Rely-Guarantee Reasoning for Automated Bound Analysis of Lock-Free Algorithms

Thomas Pani*†, Georg Weissenbacher*, Florian Zuleger*
*Formal Methods in Systems Engineering Group, TU Wien, Vienna, Austria
† Wolfgang Pauli Institute, Vienna, Austria
Email: {pani,weissenb,zuleger}@forsyte.at

***Abstract*—We present a thread-modular proof method for *complexity and resource bound analysis* of concurrent, shared-memory programs, lifting Jones' rely-guarantee reasoning to assumptions and commitments capable of expressing bounds. We automate reasoning in this logic by reducing bound analysis of concurrent programs to the sequential case. Our work is motivated by its application to *lock-free data structures*, fine-grained concurrent algorithms whose time complexity has to our knowledge not been inferred automatically before.**

I. Introduction

A. Program Complexity and Resource Bound Analysis

Program complexity and resource bounds analysis (bound analysis) aims to statically determine upper bounds on the resource usage of a program as expressions over its inputs. Despite the recent discovery of powerful bound analysis methods for *sequential* imperative programs (e.g., [1], [2], [3], [4], [5], [6], [7]), little work exists on bound analysis for *concurrent, shared-memory* imperative programs (cf. Section VII).

From a practical point of view, bound analysis is an important step towards proving functional correctness criteria of programs in resource-constrained environments: For example, in *real-time systems* intermediary results must be available within certain time bounds, or in *embedded systems* applications must not exceed hard constraints on CPU time, memory consumption, or network bandwidth.

B. Non-blocking Data Structures

We illustrate the necessity of extending bound analysis to concurrent, shared-memory programs on the example of *non-blocking data structures*: Devised to circumvent shortcomings of lock-based concurrency (like deadlocks or priority inversion), they have been adopted widely in engineering practice [8]. For example, the Michael-Scott non-blocking queue [9] is implemented in the Java standard library's `ConcurrentLinkedQueue` class.

Automated techniques have been introduced for proving both *correctness* (e.g., [10], [11], [12], [13]) and *progress* (e.g., [14], [15]) properties of non-blocking data structures. In this work, we focus on the progress property of *lock-freedom*, a liveness property that ensures absence of livelocks: Despite interleaved execution of multiple threads altering the data structure, some thread is guaranteed to complete its operation *eventually*.

From a practical, engineering point of view it is not enough to prove that a data structure operation completes *eventually*. Rather, it needs to make progress using a *bounded, measurable amount of resources*: Petrank et al. [16] formalize and study bounded lock-free progress as *bounded lock-freedom*, and discuss its relevance for practical applications. They describe its verification for a fixed number of threads and a given bound using model checking, but leave finding the bound to the user. Existing approaches for automatically proving progress properties like the ones presented in [14], [15] are limited to eventual progress. To our knowledge, bounded progress guarantees have not been inferred automatically before.

C. Overview

Reasoning about the resource consumption of non-blocking algorithms is an intricate and manually tedious problem. To illustrate this point, consider the following common design pattern for lock-free data structures: A thread aiming to manipulate the data structure starts by taking as many steps as possible without synchronization, preparing its intended update. Then, it attempts to alter the globally visible state by synchronizing on a single word in memory at a time. Interference from other threads may cause this synchronization to fail, and the thread to retry from the beginning. From the viewpoint of a single thread that accesses the data structure:

1) The *amount of interference* by other threads *directly affects its resource consumption*. In general, this means reasoning about an unbounded number of concurrent threads, even to infer resource bounds on a single thread.
2) The *point of interference* may occur at any point in the execution, due to the fine granularity of concurrency.

In this paper, we present an automated bound analysis for concurrent, shared-memory programs to remedy this situation: In particular, our method analyzes the *parameterized* system of N concurrent lock-free data structure client threads. To reason about this infinite family of systems and its interactions, we leverage and extend *rely-guarantee (RG) reasoning* [17]: RG reasoning considers each thread separately, modeling interleaved steps of other threads in an *environment assumption*. However, we will see that classic RG reasoning is too weak to obtain suitable bounds. Therefore, we extend RG reasoning to bound analysis. In the following we outline the major contributions of this paper.

D. Contributions

1) We present the first extension of rely-guarantee specifications to bound analysis (Section III).
2) We formulate inference rules to reason about these extended specifications and instantiate them to derive our method for bound analysis of concurrent programs (Section IV).
   Apart from their specific use case in this work, we believe the inference rules are interesting in their own right, for example in comparison to the reasoning rules for liveness presented in [14] (cf. the discussion in Section VII).
3) We reduce bound analysis of concurrent programs to bound analysis of sequential programs, and obtain an algorithm for rely-guarantee bound analysis (Section V).
4) We implement our algorithm in the tool COACHMAN and apply it to lock-free data structures from the literature. To our knowledge, we are the first to automatically infer runtime complexity for widely studied lock-free data structures such as Treiber's stack [18] or the Michael-Scott queue [9] (Section VI).

II. MOTIVATING EXAMPLE

We start by giving an informal explanation of our method and of the paper's main contributions on a running example.

A. Running Example: Treiber's Stack

Fig. 1 shows the implementation of a lock-free concurrent stack known as *Treiber's stack* [18]. Our input programs are represented as control-flow graphs with edges labeled by guarded commands of the form $g \rhd c$. We omit g if $g = \texttt{true}$. We assume edges are executed atomically, and that programs execute in presence of a garbage collector; the latter prevents the so-called *ABA problem* and is a common assumption in the design of lock-free algorithms [8].

Values stored on the stack do not influence the number of times its operations are executed, thus we abstract them away for readability. The stack is represented by a null-terminated singly-linked list, with the shared variable `T` pointing to the top element. The `push` and `pop` methods may be called concurrently, with synchronization occurring at the guarded commands originating in ℓ_3 for `push` and ℓ_{13} for `pop`. These low-level atomic synchronization commands are usually implemented in hardware, through instructions like *compare-and-swap* (CAS) [8].

The stack operations are implemented as follows: Initially, `T` points to `NULL`. The `push` operation (Fig. 1a)

1) allocates a new list node `n` ($\ell_0 \to \ell_1$)
2) reads the global stack pointer `T` ($\ell_1 \to \ell_2$)
3) updates the newly allocated node's `next` field to the read value of `T` ($\ell_2 \to \ell_3$)
4) atomically: compares the value read in (2) to the actual value of `T`; if equal, `T` is updated to point to `n`, otherwise the operation restarts ($\ell_3 \to \ell_4$ and $\ell_3 \to \ell_1$ respectively).

The `pop` operation (Fig. 1b) proceeds similarly.

B. Problem Statement

Consider a general data structure client $P = \texttt{op1}() \,[]\, \cdots \,[]\, \texttt{opM}()$, where $\texttt{op1}, \ldots, \texttt{opM}$ are the data structure's operations, and $[]$ denotes non-deterministic choice. We compose N concurrent client threads P_1 to P_N accessing the data structure:

$$\|_N P \stackrel{\text{def}}{=} \underbrace{P}_{P_1} \| \cdots \| \underbrace{P}_{P_N}$$

Our goal is to design an automated procedure that automatically infers upper-bounds for all system sizes N on

1) the *thread-specific* resource usage caused by a control-flow edge of a single thread P_1 when executed concurrently with $P_2 \| \cdots \| P_N$, or
2) the *total* resource usage caused by a control-flow edge in total over all threads P_1 to P_N.

Remark (Cost model)**.** To measure the amount of resource usage, bound analyses are usually parameterized by a *cost model* that assigns each operation or instruction a *cost* amounting to the resources consumed. In this paper, we adopt a *uniform cost model* that assigns a constant cost to each control-flow edge. When we speak of the *complexity* of a program, we adopt a specific uniform cost model that assigns cost 1 to each control-flow back edge and cost 0 to all other edges; this reflects the asymptotic time complexity of the program.

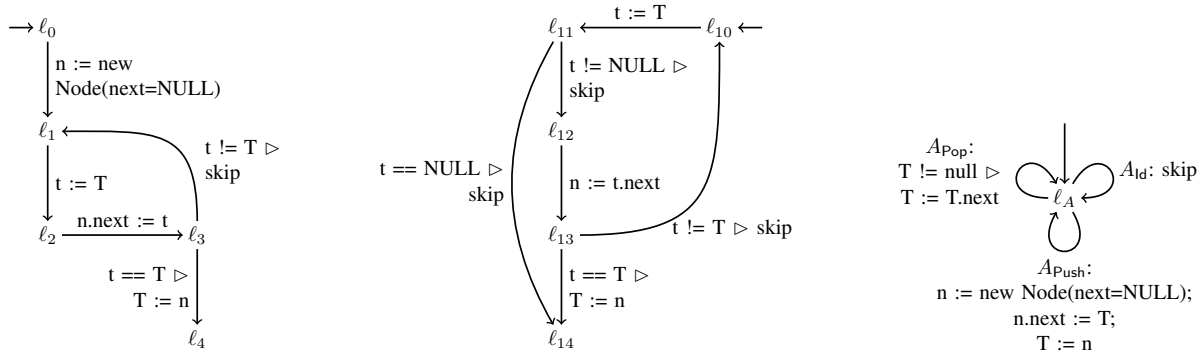
Running example. Consider N concurrent copies $P_1 \| \cdots \| P_N$ of the Treiber stack's client program $\texttt{push}() \,[]\, \texttt{pop}()$, and the `push` operation's control-flow edge $\ell_1 \to \ell_2$. A manual analysis yields a *thread-specific* bound for P_1 telling us that this edge is executed at most N times by P_1: Each time that another thread successfully modifies stack pointer `T`, P_1's copy in `t` may become outdated, causing the test at ℓ_3 to fail ($\texttt{t} \neq \texttt{T}$), and P_1 to restart. After at most $N-1$ iterations, all other threads have finished their operations and returned, and P_1 executes $\ell_1 \to \ell_2 \to \ell_3 \to \ell_4$ without interference.

Similarly, a *total* bound for $P_1 \| \cdots \| P_N$ tells us that edge $\ell_1 \to \ell_2$ is executed at most $N(N+1)/2$ times by all threads P_1 to P_N in total: The first thread to successfully synchronize at ℓ_3 sees no interference and executes $\ell_1 \to \ell_2$ once. The second thread may need to restart once due to the first thread modifying `T`, and executes $\ell_1 \to \ell_2$ at most twice, etc. The last thread to synchronize has the worst-case bound we established as thread-specific bound for P_1: it executes $\ell_1 \to \ell_2$ N times. We obtain $N(N+1)/2$ as closed form for the total bound. In the following, we illustrate how to formalize and automate this reasoning.

C. Environment Abstraction

Client program $\|_N P$ from above is *parameterized* in the number of concurrent threads N. To reason about this infinite family of parallel client programs, we base our analysis on Jones' *rely-guarantee reasoning* [17]. For each thread, RG reasoning over-approximates the following as sets of binary relations over program states (*actions*):

- the thread's effect on the global state (its *guarantee*)



(a) `push()` (b) `pop()` (c) Stateless program $P(\mathcal{A})$ for actions $\mathcal{A} = \{A_{\mathsf{Push}}, A_{\mathsf{Pop}}, A_{\mathsf{Id}}\}$ [19].

Fig. 1: Treiber's lock-free stack [18]. Stack pointer `T` is the sole global variable.

- the effect of all other threads (its *rely*) as the union of those threads' guarantees.

The effect of all other threads (the thread's *environment*) is thus effectively abstracted into a single relation. Crucially, this also abstracts away *how often* each environment action is performed, rendering Jones' RG reasoning unsuitable for concurrent bound analysis.

Running example. The program in Fig. 1c with actions $\mathcal{A} = \{A_{\mathsf{Push}}, A_{\mathsf{Pop}}, A_{\mathsf{Id}}\}$ summarizes the globally visible effect of P_1's environment $P_2 \parallel \cdots \parallel P_N$ for all $N > 0$. In particular, A_{Push} summarizes the effect of an environment thread executing edge $\ell_3 \to \ell_4$ from the point of view[1] of thread P_1, A_{Pop} that of $\ell_{13} \to \ell_{14}$, and A_{Id} that of all other edges. We discuss how to obtain $\mathcal{A}$ in Section V-A.

As is, the actions in $\mathcal{A}$ may be executed infinitely often. Our informal derivation of the bound in Section II-B however, had to determine *how often* other threads could interfere with the reference thread P_1 (altering pointer `T`) to bound its number of loop iterations.

Hence, we lift Jones' RG reasoning to concurrent bound analysis by enriching RG relations with bounds. We emphasize our focus on progress properties in this work: although our framework extends Jones' RG reasoning and can express safety properties, we only use it to reason about bounds; tighter integration is left for future work.

D. Rely-Guarantee Reasoning for Bound Analysis

In particular, relies and guarantees in our setting are maps $\{A_1 \mapsto b_1, \dots\}$ from actions A_i (which are binary relations over program states) to bound expressions b_i. Each relation describes an environment action, and the bound expression describes how often that action may occur on a run of the program.

We present a program logic for thread-modular [20] reasoning about bounds: A judgement in our logic takes the form

$$\mathcal{R}, \mathcal{G} \vdash \{S\}\ P\ \{S'\}$$

where $\{S\}\ P\ \{S'\}$ is a Hoare triple, and $\mathcal{R}, \mathcal{G}$ are a rely and guarantee. Its informal meaning is: For any execution of program P starting in a state from $\{S\}$, and environment interference described by the relations in $\mathcal{R}$ and occurring at most the number of times given by the respective bounds in $\mathcal{R}$, P changes the shared state according to the relations in $\mathcal{G}$ and at most the number of times described by the respective bounds in $\mathcal{G}$. In addition, the execution is safe (does not reach an error state) and if P terminates, its final state is in $\{S'\}$.

Running example. For readability, we focus on the analysis of Treiber's `push` method. The steps for `pop` are similar. To obtain one bound per edge, we split action A_{Id}: `skip` from Fig. 1c into several actions $A_{\mathsf{Id}}^{i,j}$: `skip`, one for each edge $\ell_i \to \ell_j$. For a rely or guarantee $\{A_{\mathsf{Id}}^{0,1} \mapsto b_1, A_{\mathsf{Id}}^{1,2} \mapsto b_2, A_{\mathsf{Id}}^{2,3} \mapsto b_3, A_{\mathsf{Id}}^{3,1} \mapsto b_4, A_{\mathsf{Push}} \mapsto b_5\}$, we fix the order of actions and write $(b_1, b_2, b_3, b_4, b_5)$ for short.

First, our method states the following RG quintuple:

$$\mathcal{R}, \mathcal{G} \vdash \{Inv\}\ P_1\ \{\mathsf{true}\}$$

where Inv is a data structure invariant stated over shared variables in a suitable assertion language (e.g., separation logic), $\mathcal{R} = (\infty, \infty, \infty, \infty, \infty)$, and $\mathcal{G} = (1, \infty, \infty, \infty, 1)$. Despite the unbounded environment $\mathcal{R}$ (which corresponds to Fig. 1c), we can already bound two edges, $\ell_0 \to \ell_1$ and $\ell_3 \to \ell_4$ of P_1, and thus the corresponding actions in $\mathcal{G}$: These edges are not part of a loop and – despite any interference from the environment – can be executed at most once.

We show how to automatically discharge (or rather, discover) such RG quintuples in Section V. Next, we use the bound information obtained in $\mathcal{G}$ to refine the environment $\mathcal{R}$ until a fixed point of the rely is reached.

Running example (continued)**.** We established that thread P_1 can perform actions $A_{\mathsf{Id}}^{0,1}$ and A_{Push} at most once. In our example, all threads are symmetric, thus each of the $N-1$ other threads can execute $A_{\mathsf{Id}}^{0,1}$ and A_{Push} at most once as well. The abstract environment representing these $N-1$ threads can

[1]Note that changes to local variables of $P_2, \dots, P_N$ are not visible to P_1.

thus execute each action $A^{0,1}_{\mathsf{ld}}$ and A_{Push} at most $N-1$ times. We obtain the *refined rely* $\mathcal{R}' = (N-1, \infty, \infty, \infty, N-1)$.

As we have reasoned in Section II-B, once the number of A_{Push} environment actions is bounded, P_1 loops only that number of times. We obtain the *refined guarantee*

$$\mathcal{G}' = (1, N, N, N-1, 1).$$

By the same reasoning as above, we multiply $\mathcal{G}'$ with $(N-1)$ (componentwise) and obtain the refined rely

$$\mathcal{R}'' = (N-1, N(N-1), N(N-1), (N-1)^2, N-1).$$

From $\mathcal{R}''$, we cannot obtain any tighter bounds, i.e., $\mathcal{G}'' = \mathcal{G}'$ is a fixed point, and we report $\mathcal{G}''$ and $\mathcal{G}'' + \mathcal{R}''$ as the thread-specific and total bounds of P_1 and $P_1 \parallel \cdots \parallel P_N$:

edge	thread-specific bound	total bound
$\ell_0 \to \ell_1$	1	N
$\ell_1 \to \ell_2$	N	N^2
$\ell_2 \to \ell_3$	N	N^2
$\ell_3 \to \ell_1$	$N-1$	$N(N-1)$
$\ell_3 \to \ell_4$	1	N

We demonstrate in Section VI that for more complex examples, more than two iterations of the rely-refinement are necessary to bound all edges. We formalize our reasoning in Sections III and IV, explain its automation in Section V, and describe further case studies in Section VI.

III. RG Specifications for Bound Analysis

In this section, we formalize the technique illustrated informally above. We start by stating our program model and formally define the kind of bounds we consider:

A. Program Model

Definition 1 (Program)**.** Let LVar and SVar be finite disjoint sets of typed *local* and *shared program variables*, and let $\mathit{Var} = \mathit{LVar} \cup \mathit{SVar}$. Let Val be a set of *values*. *Program states* $\Sigma\colon \mathit{Var} \to \mathit{Val}$ over Var map variables to values. We write $\sigma|_{\mathit{Var}'}$ where $\mathit{Var}' \subseteq \mathit{Var}$ for the projection of a state $\sigma \in \Sigma$ onto the variables in Var'. Let $GC = \mathit{Guards} \times \mathit{Commands}$ denote the set of *guarded commands* over Var and their effect be defined by $[\![\cdot]\!]\colon GC \to \Sigma \to 2^\Sigma \cup \{\bot\}$ where $\bot$ is a special error state. A *program* P over Var is a directed labeled graph $P = (L, T, \ell_0)$, where L is a finite set of *locations*, $\ell_0 \in L$ is the *initial location*, and $T \subseteq L \times GC \times L$ is a finite set of *transitions*. Let S be a predicate over Var that is evaluated over program states. We overload $[\![\cdot]\!]$ and write $[\![S]\!] \subseteq \Sigma$ for the set of states satisfying S. We represent executions of P as sequences of *steps* $r \in \Sigma \times T \times \Sigma$ and write $\sigma \xrightarrow{t} \sigma'$ for a step (σ, t, σ'). A *run* of P from S is a sequence of steps $\rho = \sigma_0 \xrightarrow{\ell_0, gc_0, \ell_1} \sigma_1 \xrightarrow{\ell_1, gc_1, \ell_2} \dots$ such that $\sigma_0 \in [\![S]\!]$ and for all $i \geq 0$ we have $\sigma_{i+1} \in [\![gc_i]\!](\sigma_i)$.

Definition 2 (Interleaving of Programs)**.** Let $P_i = (L_i, T_i, \ell_{0,i})$ for $i \in \{1, 2\}$ be two programs over $\mathit{Var}_i = \mathit{LVar}_i \cup \mathit{SVar}_i$ such that $\mathit{LVar}_1 \cap \mathit{LVar}_2 = \emptyset$. Their *interleaving* $P_1 \parallel P_2$ over $\mathit{Var}_1 \cup \mathit{Var}_2$ is defined as the program

$$P_1 \parallel P_2 = (L_1 \times L_2, T, (\ell_{0,1}, \ell_{0,2}))$$

where T is given by $((\ell_1, \ell_2), gc, (\ell_1', \ell_2')) \in T$ iff $(\ell_1, gc, \ell_1') \in T_1$ and $\ell_2 = \ell_2'$ or $(\ell_2, gc, \ell_2') \in T_2$ and $\ell_1 = \ell_1'$.

Given a program P over local and shared variables $\mathit{Var} = \mathit{LVar} \cup \mathit{SVar}$, we write $\|_N P = P_1 \parallel \cdots \parallel P_N$ where $N \geq 1$ for the N-times interleaving of program P with itself, where P_i over Var_i is obtained from P by suitably renaming local variables such that $\mathit{LVar}_1 \cap \cdots \cap \mathit{LVar}_N = \emptyset$. Given a predicate S over Var, we write $\bigwedge_N S$ for the conjunction $S_1 \wedge \cdots \wedge S_N$ where S_i over Var_i is obtained by the same renaming.

Definition 3 (Expression)**.** Let Var be a set of integer program variables. We denote by $\mathrm{Expr}(\mathit{Var})$ the set of arithmetic *expressions* over $\mathit{Var} \cup \mathbb{Z} \cup \{\infty\}$. The semantics function $[\![\cdot]\!]\colon \mathrm{Expr}(\mathit{Var}) \to \Sigma \to (\mathbb{Z} \cup \{\infty\})$ evaluates an expression in a given program state. We assume the usual expression semantics; in particular, $a \circ \infty = \infty$ and $a \leq \infty$ for all $a \in \mathbb{Z} \cup \{\infty\}$ and $\circ \in \{+, \times\}$.

Definition 4 (Bound)**.** Let $P = (L, T, \ell_0)$ be a program over variables Var, and let S over Var be a predicate describing P's initial states. Let $t \in T$ be a transition of P, and $\rho = \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots$ be a run of P from S. We use $\#(t, \rho) \in \mathbb{N}_0 \cup \{\infty\}$ to denote the number of times transition t appears on run ρ. An expression $b \in \mathrm{Expr}(\mathit{Var}_\mathbb{Z})$ over integer program variables $\mathit{Var}_\mathbb{Z} \subseteq \mathit{Var}$ is a *bound* for t on ρ iff $\#(t, \rho) \leq [\![b]\!](\sigma_0)$, i.e., if t appears at most b times on ρ.

Given a program $P = (L, T, \ell_0)$ and predicate S over local and shared variables $\mathit{Var} = \mathit{LVar} \cup \mathit{SVar}$, our goal is to compute a function $\mathrm{Bound}\colon T \to \mathrm{Expr}(\mathit{SVar}_\mathbb{Z} \cup \{N\})$, such that for all transitions $t \in T$ and all system sizes $N \geq 1$, $\mathrm{Bound}(t)$ is a bound for t of P_1 on all runs of $\|_N P = P_1 \parallel \cdots \parallel P_N$ from $\bigwedge_N S = S_1 \wedge \cdots \wedge S_N$. That is, Bound gives us the thread-specific bounds for transitions of P_1. In Section IV, we explain how to obtain total bounds on $\|_N P$ from that.

B. Extending Rely-Guarantee Reasoning for Bound Analysis

To analyze the infinite family of programs $\|_N P = P_1 \parallel \cdots \parallel P_N$, we abstract P_1's environment $P_2 \parallel \cdots \parallel P_N$: We define *actions*, which provide an abstract view of transitions by abstracting away local variables and program locations.

Definition 5 (Action, Environment Assertion)**.** Let Σ_S be a set of program states over shared variables SVar. An *action* $A \subseteq \Sigma_S \times \Sigma_S$ over SVar is a binary relation over program states. Let $\mathcal{A} = \{A_1, \dots, A_n\}$ be a finite set of actions. An *environment assertion* $\mathcal{E}_\mathcal{A}\colon \mathcal{A} \to \mathrm{Expr}(\mathit{SVar})$ over $\mathcal{A}$ is a function that maps actions to bound expressions over SVar. We omit $\mathcal{A}$ from $\mathcal{E}_\mathcal{A}$ wherever it is clear from the context.

We use sequences a of actions to describe interference: Intuitively, the bound $\mathcal{E}_\mathcal{A}(A)$ describes how often action $A \in \mathcal{A}$ is permissible in such a sequence. This is captured by the $\models$ relation defined below. We also define operations and relations on environment assertions to compose and compare them.

Definition 6 (Operations and Relations on Environment Assertions)**.** Let $\mathcal{A}$ be a finite set of actions over shared variables SVar, let $A \in \mathcal{A}$ be an action, and let a be a finite or infinite

word over actions $\mathcal{A}$. Let $\mathcal{E}_\mathcal{A}$ and $\mathcal{E}'_\mathcal{A}$ be environment assertions over $\mathcal{A}$. Let $\sigma \subseteq \Sigma_S$ be a program state over $SVar$. We overload $\#(A, a) \in \mathbb{N}_0 \cup \{\infty\}$ to denote the number of times A appears on a and define

$$a \models_\sigma \mathcal{E}_\mathcal{A} \text{ iff } \#(A, a) \leq [\![\mathcal{E}_\mathcal{A}(A)]\!](\sigma) \text{ for all } A \in \mathcal{A}.$$

Let $e \in \mathrm{Expr}(SVar)$ be an expression over $SVar$. For all actions $A \in \mathcal{A}$ we define

$$(e \times \mathcal{E}_\mathcal{A})(A) = e \times \mathcal{E}_\mathcal{A}(A), \text{and}$$
$$(\mathcal{E}_\mathcal{A} + \mathcal{E}'_\mathcal{A})(A) = \mathcal{E}_\mathcal{A}(A) + \mathcal{E}'_\mathcal{A}(A).$$

Further, let S be a predicate over $SVar$. We define

$$\mathcal{E}_\mathcal{A} \subseteq_S \mathcal{E}'_\mathcal{A} \text{ iff } [\![\mathcal{E}_\mathcal{A}(A)]\!](\sigma) \leq [\![\mathcal{E}'_\mathcal{A}(A)]\!](\sigma)$$

for all $A \in \mathcal{A}$ and all $\sigma \in [\![S]\!]$.

C. Trace Semantics of RG Quintuples

We abstract environment threads of interleaved programs with *RG quintuples* of either form

$$\mathcal{R}, \mathcal{G} \vdash \{S\}\, P\, \{S'\} \quad \text{or} \quad \mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}$$

where P and $P_1 \parallel P_2$ are programs, S and S' are predicates such that $[\![S]\!] \subseteq \Sigma$ are *inital program states*, and $[\![S']\!] \subseteq \Sigma$ are *final program states*, and *rely* $\mathcal{R}$ and *guarantees* $\mathcal{G}$ and $\mathcal{G}_1, \mathcal{G}_2$ are environment assertions over a finite set of actions $\mathcal{A}$.

Remark (Notation of environment assertions)**.** Note that the relies and guarantees of a single RG quintuple are defined over the *same* set of actions $\mathcal{A}$; in Section V-A we show how to compute a set $\mathcal{A}$ that over-approximates P (or $P_1 \parallel P_2$) in a prelimiary analysis step. We choose to write relies and guarantees as functions over $\mathcal{A}$ as it simplifies notation throughout the paper. The reader may prefer to think of environment assertions $\{A_1 \mapsto b_1, \dots\}$ as sets of pairs of an action and a bound $\{(A_1, b_1), \dots\}$, in contrast to just sets of actions $\{A_1, \dots\}$ in Jones' RG reasoning.

In particular, $\mathcal{R}$ abstracts P's or $P_1 \parallel P_2$'s environment. The guarantees $\mathcal{G}$ and $(\mathcal{G}_1, \mathcal{G}_2)$ allow us to express both thread-specific and total bounds on interleaved programs: The guarantee $\mathcal{G}$ of quintuple $\mathcal{R}, \mathcal{G} \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}$ contains total bounds for $P_1 \parallel P_2$, while the guarantees $\mathcal{G}_1, \mathcal{G}_2$ of $\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}$ contain the respective thread-specific bounds of threads P_1 and P_2.

We model executions of RG quintuples as *traces*, which abstract runs of the concrete system. In particular, for each run of the concrete system, there exists a corresponding trace of the abstract system. This allows us to over-approximate bounds by considering the traces induced by RG quintuples.

Definition 7 (Trace)**.** Let $P = (L, T, \ell_0)$ be a program of form P_1 or $P_1 \parallel P_2$ and S be a predicate over local and shared variables $Var = LVar \cup SVar$. Let $\mathcal{A}$ be a finite set of actions over $SVar$. We represent executions of P interleaved with environment actions in $\mathcal{A}$ as sequences of *trace transitions* $\delta \in (L \times \Sigma) \times (L \times \Sigma \cup \{\bot\}) \times \{1, 2, \mathsf{e}\} \times \mathcal{A}$, where the first two components define the change in program location and state, the third component defines whether the transition was taken by program P_1 (1), P_2 (2), or the environment (e), and the last component defines which action summarizes the state change. For a trace transition $\delta = ((\ell, \sigma), (\ell', \sigma'), \alpha, A)$, we write $(\ell, \sigma) \xrightarrow{\alpha:A} (\ell', \sigma')$. A *trace* $\tau = (\ell_0, \sigma_0) \xrightarrow{\alpha_1:A_1} (\ell_1, \sigma_1) \xrightarrow{\alpha_2:A_2} \dots$ is a sequence of trace transitions. Let $|\tau| \in \mathbb{N}_0 \cup \{\infty\}$ denote the number of transitions of τ. We define the set $\mathrm{traces}(S, P)$ as the set of traces such that $\sigma_0 \in [\![S]\!]$ and for $0 < i \leq |\tau|$ we have either

- $\alpha_i = 1$, $(\ell_{i-1}, gc, \ell_i) \in T_1$ for some gc, $\sigma_i \in [\![gc]\!](\sigma_{i-1})$, and $(\sigma_{i-1}{\downarrow_{SVar}}, \sigma_i{\downarrow_{SVar}}) \in A_i$, or
- $\alpha_i = 2$, $(\ell_{i-1}, gc, \ell_i) \in T_2$ for some gc, $\sigma_i \in [\![gc]\!](\sigma_{i-1})$, and $(\sigma_{i-1}{\downarrow_{SVar}}, \sigma_i{\downarrow_{SVar}}) \in A_i$, or
- $\alpha_i = \mathsf{e}$, $\ell_{i-1} = \ell_i$, $(\sigma_{i-1}{\downarrow_{SVar}}, \sigma_i{\downarrow_{SVar}}) \in A_i$, and $\sigma_{i-1}{\downarrow_{LVar}} = \sigma_i{\downarrow_{LVar}}$.

The *projection* $\tau{\downarrow_C}$ of a trace $\tau \in \mathrm{traces}(S, P)$ to components $C \subseteq \{1, 2, \mathsf{e}\}$ is the sequence of actions defined as image of τ under the homomorphism that maps $((\ell, \sigma), (\ell', \sigma'), \alpha, A)$ to A if $\alpha \in C$, and otherwise to the empty word.

We now define the meaning of RG quintuples over traces:

Definition 8 (Validity)**.** We define $\mathcal{R}, \mathcal{G} \models \{S\}\, P\, \{S'\}$ iff for all traces $\tau \in \mathrm{traces}(S, P)$ such that τ starts in state $\sigma_0 \in [\![S]\!]$ and $\tau{\downarrow_{\{\mathsf{e}\}}} \models_{\sigma_0} \mathcal{R}$ (τ's environment transitions satisfy the rely):

- if τ is finite and ends in $((\ell, \sigma), (\ell', \sigma'), \alpha, A)$ for some $\ell, \ell', \sigma, \alpha, A$ then $\sigma' \neq \bot$ (the program is safe) and $\sigma' \in [\![S']\!]$ (the program is correct), and
- $\tau{\downarrow_{\{1\}}} \models_{\sigma_0} \mathcal{G}$ (τ's P-transitions satisfy the guarantee $\mathcal{G}$).

Similarly, $\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{S\}\, P_1 \parallel P_2\, \{S'\}$ iff for all $\tau \in \mathrm{traces}(S, P_1 \parallel P_2)$ s.t. τ starts in $\sigma_0 \in [\![S]\!]$ and $\tau{\downarrow_{\{\mathsf{e}\}}} \models_{\sigma_0} \mathcal{R}$:

- if τ is finite and ends in $((\ell, \sigma), (\ell', \sigma'), \alpha, A)$ for some $\ell, \ell', \sigma, \alpha, A$ then $\sigma' \neq \bot$ and $\sigma' \in [\![S']\!]$, and
- $\tau{\downarrow_{\{1\}}} \models_{\sigma_0} \mathcal{G}_1$ and $\tau{\downarrow_{\{2\}}} \models_{\sigma_0} \mathcal{G}_2$.

$$\frac{\begin{array}{c}\mathcal{R} + \mathcal{G}_2, \mathcal{G}_1 \vdash \{S_1\}\, P_1\, \{S'_1\} \\ \mathcal{R} + \mathcal{G}_1, \mathcal{G}_2 \vdash \{S_2\}\, P_2\, \{S'_2\}\end{array}}{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S_1 \wedge S_2\}\, P_1 \parallel P_2\, \{S'_1 \wedge S'_2\}}\ \textsc{Par}$$

$$\frac{\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}}{\mathcal{R}, \mathcal{G}_1 + \mathcal{G}_2 \vdash \{S\}\, P_1 \parallel P_2\, \{S'\}}\ \textsc{Par-Merge}$$

$$\frac{\begin{array}{c}\mathcal{R}_1, \vec{\mathcal{G}}_1 \vdash \{S_1\}\, P\, \{S'_1\} \\ S_2 \Rightarrow S_1 \quad \mathcal{R}_2 \subseteq_{S_2} \mathcal{R}_1 \quad \vec{\mathcal{G}}_1 \subseteq_{S_2} \vec{\mathcal{G}}_2 \quad S'_1 \Rightarrow S'_2\end{array}}{\mathcal{R}_2, \vec{\mathcal{G}}_2 \vdash \{S_2\}\, P\, \{S'_2\}}\ \textsc{Conseq}$$

Fig. 2: Rely/guarantee proof rules for bound analysis. We write $\vec{\mathcal{G}}$ for either $\mathcal{G}$ or $(\mathcal{G}_1, \mathcal{G}_2)$. In the latter case, $\subseteq$ is applied componentwise.

IV. RG Reasoning for Bound Analysis

Similar to classic RG reasoning [17], [21], we propose inference rules to facilitate reasoning about our extended RG quintuples. Our inference rules are shown in Fig. 2:

- PAR interleaves two threads P_1 and P_2 and expresses their thread-specific guarantees in $(\mathcal{G}_1, \mathcal{G}_2)$.
- PAR-MERGE combines thread-specific guarantees $(\mathcal{G}_1, \mathcal{G}_2)$ into a total guarantee $\mathcal{G}_1 + \mathcal{G}_2$.
- CONSEQ is similar to the consequence rule of Hoare logic or RG reasoning: it allows to strengthen precondition and rely, and to weaken postcondition and guarantee(s).

We instantiate these rules to derive the main underlying principle of our bound analysis in the proof of Theorem 2.

Theorem 1 (Soundness)**.** *The rules in Fig. 2 are sound.*

Proof sketch: By Definition 7 (trace semantics of RG quintuples) and induction on the trace length. ∎

In the following, we assume existence of a procedure $\text{SYNTHG}(S, P, \mathcal{R})$ that takes a predicate S, a non-interleaved program P, and a rely $\mathcal{R}$ and computes a guarantee $\mathcal{G}$, such that $\mathcal{R}, \mathcal{G} \models \{S\}\, P\, \{\mathsf{true}\}$ holds. We present such a procedure in Section V.

Our main idea is to use SYNTHG to compute correct-by-construction guarantees for RG quintuple *fragments* of form $\mathcal{R}, ? \vdash \{Inv\}\ P_1\ \{\mathsf{true}\}$. From this, Theorem 2 stated below allows us to infer guarantees for P_1's environment $P_2 \parallel \cdots \parallel P_N$ and thus for $\|_N P = P_1 \parallel \cdots \parallel P_N$.

Theorem 2 (Generalization of Single-Thread Guarantees)**.** *Let P be a program over local and shared variables $Var = LVar \cup SVar$ and let $\|_N P = P_1 \parallel \cdots \parallel P_N$ be its N-times interleaving. Let S be a predicate over $SVar$. Let $\mathcal{A}$ over $SVar$ be the set of actions summarizing the globally visible effect of $\|_N P$ started from S, and let $\mathcal{R}$ and $\mathcal{G}$ be environment assertions over $\mathcal{A}$. Let $\mathbf{0} = (0, \ldots, 0)$ denote the empty environment.*
If

$$(N-1) \times \mathcal{G} \subseteq_S \mathcal{R} \quad \textit{and} \quad \mathcal{R}, \mathcal{G} \models \{S\}\ P_1\ \{\mathsf{true}\}$$

then

$$\mathbf{0}, (\mathcal{G}, (N-1) \times \mathcal{G}) \models \{S\}\ P_1 \parallel (P_2 \parallel \cdots \parallel P_N)\ \{\mathsf{true}\}.$$

I.e., if $(N-1) \times \mathcal{G}$ is smaller than $\mathcal{R}$, and if $\mathcal{R}, \mathcal{G} \models \{S\}\ P_1\ \{\mathsf{true}\}$ holds, then in an empty environment, P_1's environment $P_2 \parallel \cdots \parallel P_N$ executes actions $\mathcal{A}$ no more than $(N-1) \times \mathcal{G}$ times.

Proof sketch: By induction on the number of threads and repeated application of rules CONSEQ*,* PAR-MERGE*, and* PAR. ∎

Running example. Let us return to the task of computing bounds for N threads $\|_N P = P_1 \parallel \cdots \parallel P_N$ concurrently executing Treiber's `push` method. Our method starts from the RG quintuple fragment

$$\mathcal{R}, ? \vdash \{Inv\}\ P_1\ \{\mathsf{true}\} \tag{1}$$

for which it computes a correct-by-construction guarantee: It summarizes P_1's environment $P_2 \parallel \cdots \parallel P_N$ in the rely $\mathcal{R}$. At this point, it cannot safely assume any bounds on $P_2 \parallel \cdots \parallel P_N$, and thus on $\mathcal{R}$. Therefore, it lets $\mathcal{R} = (\infty, \infty, \infty, \infty, \infty)$. Next, our method runs RG bound analysis. As we have argued in Section II-D, this yields $\text{SYNTHG}(Inv, P_1, \mathcal{R}) = (1, \infty, \infty, \infty, 1)$, i.e., we have

$$(\infty, \infty, \infty, \infty, \infty), (1, \infty, \infty, \infty, 1) \models \{Inv\}\ P_1\ \{\mathsf{true}\}. \tag{2}$$

Remark (Role of Theorem 2)**.** At this point, our method cannot establish tighter bounds for P_1 unless it obtains tighter bounds for its environment $P_2 \parallel \cdots \parallel P_N$ and thus $\mathcal{R}$. In Section II-D, we informally argued that if $\mathcal{G} = (1, \infty, \infty, \infty, 1)$ is a guarantee for P_1, then $(N-1) \times \mathcal{G} = (N-1, \infty, \infty, \infty, N-1)$ must be a guarantee for the $N-1$ threads in P_1's environment $P_2 \parallel \cdots \parallel P_N$. Theorem 2 formalizes this principle: It allows us to switch the roles of reference thread and environment, i.e., to infer bounds on $P_2 \parallel \cdots \parallel P_N$ in an environment of P_1 from already computed bounds on P_1 in an environment of $P_2 \parallel \cdots \parallel P_N$.

Running example (continued)**.** Our method applies Theorem 2 to (2) and obtains

$$\begin{aligned}&\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{Inv\}\ P_1 \parallel (P_2 \parallel \cdots \parallel P_N)\ \{\mathsf{true}\} \text{ where}\\ &\quad \mathcal{R} = (0, 0, 0, 0, 0)\\ &\quad \mathcal{G}_1 = (1, \infty, \infty, \infty, 1)\\ &\quad \mathcal{G}_2 = (N-1, \infty, \infty, \infty, N-1)\end{aligned}$$

From the above, we have that $(N-1, \infty, \infty, \infty, N-1)$ is a bound for P_1's environment $P_2 \parallel \cdots \parallel P_N$ when run in parallel with P_1. Going back to the RG quintuple fragment (1), our technique refines the rely $\mathcal{R}$, which models $P_2 \parallel \cdots \parallel P_N$, by letting $\mathcal{R} = (N-1, \infty, \infty, \infty, N-1)$. Again, it runs SYNTHG, which returns $(1, N, N, N-1, 1)$. Thus,

$$\begin{aligned}&\mathcal{R}, \mathcal{G} \models \{Inv\}\ P_1\ \{\mathsf{true}\} \text{ where}\\ &\quad \mathcal{R} = (N-1, \infty, \infty, \infty, N-1)\\ &\quad \mathcal{G} = (1, N, N, N-1, 1)\end{aligned}$$

Another refinement of $\mathcal{R}$ from $\mathcal{G}$ by Theorem 2 and another run of SYNTHG gives

$$\begin{aligned}&\mathcal{R}, \mathcal{G} \models \{Inv\}\ P\ \{\mathsf{true}\} \text{ where}\\ &\quad \mathcal{R} = (N-1, N(N-1), N(N-1), (N-1)^2, N-1)\\ &\quad \mathcal{G} = (1, N, N, N-1, 1)\end{aligned}$$

This time, the guarantee has not improved any further, i.e., our method has reached a fixed point and stops the iteration. Applying Theorem 2 gives

$$\begin{aligned}&\mathcal{R}, (\mathcal{G}_1, \mathcal{G}_2) \models \{Inv\}\ P_1 \parallel (P_2 \parallel \cdots \parallel P_N)\ \{\mathsf{true}\} \text{ where}\\ &\quad \mathcal{R} = (0, 0, 0, 0, 0)\\ &\quad \mathcal{G}_1 = (1, N, N, N-1, 1)\\ &\quad \mathcal{G}_2 = (N-1, N(N-1), N(N-1), (N-1)^2, N-1)\end{aligned}$$

To compute thread-specific bounds for the transitions of P_1, our method may stop here; the bounds can be read off $\mathcal{G}_1$. For example, the second component of $\mathcal{G}_1$ indicates that transition $\ell_1 \to \ell_2$ is executed at most N times. To compute total bounds

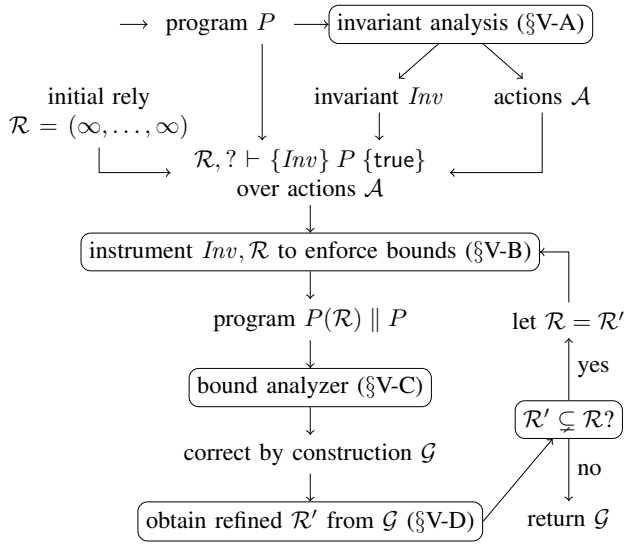


Fig. 3: Overview of our analysis.

for the transitions of the whole interleaved system $P_1 \parallel \cdots \parallel P_N$, our technique applies rule PAR-MERGE, which gives

$$\begin{aligned}&\mathcal{R}, \mathcal{G} \models \{Inv\}\ P_1 \parallel \cdots \parallel P_N\ \{\mathsf{true}\} \text{ where}\\ &\quad \mathcal{R} = (0, 0, 0, 0, 0)\\ &\quad \mathcal{G} = (N, N^2, N^2, (N-1)N, N)\end{aligned}$$

Again, bounds can be read off $\mathcal{G}$, for example the fourth component indicates that the back edge $\ell_3 \to \ell_1$ is executed at most $(N-1) \times N$ times by all N threads in total.

V. AUTOMATION

In this section, we describe the rely-guarantee bound algorithm previously presented on an example; Fig. 3 gives an overview. The algorithm builds on two main insights:

- We reduce RG bound analysis to sequential bound analysis. This allows us to implement procedure SYNTHG.
- We utilize Theorem 2 to iteratively refine bounds on environment assertions until a fixed point is reached.

A. Invariant Analysis

Given a program $P = (L, T, \ell_0)$, our algorithm starts with an invariant analysis to discover a data structure invariant Inv, a set of actions $\mathcal{A}$, and a map $\mathrm{EffectOf}\colon \mathcal{A} \to 2^T$ that indicates which transitions a given action abstracts. In our running example, each action corresponds to one transition, but in general coarser actions may be chosen. Many methods for obtaining these have been described in the literature (e.g., [22], [23], [24], [25], [19]). We use the tool TMREXP [19] as an off-the-shelf solver, which allows us to obtain $\mathcal{A}$ as a stateless program as shown in Fig. 1c.

This allows us to state the RG quintuple fragment

$$\mathcal{R}, ? \vdash \{Inv\}\ P_1\ \{\mathsf{true}\} \tag{3}$$

over $\mathcal{A}$ where $\mathcal{R} = (\infty, \ldots, \infty)$ and the guarantee is unknown. $\mathcal{R}$ soundly over-approximates P_1's environment $P_2 \parallel \cdots \parallel P_N$. We obtain a correct-by-construction guarantee from the thread-modular bound analysis described below.

B. Instrumentation

Given the RG quintuple fragment (3), our method first constructs the program $P(\mathcal{R})$: Let $\mathcal{A} = \{A_1, \ldots, A_m\}$. It starts from the stateless program

$$\textbf{while } (\mathtt{true}) \textbf{ do } A_1 \,[]\, \cdots \,[]\, A_m \textbf{ done}$$

and instruments it with counter variables ξ_A to enforce the bounds in $\mathcal{R}$:

Let $P(\mathcal{R}) = (\{\ell\}, T, \ell)$ be the program over variables $\{\xi_{A_1}, \ldots, \xi_{A_m}\}$ with initial states $[\![g_0]\!]$ where

$$T = \{(\ell, gc_A, \ell) \mid A \in \mathcal{A}\}$$

$$gc_A = \begin{cases}\xi_A > 0 \rhd \{A;\ \xi_{\mathtt{A}} := \xi_{\mathtt{A}} - 1\} & \text{if } \mathcal{R}(A) \neq \infty\\ \mathsf{true} \quad \rhd \{A\} & \text{otherwise}\end{cases}$$

$$g_0 = \bigwedge_{A \in \mathcal{A}} \begin{cases}\xi_A = \mathcal{R}(A) & \text{if } \mathcal{R}(A) \neq \infty\\ \mathsf{true} & \text{otherwise}\end{cases}$$

Proposition 1. *There exists an isomorphism between runs of $P_1 \parallel P(\mathcal{R})$ from $Inv \wedge g_0$, and traces $\{\tau \in \mathrm{traces}(Inv, P_1) \mid \tau$ starts in σ and $\tau|_{\{\mathtt{e}\}} \models_\sigma \mathcal{R}\}$, such that isomorphic runs and traces have the same length n, and for all positions $0 \leq i \leq n$ their location and state components are equal up to the instrumentation location and variables ℓ and ξ_A of $P(\mathcal{R})$.*

C. Bound Analysis

Our algorithm translates the interleaved heap-manipulating program $\hat{P} = P_1 \parallel P(\mathcal{R})$ and predicate $Inv \wedge g_0$ into an equivalent (bisimilar) integer program and predicate using the technique of [13] (alternatively one could directly compute bounds on the heap-manipulating program $\hat{P}$ using techniques such as described in [26], [27], [28]). From now on, let $\hat{P}$ and $Inv \wedge g_0$ refer to these translations.

Note that $\hat{P}$ is a sequential integer program that can be fed to an off-the-shelf sequential bound analyzer. Let $\hat{T}$ denote the transitions of $\hat{P}$. Our method runs the sequential bound analyzer on $\hat{P}$, which computes a function $\mathrm{SeqBound}\colon \hat{T} \to \mathrm{Expr}(Var_{\mathbb{Z}} \cup \{N\})$, such that for all $t \in \hat{T}$ and all $N \geq 1$, $\mathrm{SeqBound}(t)$ is a bound for t on all runs of $\hat{P}$ from $Inv \wedge g_0$.

Then, our technique maps bounds obtained on transitions of $\hat{P}$ back to the corresponding transitions of P_1 in $\hat{P} = P_1 \parallel P(\mathcal{R})$, which allows it to compute the desired guarantee for P_1: Letting

$$\mathcal{G}(A) = \sum_{t \in \mathrm{EffectOf}(A)} \mathrm{SeqBound}(t)$$

for all $A \in \mathcal{A}$ gives a correct-by-construction guarantee $\mathcal{G}$ for $\mathcal{R}, ? \vdash \{Inv\} P_1 \{\mathsf{true}\}$, i.e., we have $\mathcal{R}, \mathcal{G} \models \{Inv\} P_1 \{\mathsf{true}\}$.

D. Bound Refinement

Our algorithm then uses Theorem 2 to refine the rely of P_1 and checks if the computation has reached a fixed point yet. Let $\mathcal{R}'(A) = (N-1) \times \mathcal{G}(A)$ for all $A \in \mathcal{A}$.

1) If $\mathcal{R}' \subsetneq \mathcal{R}$, by Theorem 2 $\mathcal{R}'$ is a valid bound for $P_2 \parallel \cdots \parallel P_N$. Our algorithm iterates the computation of bounds for $\mathcal{R}', ? \vdash \{Inv\}\ P_1\ \{\texttt{true}\}$ starting from Section V-B.
2) If $\mathcal{R}' = \mathcal{R}$, the algorithm has reached a fixed point and reports the results of the analysis:
   a) For thread-specific bounds of P_1, return $\mathcal{G}$.
   b) For total bounds of $P_1 \parallel \cdots \parallel P_N$, apply Theorem 2 to get a guarantee for $P_2 \parallel \cdots \parallel P_N$, and use rule PAR-MERGE to sum up the guarantees of P_1 and $P_2 \parallel \cdots \parallel P_N$.
3) $\mathcal{R}' \not\subseteq \mathcal{R}$ can be avoided by implementing a sequential bound analyzer that is deterministic and monotonic in the sense that it always finds the same or smaller bounds on programs with further restricted transition relations.

VI. Case Studies

We have implemented the algorithm of Section V in our tool COACHMAN [29] and tested it on three well-known lock-free data structures from the literature: Treiber's stack [18], the Michael-Scott queue [9], and the DGLM queue [30]. For the sequential bound analyzer, we have implemented an algorithm similar to the one described in [7]; its implementation is available online [29].

For each data structure, our tool constructs a general client program $P = \texttt{op1}()\ [] \cdots [] \ \texttt{opM}()$, and analyzes its N-times interleaving $\|_N P = P_1 \parallel \cdots \parallel P_N$ for thread-specific bounds of a single thread P_i and total bounds of $P_1 \parallel \cdots \parallel P_N$ as described in Section II. For brevity, we only report complexity bounds here. All performance results were obtained on a single core of a 2.0GHz Intel Core i7 processor.

1) Treiber's stack [18]: We thoroughly discussed Treiber's stack in our running example (Section II). Our tool takes **2 iterations** to obtain the stack's thread-specific linear asymptotic complexity $\mathbf{O(N)}$ of a single thread P_i, and the total quadratic complexity $\mathbf{O(N^2)}$ of $P_1 \parallel \cdots \parallel P_N$ in **3 minutes**[2].

2) Michael-Scott queue [9]: This lock-free queue has, e.g., been implemented in the `ConcurrentLinkedQueue` class of the Java standard library. In contrast to Treiber's stack, the transitions of the Michael-Scott queue cannot be bounded with just a single refinement operation: It synchronizes via two CAS operations, the first one breaking/looping as in Treiber's stack, the second one located on a back edge of the main loop. Thus our algorithm cannot immediately bound the action corresponding to the second CAS. Rather, it first bounds the first CAS' action, refines and bounds the second CAS' action, and after a final refinement bounds all other edges. Our tool takes **3 iterations** to obtain the queue's thread-specific linear asymptotic complexity $\mathbf{O(N)}$, and the total quadratic complexity $\mathbf{O(N^2)}$ in **148 minutes**.

3) DGLM queue [30]: The DGLM queue is a recent, optimized version of the Michael-Scott queue. Similar to the Michael-Scott queue, our tool takes **3 iterations** to obtain the queue's thread-specific linear asymptotic complexity $\mathbf{O(N)}$, and the total quadratic complexity $\mathbf{O(N^2)}$ in **77 minutes**.

Remark (Discussion of algorithm runtime)**.** The increased runtime on the queue case studies compared to Treiber's stack is due to to their larger program LTS and doubled number of environment actions. In particular, the counter automaton produced by [13] for Treiber's stack has 531 vertices and 2,072 edges, while for the MS queue we obtain 6,165 vertices and 37,402 edges.

The runtime speedup on the DGLM queue compared to the MS queue is explained by its optimized `deq` method: Its LTS has 2 instead of 4 back edges, which drastically reduces the time spent in bound analysis.

VII. Related Work

Albert et.al. [31] describe a RG bound analysis for actor-based concurrency. They use heuristics to guess an unsound guarantee and justify it by proving that all environment actions not captured by the guarantee occur only finitely often. We note that the approach of [31] leaves environment actions not captured by the guarantee completely unconstrained, i.e., they may change the program state arbitrarily, leading to coarser than necessary bounds. In contrast, our approach includes all environment actions, recognizes that actions occurring boundedly often already carry ranking information, and leaves their handling to the sequential bound analyzer.

More closely related to our work, Gotsman et al. [14] present a general framework for expressing liveness properties in RG specifications and apply it to prove termination/unbounded lock-freedom. They give rely and guarantee as words over actions, and instantiate it for properties stating that a set of actions does not occur infinitely often. They automatically discharge such properties in an iterative proof search over the powerset of actions. Our approach differs in various aspects: First, while our RG quintuples may be formulated as words over actions, the instantiation in [14] is suitable only for termination, but too weak for bound analysis. Second, the focus on liveness properties leads to more complicated proof rules in [14], which have to account for the fact that naive circular reasoning about liveness properties is unsound [32], [33], [14]. In contrast, all sequences of actions expressable by our environment assertions are safety-closed, allowing us to use the full power of RG-style circular arguments in the premises of our reasoning rules. Finally, we obtain bounds for all actions at once in a refinement step by reduction to sequential bound analysis, rather than iteratively querying a termination prover whether a particular action occurs only finitely often.

[2] A further optimization of the bound algorithm only applicable to this case study allows us to speed up the runtime to 47 seconds.

VIII. Conclusion

We have presented the first extension of rely-guarantee reasoning to bound analysis, and automated bound analysis of concurrent programs by a reduction to sequential bound analysis. In addition, we have for the first time automatically inferred bounds for three widely-studied lock-free data structures.

IX. Future Work

While lock-freedom guarantees absence of live-locks, it does not guarantee starvation-freedom: If a thread's environment interferes infinitely often, the thread may loop forever. *Wait-freedom* is a stronger progress property that guarantees that each individual thread makes progress. Its implementation exposes global variables per thread; handling this is an interesting problem for the future.

While our framework extends Jones' RG reasoning, we have only given inference rules for parallel composition and a consequence rule and have left the concrete programming language and corresponding rules abstract. Our only requirement regarding safety is that the environment actions obtained in Section V-A over-approximate any thread's effect on the global state. Giving a full set of rules and exploring a tighter integretion between safety and (bounded) liveness properties is left for future work.

Acknowledgment

We would like to thank Erez Petrank and Ana Sokolova for comments and discussions on this work.

References

[1] S. Gulwani and F. Zuleger, "The reachability-bound problem," in *PLDI*, 2010, pp. 292–304.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Closed-form upper bounds in static cost analysis," *J. Autom. Reasoning*, vol. 46, no. 2, pp. 161–203, 2011.

[3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord, "Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs," in *SAS*, ser. Lecture Notes in Computer Science, vol. 6337. Springer, 2010, pp. 117–133.

[4] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Analyzing runtime and size complexity of integer programs," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 4, pp. 13:1–13:50, 2016.

[5] A. Flores-Montoya and R. Hähnle, "Resource analysis of complex programs with cost equations," in *APLAS*, ser. Lecture Notes in Computer Science, vol. 8858. Springer, 2014, pp. 275–295.

[6] Q. Carbonneaux, J. Hoffmann, T. W. Reps, and Z. Shao, "Automated resource analysis with coq proof objects," in *CAV*, 2017, pp. 64–85.

[7] M. Sinn, F. Zuleger, and H. Veith, "Complexity and resource bound analysis of imperative programs using difference constraints," *J. Autom. Reasoning*, vol. 59, no. 1, pp. 3–45, 2017.

[8] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[9] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*. ACM, 1996, pp. 267–275.

[10] V. Vafeiadis, "Automatically proving linearizability," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 450–464.

[11] S. Chakraborty, T. A. Henzinger, A. Sezgin, and V. Vafeiadis, "Aspect-oriented linearizability proofs," *Logical Methods in Computer Science*, vol. 11, no. 1, 2015.

[12] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine, "An integrated specification and verification technique for highly concurrent data structures," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 7795. Springer, 2013, pp. 324–338.

[13] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar, "Programs with lists are counter automata," *Formal Methods in System Design*, vol. 38, no. 2, pp. 158–192, 2011.

[14] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis, "Proving that non-blocking algorithms don't block," in *POPL*. ACM, 2009, pp. 16–28.

[15] X. Jia, W. Li, and V. Vafeiadis, "Proving lock-freedom easily and automatically," in *CPP*. ACM, 2015, pp. 119–127.

[16] E. Petrank, M. Musuvathi, and B. Steensgaard, "Progress guarantee for parallel programs via bounded lock-freedom," in *PLDI*. ACM, 2009, pp. 144–154.

[17] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, 1983, pp. 321–332.

[18] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, 1986.

[19] L. Holík, R. Meyer, T. Vojnar, and S. Wolff, "Effect summaries for thread-modular analysis - sound analysis despite an unsound heuristic," in *SAS*, ser. Lecture Notes in Computer Science, vol. 10422. Springer, 2017, pp. 169–191.

[20] C. Flanagan, S. N. Freund, and S. Qadeer, "Thread-modular verification for shared-memory programs," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 2305. Springer, 2002, pp. 262–277.

[21] Q. Xu, W. P. de Roever, and J. He, "The rely-guarantee method for verifying shared variable concurrent programs," *Formal Asp. Comput.*, vol. 9, no. 2, pp. 149–174, 1997.

[22] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv, "Thread-modular shape analysis," in *PLDI*. ACM, 2007, pp. 266–277.

[23] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv, "Thread quantification for concurrent shape analysis," in *CAV*, ser. Lecture Notes in Computer Science, vol. 5123. Springer, 2008, pp. 399–413.

[24] V. Vafeiadis, "Rgsep action inference," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 5944. Springer, 2010, pp. 345–361.

[25] A. Miné, "Static analysis of run-time errors in embedded critical parallel C programs," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 6602. Springer, 2011, pp. 398–418.

[26] T. Fiedor, L. Holík, A. Rogalewicz, M. Sinn, T. Vojnar, and F. Zuleger, "From shapes to amortized complexity," in *VMCAI*, 2018, pp. 205–225.

[27] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann, "Automatically proving termination and memory safety for programs with pointer arithmetic," *J. Autom. Reasoning*, vol. 58, no. 1, pp. 33–65, 2017.

[28] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla, "Automatic inference of resource consumption bounds," in *LPAR-18*, 2012, pp. 1–11.

[29] "Coachman," https://github.com/thpani/coachman.

[30] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm," in *FORTE*, ser. Lecture Notes in Computer Science, vol. 3235. Springer, 2004, pp. 97–114.

[31] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin, "Rely-guarantee termination and cost analyses of loops with concurrent interleavings," *J. Autom. Reasoning*, vol. 59, no. 1, pp. 47–85, 2017.

[32] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 507–534, 1995.

[33] K. L. McMillan, "Circular compositional reasoning about liveness," in *CHARME*, ser. Lecture Notes in Computer Science, vol. 1703. Springer, 1999, pp. 342–345.