure `construct` is called during the first iteration to construct an initial sequential proof and R-G relations for each thread. In subsequent iterations, the procedure `refine` uses counterexamples to refine proofs and recomputes the R-G relations.

We take a close look at the procedure `construct` in Algo. 1. The procedure accepts a concurrent program of n threads $\mathtt{c_1} \| \ldots \| \mathtt{c_n}$ and a precondition P as input, and uses them to construct the sequential proofs $\{\mathbb{P}_i\}_{i=1}^{n}$ and R-G relations $\{(\mathcal{R}_i, \mathcal{G}_i)\}_{i=1}^{n}$. To do so, first the condition P is set as the local precondition for each thread $\mathtt{c_i}$ to construct the thread's sequential proof $\mathbb{P}_i$, as at lines 2–3. Also at line 3, each Guarantee $\mathcal{G}_i$ is derived simultaneously with the construction of $\mathbb{P}_i$. This step is guided by a set of inference rules to be elaborated in §V-B. At line 4, each Rely $\mathcal{R}_i$ is computed to be the union of all Guarantees from other threads. This computation is consistent with the intuition that the environment interference of one thread is caused by the actions of other threads. Thanks to our relations being represented as sets, the above step is achieved effortlessly through set union. At line 5, the sequential proofs and R-G relations are returned as the candidate R-G proof for validation.

In subsequent iterations, the procedure `refine` uses a list of counterexamples $\{C_i\}_{i=1}^{n}$ to tackle the proof refinement. Recall that each C_i contains the missing states caused by the environment interference. Furthermore, each C_i is of the form $\bigvee_{j=1}^{i_k} U_j$ where U_j is the image of some transition $P_j \gg U_j$ in the Rely $\mathcal{R}_i$. At lines 8–10, the procedure treats each U_j as precondition to construct a sub-proof, and then merges the freshly constructed sub-proofs with the main proof $\mathbb{P}_i$. Simultaneously, fresh transitions taken from sub-proofs are added to the Guarantee $\mathcal{G}_i$, as at line 10. At line 11, the Relies are derived from the Guarantees, followed by the new candidate R-G proof being returned for validation at line 12.

B. Deductive system

Fig. 5 presents a set of viable inference rules used by **PG**. These rules are our sledgehammer to construct local proofs and Guarantee relations. We write $\mathcal{G} \triangleleft \{P\}\mathtt{c}\{Q\}$ to denote that the Hoare triple $\{P\}\mathtt{c}\{Q\}$ is sequentially valid, and it also satisfies the Guarantee $\mathcal{G}$. This notation can be defined in terms of the R-G semantics in §IV-C, where the Rely is empty, *i.e.* no interference. Formally, let $\mathcal{R}_\emptyset = \emptyset$ then:

$$\mathcal{G} \triangleleft \{P\}\mathtt{c}\{Q\} \triangleq \mathcal{R}_\emptyset, \mathcal{G} \vdash \{P\}\mathtt{c}\{Q\}$$

By ignoring the Guarantee part $\mathcal{G}\triangleleft$, the rules in Fig. 5 become familiar, as they are essentially inference rules of Hoare logic. What makes our rules unique is that they also infer the Guarantee along the way. In principle, this relation is computed by considering all transitions $P \gg Q$ taken from Hoare triples $\{P\}\mathtt{c}\{Q\}$, where the statement $\mathtt{c}$ is atomic. We now spend the rest of this section to explain these rules.

We first mention the rules for atomic statements which are quite straightforward. In [SKIP], the Hoare triple $\{P\}\mathbf{skip}\{P\}$ and the singleton Guarantee $\{P \gg P\}$ are inferred. As for the rule [ASSIGN], the triple $\{P_{[x/e]}\}\mathtt{x} := \mathtt{e}\{P\}$ and Guarantee $\{P_{[x/e]} \gg P\}$ are inferred for the assignment $\mathtt{x} := \mathtt{e}$. Here the precondition $P_{[x/e]}$ is constructed from P by replacing occurrences of x with e. Both rules [ATOM] and [AWAIT] have their triples inferred from the triples of their sub-proofs. In particular, the triple $\{P\}\mathbf{atomic}\ \mathtt{c}\{Q\}$ in [ATOM], together with the Guarantee $\{P \gg Q\}$, is inferred from the sub-proof $\{P\}\mathtt{c}\{Q\}$. In case of [AWAIT], the triple $\{P\}\mathbf{atomic}\ \mathtt{c}\{Q\}$ and Guarantee $\{P \wedge b \gg Q\}$ are inferred from $\{P \wedge b\}\mathtt{c}\{Q\}$.

We have compositional rules to combine the Guarantees together. They are [IF] for conditional statement, [SEQ] for

$$\frac{}{\{P \gg P\} \lhd \{P\}\mathbf{skip}\{P\}}\ [\textsc{skip}] \qquad \frac{}{\{P_{[x/e]} \gg P\} \lhd \{P_{[x/e]}\}\mathtt{x} := \mathtt{e}\{P\}}\ [\textsc{assign}] \qquad \frac{\{P\}\mathtt{c}\{Q\}}{\{P \gg Q\} \lhd \{P\}\mathbf{atomic}\ \mathtt{c}\{Q\}}\ [\textsc{atom}]$$

$$\frac{\{P \wedge b\}\mathtt{c}\{Q\}}{\{P \wedge b \gg Q\} \lhd \{P\}\mathbf{await}\ b\ \mathbf{do}\ \mathtt{c}\{Q\}}\ [\textsc{await}] \qquad \frac{\begin{array}{c}\mathcal{G}_1 \lhd \{P \wedge b\}\mathtt{c}_1\{Q_1\} \\ \mathcal{G}_2 \lhd \{P \wedge \neg b\}\mathtt{c}_2\{Q_2\}\end{array}}{\mathcal{G}_1 \cup \mathcal{G}_2 \lhd \{P\}\mathbf{if}\ b\ \mathbf{then}\ \mathtt{c}_1\ \mathbf{else}\ \mathtt{c}_2\{Q_1 \vee Q_2\}}\ [\textsc{if}]$$

$$\frac{\begin{array}{c}\mathcal{G}_1 \lhd \{P\}\mathtt{c}_1\{S\} \\ \mathcal{G}_2 \lhd \{S\}\mathtt{c}_2\{Q\}\end{array}}{\mathcal{G}_1 \cup \mathcal{G}_2 \lhd \{P\}\mathtt{c}_1;\mathtt{c}_2\{Q\}}\ [\textsc{seq}] \qquad \frac{\begin{array}{c}\mathcal{G}_1 \lhd \{P_1\}\mathtt{c}\{Q_1\} \\ \mathcal{G}_2 \lhd \{P_2\}\mathtt{c}\{Q_2\}\end{array}}{\mathcal{G}_1 \cup \mathcal{G}_2 \lhd \{P_1 \vee P_2\}\mathtt{c}\{Q_1 \vee Q_2\}}\ [\textsc{disj}] \qquad \frac{\begin{array}{c}\mathcal{G} \lhd \{P\}\mathtt{c}\{Q\} \\ \mathcal{G} \subseteq \mathcal{G}'\ P' \Rightarrow P\ Q \Rightarrow Q'\end{array}}{\mathcal{G}' \lhd \{P'\}\mathtt{c}\{Q'\}}\ [\textsc{cons}]$$

$$\frac{\begin{array}{c}\mathcal{G} \lhd \{I \wedge b\}\mathtt{c}_1;\mathtt{c}_2\{I\} \\ \mathcal{G}' \lhd \{I' \wedge b\}\mathtt{c}_1;\{P\}\mathtt{c}_2\{I'\}\end{array}}{\mathcal{G} \cup \mathcal{G}' \lhd \{(I \vee I') \wedge b\}\mathbf{while}\ b\ \mathbf{do}\ \{\mathtt{c}_1;\mathtt{c}_2\}\{(I \vee I') \wedge \neg b\}}\ [\textsc{inv}] \qquad \frac{\begin{array}{c}Q \wedge b \Rightarrow I \quad \mathcal{G} \lhd \{I \wedge b\}\mathtt{c}_1;\mathtt{c}_2\{I\} \\ \mathcal{G}' \lhd \{P\}\mathtt{c}_2\{Q\}\end{array}}{\mathcal{G} \cup \mathcal{G}' \lhd \{(I \vee Q) \wedge b\}\mathbf{while}\ b\ \mathbf{do}\ \{\mathtt{c}_1;\mathtt{c}_2\}\{(I \vee Q) \wedge \neg b\}}\ [\textsc{winv}]$$

Fig. 5: Deductive system for the construction of local proofs and Guarantee relations.

composition, and [DISJ] for proof weakening. In these rules, the result Guarantee is simply the union of the Guarantees from sub-proofs. Notably, the last rule [DISJ] is important for merging sub-proofs, *i.e.* by taking the disjuction of corresponding assertions from sub-proofs. We also have the rule [CONS] for rewriting proof, in which the pre/postcondtions and Guarantee are replaced by weaker conditions and relation.

Lastly, the two rules [INV] and [WINV] are viable for updating loop invariant. We abuse the notation $\{P\}\mathtt{c}_1;\{Q\}\mathtt{c}_2\{R\}$ to represent the triple $\{P\}\mathtt{c}_1;\mathtt{c}_2\{R\}$ where $\{P\}\mathtt{c}_1\{Q\}$ and $\{Q\}\mathtt{c}_2\{R\}$ both hold. The default invariant I is associated with the loop $\mathbf{while}\ b\ \mathbf{do}\ \{\mathtt{c}_1;\mathtt{c}_2\}$. The counterexample P is inside the loop body, between $\mathtt{c}_1$ and $\mathtt{c}_2$. Such counterexample can be placed at the beginning (resp. ending) of the loop body by instantiating $\mathtt{c}_1$ (resp. $\mathtt{c}_2$) with **skip**. In [INV], the refined invariant is $I \vee I'$, where I' is the fresh invariant that satisfies:

$$\{I' \wedge b\}\mathtt{c}_1;\{P\}\mathtt{c}_2\{I'\}$$

However, automatically discovering I' could be problematic due to the presence of P in the middle of the proof. Thus we invent the rule [WINV] as a workaround to update the loop invariant. The inferred invariant in [WINV], despite being more ad hoc than its peer in [INV], is actually a blessing in disguise, as it can be computed directly from the sub-proof of the loop body. To do so, the triple $\{P\}\mathtt{c}_2\{Q\}$ is derived from P such that $Q \wedge b$ implies I. The loop invariant then can be correctly updated to $I \vee Q$. Also, the Guarantees in both rules are updated by adding transitions from the derived sub-proofs.

VI. Consistency Verifier

We first elaborate on the routine steps of the component Consistency Verifier (**CV**) in §VI-A. We then discuss in §VI-B the rules used by **CV** for constructing counterexamples.

A. The procedure description

Algo. 2 describes the high-level architecture of the routine in **CV** for validating R-G proof and computing counterexamples.

Algorithm 2: Proof validation by **CV**.

Input: candidate R-G proof $\{(\mathbb{P}_i, \mathcal{R}_i, \mathcal{G}_i)\}_{i=1}^n$
Output: valid R-G proof, otherwise counterexamples

1 **foreach** *proof* $\mathbb{P}_i$ **do**
2 let P be the first basic assertion unstable under $\mathcal{R}_i$
3 construct counterexample C_i from P and $\mathcal{R}_i$
4 **if** *no counterexample is constructed* **then**
5 **return** $\{(\mathbb{P}_i, \mathcal{R}_i, \mathcal{G}_i)\}_{i=1}^n$
6 **else**
7 **return** $\{C_i\}_{i=1}^n$

Before explaining the key routine steps, we first need several definitions. A *basic statement* is an atomic statement — *i.e.* **skip**, assignment, **atomic**, **await** — that is not within the scopes of other basic statements. A proof assertion is *basic* if it is the pre/postcondition of either a basic statement, or the entire thread. These basic assertions are precisely program points where the environment interference can occur. As with the R-G semantics, basic assertions are required to be stable under the Rely, meaning that they contain the interference states specified by the Rely. Checking validity of R-G proof is therefore reduced to checking stability between basic assertions and Rely relation for each thread.

We now discuss the routine steps taken by **CV** in Algo. 2. The input is a candidate R-G proof $\{(\mathbb{P}_i, \mathcal{R}_i, \mathcal{G}_i)\}_{i=1}^n$ consisting of local proofs and R-G relations. At lines 1–3, each local proof $\mathbb{P}_i$ is examined separately, where its basic assertions are checked against the Rely $\mathcal{R}_i$ for stability condition. The first unstable basic assertion is used to construct the counterexample predicate C_i for $\mathbb{P}_i$. Detail on the checking stability and constructing counterexamples is elaborated in §VI-B. If no counterexample is constructed, then the R-G proof is valid and the candidate proof $\{(\mathbb{P}_i, \mathcal{R}_i, \mathcal{G}_i)\}_{i=1}^n$ is returned as a validated proof at line 5. Otherwise, a list of counterexamples $\{C_i\}_1^n$ is

$$\frac{\bigwedge_{R \gg Q \in \mathcal{R}} P \Rightarrow R \Rightarrow Q_{[\bar{x}/\bar{y}]} \Rightarrow \mathit{irr}(P, R \gg Q)^{\bar{x}}_{\bar{y}} \Rightarrow P_{[\bar{x}/\bar{y}]}}{\mathit{stable}(P, \mathcal{R})} \ [\textsc{st}]$$

$$\frac{C = \bigvee_{U \in \mathbb{U}} U \ \mathit{s.t.}\ \mathbb{U} = \{U \mid \exists R.\ R \gg U \in \mathit{closure}(\mathcal{R}) \wedge \mathit{sat}(R \wedge P)\}}{\mathit{stable}(P \vee C, \mathcal{R})} \ [\textsc{re}]$$

Fig. 6: Rules for constructing counterexamples.

returned to enable the next refinement cycle.

B. Construction of counterexample predicates

Fig. 6 consists of the two viable rules [ST] and [RE] used by **CV** routinely. A basic assertion is first checked for stability using [ST], and if the assertion is unstable then [RE] is applied to construct the counterexample predicate. These rules require several notations to explain. The closure of $\mathcal{R}$ is denoted by $\mathit{closure}(\mathcal{R})$, and $\mathit{sat}(\Phi)$ indicates that the formula Φ is satisfiable. The condition $\mathit{irr}(P, R \gg Q)^{\bar{x}}_{\bar{y}}$ is the conjunction of equalities $\bigwedge(x_i = y_i)$ where each variable $x_i \in \bar{x}$ is present in P but absent in $R \gg Q$, and $y_i \in \bar{y}$ is a fresh variable associated with x_i. The intuition behind this condition is that if a variable x_i is in P but not in $R \gg Q$, then any state transition (s_1, s_2) in $R \gg Q$ respects the value of x_i, *i.e.* $[\![x]\!]_{s_1} = [\![x]\!]_{s_2}$. Also, we write $P_{[\bar{x}/\bar{y}]}$ to denote the assertion P in which each variable $x_i \in \bar{x}$ is replaced by its counterpart variable $y_i \in \bar{y}$.

In rule [ST], the condition $\mathit{stable}(P, \mathcal{R})$ is established by verifying that P is stable under every transition $R \gg Q$ in $\mathcal{R}$. This sub-condition is expressible as the implication below:

$$P \rightarrow R \rightarrow Q_{[\bar{x}/\bar{y}]} \rightarrow \mathit{irr}(P, R \gg Q)^{\bar{x}}_{\bar{y}} \rightarrow P_{[\bar{x}/\bar{y}]}$$

where fresh variables $\bar{y}$ carry the effects caused by the transition $R \gg Q$ on the variables $\bar{x}$ of P. The condition $\mathit{irr}(P, R \gg Q)^{\bar{x}}_{\bar{y}}$ directs the check to focus only on the changes in shared variables between P and $R \gg Q$, *i.e.* by retaining the values of other variables. We provide the following examples to help readers gain intuition about how [ST] works. Let:

$$\mathcal{R} : \{x \geq 0 \gg x \geq 1\} \qquad P_1 : x = 0 \qquad P_2 : z = 0$$

Using [ST], the stability conditions for P_1 and P_2 are:

$$\Phi_1 : x = 0 \rightarrow x \geq 0 \rightarrow y \geq 1 \rightarrow \top \rightarrow y = 0$$
$$\Phi_2 : z = 0 \rightarrow x \geq 0 \rightarrow y \geq 1 \rightarrow z = y \rightarrow y = 0$$

It can be verified that Φ_1 is invalid while Φ_2 is valid. As a result, only the assertion P_2 is stable under $\mathcal{R}$.

The other rule [RE] plays the role of constructing counterexample predicates. Given an assertion P being unstable under the Rely $\mathcal{R}$, recall that a predicate C is a counterexample of P if the disjunction $P \vee C$ is stable under $\mathcal{R}$. In [RE], the counterexample C is computed to be $\bigvee U$, where each U is retrieved from the image of some transition $R \gg U$ in $\mathit{closure}(\mathcal{R})$, such that the constraint $R \wedge P$ is satisfiable. The procedure for over-approximating $\mathit{closure}(\mathcal{R})$ from $\mathcal{R}$ is given

Algorithm 3: Approximate transitive closure of $\mathcal{R}$.

Data: A binary relation $\mathcal{R}$
Result: Its over-approximated transitive closure $\mathcal{C}$

```
1 initiate C ← R
2 repeat
3     reset N ← ∅
4     foreach P1 ≫ Q1 ∈ C and P2 ≫ Q2 ∈ R do
5         if sat(Q1 ∧ P2) and P1 ≫ Q2 ∉ C then
6             update N ← N ∪ {P1 ≫ Q2}
7     C ← C ∪ N
8 until N = ∅
```

in Algo. 3, which we now explain. The closure $\mathcal{C}$ of $\mathcal{R}$ is initiated to $\mathcal{R}$ (line 1), and is extended iteratively (lines 4-6) by adding fresh transitions from the composition relation $\mathcal{C} \circ \mathcal{R}$; until the fixpoint condition — *i.e.* $\mathcal{C} = \mathcal{C} \circ \mathcal{R}$ — is reached. The sub-relation $\mathcal{C} \circ \mathcal{R}$ consists of transitions $P_1 \gg Q_2$, where there exist Q_1, P_2 such that $\{P_1 \gg Q_1, P_2 \gg Q_2\} \subseteq \mathcal{C}$, and $Q_1 \wedge P_2$ is satisfiable. We use a temporary relation $\mathcal{N}$ to store the fresh relations $P_1 \gg Q_2$, and to check the fixpoint condition for termination, *i.e.* $\mathcal{N} = \emptyset$.

Lastly, one important property of the rule [RE] is that it computes the weakest counterexample, as mentioned in the correctness result of Lemma 1:

Lemma 1. *The disjunction $P \vee C$ in the rule [RE] is stable under $\mathcal{R}$. Furthermore, for any C' such that $P \vee C'$ is stable under $\mathcal{R}$, we have $P \vee C \Rightarrow P \vee C'$.*

Proof sketch. Given a state s s.t. $[\![P]\!]_s \vee [\![C]\!]_s$, rule [RE] ensures that any state s' s.t. $(s, s') \in \mathit{closure}(\mathcal{R})$ — *i.e.* s' is reachable from s via $\mathcal{R}$ — is in C, and thus $[\![C]\!]_{s'}$ holds. This implies $[\![P]\!]_{s'} \vee [\![C]\!]_{s'}$. Hence, $P \vee C$ is stable under $\mathcal{R}$.

To see why C is weakest, let s be a state s.t. $[\![P]\!]_s \vee [\![C]\!]_s$. It suffices to prove that $[\![P]\!]_s \vee [\![C']\!]_s$. If $[\![P]\!]_s$ holds then we are done. Otherwise, $[\![C]\!]_s$ holds and thus s is reachable via $\mathcal{R}$ from some s^* s.t. $[\![P]\!]_{s^*}$ holds. Thus there exists a transition $R \gg Q \in \mathit{closure}(\mathcal{R})$ s.t. $[\![R]\!]_{s^*} \wedge [\![P]\!]_{s^*}$ and $[\![Q]\!]_s$. From $[\![P]\!]_{s^*}$, we arrive at $[\![P]\!]_{s^*} \vee [\![C']\!]_{s^*}$. As $(s^*, s) \in \mathit{closure}(\mathcal{R})$, the condition $\mathit{stable}(P \vee C', \mathcal{R})$ implies $[\![P]\!]_s \vee [\![C']\!]_s$. □

VII. SOUNDNESS

Having our key components **PG** and **CV** addressed in previous sections §V and §VI, we are now ready to state the main soundness result about our framework:

Theorem 1 (Soundness)**.** *The proof constructed by the framework in Fig. 1 is valid with respect to the R-G semantics.*

Proof sketch. First, notice that the construction of Relies in Algo. 1 ensures that each Guarantee necessarily implies the Relies of other threads. Thus the rule [PAR] in §II can be applied to establish the concurrent proof from local proofs.

It remains to verify that each local proof $\{P\}\mathsf{c}\{Q\}$ and its R-G relations $(\mathcal{R}, \mathcal{G})$ satisfy the R-G definition in §IV-C:

$$\mathcal{R}, \mathcal{G} \vdash \{P\}\mathsf{c}\{Q\}$$

Let $\langle \mathtt{c}, s\rangle$ s.t. $[\![P]\!]_s$ holds. First, to see why $\mathcal{R}, \mathcal{G} \models_n \langle \mathtt{c}, s\rangle$ is true for arbitrary n, recall that our construction (*i.e.* Algo. 2) ensures that all basic assertions — program points where the environment interference can occur — are stable under $\mathcal{R}$. As a result, every program execution starting from $\langle \mathtt{c}, s\rangle$ has its external steps satisfy the Rely $\mathcal{R}$. Meanwhile, the condition $\mathcal{G} \triangleleft \{P\}\mathtt{c}\{Q\}$ implies that such program execution also satisfies the Guarantee $\mathcal{G}$. Hence, $\mathcal{R}, \mathcal{G} \models_n \langle \mathtt{c}, s\rangle$ holds.

Now let $\langle \mathbf{skip}, s'\rangle$ s.t. $\langle \mathtt{c}, s\rangle \overset{*}{\underset{\mathcal{R}}{\rightsquigarrow}} \langle \mathbf{skip}, s'\rangle$, we need to prove $[\![Q]\!]_{s'}$ holds. Notice that the proof construction in Algo. 1 satisfies the condition $\mathcal{G} \triangleleft \{P\}\mathtt{c}\{Q\}$. This implies $\{P\}\mathtt{c}\{Q\}$ is a valid Hoare triple, and thus $[\![Q]\!]_{s'}$ can be deduced. □

Discussion on completeness. The proof construction of our framework is incomplete. That is, the R-G proof for a given program is possibly nonexistent, and thus the refinement step fails to terminate. This shortcoming essentially boils down to the over-approximation of R-G relations as unordered sets of transitions. This simple representation is advantageous for computation, such as in the construction of Relies and counterexamples. However, it also shatters the total order of the program execution, making the reasoning infeasible in certain scenarios. For instance, our framework fails to establish the valid proof $\{x = 0\}\mathtt{x} := \mathtt{x} + 1 \| \mathtt{x} := \mathtt{x} + 1\{x = 2\}$ since the computation of R-G relations fails to converge. This is mainly because the locality and frequency of the statement $\mathtt{x} := \mathtt{x} + 1$ are lost as a consequence of the over-approximation.

In fact, the above problem of over-approximation is well-known in R-G framework, *e.g.* [9], [38], [39]. To tackle this problem, one common solution is to use *auxiliary* (or *ghost*) variables (*e.g.* [9], [38], [39]) for the bookkeeping of state transitions. By doing so, the order of execution can be encoded into the R-G relations for sensible reasoning. We adopt the above idea by introducing the rule [AU] below as a workaround to label necessary transitions:

$$\frac{\begin{array}{c}\mathit{fresh}(a) \qquad \mathit{atomic}(\mathtt{c}) \qquad \mathcal{G}_1 = \{P \gg Q\} \\ \mathcal{G}_2 = \{(P \wedge x = 0) \gg (Q \wedge x = 1)\}\end{array}}{\begin{array}{c}\mathcal{R}, \mathcal{G}_1 \vdash \{P\}\mathtt{c}\{Q\} \Leftrightarrow \\ \mathcal{R}, \mathcal{G}_2 \vdash \{P \wedge x = 0\}\mathbf{atomic}\ \{\mathtt{c}; \mathtt{x} := 1\}\{Q \wedge x = 1\}\end{array}}\ [\text{AU}]$$

Using a fresh variable x initiated to 0, the rule [AU] transforms the triple $\{P\}\mathtt{c}\{Q\}$, where c is atomic, into the triple in which $\mathtt{x} := 1$ is grouped with c into an atomic block:

$$\{P \wedge x = 0\}\mathbf{atomic}\ \{\mathtt{x} := 1; \mathtt{c}\}\{Q \wedge x = 1\}$$

As x is updated from 0 to 1 along with c, it helps to encode the transition $P \gg Q$ into $(P \wedge x = 0) \gg (Q \wedge x = 1)$. This allows the transition, as being associated with c, to be identified with ease when it was added to the R-G relations.

VIII. Evaluation

A. Implementation and optimization

The framework in Fig. 1 is realized as a Java prototype dubbed ReGaSol (**Re**ly **Gua**rantee **Sol**ver). Our prototype consists of 17 component classes with more than 4500 lines of code en masse. The SMT solver Z3 [10] is deployed as back-end to handle SAT constraints generated during stability check and counterexample construction. To improve the performance of ReGaSol, we have designed specifically several optimization strategies for our framework. Their key principles are briefly discussed below.

Dynamic programming. We propose a new method to compute the new Rely closure by reusing the previous version from the last iteration. In detail, the new closure $\mathcal{C}_{\text{new}}$ is extended gradually from $\mathcal{C}_{\text{old}}$ by considering one fresh transition $P \gg Q$ at a time. This task is accomplished by computing first the composed relation $\mathcal{P} = \{P \gg Q\} \circ \mathcal{C}$, and then the relation $\mathcal{C} \circ \mathcal{P}$. In practice, this approach helps to reduce the total number of SAT constraints, particularly when the number of transitions in Rely is large.

Symmetry reduction. We say that a thread T_1 is *symmetric* to another thread T_2 if there exists a bijective substitution M mapping variables from T_1 to T_2, such that T_1 becomes T_2 after the substitution. As a result, the mapping M can be used to transform the proof of T_1 and the respective R-G relations into their counterparts in T_2, effectively reducing the workload associated with T_2. Thus this approach can improve run-time performance when verifying programs with symmetries.

Parallelization. For each thread, the local proof construction and stability check are performed independently. As a result, we can create multiple instances of **PG** and **CV**, one per thread, to accomplish the above tasks in parallel. As per iteration, synchronization only occurs when the fresh Relies are constructed, since these relations are derived from the Guarantees of other threads.

B. Experiment design

Table I presents our small benchmark of 12 practical multi-threaded programs split equally into category 1 (P1–P6) and 2 (P7–P12). The first category (P1–P6) consists of standard algorithms for mutual exclusion, namely Peterson [2], naïve Bakery [28] and its complete version [25], Szymanski [37], readers-writers lock [12], time-varying mutex [12]. Meanwhile, programs P7–P12 in category 2 contain loops to test the scalability of our framework. In detail, P7 is the example program in §III, and P8 is modified from P7 by changing the loop condition to $x < 100$. Program P9 contains two threads $\{\mathbf{while}\ x < 10\ \mathbf{do}\ \mathtt{x} := \mathtt{x} + 1\}$ and $\{\mathbf{while}\ y < 20\ \mathbf{do}\ \mathtt{y} := \mathtt{y} + 1\}$. Its loop conditions are changed in P10 to $x < 100$ and $y < 200$. Program P11 is generated from P9 by adding $\{\mathbf{await}\ x = 10 \&\& y = 20\ \mathbf{do}\ \mathtt{z} := \mathtt{x} + \mathtt{y}\}$ as the third thread. Lastly, P12 is generated from P10 by adding the third thread $\{\mathbf{await}\ x = 100 \&\& y = 200\ \mathbf{do}\ \mathtt{z} := \mathtt{x} + \mathtt{y}\}$.

We use the above benchmark to evaluate ReGaSol, and the optimized version ReGaSol+ with features discussed in §VIII-A. The benchmark is also run by verification tools Threader [17] and Lazy-Cseq [22] for comparison. The tool Threader applies both CEGAR and R-G technique to prove program safety. Technically, it does so by constructing the unreachability proof of error states via Horn-clause abstraction refinement, starting from the most general abstraction. By contrast, our approach starts at a partially constructed

No	Name	THREADER	LAZY-CSEQ	REGASOL	REGASOL+
P1	peterson	2	**0.92**	1.7	1.22
P2	bakery-simpl	2.16	0.8	0.25	**0.17**
P3	bakery	61.2	7.07	1.8	**1.1**
P4	read-write-lock	0.14	0.59	0.1	**0.11**
P5	szymanski	8.02	**2.8**	3.9	2.9
P6	time-var-mutex	5.68	0.92	0.13	**0.11**
P7	loop1-10-25	0.22	0.71	**0.04**	0.05
P8	loop1-100-25	T/O	36.92	**0.03**	0.05
P9	loop2-10-20	1.14	0.87	**0.02**	0.03
P10	loop2-100-200	T/O	33.92	**0.02**	0.04
P11	loop3-10-20	T/O	1.81	**0.17**	**0.17**
P12	loop3-100-200	T/O	144.41	**0.17**	0.18

TABLE I: Experiment results (in seconds) among tools.

abstractions of R-G relations, and gradually generalizes them via proof refinement. On the other hand, the tool LAZY-CSEQ detects bugs in a concurrent program by under-approximating it as a nondeterministic sequential program with scheduler, which is handled by model checking techniques. As for our framework, while the local proofs are also constructed in a sequential manner, global correctness of the concurrent program is safely persevered, thanks to the R-G principles. As a result, all successfully verified programs by our framework are true-positive, *i.e.* the constructed correctness proofs are sound, as compared to LAZY-CSEQ where false-positive cases — *i.e.* buggy programs being reported safe — are possible.

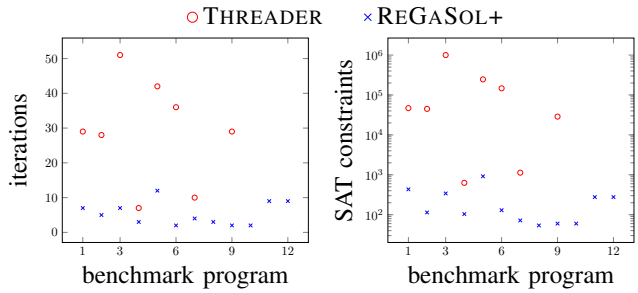
We run the benchmark on a computer with 6-core 3.6GHz processor and 32GB memory. Each program is given equally 300 seconds (s) to be verified before timeout (T/O). Results with best running time are also **highlighted**. As a technicality, the round parameter in LAZY-CSEQ is set to be the number of statements to allow sufficient interleavings.

C. Experiment results

We now discuss the experiment results reported in Table I. While REGASOL+ outperforms REGASOL in category 1 (5.59s *vs.* 7.88s), its performance is slightly worse than REGASOL's in category 2 (0.45s *vs.* 0.54s). We conjecture that the synchronization overheads of parallel proofs accounts for this performance penalty, especially in small programs. In both tools, the performance gaps between P7&P8, P9&P10, and P11&P12 are insignificant, *i.e.* no more than 0.02s for each pair, thanks to the use of loop invariants.

As for the other tools, THREADER and LAZY-CSEQ need 79.2s and 13.1s respectively to verify all programs P1-P6 in category 1. These performances are worse than the performances of REGASOL (5.59s) and REGASOL+ (7.88s). Noticeably, LAZY-CSEQ outperforms our tools in programs P1 and P5. In category 2, THREADER manages to verify programs with small loops, *i.e.* P7 and P9 in 1.36s. However, it times out for the rest. Meanwhile, LAZY-CSEQ verifies programs P7-P12 within 218.64s, which is considerably slower than REGASOL (0.45s) and REGASOL+ (0.54s).

We further investigate the numbers of refinement iterations and SAT constraints generated by THREADER and REGASOL+. The data are plotted in Fig. 7 for comparison. Among programs being successfully verified by both tools, THREADER, on average, requires 29 iterations and generates



(a) Required iterations to terminate (b) Generated SAT constraints

Fig. 7: Comparison between THREADER and REGASOL+.

189,719 SAT constraints for each program. These numbers are 5 iterations and 271 SAT constraints respectively for REGASOL+. Such statistics help to explain the performance gap between the two tools over the benchmark.

To summarize, the experiment results demonstrate that our framework is efficient in verifying concurrent programs, and the strategies in §VIII-A improve the overall performance.

IX. RELATED WORK AND CONCLUSION

Correctness of parallel and concurrent software was first addressed in the Owicki-Gries [30] and shortly later in the Rely-Guarantee [23] proof systems. Whilst the former establishes some concepts that will be used for the verification of concurrent systems such as the notion of stability, parallel compositionality makes R-G a reference for non-automatic [29], [33] and automatic proof systems [13], [8], [16], [18], [40]. The approach based on Horn-clauses and CEGAR-like refinement followed by [15] makes it also comparable with REGASOL.

Our work combines features from both approaches, establishing a set of reasoning rules that allows compositional reasoning on parallel programs, which is automatically carried out using abstraction and refinement. Verification of non-automatic approaches [29], [33] requires to manually construct the rely and guarantee relations, and to infer the properties to be locally verified from the global property. Our approach, in addition to providing a *push-button* solution, automatically constructs the relations modelling the environment and component behaviours, also calculating adequate local preconditions that satisfy the conditions for parallel compositionality.

When comparing with the automatic proof systems in [13], [15], [16], [18], [40] for the verification of concurrent systems, our work is not based on state transition systems but on predicate transformations. As a consequence, our approach has a better performance on systems with a large state space, which can be easily observed on the analysis of programs with loops having a high number of iterations. Although the current setting requires us to manually provide an initial loop invariant, this step can be easily automated with the help of existing invariant inference tools such as [26]. While the work in [13] is incomplete, the addition of a rule for auxiliary variables makes our work complete. The abstraction

performed in [8] may be too precise declaring local variables as global to keep completeness, but our work calculates only the necessary over-approximation to guarantee stability of the predicates on the current refinement iteration to guarantee correctness. An additional side effect of our approach is that it generates a certification of the correctness of the parallel programs. The works in [16], [15] both use Horn-clauses and CEGAR-like refinement. While [16] translate concurrent C programs into a transition system that is evaluated by their refinement-based verification engine, the work in [15] provides a general approach taking as input a transition system, serving as backend for other tools. Hence, the verification of concurrent programs in [15] requires to translate the program to be verified into a set of transitions as a constraints, and also to provide constraints as proof rules modelling the semantics for concurrency.

In conclusion, we proposed a novel framework to verify concurrent programs based on R-G technique, by automatically constructing the Hoare-style proof and inferring the valid R-G relations through iterative refinement. The experiments demonstrate that our prototypes are capable of verifying practical concurrent programs efficiently. For future work, we plan to extend the framework to reason about programs with data structures such as lists and arrays.

Acknowledgment. This work is supported by the Ministry of Education, Singapore under its Tier-2 Project (Award No. MOE2018-T2-1-068) and partially supported by the National Satellite of Excellence in Trustworthy Software Systems (Award No. NRF2018NCR-NSOE003), and award NRF Investigatorship NRFI06-2020-0022, funded by NRF Singapore under National Cyber-security R&D (NCR) programme.

REFERENCES

[1] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. 2014.

[2] Yoah Bar-David and Gadi Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Distributed Computing*, pages 136–150, 2003.

[3] Richard Bornat. Proving pointer programs in hoare logic. In *MPC*, pages 102–126, 2000.

[4] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, Apr 2007.

[5] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI*, pages 270–281, 2014.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr 1986.

[7] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[8] Ariel Cohen and Kedar S. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 34(2):104–125, Apr 2009.

[9] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Logic. Comput.*, 17(4):807–841, Aug 2007.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[11] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic verification for fine-grained concurrency. In *ESOP*, pages 420–447, 2017.

[12] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, pages 262–277, 2002.

[13] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Model Checking Software*, pages 213–224, 2003.

[14] P. Fonseca, Cheng Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, pages 221–230, 2010.

[15] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[16] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.

[17] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, pages 412–417, 2011.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *CAV*, pages 262–274, 2003.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct 1969.

[20] C. A. R. Hoare. Communicating sequential processes. In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 413–443, 2002.

[21] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536, 2013.

[22] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded c programs via lazy sequentialization. In *CAV*, pages 585–602, 2014.

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

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *SOSP*, pages 207–220, 2009.

[25] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug 1974.

[26] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. Fib: Squeezing loop invariants by interpolation between forward/backward predicate transformers. In *ASE*, pages 793–803, 2017.

[27] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[28] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg, 1995.

[29] Leonor Prensa Nieto. The rely-guarantee method in isabelle/hol. In *ESOP*, pages 348–362, 2003.

[30] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6(4):319–340, Dec 1976.

[31] Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. Verifying concurrent graph algorithms. In *APLAS*, pages 314–334, 2016.

[32] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[33] David Sanán, Yongwang Zhao, Zhe Hou, Fuyuan Zhang, Alwen Tiu, and Yang Liu. Csimpl: A rely-guarantee-based framework for verifying concurrent programs. In *TACAS*, pages 481–498, 2017.

[34] Norbert Schirmer. A verification environment for sequential imperative programs in isabelle/hol. In *LPAR*, pages 398–414, 2005.

[35] Ilya Sergey, Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, pages 333–358, 2015.

[36] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, pages 703–719, 2011.

[37] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *ICS*, pages 621–626, 1988.

[38] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[39] Qiwen Xu, Willem Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, Mar 1997.

[40] Fuyuan Zhang, Yongwang Zhao, David Sanán, Yang Liu, Alwen Tiu, Shang-Wei Lin, and Jun Sun. Compositional reasoning for shared-variable concurrent programs. In *FM*, pages 523–541, 2018.