Automated and modular refinement reasoning for concurrent programs

Chris Hawblitzel, Microsoft
Erez Petrank, Technion
Shaz Qadeer, Microsoft
Serdar Tasiran, Koç University

Abstract

We present CIVL, a language and verifier for concurrent programs based on automated and modular refinement reasoning. CIVL supports reasoning about a concurrent program at many levels of abstraction. Atomic actions in a high-level description are refined to fine-grain and optimized lower-level implementations. A novel combination of automata theoretic and logic-based checks is used to verify refinement. Modular specifications and proof annotations, such as location invariants and procedure pre- and post-conditions, are specified separately, independently at each level in terms of the variables visible at that level. We have implemented CIVL as an extension to the BOOGIE language and verifier. We have used CIVL to refine a realistic concurrent garbage collection algorithm from a simple high-level specification down to a highly-concurrent implementation described in terms of individual memory accesses.

1 Introduction

We present a technique for verifying a refinement relation between two concurrent, shared-memory multithreaded programs. Our work is inspired by stepwise refinement [53], where a high-level description is systematically refined, potentially via several intermediate descriptions, down to a detailed implementation. Refinement checking is a classical problem in verification and has been investigated in many contexts, including hardware verification [15] and verification of cache-coherence protocols and distributed algorithms [39]. In the realm of sequential software, notable successes using the refinement approach include the work of Abrial et al. [2] and the proof of full functional correctness of the seL4 microkernel [37]. This paper presents the first general and automated proof system for refinement verification of shared-memory multithreaded software.

We present our verification approach in the context of CIVL, an idealized concurrent programming language. In CIVL, a program is described as a collection of procedures whose implementation can use the standard features such as assignment, conditionals, loops, procedure calls, and thread creation. Each procedure accesses shared global variables only through invocations of atomic actions. A subset of the atomic actions may be refined by new procedures and a new program is obtained by replacing the invocation of an atomic action by a call to the corresponding procedure refining the action. Several layers of refinement may be performed until all atomic actions in the final program are directly implementable primitives. Unlike classical program verifiers based on Floyd-Hoare reasoning [24, 34] that manipulate a program and annotations, the CIVL verifier manipulates multiple operational descriptions of a program, i.e., several layers of refinement are specified and verified at once.

To prove refinement in CIVL, a simulation relation between a program and its abstraction is inferred from checks on each procedure, thus decomposing a whole-program refinement problem into per-procedure verification obligations. The computation inside each such procedure is partitioned into "steps" such that one step behaves like the atomic specification and all other steps have no effect on the visible state. This partitioning follows the syntactic structure of the code in a way similar in spirit to Floyd-Hoare reasoning. To be able to express the per-procedure verification obligation in terms of a collection of per-step verification tasks, the CIVL verifier needs to address two issues. First, the notion of a "step" in the implementation must be defined. The definition of a step can deeply affect the number of checks that need to be performed and the number of user annotations. Second, it is typically not possible to show the correctness of a step from an arbitrary state. A precondition for the step in terms of shared variables must be supplied by the programmer and mechanically checked by the verifier.

To address the first problem, CIVL lets the programmer define the granularity of a step, allowing the user to specify a semantics with larger atomic actions. A *cooperative* semantics for the program is explicitly introduced by the programmer through the use of a new primitive *yield* statement; in this semantics a thread can be scheduled out only when it is about to execute a yield statement. The *preemptive* semantics of the program is sequentially consistent execution; all threads are imagined to execute on a single processor and preemption, which causes a thread to be scheduled out and a nondeterministically chosen thread to be scheduled in, may occur before any instruction.[1] Given a program P, CIVL verifies that the safety of the cooperative semantics of P implies the safety of the preemptive semantics of P. This verification is done by computing an automata-theoretic simulation check [30] on an abstraction of P in which each atomic action of P is represented by only its mover type [43, 21]. The mover types themselves are verified separately and automatically using an automated theorem prover [10].

To address the second problem that refinement verification for each step requires invariants about the program execution, CIVL allows the programmer to specify location invariants, attached either to a yield statement or to a procedure as its pre- or postcondition. Each location invariant must be correct for all executions and must continue to hold in spite of potential interference from concurrently executing threads. We build upon classical work [48, 36] on reasoning about non-interference with two distinct innovations. First, we do not require the annotations to be strong enough to prove program correctness but only strong enough to provide the context for refinement checking. Program correctness is established via a sequence of refinement layers from an abstract program that cannot fail. Second, to establish a postcondition of a procedure, we do not need to propagate a precondition through all the yield annotations in the procedure body. The correctness of an atomic action specification gives us a simple frame rule—the precondition only needs to be propagated across the atomic action specification. CIVL further simplifies the manual annotations required for logical non-interference checking by providing a linear type system [52] that enables logical encoding of thread identifiers, permissions [8], and disjoint memory [38].

Finally, CIVL provides a simple module system. Modules can be verified separately, in parallel or at different times, since the module system soundly does away with checks that pertain to cross-module interactions. This feature is significant since commutativity checks and non-interference checks for location invariants are quadratic, whole program checks involving all pairs of yield locations and atomic blocks, or all pairs of actions from a program. Using the module system, the number of checks is reduced; they become quadratic in the number of yields and atomic blocks within each module rather than the entire program.

We have implemented CIVL as a conservative extension of the BOOGIE verifier. We have used it to verify a collection of microbenchmarks and benchmarks from the literature [7, 17, 18, 19, 23, 33]. The most challenging case study with CIVL was carried out concurrently with CIVL's development and served as a design driver. We verified a concurrent garbage collector, through six layers of refinement, down to atomic actions corresponding to individual memory accesses. The level of granularity of the lowest-level implementation distinguishes this verification effort from previous attempts in the literature.

In conclusion, CIVL is the first automated verifier for shared-memory multithreaded programs that provides the capability to establish a multi-layered refinement proof. This novel capability is enabled by

[1]In this paper, we focus our attention on sequential consistency and leave consideration of weak memory models to future work.

two important innovations in core verification techniques for reducing the complexity of invariants supplied by the programmer and the verification conditions solved by the prover.

- Reasoning about preemptive semantics is replaced by simpler reasoning about cooperative operational semantics by exploiting automata-theoretic simulation checking. We are not aware of any other verifier that combines automata-based and logic-based reasoning in this style.

- A linear type system establishes invariants about disjointness of permission sets associated with values contained in program variables. These invariants, communicated to the prover as free assumptions, significantly reduce the overhead of program annotations. We are not aware of any other verifier that combines type-based and logic-based reasoning in this style.

2 Overview

We present an overview of our approach to refinement on the program in Figure 1, a simplified version of the write barrier in a concurrent garbage collector (GC). In a concurrent GC, a color (either `WHITE`, `GRAY`, or `BLACK`) is associated with each object on the heap. Before writing to an address `addr`, a mutator executes a write barrier. It checks `addr` has color `WHITE` and sets it to `GRAY`, indicating that the object at `addr` and objects reachable from it should not be garbage collected.

Procedure `WB` implements the write barrier. To simplify exposition, we consider a single object whose color is stored in the shared variable `Color`. `WB` first reads `Color` without holding a lock, to avoid, when possible, the cost of acquiring and releasing a lock for each address encountered by a mutator. If `Color` is `WHITE`, it calls the more expensive procedure `WBSlow` to re-examines and possibly update `Color` while holding the lock. CIVL simplifies reasoning about `WB` and `WBSlow` by allowing us to compactly express their specification as the following atomic action:

```
var Color: int; // WHITE=1, GRAY=2, BLACK=3
procedure WB(linear tid:Tid)
atomic [if (Color == WHITE) Color := GRAY];
{
  var cNoLock:int;
  cNoLock := GetColorNoLock(tid);
  yield Color >= cNoLock;
  if (cNoLock == WHITE)
    call WBSlow(tid);
}
procedure WBSlow(linear tid:Tid)
atomic [if (Color == WHITE) Color := GRAY];
{
  var cLock:int;
  call AcquireLock(tid);
  cLock := GetColorLocked(tid);
  if (cLock == WHITE)
    call SetColorLocked(tid, GRAY);
  call ReleaseLock(tid);
}
procedure GetColorNoLock(linear tid:Tid)
  returns (cl:int) atomic [...];
procedure AcquireLock(linear tid:Tid)
  right [...];
procedure ReleaseLock(linear tid:Tid)
  left [...];
procedure GetColorLocked(linear tid:Tid)
  returns (cl:int) both [...];
procedure SetColorLocked(linear tid:Tid,
  cl: int) atomic [...];
```

Figure 1: Write barrier

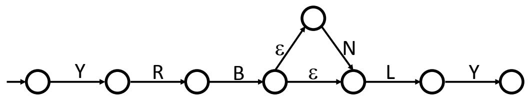


Figure 2: Abstraction of `WBSlow`

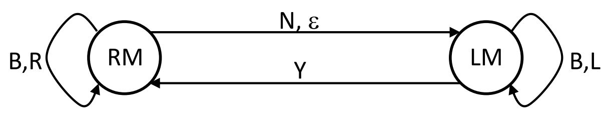


Figure 3: Yield sufficiency automaton (YSA)

```
[if (Color == WHITE) Color := GRAY]
```

This specification indicates that regardless of the different implementations of `WB` and `WBSlow` and regardless of how the environment interferes with their execution, to their respective callers it appears as if they atomically execute the above code.

Per-procedure simulation, non-interference via invariants. The correctness of `WB` is not obvious, and its verification requires a combination of techniques as discussed next. Consider the following potential scenario. `WB`, not holding a lock, reads `Color` and sets `cNoLock` to `GRAY` and then yields. Another thread sets `Color` to `WHITE`. `WB` resumes, but does nothing and exits, because the procedure-local variable `cNoLock` is `GRAY`. In this scenario, the atomic action specification of `WB` would not be satisfied. However, in the GC this scenario is not possible. The yield predicate (location invariant) expresses the fact that other threads in the environment of the thread running `WB` can only modify `Color` to a higher (darker) value – CIVL verifies the correctness of this location invariant. Using this location invariant, CIVL then verifies atomicity refinement for `WB` by verifying the existence of a particular simulation-relation: for every control path through `WB`, exactly one `yield`-to-`yield` execution fragment is simulated by the atomic action specification and other fragments do not modify global state. This proof requires both correct modeling of environment interference in the yield predicate and the atomic action specification for the called procedure `WBSlow`. The CIVL verifier automatically computes a logical verification condition capturing these proof obligations from the body and specification of `WB`.

Just as the verification of `WB` builds on the specification of `WBSlow`, the verification of `WBSlow` builds on other refinement proofs (not shown) of the procedures called in `WBSlow`; these procedures are shown at the bottom of the figure. This example shows only one procedure at this layer. In programs with many procedures with atomic specifications at each layer, CIVL combines the per-procedure refinement proofs soundly into a whole-program refinement proof.

Reduction, preemptive vs collaborative semantics. The verification of `WBSlow` highlights reduction-related features in CIVL. Refinement checking is performed on cooperative semantics in which a `yield`-to-`yield` execution fragment of code is executed atomically. However, in a real execution, control can switch between threads at any point in the code. A naive modeling of a real execution would put a yield statement before every instruction in the code. The absence of a yield statement before every instruction is justified by reasoning about mover types [21]. The procedures called in `WBSlow` have the mover types claimed in their declarations and verified by CIVL. For example, the mover type of `Acquirelock` is `right` which indicates that it commutes later in time against concurrently executing environment actions. These mover types are checked by constructing verification conditions from each pair of atomic actions.

The use of movers is entirely optional in CIVL, but very beneficial in our experience. One can avoid mover and commutativity reasoning, simply annotating atomic action specifications with the mover type `atomic`. However, without mover reasoning, a yield statement and an accompanying predicate must be inserted before every invocation of an atomic action. It has been demonstrated in work in the literature that mover reasoning simplifies many proofs [17]. In our experience with CIVL, we have observed that using more yield predicates rather than mover reasoning can make proofs difficult in two ways. First, the annotation burden goes up because sophisticated ghost variables may need to be introduced in the program semantics.[2] Second, the computational cost of the pairwise mover reasoning is replaced by

[2]Well-scoped location invariants that cannot refer to the state of other threads are known to be incomplete, both in theory and in practice.

the cost of pairwise non-interference checks between yield predicates and concurrently executing atomic actions. CIVL does not force the use of mover reasoning but provides automation for this important verification feature and its use in conjunction with other techniques illustrated in this section.

Given verified mover types for actions, CIVL verifies reduction, i.e., the correctness of the placement of `yield`s using a novel approach. A *Yield Sufficiency Automaton* (*YSA* in Figure 3) encodes all sequences of atomic actions and yields for which safety of cooperative semantics is sufficient for safety of preemptive semantics. Each "transaction" starts with a sequence of right movers (or both movers) and ends with a sequence of left movers (or both movers). In the middle, it can have at most one non mover. Transactions must be separated by `yield`s. CIVL then interprets the control-flow graph of each procedure as an automaton with mover types as edge labels. This abstraction for `WBSlow` is shown in Figure 2. CIVL verifies that this automaton is simulated by the yield-sufficiency automaton using an existing algorithm for computing simulation relations [30].

Linear variables. In Figure 1, thread identifier (`tid`) variables are declared `linear` to indicate that two threads cannot possess the same thread identifier simultaneously. To enforce unique ownership of linear resources, the CIVL type checker prohibits duplication of linear resources [52]. Interference checking and commutativity checking leverage this linearity by automatically inserting assumptions about disjointness of linear resources into verification conditions, making it easier to prove non-interference and commutativity. Linearity is general enough to support much more than just fixed thread identifiers: CIVL also uses it to express separation of memory (in the style of separation logic [49]; see [38]) and to express permissions [8] that may be transferred but not duplicated between threads. Our verified GC, for example, expresses mutual exclusion during initialization and root scanning by temporarily transferring permissions from mutator threads to the GC thread.

Variable hiding. The atomic action specification of `WBSlow` makes no reference to the lock variable, although its implementation involves a lock. When verifying refinement for `WBSlow`, the lock variable has been hidden. CIVL allows the programmer to both introduce and hide variables in each refinement step, thereby providing the capability to perform data refinement. The ability to introduce and hide variables and write yield predicates specific to each refinement step facilitates proofs spanning a large range of abstraction.

$$
\begin{array}{rcl}
g & \in & \mathit{Global} \subseteq \mathit{AllVar} \\
tl & \in & \mathit{ThreadLocal} \subseteq \mathit{AllVar} \\
l & \in & \mathit{Local} \subseteq \mathit{AllVar} \\
x, y & \in & \mathit{Var} = \mathit{Global} \cup \mathit{ThreadLocal} \cup \mathit{Local} \\
v & \in & \mathit{Value} \\
\sigma & \in & \mathit{Store} = \mathit{Var} \rightarrow \mathit{Value} \\
G & \in & \mathit{GStore} = \mathit{Global} \rightarrow \mathit{Value} \\
TL & \in & \mathit{TLStore} = \mathit{ThreadLocal} \rightarrow \mathit{Value} \\
L & \in & \mathit{LStore} = \mathit{Local} \rightarrow \mathit{Value} \\
e, \phi, \psi, \rho, le & \in & \mathit{Expr} = 2^{\mathit{Store}} \\
\alpha, \beta & \in & \mathit{TransExpr} = 2^{(\mathit{Store}, \mathit{Store})} \\
P & \in & \mathit{Proc} \\
A & \in & \mathit{Action} \\
m & \in & \mathit{Mover} = \{B, R, L, N\} \\
as & \in & \mathit{Action} \rightarrow (\mathit{Expr}, \mathit{TransExpr}, \mathit{Mover}) \\
ps & \in & \mathit{Proc} \rightarrow (\mathit{Expr}, 2^{\mathit{ThreadLocal}}, \mathit{Expr}, \mathit{Stmt}) \\
\lambda & \in & \mathit{LinearVar} = 2^{\mathit{Global} \cup \mathit{ThreadLocal}} \\
ls & \in & (\mathit{Action} \cup \mathit{Proc}) \rightarrow (\mathit{LinearVar}, \mathit{LinearVar}) \\
a & \in & \mathit{InsideABlock} ::= \mathrm{a}^{+} \mid \mathrm{a}^{-} \\
RS & \in & \mathit{Proc} \rightharpoonup \mathit{Action} \\
HV & \in & 2^{\mathit{Global}} \\
HA & \in & 2^{\mathit{Action}} \\
HP & \in & 2^{\mathit{Proc}} \\
Perm & \in & \mathit{Value} \rightarrow 2^{\mathit{Value}}
\end{array}
$$

Figure 4: Definitions

3 A concurrent programming language

In this paper, we will formalize our verification method on a core concurrent programming language, CIVL. In this section, we present the syntax and operational semantics of our language. Figure 4 contains basic definitions. Figure 5 uses these definitions to define the syntax of a CIVL program. Informally, a CIVL program is comprised of threads each of which has a stack, global variables shared across threads, thread-local variables that are shared across procedures, and local variables inside a procedure. Thus, a program contains all information to represent not only the static program written by the programmer but also the dynamic state of the program as it ex-

$$
\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & \mathit{skip} \mid \mathit{yield}\ e, \lambda \mid \mathit{call}\ A \mid \mathit{call}\ P \mid \\
 & & \mathit{async}\ P \mid \mathit{ablock}\ \{e, \lambda\}\ s \mid s;\ s \mid \\
 & & \mathit{if}\ \mathit{le}\ \mathit{then}\ s\ \mathit{else}\ s \mid \mathit{while}\ \{e, \alpha\}\ \mathit{le}\ \mathit{do}\ s
\end{array}
$$

$$
\begin{array}{rcl}
F \in \mathit{Frame} & ::= & (P, L, s) \\
T \in \mathit{Thread} & ::= & (\mathit{TL}, \overrightarrow{F}) \\
\mathit{SC} \in \mathit{StmtCtxt} & ::= & []_{\mathit{Stmt}} \mid \mathit{SC}; s \\
\mathit{FC} \in \mathit{FrameCtxt} & ::= & (P, []_{\mathit{LStore}}, \mathit{SC}) \\
\mathit{TC} \in \mathit{ThreadCtxt} & ::= & ([]_{\mathit{TLStore}}, \mathit{FC} \cdot \overrightarrow{F}) \\
\mathit{PC} \in \mathit{ProgCtxt} & ::= & (\mathit{ps}, \mathit{as}, \mathit{ls}, []_{\mathit{GStore}}, \overrightarrow{T} \cdot \mathit{TC} \cdot \overrightarrow{T}) \\
\mathit{Prog} \in \mathit{Program} & ::= & \mathit{PC}[G][\mathit{TL}][L][s]
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{YT} \in \mathit{YieldingThread} & ::= & (\mathit{TL}, (P, L, \mathit{SC}[\mathit{yield}\ e, \lambda]) \cdot \overrightarrow{F}) \mid \\
 & ::= & (\mathit{TL}, (P, L, \mathit{SC}[\mathit{call}\ P']) \cdot \overrightarrow{F}) \mid \\
 & ::= & (\mathit{TL}, (P, L, \mathit{skip}) \cdot \overrightarrow{F}) \\
\mathit{YProgram} & ::= & (\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{\mathit{YT}} \cdot T \cdot \overrightarrow{\mathit{YT}}) \\
\mathit{CProgram} & ::= & (\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{\mathit{YT}})
\end{array}
$$

Figure 5: Syntax

$$\frac{\mathit{as}(A) = (\rho, \alpha, m) \qquad \sigma \notin \rho}{\mathit{as} \vdash (\sigma, \mathit{call}\ A) \longrightarrow \mathit{error}} \quad (\textsc{Atomic-False})$$

$$\frac{\mathit{as}(A) = (\rho, \alpha, m) \qquad \sigma \in \rho \qquad (\sigma, \sigma') \in \alpha}{\mathit{as} \vdash (\sigma, \mathit{call}\ A) \longrightarrow (\sigma', \mathit{skip})} \quad (\textsc{Atomic-True})$$

$$\frac{}{\mathit{as} \vdash (\sigma, \mathit{yield}\ e, \lambda) \longrightarrow (\sigma, \mathit{skip})} \quad (\textsc{Yield})$$

$$\frac{}{\mathit{as} \vdash (\sigma, \mathit{ablock}\ \{e, \lambda\}\ s) \longrightarrow (\sigma, s)} \quad (\textsc{AtomicBlock})$$

$$\frac{}{\mathit{as} \vdash (\sigma, \mathit{skip}; s) \longrightarrow (\sigma, s)} \quad (\textsc{Seq})$$

$$\frac{\sigma \notin \mathit{le}}{\mathit{as} \vdash (\sigma, \mathit{if}\ \mathit{le}\ \mathit{then}\ s_1\ \mathit{else}\ s_2) \longrightarrow (\sigma, s_2)} \quad (\textsc{If-False})$$

$$\frac{\sigma \in \mathit{le}}{\mathit{as} \vdash (\sigma, \mathit{if}\ \mathit{le}\ \mathit{then}\ s_1\ \mathit{else}\ s_2) \longrightarrow (\sigma, s_1)} \quad (\textsc{If-True})$$

$$\frac{\sigma \notin \mathit{le}}{\mathit{as} \vdash (\sigma, \mathit{while}\ \{e, \alpha\}\ \mathit{le}\ \mathit{do}\ s) \longrightarrow (\sigma, \mathit{skip})} \quad (\textsc{While-False})$$

$$\frac{\sigma \in \mathit{le} \qquad s' = \mathit{while}\ \{e, \alpha\}\ \mathit{le}\ \mathit{do}\ s}{\mathit{as} \vdash (\sigma, s') \longrightarrow (\sigma, s; s')} \quad (\textsc{While-True})$$

Figure 7: Operational semantics for statements

$$\frac{\begin{array}{c} \mathit{PC}[G][\mathit{TL}][L][s] = (\_, \mathit{as}, \_, \_, \_) \\ \mathit{as} \vdash (G \cdot \mathit{TL} \cdot L, s) \longrightarrow (G' \cdot \mathit{TL}' \cdot L', s') \end{array}}{\mathit{PC}[G][\mathit{TL}][L][s] \longrightarrow \mathit{PC}[G'][\mathit{TL}'][L'][s']} \quad (\textsc{Step})$$

$$\frac{\begin{array}{c} \mathit{PC}[G][\mathit{TL}][L][s] = (\_, \mathit{as}, \_, \_, \_) \\ \mathit{as} \vdash (G \cdot \mathit{TL} \cdot L, s) \longrightarrow \mathit{error} \end{array}}{\mathit{PC}[G][\mathit{TL}][L][s]) \longrightarrow \mathit{error}} \quad (\textsc{Fail})$$

$$\frac{\begin{array}{c} \mathit{ps}(P) = (\phi, M, \psi, s) \\ \mathit{ls}(P) = (\lambda, \lambda') \qquad T' = (\mathit{TL}, (P, L, \mathit{yield}\ \phi, \lambda; s)) \\ \overrightarrow{T} = \overrightarrow{T_1} \cdot \mathit{TC}[\mathit{TL}][L][\mathit{async}\ P] \cdot \overrightarrow{T_2} \\ \overrightarrow{T}' = \overrightarrow{T_1} \cdot \mathit{TC}[\mathit{TL}][L][\mathit{skip}] \cdot \overrightarrow{T_2} \end{array}}{(\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T}) \longrightarrow (\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T}' \cdot T')} \quad (\textsc{Async})$$

$$\frac{T = (\mathit{TL}, \epsilon)}{(\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T} \cdot T \cdot \overrightarrow{T'}) \longrightarrow (\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T} \cdot \overrightarrow{T'})} \quad (\textsc{End})$$

$$\frac{\begin{array}{c} \mathit{ps}(P) = (\phi, M, \psi, s) \\ \mathit{ls}(P) = (\lambda, \lambda') \qquad T = (\mathit{TL}, (P', L, \mathit{SC}[\mathit{call}\ P]) \cdot \overrightarrow{F}) \\ T' = (\mathit{TL}, (P, L, s) \cdot (P', L, \mathit{SC}[\mathit{skip}]) \cdot \overrightarrow{F}) \end{array}}{(\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T} \cdot T \cdot \overrightarrow{T'}) \longrightarrow (\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T} \cdot T' \cdot \overrightarrow{T'})} \quad (\textsc{Call})$$

$$\frac{T = (\mathit{TL}, (P, L, \mathit{skip}) \cdot \overrightarrow{F}) \qquad T' = (\mathit{TL}, \overrightarrow{F})}{(\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T} \cdot T \cdot \overrightarrow{T'}) \longrightarrow (\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T} \cdot T' \cdot \overrightarrow{T'})} \quad (\textsc{Return})$$

Figure 6: Operational semantics for program

ecutes. The sets *Global*, *ThreadLocal*, and *Local* are the names of global, thread-local, and procedure-local variables respectively. These sets are mutually disjoint and *Var* denotes their union. There is an uninterpreted set *Value* of values that may be stored in these variables. *Store*, *GStore*, *TLStore*, and *LStore* are all sets of maps into *Value* from *Var*, *Global*, *ThreadLocal*, and *Local* respectively. We denote elements of *Store*, *GStore*, *TLStore*, and *LStore* by σ, G, TL, and L respectively. Formally, a CIVL program is a tuple $(\mathit{ps}, \mathit{as}, \mathit{ls}, G, \overrightarrow{T})$ with the following components.

Procedures. ps maps a procedure name $P \in \mathit{Proc}$ to a tuple (ϕ, M, ψ, s), where ϕ is the precondition of P, M is the set of thread-local variables potentially modified by P, ψ is the postcondition of P, and s is the body of P. The predicates ϕ and ψ cannot refer to procedure-local variables. Procedures do not have parameters, but parameters may be modeled using thread-local variables.

Actions. as maps an action name $A \in \mathit{Action}$ to a tuple (ρ, α, m). Actions are used inside procedure bodies to access global and thread-local variables. The predicate ρ over *Store* is the gate and the predicate α over *Store* $\times$ *Store* is the transition relation of A. If the gate ρ does not hold on the current state, the program fails; otherwise, the program makes progress by executing the action A based on the transition relation α. Finally, m is one of four values in $\{B, R, L, N\}$; it denotes the commutativity type of the action, B for both mover, R for right mover, L for left mover, and N for non mover [21].

Linear interfaces. *ls* maps each procedure and action name in $Proc \cup Action$ to a linear interface [52] (λ, λ'), where each of λ and λ' is a set comprising global and thread-local variables. To exploit linearity during verification, the programmer provides a function *Perm* from *Value* to 2^{Value}. The set $Perm(v)$ is the set of *permissions* associated with a value $v \in Value$. The CIVL type checker (Section 4.1) computes a set of *available* linear variables at each control location such that the multiset union of the permissions associated with the available variables is guaranteed to be a set, i.e., permissions are never duplicated among linear variables. The linear interface (λ, λ') of a procedure (or action) indicates the available sets at its beginning (λ) and end (λ'), respectively. Linear interfaces allow CIVL to reason about each procedure separately.

Global store. G is the global store of the program.

Threads. $\overrightarrow{T}$ is a sequence of threads. Each thread T in $\overrightarrow{T}$ is a pair $(TL, \overrightarrow{F})$ where TL is the thread-local store and $\overrightarrow{F}$ is a stack of frames comprising the continuation of T. Each frame (P, L, s) in $\overrightarrow{F}$ comprises a procedure P, its procedure-local store L, and a statement s.

3.1 Operational semantics

Figures 6 and 7 together present the operational semantics as a relation $\longrightarrow$ between a pair of programs. The rules for the relation extend the syntax of statements with an embedded context [54]. A statement context SC is either a context $[]_{Stmt}$ or the sequential composition of a statement context and a statement. Thus, each statement context has a unique context $[]_{Stmt}$ inside it; this context encodes the current position of the execution of a thread. The result of substituting a statement s inside the statement context SC is another statement and is denoted by $SC[s]$. Similarly, we define a frame context FC with two contexts embedded inside it, $[]_{LStore}$ for substituting a procedure-local store and $[]_{Stmt}$ for substituting a statement. The result of substituting a procedure-local store L and a statement s inside FC is a frame and is denoted by $FC[L][s]$; here, L is the procedure-local store for the procedure containing the currently executed statement s. A thread context TC has three contexts embedded inside it, $[]_{TLStore}$ for substituting a thread-local store, $[]_{LStore}$ for substituting a procedure-local store, and $[]_{Stmt}$ for substituting a statement. The result of substituting a thread-local store TL, a procedure-local store L, and a statement s inside TC is a thread and is denoted by $TC[TL][L][s]$. Finally, the result of substituting a global store G, a thread-local store TL, a procedure-local store L, and a statement s inside PC is a program and is denoted by $PC[G][TL][L][s]$.

The operational semantics use the notation $G \cdot TL \cdot L$ to denote the concatenation of the stores G, TL, and L. Rule STEP in Figure 6 allows the program $PC[G][TL][L][s]$ to move to $PC[G'][TL'][L'][s']$ if $(G \cdot TL \cdot L, s)$ can move to $(G' \cdot TL' \cdot L', s)$ according to any rule in Figure 7 except for rule ATOMIC-FALSE. Similarly, rule FAIL allows $PC[G][TL][L][s]$ to fail if $(G \cdot TL \cdot L, s)$ fails according to rule ATOMIC-FALSE. Rule ASYNC describes the creation of a thread via *async* P. The new thread is added to the sequence of threads in the program. Rule END describes the termination of a thread; the terminating thread is removed from the sequence of threads in the program. Rules CALL and RETURN describe the call and return of a procedure via *call* P.

The rules in Figure 7 provide the operational semantics for the statements in CIVL. The statement *skip* has no side effect and is used as a marker in the formal operational semantics. The statement *yield* e, λ contains a predicate e and the linear permissions λ. The yield predicate e is expected to be established by the executing thread and preserved by other threads; the linear permissions λ are expected to be available as long as the control of the thread is at the yield statement. The yield statement is the mechanism by which the programmer specifies the cooperative semantics over which the refinement check is performed. The statement *call* A invokes the action A. Suppose $as(A) = (\rho, \alpha, m)$ and $ls(A) = (\lambda, \lambda')$. If ρ does not hold, the statement *call* A fails, otherwise the store is updated according to the transition relation α. The permissions in λ are expected to be available at the beginning and the permissions in λ' are guaranteed to be available at the end. The statement *call* P invokes the procedure P. Suppose $ps(P) = (\phi, M, \psi, s)$ and $ls(P) = (\lambda, \lambda')$. It is expected that the precondition ϕ holds and the permissions λ are available at the beginning. Then, the statement s executes. When s terminates, the postcondition ψ holds and the permissions λ' are available. The predicates ϕ and ψ are expected to be preserved by other threads in the environment. The statement *async* P creates a new thread that begins by invoking the procedure P. The statement *ablock* $\{e, \lambda\}$ s is useful only for compactly encoding non-interference checks (Section 4.2.2); the predicate e is expected to hold and the permissions λ expected to be available at the beginning of s. In CIVL, yield predicates, preconditions, and postconditions are the three sources of location invariants [48].

Finally, we have the standard primitives for sequencing, conditional execution, and looping. The statement s_1; s_2 executes s_1 followed by s_2. The statement *if le then* s_1 *else* s_2 executes s_1 if *le* is true and otherwise executes s_2. The statement *while* $\{e, \alpha\}$ *le do s* executes s repeatedly until *le* becomes false. The expression e is a loop invariant and the expression α is a summary for failure-free executions of the loop that do not go through a yield statement (Section 4.2.2). The conditional expression *le* in *if* and *while* statements cannot refer to global variables.

We formalize the preemptive operational semantics of CIVL as a relation $\longrightarrow$ with domain *Program* and codomain *Program* $\cup \{error\}$. We use four contexts—*SC*, *FC*, *TC*, and *PC*—to formalize $\longrightarrow$ as a structured operational semantics [54]. In this semantics, a thread is chosen nondeterministically to execute a single statement. The statement *skip* has no side effect. The statement *yield* e, λ contains a predicate e and a set λ comprising global and thread-local variables. The yield predicate e is expected to be established by the executing thread and preserved by other threads; the variables λ are expected to be available at the yield statement. The yield statement is the mechanism by which the programmer specifies the cooperative semantics (described later) over which the refinement check is performed. In CIVL, yield predicates, preconditions, and postconditions are the three sources of location invariants [48]. The statement *call* A invokes the action A. Suppose $as(A) = (\rho, \alpha, m)$. If ρ does not hold, the statement *call* A fails and the program evolves to *error*, otherwise the store is updated according to the transition relation α. The statement *call* P invokes the procedure P. Suppose $ps(P) = (\phi, M, \psi, s)$. It is expected that the precondition ϕ holds, then statement s executes, and finally the postcondition ψ holds. The predicates ϕ and ψ are expected to be preserved by other threads in the environment. The statement *async* P creates a new thread that begins by invoking the procedure P. The statement *ablock* $\{e, \lambda\}$ s is useful only for compactly encoding non-interference checks; the predicate e is expected to hold and the set λ is expected to be available at the beginning of s. Finally, we have the standard primitives for sequencing, conditional execution, and looping.

We now define the cooperative operational semantics induced by the presence of yield statements in the program. A *yielding thread* (*YieldingThread* in Figure 5) is one that is waiting to execute at a yield statement, a call, or a return from a call. A program in *YProgram* is one in which all threads except at most one thread are yielding threads. A program in *CProgram* is one in which all threads are yielding threads. Thus, we have *CProgram* $\subseteq$ *YProgram* $\subseteq$ *Program*. Let $\hookrightarrow$ be the relation obtained by restricting the domain and codomain of $\longrightarrow$ to *YProgram*. In a $\hookrightarrow$-execution, the one (possibly) non-yielding thread is the only one allowed to execute until it reaches a yield statement and becomes a yielding thread; at that point, all threads are yielding and any one can be picked for execution. We define the relation $\longmapsto$ with domain *CProgram* and codomain *CProgram* $\cup \{error\}$ as follows:

1. *Prog* $\longmapsto$ *Prog*$'$ if there is a $\hookrightarrow$-execution from *Prog* to *Prog*$'$ such that every program on the

execution except for *Prog* and *Prog'* is not in *CProgram*.

2. *Prog* $\longmapsto$ *error* if there is a $\hookrightarrow$-execution from *Prog* to *error* such that every program on the execution except for *Prog* is not in *CProgram*.

Consider a program $Prog \in CProgram$. We write *Safe*(*Prog*) if it is not the case that $Prog \longrightarrow error$. We write $Safe^*(Prog)$ if $Safe(Prog')$ for all $Prog'$ such that $Prog \longrightarrow^* Prog'$. We write *CSafe*(*Prog*) if it is not the case that $Prog \longmapsto error$. We write $CSafe^*(Prog)$ if $CSafe(Prog')$ for all $Prog'$ such that $Prog \longmapsto^* Prog'$. We write *Cooperative*(*Prog*) if for each infinite $\hookrightarrow$-execution $Prog \hookrightarrow Prog_0 \hookrightarrow Prog_1 \hookrightarrow \cdots$ starting from *Prog*, there is some $i \geq 0$ such that $Prog_i \in CProgram$. We write $Cooperative^*(Prog)$ if $Cooperative(Prog')$ for all $Prog' \in CProgram$ such that $Prog \longmapsto^* Prog'$.

3.2 Definitions

We conclude this section by presenting a few definitions of concepts that are used in Section 4 to formalize our verification method. A predicate ρ is *independent* of a variable $x \in Var$ if for all $\sigma \in Store$ and $v \in Value$, if $\sigma \in \rho$ then $\sigma[x \to v] \in \rho$. A transition relation α is *independent* of a variable $x \in Var$ if for all $\sigma, \sigma' \in Store$ and $v, v' \in Value$, if $(\sigma, \sigma') \in \alpha$ then $(\sigma[x \to v], \sigma(x \to v')) \in \alpha$. We define $Acc(\rho)$ to be the smallest set $X \subseteq Var$ such that ρ is independent of x for all $x \in Var \setminus X$. We also define $Acc(\alpha)$ similarly. For any transition relation α, let $\exists\exists Local.\ \alpha$ be the transition relation

$$\{(GTLL, G'\cdot TL'\cdot L') \mid \exists L_1, L_2.\ (GTLL_1, G'\cdot TL'\cdot L_2) \in \alpha\}.$$

For any $X \subseteq Var$, let $Havoc(X)$, be the transition relation $\{(\sigma, \sigma') \mid \forall x \in Var \setminus X.\ \sigma(x) = \sigma'(x)\}$.

Given $\rho_1, \rho_2 \in Expr$, we write $\rho_1 \vee \rho_2$ for the union of ρ_1 and ρ_2, $\rho_1 \wedge \rho_2$ for the intersection of ρ_1 and ρ_2, and $\neg\rho_1$ for $Expr \setminus \rho_1$. Similarly, given $\alpha_1, \alpha_2 \in TransExpr$, we write $\alpha_1 \vee \alpha_2$ for the union of α_1 and α_2, $\alpha_1 \wedge \alpha_2$ for the intersection of α_1 and α_2, and $\neg\alpha_1$ for $TransExpr \setminus \alpha_1$. We denote the relational composition of α_1 and α_2 by $\alpha_1 \circ \alpha_2$. Given $\rho \in Expr$ and $\alpha \in TransExpr$, we denote by $\rho \to \alpha$ the set $\{(\sigma, \sigma') \mid \sigma \in \rho \Rightarrow (\sigma, \sigma') \in \alpha\}$. We let $\rho \circ \alpha$ denote the relation obtained by restricting the domain of α to ρ. Similarly, we let $\alpha \circ \rho$ denote the relation obtained by restricting the codomain of α to ρ. We also sometimes use *false* to represent the empty set of states or the empty set of transitions. Given a relation R (over a pair of sets), we denote the domain of R by $dom(R)$ and the codomain of R by $cod(R)$.

It is often convenient to collect the permissions in a set of available variables. To that end, we define the function *Collect*. Given a partial map π from *AllVar* to *Value* and a set $X \subseteq AllVar$, let $Collect(\pi, X)$ be the multiset $\bigcup\{Perm(\pi(x)) \mid x \in X \wedge dom(\pi)\}$. For any multiset Λ over *Value*, we also define $IsSet(\Lambda)$ be true iff Λ is a set.

4 Verification

Suppose a program *Prog'* has been proved to be safe. However, it is implemented using atomic actions that are too coarse to be directly implementable. To carry over the safety of *Prog'* to a realizable implementation *Prog*, these coarse atomic actions must be refined down to lower-level actions. During this refinement, an invocation *call* A of a high-level atomic action A is transformed into an invocation *call* P of a procedure that is implemented using low-level actions. The main contribution of this paper is a verification method that allows us to safely refine program *Prog'* to another program *Prog* (or abstract *Prog* to *Prog'*) so that safety properties proved on *Prog'* continue to hold on *Prog* as well.

The safety of the program *Prog*, defined in Section 3 as $Safe^*(Prog)$, depends on the preemptive semantics of *Prog*. Our overall goal is to conclude $Safe^*(Prog)$ from $Safe^*(Prog')$; we decompose this goal into two sub-goals:

1. We establish a simulation relation from the cooperative semantics of *Prog* to the preemptive semantics of *Prog'*, which ensures that $Safe^*(Prog')$ is sufficient for $CSafe^*(Prog)$.

2. We exploit commutativity reasoning to compute a simulation relation from *Prog* to *YSA* (Fig-

(Skip)
$$\frac{}{\lambda \vdash_l \mathit{skip} : \lambda}$$

(Atomic)
$$\frac{\mathit{ls}(A) = (\lambda, \lambda')}{\lambda \vdash_l \mathit{call}\ A : \lambda'}$$

(Yield)
$$\frac{\lambda_y \subseteq \lambda}{\lambda \vdash_l \mathit{yield}\ e, \lambda_y : \lambda}$$

(Ablock)
$$\frac{\lambda \vdash_l s : \lambda' \qquad \lambda_a \subseteq \lambda}{\lambda \vdash_l \mathit{ablock}\ \{e, \lambda_a\}\ s : \lambda'}$$

(Call)
$$\frac{\mathit{ls}(P) = (\lambda, \lambda')}{\lambda \vdash_l \mathit{call}\ P : \lambda'}$$

(Async)
$$\frac{\lambda_G \subseteq \mathit{Global} \qquad \lambda \cup \lambda_P \cup \lambda'_P \subseteq \mathit{ThreadLocal} \qquad \mathit{ls}(P) = ((\lambda_G, \lambda_P), (\lambda_G, \lambda'_P))}{\lambda_G, \lambda, \lambda_P \vdash_l \mathit{async}\ P : \lambda_G, \lambda}$$

(Seq)
$$\frac{\lambda \vdash_l s_1 : \lambda' \qquad \lambda' \vdash_l s_2 : \lambda''}{\lambda \vdash_l s_1; s_2 : \lambda''}$$

(Ite)
$$\frac{\lambda \vdash_l s_1 : \lambda' \qquad \lambda \vdash_l s_2 : \lambda'}{\lambda \vdash_l \mathit{if}\ \mathit{le}\ \mathit{then}\ s_1\ \mathit{else}\ s_2 : \lambda'}$$

(While)
$$\frac{\lambda \vdash_l s : \lambda}{\lambda \vdash_l \mathit{while}\ \{e, \alpha\}\ \mathit{le}\ \mathit{do}\ s : \lambda}$$

(Procedure)
$$\frac{\mathit{ls}(P) = (\lambda, \lambda') \qquad \mathit{ps}(P) = (\phi, M, \psi, s) \qquad \lambda \vdash_l s : \lambda'}{\vdash_l P}$$

(Action)
$$\frac{\begin{array}{c}\mathit{ls}(A) = (\lambda, \lambda') \qquad \lambda \cap \mathit{Global} = \lambda' \cap \mathit{Global} \qquad \mathit{as}(A) = (\rho, \alpha, m) \\ \forall (\sigma, \sigma') \in \rho \circ \alpha.\ \mathit{Collect}(\sigma', \lambda') \subseteq \mathit{Collect}(\sigma, \lambda)\end{array}}{\vdash_l A}$$

(StackFrame)
$$\frac{\lambda \vdash_l s : \lambda'}{\lambda \vdash_l (P, L, s) : \lambda'}$$

(Thread)
$$\frac{\forall 1 \leq i \leq n.\ \lambda_i \vdash_l F_i : \lambda_{i+1}}{\lambda_1 \vdash_l (\mathit{TL}, F_1 \ldots F_n)}$$

(Program)
$$\frac{\begin{array}{c}\forall P \in \mathit{Proc}.\ \vdash_l P \qquad \forall A \in \mathit{Action}.\ \vdash_l A \qquad \lambda_G \subseteq \mathit{Global} \qquad \forall 1 \leq i \leq n.\ \lambda_i \subseteq \mathit{ThreadLocal} \\ \forall 1 \leq i \leq n.\ \lambda_G, \lambda_i \vdash_l T_i \qquad \forall 1 \leq i \leq n.\ T_i = (\mathit{TL}_i, \ldots) \qquad \mathit{IsSet}(\cup_{1 \leq i \leq n} \mathit{Collect}(\mathit{TL}_i, \lambda_i) \cup \mathit{Collect}(G, \lambda_G))\end{array}}{\vdash_l (\mathit{ps}, \mathit{as}, \mathit{ls}, G, T_1 \ldots T_n)}$$

Figure 8: Linear variables

ure 3), which ensures that $\mathit{CSafe}^*(\mathit{Prog})$ is sufficient for $\mathit{Safe}^*(\mathit{Prog})$

Both sub-goals depend on the auxiliary judgment $\vdash_l \mathit{Prog}$ for checking that *Prog* uses linear variables appropriately. This judgment computes available linear variables at each program location; this information is encoded logically as free disjointness assumptions that increase precision of non-interference and commutativity reasoning at low annotatation overhead.

The first sub-goal is discharged by the judgments $\mathit{RS}; \mathit{HP}; \mathit{HV}; \mathit{HA}; \overrightarrow{b} \vdash \mathit{Prog} \rightsquigarrow \mathit{Prog}'$, $\mathit{RS}; \mathit{HP}; \mathit{HV}; \overrightarrow{b} \vdash_r \mathit{Prog}$, and $\mathit{InterferenceSafe}(\mathit{Prog})$. The first judgment performs a rewrite of *Prog* to Prog'; the second judgment checks each procedure separately against location invariants and atomic action specifications; the third judgment checks that the location invariants are stable against interference from concurrently-executing threads. The map $\mathit{RS} \in \mathit{Proc} \rightharpoonup \mathit{Action}$ indicates for a procedure $P \in \mathit{dom}(\mathit{RS})$ the action $\mathit{RS}(P)$ that abstracts it; in Prog', every occurrence of *call P* is replaced by *call* $\mathit{RS}(P)$. The sets $\mathit{HP} \in 2^{\mathit{Proc}}$, $\mathit{HV} \in 2^{\mathit{Global}}$, and $\mathit{HA} \in 2^{\mathit{Action}}$ are, respectively, the procedure, the global variables, and atomic actions that are introduced for refining Prog' to *Prog*; we can also think that these entities are hidden in the abstract program Prog'. The set *HP* includes $\mathit{dom}(\mathit{RS})$ but may also contain other procedures whose invocations are replaced by *skip* in Prog'. The vector $\overrightarrow{b}$ contains a *Boolean* value b_i for each thread T_i in *Prog*. If b_i is true, it indicates that there is a partially executed procedure on the stack of T_i such that $P \in \mathit{dom}(\mathit{RS})$ but the code inside P that corresponds to the execution of the atomic action $\mathit{RS}(P)$ has yet to be executed. The value of b_i is used to rewrite the stack of T_i while transforming *Prog* to Prog'.

The validity of the transformation from *Prog* to Prog' depends on establishing a simulation relation between them. The base case of the simulation must check that $\mathit{CSafe}(\mathit{Prog})$; The inductive case must check that if *Prog* can evolve to Prog_1 (via cooperative semantics) by executing from the current yield statement to the next yield statement, then there is Prog'_1 and b_1 such that Prog' can evolve in zero or more steps to Prog'_1 and $RS; HP; HV; HA; \overrightarrow{b_1} \vdash \mathit{Prog}_1 \rightsquigarrow \mathit{Prog}'_1$. The judgments $RS; HP; HV; \overrightarrow{b} \vdash_r \mathit{Prog}$ and $\mathit{InterferenceSafe}(\mathit{Prog})$ establish this property via a collection of verification conditions, one for each procedure and one for each pair of yield statement and atomic block in the program.

The second sub-goal is discharged by the two judgments $RS \vdash_y \mathit{Prog}$ and $\mathit{CommutativitySafe}(\mathit{Prog})$. Suppose $\mathit{Prog} = (ps, as, ls, G, \overrightarrow{T})$. To justify that $\mathit{CSafe}^*(\mathit{Prog})$ is sufficient for $\mathit{Safe}^*(\mathit{Prog})$, we exploit commutativity information available from the mover types of atomic actions. The *Yield Sufficiency Automaton* (*YSA*) from Figure 3 encodes all sequences of atomic actions and yields for which reasoning about cooperative semantics is sufficient. For example, the *YSA* automaton indicates that a yield after a right mover or a yield before a left mover is unnecessary. The judgment $\vdash_y \mathit{Prog}$ checks that the code of *Prog* can be simulated by *YSA*; it depends on the judgment $\mathit{CommutativitySafe}(\mathit{Prog})$ which checks that all mover annotations are correct.

Theorem 1 *Let* $\mathit{Prog}, \mathit{Prog}' \in \mathit{CProgram}$ *be such that* $\mathit{Cooperative}^*(\mathit{Prog})$ *and* $\mathit{Safe}^*(\mathit{Prog}')$. *Let* $RS \in \mathit{Proc} \rightharpoonup \mathit{Action}$, $HP \in 2^{\mathit{Proc}}$, $HV \in 2^{\mathit{Global}}$, $HA \in 2^{\mathit{Action}}$, *and* $\overrightarrow{b} \in \overrightarrow{\mathit{Boolean}}$ *be such that the following hold:*

1. $\vdash_l \mathit{Prog}$.
2. $RS; HP; HV; HA; \overrightarrow{b} \vdash \mathit{Prog} \rightsquigarrow \mathit{Prog}'$,
   $RS; HP; HV; \overrightarrow{b} \vdash_r \mathit{Prog}$,
   and $\mathit{InterferenceSafe}(\mathit{Prog})$.
3. $\vdash_y \mathit{Prog}$ *and* $\mathit{CommutativitySafe}(\mathit{Prog})$.

Then, we have $\mathit{Safe}^*(\mathit{Prog})$.

Theorem 1 is our main soundness theorem; it concludes the safety of *Prog* from the safety of Prog'. In general, the correctness proof of a large program is obtained by chaining together multiple instances of this theorem connecting a sequence of programs.

4.1 Using linear variables

In this section, we formalize the judgment $\vdash_l \mathit{Prog}$. The goal of this judgment is to check that linear variables and atomic blocks are used appropriately in *Prog*. The judgment $\vdash_l \mathit{Prog}$ enforces that linear permissions are never duplicated during the execution of *Prog*. Rule Program checks that the disjointness invariant holds in the initial state of *Prog*. Rule Action checks that each action A preserves the disjointness invariant. Consequently, the invariant holds throughout the execution of *Prog*. There are three conditions being checked by rule Action. First, the set of global linear permissions does not change. Second, if the disjointness invariant holds for input permissions it also holds for output permissions. Finally, the union of the sets constructed from output permissions is a subset of the union of sets constructed from input permissions. This last condition is important because it allows via local checking to conclude that the disjointness invariant holds globally for linear permissions held by all threads. The rule Async splits the thread-local permissions λ of the caller of *async P* into λ and λ_P, passing λ_P to the new thread and continuing with λ. Note that all global permissions in λ_G are also made available to the new thread; there is no duplication because all threads refer to the same set of global variables.

The following lemma states that if a program is well-typed it remains remains well-typed after executing a step. By induction over execution steps, this lemma indirectly ensures that all programs reaachable from a well-typed program are well-typed.

Lemma 1 *Let* $\mathit{Prog}, \mathit{Prog}_1 \in \mathit{Program}$ *such that* $\vdash_l \mathit{Prog}$ *and* $\mathit{Prog} \longrightarrow \mathit{Prog}_1$. *Then, we have* $\vdash_l \mathit{Prog}_1$.

Proof Sketch: Suppose $\mathit{Prog} = (ps, as, ls, G, T_1 \ldots T_n)$, $T_i = (TL_i, (P_i, L_i, SC[s]) \cdot \overrightarrow{F_i})$, and thread T_i takes a step in going from *Prog* to Prog_1. Suppose the judgment $\vdash_l \mathit{Prog}$ used available sets λ_G and $\lambda_1, \ldots, \lambda_i, \ldots, \lambda_n$ to check *Prog*. To type check Prog_1 only the available set λ_i for the thread T_i that took the step needs to change. To pick λ_i, we perform a case analysis on s. Let us consider a few cases. Suppose $s = \mathit{call}\ A$ and $ls(A) = (\lambda, \lambda')$. By rule Atomic, we have $\lambda = \lambda_i$ and we pick λ' for typing $Prog_1$. Suppose $s = call\ P$, $ls(P) = (\lambda, \lambda')$, and $ps(P) = (\phi, M, \psi, s')$. Then we push the stack frame $(P, L, yield\ \phi, \lambda; s')$ onto the stack of T_i. By rule CALL, we have $\lambda = \lambda_i$, We exploit the proof of $\vdash_l P$ (rule PROCEDURE) to construct the proof for the new stack. Suppose $s = async\ P$, $ls(P) = (\lambda, \lambda')$, and $ps(P) = (\phi, M, \psi, s')$. Then we create a new thread with a single stack frame $(P, L, yield\ \phi, \lambda; s')$. By rule ASYNC, we have $\lambda \subseteq \lambda_i$; we pick λ for the new thread and $\lambda_i \setminus \lambda$ for thread T_i. Other cases are handled similarly. ■

The following lemma states that in a well-typed program that if two different threads are about to execute a yield statement and an atomic block statement, respectively, the available linear variables annotating these statements have disjoint permissions. Together with Lemma 1, this lemma ensures the soundness of the judgment *InterferenceSafe*(*Prog*).

Lemma 2 *Let* $Prog \in Program$ *such that* $\vdash_l Prog$. *Suppose* $Prog = (ps, as, ls, G, T_1 \ldots T_n)$ *and* $i, j \in [1, n]$ *such that* $T_i = (TL_i, (P_i, L_i, SC[ablock\ \ \{e_i, \lambda_i\}\ \ ]) \cdot \overrightarrow{F_i})$ *and* $T_j = (TL_j, (P_j, L_j, SC[yield\ \ e_j, \lambda_j]) \cdot \overrightarrow{F_j})$. *Then* $IsSet(Collect(G, \lambda_i) \cup Collect(TL_i, \lambda_i) \cup Collect(TL_j, \lambda_j))$.

Proof Sketch: The proof follows from rules PROGRAM, ABLOCK, and YIELD. ■

The following lemma states that in a well-typed program that if two different threads are each about to execute an atomic action, the available linear variables at the input of these actions have disjoint permissions. Together with Lemma 1, this lemma ensures the soundness of the judgment *CommutativitySafe*(*Prog*).

Lemma 3 *Let* $Prog \in Program$ *such that* $\vdash_l Prog$. *Suppose* $Prog = (ps, as, ls, G, T_1 \ldots T_n)$ *and* $i, j \in [1, n]$ *such that* $T_i = (TL_i, (P_i, L_i, SC[call\ \ A_i]) \cdot \overrightarrow{F_i})$, $ls(A_i) = (\lambda_i, \lambda'_i)$, $T_j = (TL_j, (P_j, L_j, SC[call\ \ A_j]) \cdot \overrightarrow{F_j})$, *and* $ls(A_j) = (\lambda_j, \lambda'_j)$. *Then* $IsSet(Collect(G, \lambda_i) \cup Collect(TL_i, \lambda_i) \cup Collect(TL_j, \lambda_j))$.

Proof Sketch: This proof follows from rules PROGRAM, ACTION, and ATOMIC. ■

4.2 Reasoning about cooperative semantics

In this section, we attack the first sub-goal discussed earlier. We establish a simulation relation from the cooperative semantics of *Prog* to the preemptive semantics of *Prog'*, which ensures that $Safe^*(Prog')$ is sufficient for $CSafe^*(Prog)$. This verification depends on three judgments. The judgment $RS; HP; HV; HA; \overrightarrow{b} \vdash Prog \rightsquigarrow Prog'$ is discussed in Section 4.2.1. The judgments $RS; HP; HV; \overrightarrow{b} \vdash_r Prog$ and *InterferenceSafe*(*Prog*) are discussed in Section 4.2.2.

4.2.1 Program transformation

In this section, we take a closer look at the judgment $RS; HP; HV; HA; \overrightarrow{b} \vdash Prog \rightsquigarrow Prog'$. In this judgment, $RS \in Proc \rightharpoonup Action$ is a partial function from procedure names to action names, $HP \in 2^{Proc}$ is a set of procedure names, $HV \in 2^{Global}$ is a set of global variables, and $HA \in 2^{Action}$ is a set of action names. The map RS indicates for a procedure $P \in dom(RS)$ the action $RS(P)$ that abstracts it; in the abstract program, every occurrence of *call P* is replaced by *call RS(P)*. The set HP contains all procedures that are hidden as a result of the abstraction. This set must include $dom(RS)$ but can also contain other procedures whose invocations are replaced by *skip* in the abstract program. The sets HV and HA are the global variables and atomic actions, respectively, that are hidden in the abstract program.[3] The vector $\overrightarrow{b}$ contains a *Boolean* value b_i for each thread T_i in *Prog*. If b_i is true, it indicates that there is a partially executed procedure on the stack of T_i such that $P \in dom(RS)$ but the code inside P that corresponds to the execution of the atomic action $RS(P)$ has yet to be executed. The value of b_i is used to rewrite the stack of T_i.

We now discuss the transformation rules in Figure 9 starting from the bottom. Rule PROGRAM first checks sanity conditions involving RS, HP, HV, and

[3]In our formalization, hiding a procedure, global variable, or action simply means that the abstract program is forbidden from using it. Technically, these hidden entities continue to exist since the sets *Proc*, *Global*, and *Action* do not change.

$$\text{(Skip)}\quad \frac{}{RS; HP; HV; HA \vdash \mathit{skip} \rightsquigarrow \mathit{skip}}$$

$$\text{(Atomic)}\quad \frac{A \notin HA}{RS; HP; HV; HA \vdash \mathit{call}\ A \rightsquigarrow \mathit{call}\ A}$$

$$\text{(Yield)}\quad \frac{}{RS; HP; HV; HA \vdash \mathit{yield}\ e, \lambda \rightsquigarrow \mathit{yield}\ e', \lambda'}$$

$$\text{(Ablock1)}\quad \frac{RS; HP; HV; HA \vdash s \rightsquigarrow s'}{RS; HP; HV; HA \vdash \mathit{ablock}\ \{e, \lambda\}\ s \rightsquigarrow s'}$$

$$\text{(Ablock2)}\quad \frac{RS; HP; HV; HA \vdash s \rightsquigarrow s'}{RS; HP; HV; HA \vdash s \rightsquigarrow \mathit{ablock}\ \{e, \lambda\}\ s'}$$

$$\text{(Call1)}\quad \frac{P \notin HP}{RS; HP; HV; HA \vdash \mathit{call}\ P \rightsquigarrow \mathit{call}\ P}$$

$$\text{(Call2)}\quad \frac{P \in HP \setminus \mathit{dom}(RS)}{RS; HP; HV; HA \vdash \mathit{call}\ P \rightsquigarrow \mathit{skip}}$$

$$\text{(Call3)}\quad \frac{P \in \mathit{dom}(RS)}{RS; HP; HV; HA \vdash \mathit{call}\ P \rightsquigarrow \mathit{call}\ RS(P)}$$

$$\text{(Async1)}\quad \frac{P \notin HP}{RS; HP; HV; HA \vdash \mathit{async}\ P \rightsquigarrow \mathit{async}\ P}$$

$$\text{(Async2)}\quad \frac{P \in HP \setminus \mathit{dom}(RS)}{RS; HP; HV; HA \vdash \mathit{async}\ P \rightsquigarrow \mathit{skip}}$$

$$\text{(Seq)}\quad \frac{RS; HP; HV; HA \vdash s_1 \rightsquigarrow s_1' \qquad RS; HP; HV; HA \vdash s_2 \rightsquigarrow s_2'}{RS; HP; HV; HA \vdash s_1; s_2 \rightsquigarrow s_1'; s_2'}$$

$$\text{(Ite)}\quad \frac{RS; HP; HV; HA \vdash s_1 \rightsquigarrow s_1' \qquad RS; HP; HV; HA \vdash s_2 \rightsquigarrow s_2'}{RS; HP; HV; HA \vdash \mathit{if}\ le\ \mathit{then}\ s_1\ \mathit{else}\ s_2 \rightsquigarrow \mathit{if}\ le\ \mathit{then}\ s_1'\ \mathit{else}\ s_2'}$$

$$\text{(While)}\quad \frac{RS; HP; HV; HA \vdash s \rightsquigarrow s'}{RS; HP; HV; HA \vdash \mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s \rightsquigarrow \mathit{while}\ \{e', \alpha'\}\ le\ \mathit{do}\ s'}$$

$$\text{(Procedure)}\quad \frac{RS; HP; HV; HA \vdash s \rightsquigarrow s'}{RS; HP; HV; HA \vdash (\phi, M, \psi, s) \rightsquigarrow (\phi', M', \psi', s')}$$

$$\text{(StackFrame)}\quad \frac{RS; HP; HV; HA \vdash s \rightsquigarrow s'}{RS; HP; HV; HA \vdash (P, L, s) \rightsquigarrow (P, L, s')}$$

$$\text{(Thread1)}\quad \frac{\forall 1 \le j \le n.\ F_j = (P_j, L_j, s_j) \qquad 1 \le i \le n \qquad \forall 1 \le j \le n.\ P_j \in HP \Leftrightarrow j \le i \qquad \forall i < j \le n.\ RS; HP; HV; HA \vdash F_j \rightsquigarrow F_j'}{RS; HP; HV; HA; \mathit{true} \vdash (TL, F_1 \ldots F_{i+1} \ldots F_n) \rightsquigarrow (TL, F_{i+1}' \ldots F_n')}$$

$$\text{(Thread2)}\quad \frac{\begin{array}{c}\forall 1 \le j \le n.\ F_j = (P_j, L_j, s_j) \qquad 1 < i \le n \qquad \forall 1 \le j \le n.\ P_j \in HP \Leftrightarrow j < i \qquad \forall i < j \le n.\ RS; HP; HV; HA \vdash F_j \rightsquigarrow F_j' \\ P_{i-1} \in \mathit{dom}(RS) \qquad RS; HP; HV; HA \vdash s_i \rightsquigarrow s_i'\end{array}}{RS; HP; HV; HA; \mathit{false} \vdash (TL, F_1 \ldots F_n) \rightsquigarrow (TL, (P_i, L_i, \mathit{call}\ RS(P_{i-1}); s_i') \cdot F_{i+1}' \ldots F_n')}$$

$$\text{(Program)}\quad \frac{\begin{array}{c}\mathit{dom}(RS) \subseteq HP \qquad \mathit{cod}(RS) \cap HA = \{\} \qquad \forall (\rho, \alpha, m) \in \mathit{cod}((\mathit{Action} \setminus HA) \circ as).\ (\mathit{Acc}(\rho) \cup \mathit{Acc}(\alpha)) \cap HV = \emptyset \\ \forall A \in \mathit{Action} \setminus HA.\ as(A) = as'(A) \qquad \forall P \notin HP.\ RS; HP; HV; HA \vdash ps(P) \rightsquigarrow ps'(P) \qquad \forall g \in \mathit{Global} \setminus HV.\ G(g) = G'(g) \\ \forall (\rho, \alpha, m) \in \mathit{cod}(RS \circ as).\ (\mathit{Acc}(\rho) \cap \mathit{Local} = \emptyset) \wedge (\alpha = ((\exists\exists \mathit{Local}.\ \alpha) \wedge \mathit{Same}(\mathit{Local}))) \\ \forall 1 \le i \le n.\ RS; HP; HV; HA; b_i \vdash T_i \rightsquigarrow T_i' \qquad T_1' \ldots T_n' \rightsquigarrow \overrightarrow{T'}\end{array}}{RS; HP; HV; HA; b_1; \ldots; b_n \vdash (ps, as, ls, G, T_1 \ldots T_n) \rightsquigarrow (ps', as', ls', G', \overrightarrow{T'})}$$

Figure 9: Program transformation

HA. These include checks that $dom(RS)$ is a subset of HP, $cod(RS)$ is disjoint from HA, and the atomic actions that are available in the abstract program do not access any hidden variable in HV. Next, it connects actions, procedure bodies, and global state in the concrete and the abstract program; the hidden procedures, global variables, and actions are unconstrained in the abstract program. Next, it checks that every action in $cod(RS)$ has a gate that is independent of procedure-local variables and a transition relation that leaves procedure-local variables unchanged. This check is meaningful because the action $RS(P)$ is written from the perspective of the caller of P. Since the procedure-local variables of the caller of P are distinct from those of P, these variables are quantified out from the transition relation of $RS(P)$ when the body of P is being checked (see Section 4.2.2). Finally, this rule rewrites the code of each thread in the program using the auxiliary judgment $\overrightarrow{T} \rightsquigarrow \overrightarrow{T'}$ that rewrites $\overrightarrow{T}$ by removing all empty threads from it.

$$\frac{}{\epsilon \rightsquigarrow \epsilon} \qquad \frac{\overrightarrow{T} \rightsquigarrow \overrightarrow{T'}}{\overrightarrow{T} \cdot (TL, \epsilon) \rightsquigarrow \overrightarrow{T'}} \qquad \frac{\overrightarrow{F} \neq \epsilon \qquad \overrightarrow{T} \rightsquigarrow \overrightarrow{T'}}{\overrightarrow{T} \cdot (TL, \overrightarrow{F}) \rightsquigarrow \overrightarrow{T'} \cdot (TL, \overrightarrow{F})}$$

Rules Thread1 and Thread2 together rewrite a thread using the judgment $RS; HP; HV; HA; b_i \vdash T_i \rightsquigarrow T'_i$. The verification rules justifying the correctness of the program rewriting transformation, described in Section 4.2.2, guarantee that any stack frames for procedures in HP are located contiguously at the top of the stack. Furthermore, this substack at the top either entirely consists of procedures in $HP \setminus dom(RS)$ or it has exactly one procedure $P \in dom(RS)$ at its bottom. Both these cases have to be appropriately simulated by the rewritten program. In the former case, there is certainly no pending atomic action in the abstraction. This case is handled by Thread1 by erasing the sub-stack. In the latter case, if b_i is false then Thread1 erases the sub-stack; otherwise, if b_i is true then Thread2 replaces the sub-stack with *call* $RS(P)$.

We discuss the remainder of the rules starting from the top of the figure. Rule Atomic ensures that the abstract program only uses actions not in HA but otherwise leaves *call* A unchanged. Rule Yield allows the annotations associated with the yield statement to be rewritten arbitrarily. Rules Ablock1 and Ablock2 together allow atomic blocks in the concrete program to be eliminated and new atomic blocks in the abstract programs to be created. Rules Call1, Call2, and Call3 together handle *call* P, leaving the statement unchanged if $P \notin HP$, rewriting to *skip* if $P \in HP \setminus dom(RS)$, and rewriting to *call* $RS(P)$ if $P \in dom(RS)$. Rules Async1 and Async2 together handle *async* P and are similar to rules Call1 and Call2 respectively. Rules Seq, Ite, and While rewrite the statement recursively; the loop annotation in While may be rewritten arbitrarily. Rule Procedure rewrites the body of a procedure recursively and allows the procedure specifications to be rewritten arbitrarily. Rules StackFrame rewrites only the statement in the stack frame and leaves other components unchanged.

4.2.2 Refinement

In this section, we formalize the judgments $RS; HP; HV; \overrightarrow{b} \vdash_r Prog$ and $InterferenceSafe(Prog)$. Together these judgments check that the body of each procedure P in $dom(RS)$ behaves like the atomic action $RS(P)$. Our strategy for P is to check that along all control paths in its body, there occurs exactly one atomic block *ablock* $\{e, \lambda\}$ s that is simulated by the atomic action $RS(P)$. All other atomic blocks before or after this unique block are simulated by the transition relation $Havoc(HV \cup Local)$ which allows procedure-local and hidden global variables to be modified arbitrarily but requires that other global variables and thread-local variables are not modified. In addition, these judgments also check that each atomic block inside a procedure in $HP \setminus dom(RS)$ is simulated by the transition relation $Havoc(HV \cup Local)$. In general, the appropriate simulation check may not be provable by examining only the body s of the atomic block. Contextual information such as the predicate e that is expected to hold when the atomic block begins execution may also be required. Annotations such as preconditions and postconditions, loop invariants, and yield predicates are leveraged to prove such predicates throughout the program.

The crux of the rules in Figure 10 is the statement-

(Skip)

$$\frac{}{RS; HP; HV; ps; as; P; a \vdash_r \{b, \phi\}\mathit{skip}\{b, \phi\}; \{\}}$$

(Atomic)

$$\frac{\begin{array}{c} as(A) = (\rho, \alpha, m) \qquad M \subseteq \mathit{ThreadLocal} \cup \mathit{Local} \qquad (\phi \wedge \rho) \circ \alpha \subseteq \mathit{Havoc}(\mathit{Global} \cup M) \\ P \in HP \Rightarrow \phi \subseteq \rho \qquad \mathit{cod}((\phi \wedge \rho) \circ \alpha) \subseteq \psi \end{array}}{RS; HP; HV; ps; as; P; \mathrm{a}^{+} \vdash_r \{b, \phi\}\mathit{call}\ A\{b, \psi\}; M}$$

(Yield)

$$\frac{}{RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{b, e\}\mathit{yield}\ e, \lambda\{b, e\}; \{\}}$$

(Ablock1)

$$\frac{\begin{array}{c} P \notin HP \\ RS; HP; HV; ps; as; P; \mathrm{a}^{+} \vdash_r \{b, \phi_1 \wedge e\}s\{b, \phi_2\}; M \end{array}}{RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{b, \phi_1 \wedge e\}\mathit{ablock}\ \{e, \lambda\}\ s\{b, \phi_2\}; M}$$

(Ablock2)

$$\frac{\begin{array}{c} P \in HP \qquad [s] \subseteq e \rightarrow \mathit{Havoc}(HV \cup \mathit{Local}) \\ RS; HP; HV; ps; as; P; \mathrm{a}^{+} \vdash_r \{b, \phi_1 \wedge e\}s\{b, \phi_2\}; M \end{array}}{RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{b, \phi_1 \wedge e\}\mathit{ablock}\ \{e, \lambda\}\ s\{b, \phi_2\}; M}$$

(Ablock3)

$$\frac{P \in \mathit{dom}(RS) \qquad as(RS(P)) = (\rho, \alpha, m) \qquad [s] \subseteq e \rightarrow \exists\exists \mathit{Local}.\ \alpha \qquad RS; HP; HV; ps; as; P; \mathrm{a}^{+} \vdash_r \{b, \phi_1 \wedge e\}s\{b, \phi_2\}; M}{RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{\mathit{false}, \phi_1 \wedge e\}\mathit{ablock}\ \{e, \lambda\}\ s\{\mathit{true}, \phi_2\}; M}$$

(Call)

$$\frac{P \in HP \Rightarrow P' \in HP \setminus \mathit{dom}(RS) \qquad ps(P') = (\phi, M, \psi, s)}{RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{b, \phi\}\mathit{call}\ P'\{b, \psi\}; M}$$

(Async)

$$\frac{P \in HP \Rightarrow P' \in HP \setminus \mathit{dom}(RS) \qquad ps(P') = (\phi, M, \psi, s)}{RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{b, \phi\}\mathit{async}\ P'\{b, \phi\}; \{\}}$$

(Seq)

$$\frac{\begin{array}{c} RS; HP; HV; ps; as; P; a \vdash_r \{b_1, \phi_1\}s_1\{b_2, \phi_2\}; M_1 \\ RS; HP; HV; ps; as; P; a \vdash_r \{b_2, \phi_2\}s_2\{b_3, \phi_3\}; M_2 \end{array}}{RS; HP; HV; ps; as; P; a \vdash_r \{b_1, \phi_1\}s_1; s_2\{b_3, \phi_3\}; M_1 \cup M_2}$$

(Ite)

$$\frac{\begin{array}{c} RS; HP; HV; ps; as; P; a \vdash_r \{b, le \wedge \phi_1\}s_1\{b', \phi_2\}; M_1 \\ RS; HP; HV; ps; as; P; a \vdash_r \{b, \neg le \wedge \phi_1\}s_2\{b', \phi_2\}; M_2 \end{array}}{RS; HP; HV; ps; as; P; a \vdash_r \{b, \phi_1\}\mathit{if}\ le\ \mathit{then}\ s_1\ \mathit{else}\ s_2\{b', \phi_2\}; M_1 \cup M_2}$$

(While)

$$\frac{RS; HP; HV; ps; as; P; a \vdash_r \{b, e \wedge le\}s\{b, e\}; M}{RS; HP; HV; ps; as; P; a \vdash_r \{b, e\}\mathit{while}\ \{e, \alpha\}\ le\ \mathit{do}\ s\{b, e \wedge \neg le\}; M}$$

(Weaken)

$$\frac{\begin{array}{c} \phi \subseteq \phi' \qquad \psi' \subseteq \psi \\ RS; HP; HV; ps; as; P; a \vdash_r \{b, \phi'\}s\{b', \psi'\}; M \end{array}}{RS; HP; HV; ps; as; P; a \vdash_r \{b, \phi\}s\{b', \psi\}; M}$$

(Frame1)

$$\frac{\begin{array}{c} RS; HP; HV; ps; as; P; a \vdash_r \{b, \phi\}s\{b', \psi\}; M \\ \mathit{Acc}(\rho) \cap (\mathit{Global} \cup M) = \{\} \end{array}}{RS; HP; HV; ps; as; P; a \vdash_r \{b, \rho \wedge \phi\}s\{b', \rho \wedge \psi\}; M}$$

(Frame2)

$$\frac{\begin{array}{c} RS; HP; HV; ps; as \vdash_r \{b, \phi\}F\{b', \psi\}; M \\ \mathit{Acc}(\rho) \cap (\mathit{Global} \cup \mathit{Local} \cup M) = \{\} \end{array}}{RS; HP; HV; ps; as \vdash_r \{b, \rho \wedge \phi\}F\{b', \rho \wedge \psi\}; M}$$

(Procedure)

$$\frac{\begin{array}{c} ps(P) = (\phi, M, \psi, s) \qquad (\mathit{Acc}(\phi) \cup \mathit{Acc}(\psi)) \cap \mathit{Local} = \{\} \\ M' \cap \mathit{ThreadLocal} \subseteq M \\ RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{P \notin \mathit{dom}(RS), \phi\}s\{\mathit{true}, \psi\}; M' \end{array}}{RS; HP; HV; ps; as \vdash_r P}$$

(StackFrame)

$$\frac{\begin{array}{c} RS; HP; HV; ps; as; P; \mathrm{a}^{-} \vdash_r \{b, \phi'\}s\{b', \psi\}; M \\ \forall G, TL, L'.\ G \cdot TL \cdot L' \in \phi \Rightarrow G \cdot TL \cdot L \in \phi' \end{array}}{RS; HP; HV; ps; as \vdash_r \{b, \phi\}(P, L, s)\{b', \psi\}; M \cap \mathit{ThreadLocal}}$$

(Thread)

$$\frac{\begin{array}{c} \forall 1 \leq i \leq n.\ \mathit{Acc}(\phi_i) \cap \mathit{Local} = \{\} \qquad \forall L'.\ G \cdot TL \cdot L' \in \phi_1 \\ \forall 1 \leq i \leq n.\ RS; HP; HV; ps; as \vdash_r \{b_i, \phi_i\}F_i\{b_{i+1}, \phi_{i+1}\}; M_i \end{array}}{RS; HP; HV; ps; as; G; b_1 \vdash_r (TL, F_1 \ldots F_n)}$$

(Program)

$$\frac{\begin{array}{c} \forall P \in \mathit{Proc}.\ RS; HP; HV; ps; as \vdash_r P \\ \forall 1 \leq i \leq n.\ RS; HP; HV; ps; as; G; b_i \vdash_r T_i \end{array}}{RS; HP; HV; b_1; \ldots; b_n \vdash_r (ps, as, ls, G, T_1 \ldots T_n)}$$

Figure 10: Refinement

level judgment $RS; HP; HV; ps; as; P; a \vdash_r \{b, \phi\} s \{b', \psi\}; M$. In this judgment, P is the procedure being checked. The argument a is either a^- which indicates that s is outside any atomic block or a^+ which indicates that s is inside some atomic block. This information enables the verification rules to check two properties. First, each atomic block must not contain any other atomic block, yield, or (ordinary and asynchronous) call statements inside it, ensuring that the computation of each *ablock* $\{e, \lambda\}$ s' can be summarized by a transition relation, denoted by $[s']$. Second, any invocation of an atomic action (the only computation that can modify the store) must occur inside an atomic block. This check ensures that checking non-interference of a yield predicate against all atomic blocks concurrently executing in the environment will preserve the yield predicate across the entire computation that happens in the environment during a context switch.

On the right side of the judgment, we have a generalization of the Floyd-Hoare triple—$\{b, \phi\} s \{b', \psi\}$. It indicates that s is being verified assuming it executes from a state in ϕ and ensuring it ends in a state satisfying ψ. The *Boolean* values b and b' are used to track refinement checking, which is performed only if $P \in HP$. If b is *true*, then each atomic block in s is expected to be simulated by $Havoc(HV \cup Local)$ and b' remains *true*. A statement in which all atomic blocks refine $Havoc(HV \cup Local)$ is said to *stutter*. If b is *false*, then it must be the case that $P \in dom(RS)$, at most one atomic block in s is simulated by $RS(P)$ and all other blocks are simulated by $Havoc(HV \cup Local)$. The value of b' is allowed to be *true* if some atomic block in s is simulated by $RS(P)$. In the discussion below, we will refer to the pair (b, ϕ) as the precondition and (b', ψ) as the postcondition of s, respectively. Finally, the set M contains thread-local and procedure-local variables that are potentially modified by s; this information is useful in the frame rules, Frame1 and Frame2, explained later.

We now examine the rules for the the judgment $RS; HP; HV; \overrightarrow{b} \vdash_r Prog$ in Figure 10, starting from the bottom. In this judgment, the vector $\overrightarrow{b}$ contains a *Boolean* value b_i for each thread T_i. Rule Program checks each thread and each procedure separately. The value b_i is passed to the judgment for checking a thread. The rule Thread checks each frame on the stack of a thread modularly using a sequence of $b_1, \ldots, b_{n+1}$ of *Boolean* values and a sequence $\phi_1, \ldots, \phi_{n+1}$ of predicates that may not refer to procedure-local variables. The pair (b_i, ϕ_i) acts as the precondition to the i-the stack frame (from the top) of the thread. The rule StackFrame checks a stack frame (P, L, s) by checking s recursively. The precondition ϕ of the stack frame may be specialized to the local state L to obtain the precondition for s. The rule Procedure checks the body of the procedure P with the precondition $(P \notin dom(RS), \phi)$ and postcondition $(true, \psi)$, where ϕ and ψ are the precondition and postconditions of P respectively. This check ensures that if P has an atomic action specification, the body refines it appropriately. The rule also checks that the body of P modifies only those thread-local variables mentioned in the specification of P and that the precondition and postcondition of P does not refer to procedure-local variables.

The rules Weaken, Frame1, and Frame2 are standard. The rules While, Ite, and Seq extend the analogous Floyd-Hoare rules with refinement checking. The rule While checks that the body of the loop stutters and concludes that the loop itself stutters. The rule Ite checks that both branches behave in the same way. The rule Seq for $s_1; s_2$ chains the refinement behavior of s_1 with that of s_2.

Rules Call and Async handle procedure calls. Both rules require that if the procedure P being checked is in $dom(RS)$, then the called procedure P' must be in $HP \setminus dom(RS)$. Thus, the call relation over the set of procedures being introduced during refinement (or hidden during abstraction) must be a subset of $HP \times (HP \setminus dom(RS))$. This check is important for ensuring that *call* P can be rewritten either to *call* $RS(P)$ if $P \in dom(RS)$ or to *skip* if $P \in HP \setminus dom(RS)$.

Rules Ablock1, Ablock2, and Ablock3 handle an atomic block *ablock* $\{e, \lambda\}$ s. Rule Ablock1 considers the case when $P \notin HP$ and performs only the standard sequential correctness check. Rules Ablock2 and Ablock3 consider the two cases when $P \in HP$ and handle the refinement of

$HavocHV \cup Local$ and the atomic action $RS(P)$, respectively. Rule Yield handles the yield statement; only the yield predicate is available in the postcondition of the statement.

Rule Atomic handles the call of an atomic action. It computes in M the set of thread-local and procedure-local variables possibly modified by the action. It checks that the gate of the action holds but only when the call occurs inside the body of a procedure in HP. In other words, a gate of an atomic action A, the only source of safety assertions in our operational semantics, is discharged only when A is invoked inside a procedure that is hidden as a result of abstraction. Since we are only interested in proving a simulation relation between the concrete and abstract semantics, it is safe to leave unverified the gates at all other invocations of atomic actions which are present in both the concrete and abstract program. Finally, this rule verifies the postcondition ψ from the precondition ϕ using the semantics of the atomic action.

We now move on to discuss the judgment $\mathit{InterferenceSafe}(\mathit{Prog})$, where $\mathit{Prog} = (ps, as, ls, G, \overrightarrow{T})$. Let $\mathit{Yields}(\mathit{Prog})$ be the union of the following sets:

- $\{(\phi, \lambda) \mid \mathit{yield}\ \phi, \lambda$ appears in $\mathit{Prog}\}$.
- $\{(\phi, \lambda) \mid P \in \mathit{Proc} \wedge ps(P) = (\phi, M, \psi, s) \wedge ls(P) = (\lambda, \lambda')\}$.
- $\{(\psi, \lambda') \mid P \in \mathit{Proc} \wedge ps(P) = (\phi, M, \psi, s) \wedge ls(P) = (\lambda, \lambda')\}$.

Let $\mathit{Ablocks}(\mathit{Prog})$ be the set of atomic blocks in Prog. The program Prog is interference-free, denoted by $\mathit{InterferenceSafe}(\mathit{Prog})$, if for each predicate $(\phi, \lambda_y) \in \mathit{Yields}(\mathit{Prog})$ and for each atomic block $\mathit{ablock}\ \{e, \lambda\}\ s \in \mathit{Ablocks}(\mathit{Prog})$, we have

$$\begin{array}{l}
\forall G, TL, L, G', TL', L', TL_y, L_y. \\
\begin{array}{lll}
\mathit{let}\ \Lambda = & \begin{array}{ll} \cup & \mathit{Collect}(G, \lambda) \\ \cup & \mathit{Collect}(TL, \lambda) \\ \cup & \mathit{Collect}(TL_y, \lambda_y) \end{array} & \\
& \begin{array}{ll} \wedge & \mathit{IsSet}(\Lambda) \\ \wedge & G \cdot TL \cdot L \in e \\ \wedge & (G \cdot TL \cdot L, G' \cdot TL' \cdot L') \in [s] \\ \wedge & G \cdot TL_y \cdot L_y \in \phi \end{array} & \Rightarrow G' \cdot TL_y \cdot L_y \in \phi. \\
\mathit{in} & &
\end{array}
\end{array}$$

We summarize the contributions of the judgments discussed in this section in Lemma 4 and Lemma 5. Both lemmas talk about Prog and its abstraction Prog' and assume that Prog' is safe. Lemma 4 states that Prog cannot fail while executing from the current yield point to the next. Lemma 5 states that the yield-to-yield step taken by Prog to get to Prog_1 can be simulated by zero or more steps taken by Prog' to get to Prog'_1 such that Prog_1 is abstracted by Prog'_1. Together, these lemmas establish the simulation relation between Prog and Prog', and allows us to conclude $\mathit{CSafe} * (\mathit{Prog})$.

Lemma 4 *Let* $\mathit{Prog} \in \mathit{CProgram}$, $\mathit{Prog}' \in \mathit{Program}$, $RS \in \mathit{Proc} \rightharpoonup \mathit{Action}$, $HP \in 2^{\mathit{Proc}}$, $HV \in 2^{\mathit{Global}}$, $HA \in 2^{\mathit{Action}}$, *and* $\overrightarrow{b} \in \overrightarrow{\mathit{Boolean}}$ *be such that* $\vdash_l \mathit{Prog}$, $\mathit{Safe}^*(\mathit{Prog}')$, $RS; HP; HV; HA; \overrightarrow{b} \vdash \mathit{Prog} \rightsquigarrow \mathit{Prog}'$, $RS; HP; HV; \overrightarrow{b} \vdash_r \mathit{Prog}$, *and* $\mathit{InterferenceSafe}(\mathit{Prog})$. *Then, we have* $\mathit{CSafe}(\mathit{Prog})$.

Proof idea: Let us pick an arbitrary thread to execute from Prog. Since $\mathit{Prog} \in \mathit{CProgram}$, this thread is waiting to execute at a yield point, i.e., yield statement or a call or a return. As it executes from the current yield point to the next yield point, it executes a sequence of atomic blocks, all which must be inside the same procedure, say P, by our definition of yield point. Since each invocation of an atomic action must reside inside an atomic block, failure can only occur during the execution of one of these atomic blocks. We need to prove that there is no failure during this execution. If $P \notin HP$, the execution in Prog can be simulated by an identical execution in Prog'. Since $\mathit{Safe}^*(\mathit{Prog}')$, we conclude that $\mathit{CSafe}(\mathit{Prog})$. If $P \in HP$, we note from rule Atomic for *call* A that the gate of A is discharged. Therefore, in this case also we conclude that $\mathit{CSafe}(\mathit{Prog})$. ■

Lemma 5 *Let* $\mathit{Prog}, \mathit{Prog}_1 \in \mathit{CProgram}$, $\mathit{Prog}' \in \mathit{Program}$, $RS \in \mathit{Proc} \rightharpoonup \mathit{Action}$, $HP \in 2^{\mathit{Proc}}$, $HV \in 2^{\mathit{Global}}$, $HA \in 2^{\mathit{Action}}$, *and* $\overrightarrow{b} \in \overrightarrow{\mathit{Boolean}}$ *be such that* $\vdash_l \mathit{Prog}$, $\mathit{Safe}^*(\mathit{Prog}')$, $RS; HP; HV; HA; \overrightarrow{b} \vdash \mathit{Prog} \rightsquigarrow \mathit{Prog}'$, $RS; HP; HV; \overrightarrow{b} \vdash_r \mathit{Prog}$, $\mathit{InterferenceSafe}(\mathit{Prog})$, *and* $\mathit{Prog} \longmapsto \mathit{Prog}_1$. *Then, there exists* $\mathit{Prog}'_1 \in \mathit{Program}$ *and* $\overrightarrow{b_1} \in \overrightarrow{\mathit{Boolean}}$ *such that* $RS; HP; HV; HA; \overrightarrow{b_1} \vdash \mathit{Prog}_1 \rightsquigarrow \mathit{Prog}'_1$, $RS; HP; HV; \overrightarrow{b_1} \vdash_r \mathit{Prog}_1$, $\mathit{InterferenceSafe}(\mathit{Prog}_1)$, *and* $\mathit{Prog}' \longrightarrow^* \mathit{Prog}'_1$.

Proof idea: Suppose thread i executed from a yield point to the next yield point in moving from *Prog* to $Prog_1$, executing a sequence of atomic blocks all which were in the same procedure, say P. We first show how to pick the values $\overrightarrow{b_1}$. For threads j other than i, we pick $\overrightarrow{b}(j)$. For thread i, we perform a case analysis. If the next yield point is a yield statement, we pick the first component of the precondition (b, ϕ) of the yield statement from the proof tree for $RS; HP; HV; \overrightarrow{b} \vdash_r Prog$. If the next yield point is *call* P' and $P' \in dom(RS)$, we pick *false*. If the next yield point is *call* P' and $P' \notin dom(RS)$, we pick the first component of the precondition (b, ϕ) of the call statement from the proof tree for $RS; HP; HV; \overrightarrow{b} \vdash_r Prog$. If the next yield point is return from a procedure $P \in dom(RS)$, we pick *true*. If the next yield point is return from a procedure $P \notin dom(RS)$, we pick $\overrightarrow{b}(i)$.

Next, we show how to simulate the execution from *Prog* to $Prog_1$ with an execution from $Prog'$ to $Prog'_1$. Suppose $\overrightarrow{b}(i) = true$. In this case, rule THREAD1 must have been used. Then it must be the case that in rule THREAD, the precondition of all stack frames of thread i must have the *Boolean* value *true*. Suppose the topmost stack frame is (P, L, s). If $P \in HP$, then the execution from *Prog* to $Prog_1$ must not have modified any variables except those in *Local* and HV, $Prog'$ does not have to move, and $Prog' = Prog'_1$. If $P \notin HP$, then the stacks of thread i in *Prog* and $Prog'$ look the same except for the subsitution applied to calls; the execution from *Prog* to $Prog_1$ can be simulated by an identical execution from $Prog'$ to $Prog'_1$.

Suppose $\overrightarrow{b}(i) = false$. In this case, rule THREAD2 must have been used. Suppose the topmost stack frame is (P, L, s); we know that $P \in HP$. If $P \in HP \backslash dom(RS)$, then the execution from *Prog* to $Prog_1$ must not have modified any variables except those in *Local* and HV, $Prog'$ does not have to move, and $Prog' = Prog'_1$. If $P \in dom(RS)$, then there are two cases for picking the *Boolean* value for thread i (see first paragraph). If we picked *false*, the execution from *Prog* to $Prog_1$ must not have modified any variables except those in *Local* and HV, $Prog'$ does not have to move, and $Prog' = Prog'_1$. If we picked *true*, some atomic block in the execution must have behaved like the atomic action $RS(P)$. We can simulate this behavior by stepping through *call* $RS(P)$ inserted in the stack frame just below the top. ■

4.3 From preemptive to cooperative semantics

In this section, we outline the verification required for justifying that it suffices to reason about the cooperative semantics of *Prog*, which ensures that $CSafe^*(Prog)$ is sufficient for $Safe^*(Prog)$ Let $Prog = (ps, as, ls, G, \overrightarrow{T})$ be a program. The map *as* maps each atomic action to a triple (ρ, α, m), the last component of which denotes type of atomic action—B for *both mover*, R for *right mover*, L for *left mover*, and N for *non mover*. Informally, an action labeled N does not commute with other concurrent actions, an action labeled L commutes to the left (or earlier in time) of other concurrent actions, an action labeled R commutes to the right (or later in time) of other concurrent actions, and an action labeled B commutes both to the left and the right of other concurrent actions. The *Yield Sufficiency Automaton* (*YSA*) from Figure 3 encodes all sequences of atomic actions and yields for which reasoning about cooperative semantics is sufficient. For example, the *YSA* automaton indicates that a yield after a right mover or a yield before a left mover is unnecessary. The judgment $\vdash_y Prog$ checks that the code of *Prog* can be simulated by *YSA*; it depends on the judgment *CommutativitySafe*(*Prog*) which checks that all mover annotations are correct.

The essence of the rules in Figure 11 is to capture the effect of a computation as a pair (x, y) where $x, y \in \{RM, LM\}$; the meaning is that to simulate the computation the automaton moves from state x to state y. Rule PROGRAM performs checking for all threads and for all procedures. Rules THREAD, STACKFRAME, and PROCEDURE are straightforward. Rule WHILE checks that the body of the loop leaves the state of the automaton unchanged. Rule ITE checks that the effect both branches is the same. Rule SEQ composes the effect of s_1 with the effect of s_2.

$$\text{(Skip)}\ \frac{}{as \vdash_y skip : (x,x)} \qquad \text{(Both)}\ \frac{as(A) = (\rho, \alpha, B)}{as \vdash_y call\ A : (x,x)} \qquad \text{(Right)}\ \frac{as(A) = (\rho, \alpha, R)}{as \vdash_y call\ A : (RM, RM)} \qquad \text{(Left)}\ \frac{as(A) = (\rho, \alpha, L)}{as \vdash_y call\ A : (x, LM)} \qquad \text{(Non)}\ \frac{as(A) = (\rho, \alpha, N)}{as \vdash_y call\ A : (RM, LM)}$$

$$\text{(Yield)}\ \frac{}{as \vdash_y yield\ e, \lambda : (x, RM)} \qquad \text{(Ablock)}\ \frac{as \vdash_y s : (x,y)}{as \vdash_y ablock\ \{e, \lambda\}\ s : (x,y)} \qquad \text{(Call)}\ \frac{}{as \vdash_y call\ P : (x, RM)} \qquad \text{(Async)}\ \frac{}{as \vdash_y async\ P : (x, LM)}$$

$$\text{(Seq)}\ \frac{as \vdash_y s_1 : (x,y) \qquad as \vdash_y s_2 : (y,z)}{as \vdash_y s_1; s_2 : (x,z)} \qquad \text{(Ite)}\ \frac{as \vdash_y s_1 : (x,y) \qquad as \vdash_y s_2 : (x,y)}{as \vdash_y if\ le\ then\ s_1\ else\ s_2 : (x,y)} \qquad \text{(While)}\ \frac{as \vdash_y s : (x,x)}{as \vdash_y while\ \{e, \alpha\}\ le\ do\ s : (x,x)}$$

$$\text{(Procedure)}\ \frac{ps(P) = (\phi, M, \psi, s) \qquad as \vdash_y s : (x,y)}{as \vdash_y P} \qquad \text{(StackFrame)}\ \frac{as \vdash_y s : (x,y)}{as \vdash_y (P, L, s) : (x,y)} \qquad \text{(Thread)}\ \frac{\forall 1 \leq i \leq n.\ as \vdash_y F_i : (x_i, x_{i+1})}{as \vdash_y (TL, F_1 \ldots F_n)}$$

$$\text{(Program)}\ \frac{\forall P \in Proc.\ as \vdash_y P \qquad \forall 1 \leq i \leq n.\ as \vdash_y T_i}{\vdash_y (ps, as, ls, G, T_1 \ldots T_n)}$$

Figure 11: Yield sufficiency

Rule Async is interesting because it treats an asynchronous call as a left mover. Rules Yield and Call treat their statements as a yield. Rule Ablock calculates the effect of the body of the block. Rules Both, Right, Left, and Non essentially examine the available edges in the automaton to validate the statement and calculate its effect. Rule Skip leave the state of the automaton unchanged.

The judgment $\vdash_y$ *Prog* is not quite enough to check the sufficiency of yields soundly. Consider the following program with a global variable x and two threads.

```
   [ x := 0 ];           ||   [ x := x + 1 ];
   [ assert x == 0 ];         while (true) ;
```

This program violates the assertion in it. The first thread sets x to 0 in the atomic action indicated [x := 0]; the second thread preempts the first thread and changes x to 1; the first thread preempts the second thread and fails the assert x == 0. However, the program does not fail if the second thread is executed without preemption, even though the code of the second thread satisfies the yield sufficiency check described earlier. Our verification method is applicable to a program *Prog* only if $Cooperative^*(Prog)$ holds, which is false for the program in the example above. Verifying this condition is similar to proving termination and is orthogonal to the contribution of this paper.

Of course, the judgment $\vdash_y$ *Prog* depends on the mover annotations on the atomic actions being correct. We capture this requirement in the judgmen *CommutativitySafe*(*Prog*). Formally, the program *Prog* is commutativity-safe, denoted by *CommutativitySafe*(*Prog*), if for all $A_1, A_2 \in$ *Action* such that $as(A_1) = (\rho_1, \alpha_1, m_1)$, $as(A_2) = (\rho_2, \alpha_2, m_2)$, $ls(A_1) = (\lambda_1, \lambda'_1)$, and $ls(A_2) = (\lambda_2, \lambda'_2)$, then all of the following conditions are satisfied:

- **Commutativity.** This condition checks that if action A_1 is a right mover or action A_2 is a left mover, then the effect of executing A_1 followed by A_2 in two different threads can be achieved by executing A_2 followed by A_1. Formally, if

$m_1 \in \{B, R\}$ or $m_2 \in \{B, L\}$, then

$$
\begin{array}{l}
\forall G^-, G, G^+, TL_1, L_1, TL'_1, L'_1, TL_2, L_2, TL'_2, L'_2.\ \exists G'. \\
\begin{array}{lll}
\mathit{let}\ \Lambda = & \cup & \mathit{Collect}(G^-, \lambda_1) \\
 & \cup & \mathit{Collect}(TL_1, \lambda_1) \\
 & \cup & \mathit{Collect}(TL_2, \lambda_2)
\end{array} \\
\mathit{in} \\
\quad \begin{array}{ll}
\wedge & \mathit{IsSet}(\Lambda) \\
\wedge & G^- \cdot TL_1 \cdot L_1 \in \rho_1 \\
\wedge & G^- \cdot TL_2 \cdot L_2 \in \rho_2 \\
\wedge & (G^- \cdot TL_1 \cdot L_1, G \cdot TL'_1 \cdot L'_1) \in \alpha_1 \\
\wedge & (G \cdot TL_2 \cdot L_2, G^+ \cdot TL'_2 \cdot L'_2) \in \alpha_2
\end{array} \\
\Rightarrow \\
\quad \begin{array}{ll}
\wedge & (G^- \cdot TL_2 \cdot L_2, G' \cdot TL'_2 \cdot L'_2) \in \alpha_2 \\
\wedge & (G' \cdot TL_1 \cdot L_1, G^+ \cdot TL'_1 \cdot L'_1) \in \alpha_1
\end{array}
\end{array}
$$

- **Forward preservation.** This condition checks that if A_1 is a right mover or A_2 is a left mover, the failure of A_2 immediately after the execution of A_1 is sufficient to imply that A_2 must also fail before the exection of A_1. This condition is equivalent to forward preservation of the gate of A_2 by the execution of A_1. Formally, if $m_1 \in \{B, R\}$ or $m_2 \in \{B, L\}$, then

$$
\begin{array}{l}
\forall G, G', TL_1, L_1, TL'_1, L'_1, TL_2, L_2. \\
\begin{array}{lll}
\mathit{let}\ \Lambda = & \cup & \mathit{Collect}(G, \lambda_1) \\
 & \cup & \mathit{Collect}(TL_1, \lambda_1) \\
 & \cup & \mathit{Collect}(TL_2, \lambda_2)
\end{array} \\
\mathit{in} \\
\quad \begin{array}{ll}
\wedge & \mathit{IsSet}(\Lambda) \\
\wedge & G \cdot TL_1 \cdot L_1 \in \rho_1 \\
\wedge & (G \cdot TL_1 \cdot L_1, G' \cdot TL'_1 \cdot L'_1) \in \alpha_1 \\
\wedge & G \cdot TL_2 \cdot L_2 \in \rho_2
\end{array} \quad \Rightarrow \quad G' \cdot TL_2 \cdot L_2 \in \rho_2
\end{array}
$$

- **Backward preservation.** This condition checks that if A_1 is a left mover and A_2 is an arbitrary action, then the failure of A_2 immediately before the execution of A_1 is sufficient to imply that A_2 must also fail immediately after the execution of A_1. This condition is equivalent to backward preservation of the gate of A_2 by the execution of A_1. Formally, if $m_1 \in \{B, L\}$, then

$$
\begin{array}{l}
\forall G, G', TL_1, L_1, TL'_1, L'_1, TL_2, L_2. \\
\begin{array}{lll}
\mathit{let}\ \Lambda = & \cup & \mathit{Collect}(G, \lambda_1) \\
 & \cup & \mathit{Collect}(TL_1, \lambda_1) \\
 & \cup & \mathit{Collect}(TL_2, \lambda_2)
\end{array} \\
\mathit{in} \\
\quad \begin{array}{ll}
\wedge & \mathit{IsSet}(\Lambda) \\
\wedge & G \cdot TL_1 \cdot L_1 \in \rho_1 \\
\wedge & (G \cdot TL_1 \cdot L_1, G' \cdot TL'_1 \cdot L'_1) \in \alpha_1 \\
\wedge & G' \cdot TL_2 \cdot L_2 \in \rho_2
\end{array} \quad \Rightarrow \quad G \cdot TL_2 \cdot L_2 \in \rho_2
\end{array}
$$

- **Nonblocking.** This condition checks that if A_1 is a left mover, then it must be non-blocking. If $m_1 \in \{B, L\}$, then $\forall \sigma \in \rho_1.\ \exists \sigma'.\ (\sigma, \sigma') \in \alpha_1$.

Finally, we summarize the results of this section in the following lemma connecting the safety of preemptive and cooperative semantics of *Prog*.

Lemma 6 *Let Prog* $\in$ *CProgram be such that* $\vdash_l$ *Prog, CommutativitySafe(Prog), and* $\vdash_y$ *Prog. If Cooperative***(Prog) and CSafe***(Prog), then Safe***(Prog).*

Proof idea: The proof of this lemma is similar to other such proofs done in the literature [21, 17]. We have to show that if *Cooperative**(*Prog*) and a preemptive $\longrightarrow$-execution from *Prog* ends in *error*, then there is a cooperative $\longmapsto$-exection from *Prog* ending in *error*. We know that every thread in *Prog* starts execution from a yield point (yield statement or call or return) By checking simulation against *YSA*, it is also the case that any execution of a thread from a yield point to another yield point is a (possibly empty) sequence of right movers and local actions, followed by zero or one non mover, followed by a (possibly empty) sequence of left movers and local actions. A successful simulation check against *YSA* labels each statement with either *RM* or *LM*.

We pick a preemptive $\longrightarrow$-execution leading to *error* such that there is no shorter such execution. Starting from this execution, it is possible to swap adjacent steps by different threads while preserving the failure at the end of the execution. Each swap is one of the following: (1) moving an *RM*-labeled step by one thread after a step of another thread, or (2) moving an *LM*-labeled step by one thread before a step of another thread. Eventually, when no more swaps are possible, the final execution is such that all context switches from thread i to a different thread j happen at a yield point, but for the possible exception of the unique switch subsequent to which thread i never executes. If there exists such a switch from thread i, then it is also the case that the statement at which the switch happened is labeled with *LM*. This implies that it is possible to continue executing thread i via a sequence of left-moving steps.

In the next step, we patch these exceptional switches one by one. To patch the switch from thread i, we extend the execution of thread i at the end of the execution by a left mover and then commute it earlier by repeated swaps until it reaches the exceptional context switch. We repeat this process until thread i reaches a yield point; it is guaranteed to reach a yield point because of the $\mathit{Cooperative}^*(\mathit{Prog})$ property. This process is continued until all switches are patched resulting in an execution in which all context switches happen at yield points. ■

5 Implementation

We have implemented the method described in this section as a conservation extension of the Boogie [5] language and verifier. Our implementation provides new language primitives for linear variables, asynchronous and parallel procedure calls, yields, atomic actions as procedure specifications, expressing refinement layers, and hiding of global variables and procedures.

At its core, Boogie is an unstructured language comprising code blocks and goto statements. Our implementation handles the complexity of unstructured control flow. To simplify the exposition, our formalization uses Floyd-Hoare triples to present sequential correctness and annotated atomic code blocks to present refinement and non-interference checks. However, our implementation is considerably more automated. All the annotations, except those at yields, loops, and procedure boundaries, are automatically generated using the technique of verification conditions [6]. Annotated atomic code blocks are also inferred automatically.

Unlike the formalization in which non-interference checks are performed separately, our implementation inserts the non-interference checks for all yield statements inside the body of each source procedure. This strategy increases the precision of the verification since the entire context of the source procedure is available in the verification condition.

The judgment $\vdash_y$ *Prog* from Section 4.3 is fully automated. We adapted an algorithm by Henzinger et al.[30] for computing the similarity relation of labeled graphs to check that the control flow graph of a procedure is simulated by the *YSA* automaton. The complexity of the algorithm is $O(n*m)$, where n and m are the number of control-flow graph nodes and edges. In practice, this part of the verification is fast.

```
var x:int;

procedure p()
  requires x >= 5;
  ensures  x >= 8;
{
  yield x >= 5;  x := x + 1;
  yield x >= 6;  x := x + 1;
  yield x >= 7;  x := x + 1;
}
procedure q() { x := x + 3; }
```

Figure 12: Program 1

A large proof usually comprises multiple layers of refinement chained together. Our implementation allows the specification of multiple views of a program in a single file by using the mechanism of *layers*. The programmer may attach a positive layer number to each annotation and procedure; version i of the program is constructed from annotations labeled i and procedures labeled at least i. We have implemented a type checker to make sure that layer numbers are used appropriately, e.g., it is illegal for a procedure with layer i to call a procedure with layer j greater than i.

Our implementation also provides knobs for selectively verifying certain refinement layers or only the commutativity checks. The verification is fast in general; the presence of these knobs further reduces verification time to facilitate interactive program verification and development.

6 Examples

In this section, we present a collection of examples that illustrate specification and verification features supported by the CIVL verifier.

6.1 Reasoning about interference

We present an overview of the CIVL language and verifier through a sequence of examples. Figure 12 shows Program 1 containing a procedure `p` executing

```
procedure yield_x(n:int)
  requires x >= n;
  ensures  x >= n;
{
  yield x >= n;
}
procedure p()
  requires x >= 5;
  ensures  x >= 8;
{
  call yield_x(5);  x := x + 1;
  call yield_x(6);  x := x + 1;
  call yield_x(7);  x := x + 1;
}
```

Figure 13: Program 2

```
procedure yield_x()
  ensures  x >= old(x);
{
  yield x >= old(x);
}
procedure p()
  requires x >= 5;
  ensures  x >= 8;
{
  call yield_x();  x := x + 1;
  call yield_x();  x := x + 1;
  call yield_x();  x := x + 1;
}
```

Figure 14: Program 3

concurrently with another procedure q. As explained earlier, a CIVL program is verified with respect to its cooperative semantics; a thread explicitly yields control to the scheduler via the yield statement following which execution continues on a nondeterministically chosen thread. The yield statement has a predicate φ attached to it. The yielding thread must establish φ when it yields and the execution of other threads must preserve φ; these two requirements in Owicki-Gries-style reasoning [48] are usually known as *sequential correctness* and *non-interference*, respectively. To check these requirements, the CIVL verifier creates verification conditions, whose number is at most quadratic in the number of yield statements in the program. For example, in Program 1 each yield predicate in p must be checked against the action x := x + 3 in q.

From quadratic to linear verification conditions. Figure 13 shows Program 2, a variation of Program 1 in which the procedure yield_x contains a single yield statement and p calls yield_x instead of yielding directly. If the calls to yield_x are inlined in Program 2, then we will get Program 1. Both Program 1 and 2 are verifiable in CIVL but the cost of verifying Program 2 is less because it has fewer yield statements. In fact, if it is possible to capture all interference in a concurrent program in a single yield predicate, then the trick in Program 2 can be used to verify the program with a linear number of verification conditions.

Encoding rely-guarantee specifications. Figure 14 shows Program 3, yet another variation of Programs 1 and 2 which shows how to encode a rely-guarantee-style [36] (two-state invariant) proof using CIVL's one-state yield statements. The standard rely-guarantee specification to prove the assertions in p is that the environment of p may only increase x. We can encode this in CIVL by factoring out the yield statement in a separate procedure and then referring old(x), the value of x when yield_x is entered.

Parallel calls. Program 4 in Figure 15 illustrates parallel calls, supported by Boogie based on the standard Owicki-Gries rules for parallel composition of threads. The statement call incr_x() | yield_y() | yield_z() in p creates three threads executing incr_x, yield_y, and yield_z respectively, yields control to the scheduler, and blocks until all three threads have terminated. For a procedure to be invoked in a parallel call, its preconditions and postconditions must be stable against interference. This requirement ensures that it is safe to assume the precondition in the callee and the postcondition in the caller.

The threads created by p for yield_y and yield_z are not doing any interesting computation; their only purpose is to make available to their parent the conjunction of their respective postconditions (following

```
var x:int, y:int, z:int;

procedure incr_x()
  ensures  x >= old(x) + 1;
{
  yield x >= old(x);
  x := x + 1;
  yield x >= old(x) + 1;
}

procedure yield_y()
  ensures  y >= old(y);
{
  yield y >= old(y);
}

procedure yield_z()
  ensures  z >= old(z);
{
  yield z >= old(z);
}

procedure p()
  requires x == 3 && y == 5 && z == 7;
{
  call incr_x() | yield_y() | yield_z();
  assert x >= 4 && y >= 5 && z >= 7;
}
```

Figure 15: Program 4

Owicki-Gries rules for parallel composition). In this example, the postconditions of `yield_y` and `yield_z` preserve information about variables `y` and `z` that would otherwise be lost during the call to `incr_x`, whose postcondition only supplies information about `x` even though its yield statements potentially cause all global variables, including `y` and `z`, to change. This example demonstrates modular proof structuring by factoring out yield assertions into a collection of procedures; the declaration of `incr_x` can focus on changes to `x`, without having to explicitly preserve invariants about all other variables in the program.

```
type Tid;
const nil:Tid;
procedure Allocate()
  returns (linear tid:Tid);
  ensures tid != nil;

var a:[Tid]int;

procedure main()
{
  while (true) {
    var linear tid:Tid := Allocate();
    async call P(tid);
    yield true;
  }
}
procedure P(linear tid: Tid)
  requires tid != nil;
  ensures a[tid] == old(a)[tid] + 1;
{
  var t:int := a[tid];
  yield t == a[tid];
  a[tid] := t + 1;
}
```

Figure 16: Program 5

6.2 Linear variables

Program 5 in Figure 16 introduces linear variables, a feature of CIVL that is useful for encoding disjointness among values contained in different variables. This example uses this feature for encoding the concept of an identifier that is unique to each thread. Program 5 contains a shared global array `a` indexed by an uninterpreted type `Tid` representing the set of thread identifiers. A collection of threads are executing procedure `P` concurrently. The identifier of the thread executing `P` is passed in as the parameter `tid`. A thread with identifier `tid` owns `a[tid]` and can increment it without danger of interference. The yield predicate `t == a[tid]` in `P` indicates this expectation, yet it is not possible to prove it unless the reasoning engine knows that the value of `tid` in one thread is distinct its value in a different thread.

```
type lmap = LMap(dom:[int]bool,
                 map:[int]int);
const empty:lmap;
axiom empty.dom == {};

procedure Load(linear l:lmap, i:int)
  returns(v:int);
  requires l.dom[i];
  ensures v == l.map[i];

procedure Store(linear_in l_in:lmap,
                i:int, v:int)
  returns(linear l:lmap);
  requires l_in.dom[i];
  ensures l.dom == l_in.dom;
  ensures l.map == l_in.map[i := v];

procedure Move(linear_in l1_in:lmap)
  returns(linear l1:lmap, linear l2:lmap);
  ensures l1 == empty && l2 == l1_in;

procedure Transfer(linear_in l1_in:lmap,
                   linear_in l2_in:lmap,
                   i:int)
  returns(linear l1:lmap, linear l2:lmap);
  ensures l1.dom == l1_in.dom - {i};
  ensures l1.map == l1_in.map;
  ensures l2.dom == l2_in.dom + {i};
  ensures l2.map == l2_in.map[i:=l1.map[i]];
```

Figure 17: Encoding linear maps

```
var linear g:lmap;
var b:bool;

procedure P()
{
  var t: int;
  var linear l:lmap;
  while (b) { call Yield(); }
              b := true;
              call g, l := Move(g);
  call Yield(); call t := Load(l, p);
  call Yield(); call l := Store(l, p, t+1);
  call Yield(); call t := Load(l, p+4);
  call Yield(); call l := Store(l, p+4, t+1);
  call Yield(); call l, g := Move(l);
              b := false;
  call Yield();
}

procedure Yield()
{
  yield b || (g.dom == {p,p+4} &&
            g.map[p] == g.map[p+4]);
}
```

Figure 18: Program 6

Instead of building a notion of thread identifiers into CIVL, we provide a more primitive and general notion of linear variables. The CIVL type system ensures that values contained in linear variables cannot be duplicated. Consequently, the parameter `tid` of distinct concurrent calls to `P` are known to be distinct; the CIVL verifier exploits this invariant while checking for non-interference.

Linear maps. Figure 17 illustrates how linear maps [38] can be encoded in CIVL. A linear map `l` is a pair comprising `l.map`), an array of integers, and `l.dom`, a set of integers representing the locations where it is legal to access `l.map`. In the parlance of separation logic, a linear map can be thought of as a collection of *points-to* facts partitioned by separating conjunction; furthermore, there is an implicit separating conjuction between the values contained in two distinct linear maps. Figure 17 shows the primitive operations on linear maps. (The notation `linear_in` indicates that a procedure consumes an argument, making it unavailable to the caller after the call, while a parameter marked `linear` is only borrowed from the caller and remains available to the caller after the call.) If a program is written using only these primitive operations, it is guaranteed all occurrences of `Load` and `Store` can be erased to loads and stores, respectively, to a single global memory. In addition, all occurrences of `Move` and `Transfer` can be erased completely. Thus, linear maps provide a mechanism to program functionally yet execute imperatively. At the same time, as is evident from the specifications of the primitive operations, linear maps can be encoded using classical logic and verification of programs using them can benefit seamlessly from advances in first-order automated theorem proving.

Program 6 uses a linear map `g` to represent two adjacent memory location starting at offset `p`. The variable `g` is protected by a lock encoded with a boolean variable `b`. The procedure `P` acquires the lock `b` using atomic test-and-set, moves the content of `g` into a local linear map `l`, increments both memory locations in `l`, and then moves `l` back into `g`. The yield statement is encapsulated in the `Yield` procedure; it amounts to a simple global invariant asserting that whenever the lock `b` is not held, the domain of `g` must contain the two adjacent address `p` and `p+4` and the values stored in these addresses are identical. This program captures the essence of programming with monitors [35] and demonstrates that this reasoning can be encoded easily in CIVL.

Permissions. Because linear variables cannot be duplicated, linear variables can safely express exclusive access to a resource, such as a section of memory. Boyland [8] generalized linear type systems with a notion of fractional permissions: although linear variables still cannot be duplicated, fractions of a linear resource may be shared among multiple linear variables, as long as the fractions are all greater than 0 and always add up to 100%. For example, two variables could possess 50% of a resource, or one variable could possess 100% of a resource. The latter case (100%) corresponds to the exclusive access in traditional linear type systems. Fractional permissions are useful for distinguishing exclusive access, such as exclusive read-write access to memory, from shared access, such as shared read-only access to memory.

CIVL's linear variables can encode fractional permissions. As an example of this encoding, Figure 19 extends Figure 17's linear map type with a fraction indicating partial access to the linear map. For soundness, the `Store` operation still requires exclusive access (`fraction` = 100%), as in Figure 17, but `Load` now allows shared access (`fraction` may be anything), since multiple concurrent loads do not interfere with each other. The `Split` procedure splits a single linear map into two subfractions, while the `Combine` procedures combines two fractions into a single linear map.

Rather than using real-number or rational-number fractions, as in [8], we can also use sets in place of fractions, as shown in Figure 20. Set union then replaces numerical addition, and set difference replaces numerical subtraction. The type of the set doesn't matter much; Figure 20 uses sets of real numbers. With sets of real numbers, we can use the real interval [0, 1) to indicate exclusive access, in analogy to the 100% fraction, and subintervals like [0, 0.5) and [0.5, 1) to indicate shared access, in analogy to 50% fractions.

This extra level of detail allows CIVL to exploit disjointness of the subintervals: we know that two threads can't both contain a linear map with the same domain and the same subinterval (one thread might have subinterval [0, 0.5) and another [0.5, 1), but they can't both have [0, 0.5)). Section 8.2 describes how the GC verification uses this strategy to transfer fractions of thread identifiers between threads and global variables in order to aid verification of non-interference.

7 Modules

Section 6 used many small examples to illustrate various verification techniques supported by CIVL. Nevertheless, it's also important to show that CIVL's features can scale up to larger programs in a modular way: we should be able to check a large program by breaking it into smaller pieces and checking the pieces independently. Many languages built on Boogie, including Dafny [41] and BoogieX86 [27], encode modules in Boogie, so it's important that concurrent extensions to Boogie continue to support modular programming. This section describes a simple module system built on CIVL that allows separate verification of modules, allowing programmers to make changes to the private implementation of one module without disturbing the verification of other modules. Although the focus is on separate verification, this module system can easily be extended with other traditional module features, such as abstraction of types and hiding of definitions.

A key challenge for modular verification in CIVL

```
type lmap = LMap(dom:[int]bool,
                 map:[int]int,
                 fraction:real);

procedure Split(linear_in l:lmap, f1:real)
  returns(linear l1:lmap, linear l2:lmap)
  requires 0.0 < f1 && f1 < l.fraction;
  ensures l1.dom == l.dom && l1.map == l.map;
  ensures l2.dom == l.dom && l2.map == l.map;
  ensures l1.fraction == f1;
  ensures l2.fraction == l.fraction - f1;

procedure Combine(linear_in l1:lmap,
                  linear_in l2:lmap)
  returns (linear l:lmap)
  requires l1.dom == l2.dom;
  ensures l.dom == l1.dom && l.map == l1.map;
  ensures l.fraction == l1.fraction
                      + l2.fraction;

procedure Load(linear l:lmap, i:int)
  ...

procedure Store(linear_in l_in:lmap,
                i:int, v:int)
  ...
  requires l_in.fraction == 1.0;
  ...

procedure Move(linear_in l1_in:lmap)
  ...

procedure Transfer(linear_in l1_in:lmap,
                   linear_in l2_in:lmap,
                   i:int)
  ...
  requires l1_in.fraction == l2_in.fraction;
  ...
```

Figure 19: Fractional permissions

```
type lmap = LMap(dom:[int]bool,
                 map:[int]int,
                 fraction:[real]bool);

procedure Split(linear_in l:lmap, f1:set)
  returns(linear l1:lmap, linear l2:lmap)
  requires f1 != {} && f1 != l.fraction;
  requires f1 isSubsetOf l.fraction;
  ensures l1.dom == l.dom && l1.map == l.map;
  ensures l2.dom == l.dom && l2.map == l.map;
  ensures l1.fraction == f1;
  ensures l2.fraction == l.fraction - f1;

procedure Combine(linear_in l1:lmap,
                  linear_in l2:lmap)
  returns (linear l:lmap)
  requires l1.dom == l2.dom;
  ensures l.dom == l1.dom && l.map == l1.map;
  ensures l.fraction == l1.fraction
                      + l2.fraction;
```

Figure 20: Subset permissions

is the *CommutativitySafe* and *InterferenceSafe* judgments. Section 4 defines these as whole-program judgments, quadratically checking all pairs of actions or all pairs of yields and atomic blocks from an entire program. To achieve separate verification for CIVL, we must be able to check these judgments on a per-module basis. To achieve this, we observe that *CommutativitySafe* and *InterferenceSafe* are trivially true for operations that act on disjoint sets of global variables. If an atomic block modifies only variables g_1 and g_2, it will not interfere with a yield that refers only to variables g_3 and g_4. More generally, let each module M own a set of global variables, such that each global variable is owned by exactly one module, and decree that only M's procedures and actions can access M's global variables. Formally, define ownership for global variables, procedures and actions as:

$$\begin{array}{rcl} M & \in & \mathit{Module} \\ \mathit{own}_{\mathit{Global}} & \in & \mathit{Global} \rightarrow \mathit{Module} \\ \mathit{own}_{\mathit{Proc}} & \in & \mathit{Proc} \rightarrow \mathit{Module} \\ \mathit{own}_{\mathit{Action}} & \in & \mathit{Action} \rightarrow \mathit{Module} \end{array}$$

If $\mathit{own}_{\mathit{Proc}}(P) = M$, then P's preconditions, postcon-

```
module Lock1
  var lock1:Tid;
  procedure Acq1(linear tid:Tid)
    right [assume lock1 == none;
           lock1 := tid;]
  {...}
  procedure Rel1(linear tid:Tid)
    left  [assert lock1 == tid;
           lock1 := none;]
  {...}
  ...

module Lock2
  var lock2:Tid;
  procedure Acq2(linear tid:Tid)
    right [assume lock2 == none;
           lock2 := tid;]
  {...}
  procedure Rel2(linear tid:Tid)
    left  [assert lock2 == tid;
           lock2 := none;]
  {...}
  ...
```

Figure 21: Lock modules

ditions, and statements only refer to global g where $own_{Global}(g) = M$. Similarly, if $own_{Proc}(A) = M$, then A's gate and transition relation can depend only on g where $own_{Global}(g) = M$. With this restriction, *CommutativitySafe* and *InterferenceSafe* can be checked for each module M in isolation; no other module M' could interfere with any of the global variables owned by M.

As a more concrete example, suppose modules `Lock1` and `Lock2` independently implement locks, expressed as atomic actions on variables `lock1` and `lock2` (Figure 21).

By defining own_{Global}(`lock1`) = `Lock1` and own_{Global}(`lock2`) = `Lock2`, the `Lock1` and `Lock2` modules can be verified independently.

Nevertheless, a fixed ownership assignment is too inflexible to allow effective sharing of global state between modules. Therefore, it's also important to notice that own_{Global}, own_{Proc}, and own_{Action} can change across refinement layers. For example, suppose that in the first (lowest) layer, we verify `Lock1` and `Lock2`, replacing the `Acq1`, `Acq2`, `Rel1`, and `Rel2` procedures with `Acq1`, `Acq2`, `Rel1`, and `Rel2` actions, and hiding the implementations of the bodies of these procedures. Then suppose that in the second layer, we want to use `Lock1` and `Lock2` in a module `Counter` (Figure 22).

`Counter` uses `Lock1` and `Lock2` to implement an atomic counter supporting increment and decrement, using only primitive increment operations `Inc1` and `Inc2` to access its variables `x1` and `x2`. Since `Inc1` and `Inc2` refer to `lock1` and `lock2`, the owner of `Inc1` and `Inc2` must also own `lock1`, and `lock2`. Fortunately, ownership is free to migrate between layers, so in the second layer we define own_{Global}(`lock1`) = `Counter` and own_{Action}(`Acq1`) = `Counter` and so on. (Note that own_{Proc}(`Acq1`) doesn't matter in the second layer, since the procedure `Acq1` was hidden in the first layer.)

After verifying `Counter` in the second layer, we hide the `lock1` and `lock2` variables entirely, along with the `Inc1`, `Inc2`, `Acq1`, `Acq2`, `Rel1`, and `Rel2` actions and the `IncCnt`, `DecCnt`, and `ReadCnt` procedures. This leaves just the `x1` and `x2` variables and the `IncCnt`, `DecCnt`, and `ReadCnt` atomic actions.

We then repeat this process, transferring ownership of `x1`, `x2`, `IncCnt`, `DecCnt`, and `ReadCnt` to another module in a third layer of refinement/abstraction:

```
module Client
  ..call IncCnt..call DecCnt..call ReadCnt..
```

7.1 Functors

For the sake of explicitness, the example above used two replicas of a lock module (`Lock1` and `Lock2`), and verified each replica separately. In practice, we only want to want to write and verify such modules once. Borrowing from Standard ML [46], we introduce a notion of *functors* that generate modules:

```
functor LockFunctor()
  var lock:Tid;
  procedure Acq(linear tid:Tid) right [...]
```

```
module Counter
  var x1:int, x2:int;
  ...
  procedure Inc1(linear tid:Tid)
    returns(n1:int)
    both [assert lock1 == tid;
          n1 := x1; x1 := x1 + 1;]

  procedure Inc2(linear tid:Tid)
    returns(n2:int)
    both [assert lock2 == tid;
          n2 := x2; x2 := x2 + 1;]

  procedure IncCnt(linear tid:Tid)
    atomic [...]
  { call Acq1(tid);
    call _ := Inc1(tid);
    call Rel1(); }

  procedure DecCnt(linear tid:Tid)
    atomic [...]
  { call Acq2(tid);
    call _ := Inc2(tid);
    call Rel2(); }

  procedure ReadCnt(linear tid:Tid)
    returns(n:int)
    atomic [...]
  { call Acq1(tid);
    call Acq2(tid);
    var n1 := Inc1(tid);
    var n2 := Inc2(tid);
    n := n1 - n2;
    call Rel2();
    call Rel1(); }
```

Figure 22: Counter module

```
  procedure Rel() left [...]
  ...

module Lock1 := LockFunctor();
module Lock2 := LockFunctor();
```

Following Standard ML's approach, the `Lock1` and `Lock2` modules each get their own copies of the `lock` variable (named `Lock1.lock` and `Lock2.lock`) and their own copies of `Acq` and `Rel`. Nevertheless, we only have to verify `LockFunctor` once, rather than verifying each instantiation of `LockFunctor` separately (similar to Standard ML, which only has to type-check a functor once, rather than type-checking each functor instantiation separately).

In some situations, we might want to create an unbounded number of locks at run-time; static instantiation of a bounded number of modules doesn't suffice for this. Instead, without adding any new features to CIVL, we can simply change the `lock` variable to represent an array of locks (as an array of `Tid`s), and implement acquire/release operations on elements of the array. Note that the functor approach and array approach are complementary and can be combined. The functor approach has the advantage of creating multiple independent `lock` variables, whose ownership can be independently transferred to other modules. This allows, say, module M_1 to own one `lock` variable replica, while M_2 owns a different `lock` variable replica, so that M_1's `lock` variable cannot interfere with M_2's `lock` variable. The array approach, on the other hand, is useful for supporting dynamically allocated locks within a single module.

7.2 Module invariants

Modules often need to maintain invariants about their variables. For example, the module in Figure 23 maintains an invariant `x >= 0` that is true after initialization. Other modules may need to maintain `X`'s invariant to call `X`'s procedures. However, if `X` owns `x`, mentioning `x` from another module is prohibited:

```
module Y
  procedure P()
  {
```

```
module X
  var x:int;
  procedure Init()
    ensures x >= 0;
  { x := 0; }

  procedure Inc()
    requires x >= 0;
    ensures  x >= 0;
  { x := x + 1; }

  procedure Yield()
    ensures  old(x) >= 0 ==> x >= 0;
  { yield old(x) >= 0 ==> x >= 0; }
```

Figure 23: Module with invariant

```
    call Init();
    ...
    yield x >= 0; // not allowed
    ...
    call Inc();
  }
```

Instead, following the approach from Figure 14, the other module calls a procedure in module `X` that yields, since that procedure can refer to `x`:

```
module Y
  procedure P()
  {
    call Init();
    ...
    call Yield(); // allowed
    ...
    call Inc();
  }
```

Intuitively, module `X` is responsible for declaring its invariants in `Yield`, and these invariants are checked when verifying `X`. Other modules then maintain these invariants indirectly through `Yield`. To maintain invariants from multiple modules (say, `X1` and `X2`), we use parallel calls to multiple yield procedures simultaneously, as in Figure 15:

```
module X1 ...
module X2 ...

module Y
  procedure P()
  {
    call Init1();
    ...
    // parallel call:
    call Yield1() | Yield2();
    ...
    call Inc2();
  }
```

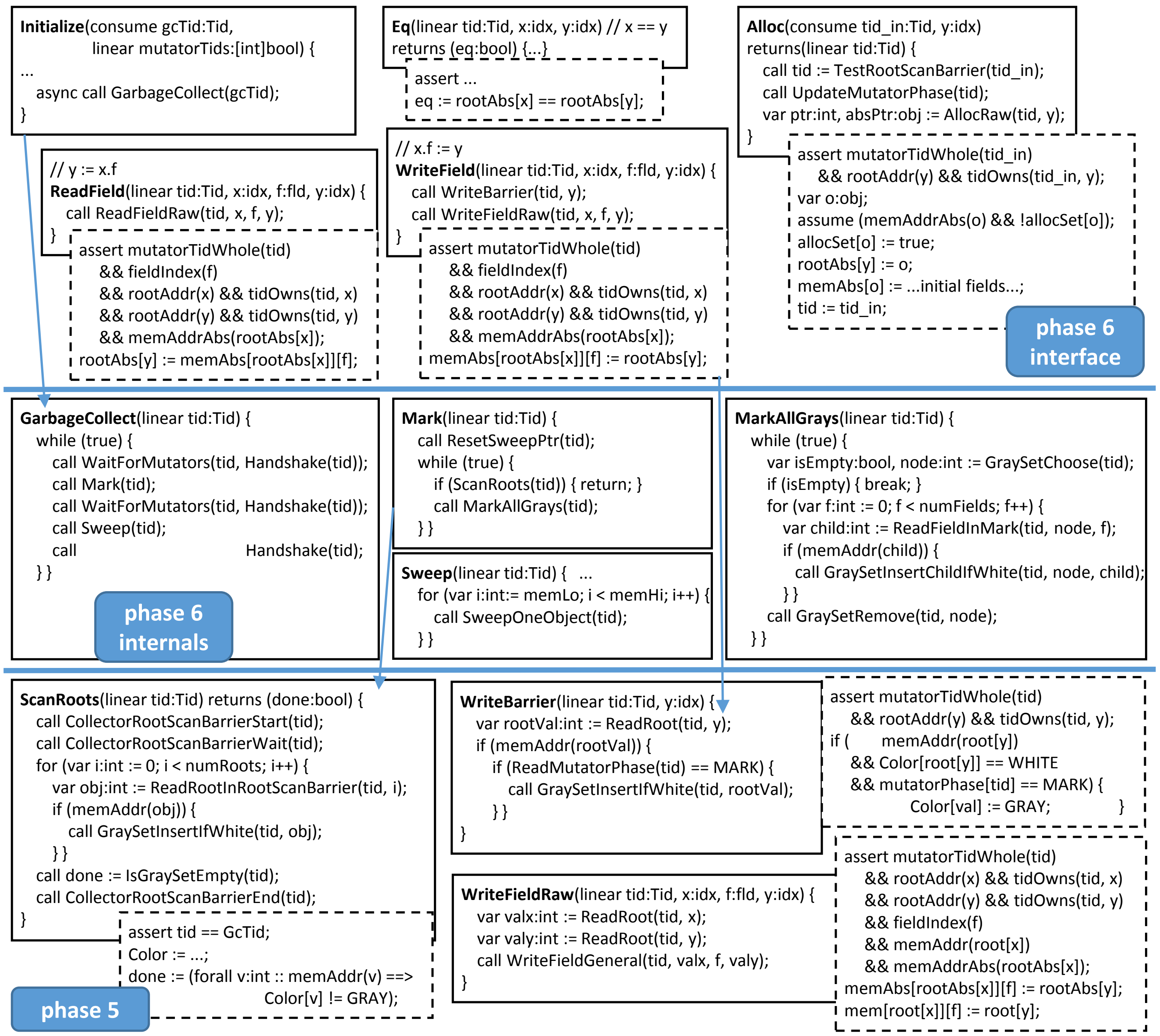
8 Case study: a verified concurrent GC algorithm

The CIVL verifier has been under development for around two years. Over that period, we have developed a collection of 32 benchmarks, ranging in size from 17 to 539 LOC, to illustrate various features of CIVL and for regression testing as we evolved the verifier. In addition to microbenchmarks, this collection also includes standard benchmarks from the literature such as a multiset implementation [18], the ticket algorithm [19], Treiber stack [33], work-stealing queue [7], device cache [17], and lock-protected increment [23]. The CIVL verifier is fast; the entire benchmark set verifies in 20 seconds on a standard 4-core Windows PC (2.8GHz, 8GB) with no benchmark requiring more than a few seconds.

In addition to these 32 small benchmarks, we also verified a larger algorithm: a concurrent mark-sweep garbage collector (GC). The rest of this section discusses the GC and its verification in detail.

8.1 Garbage collector

We demonstrate the verification methodology and tool on a realistic modern concurrent garbage collector algorithm (available at [29]). Our algorithm builds on the concurrent collector of Dijkstra et al. [11]. Dijkstra's collector is attractive for verification because it maintains a simple tri-color invariant on the heap objects (in contrast to snapshot-oriented



```
Initialize(consume gcTid:Tid,
           linear mutatorTids:[int]bool) {
...
  async call GarbageCollect(gcTid);
}
```

```
Eq(linear tid:Tid, x:idx, y:idx) // x == y
returns (eq:bool) {...}
```

```
assert ...
eq := rootAbs[x] == rootAbs[y];
```

```
Alloc(consume tid_in:Tid, y:idx)
returns(linear tid:Tid) {
  call tid := TestRootScanBarrier(tid_in);
  call UpdateMutatorPhase(tid);
  var ptr:int, absPtr:obj := AllocRaw(tid, y);
}
```

```
assert mutatorTidWhole(tid_in)
    && rootAddr(y) && tidOwns(tid_in, y);
var o:obj;
assume (memAddrAbs(o) && !allocSet[o]);
allocSet[o] := true;
rootAbs[y] := o;
memAbs[o] := ...initial fields...;
tid := tid_in;
```

```
// y := x.f
ReadField(linear tid:Tid, x:idx, f:fld, y:idx) {
  call ReadFieldRaw(tid, x, f, y);
}
```

```
assert mutatorTidWhole(tid)
    && fieldIndex(f)
    && rootAddr(x) && tidOwns(tid, x)
    && rootAddr(y) && tidOwns(tid, y)
    && memAddrAbs(rootAbs[x]);
rootAbs[y] := memAbs[rootAbs[x]][f];
```

```
// x.f := y
WriteField(linear tid:Tid, x:idx, f:fld, y:idx) {
  call WriteBarrier(tid, y);
  call WriteFieldRaw(tid, x, f, y);
}
```

```
assert mutatorTidWhole(tid)
    && fieldIndex(f)
    && rootAddr(x) && tidOwns(tid, x)
    && rootAddr(y) && tidOwns(tid, y)
    && memAddrAbs(rootAbs[x]);
memAbs[rootAbs[x]][f] := rootAbs[y];
```

```
GarbageCollect(linear tid:Tid) {
  while (true) {
    call WaitForMutators(tid, Handshake(tid));
    call Mark(tid);
    call WaitForMutators(tid, Handshake(tid));
    call Sweep(tid);
    call                      Handshake(tid);
} }
```

```
Mark(linear tid:Tid) {
  call ResetSweepPtr(tid);
  while (true) {
    if (ScanRoots(tid)) { return; }
    call MarkAllGrays(tid);
} }
```

```
Sweep(linear tid:Tid) {  ...
  for (var i:int:= memLo; i < memHi; i++) {
    call SweepOneObject(tid);
} }
```

```
MarkAllGrays(linear tid:Tid) {
  while (true) {
    var isEmpty:bool, node:int := GraySetChoose(tid);
    if (isEmpty) { break; }
    for (var f:int := 0; f < numFields; f++) {
      var child:int := ReadFieldInMark(tid, node, f);
      if (memAddr(child)) {
        call GraySetInsertChildIfWhite(tid, node, child);
    } }
    call GraySetRemove(tid, node);
} }
```

```
ScanRoots(linear tid:Tid) returns (done:bool) {
  call CollectorRootScanBarrierStart(tid);
  call CollectorRootScanBarrierWait(tid);
  for (var i:int := 0; i < numRoots; i++) {
    var obj:int := ReadRootInRootScanBarrier(tid, i);
    if (memAddr(obj)) {
      call GraySetInsertIfWhite(tid, obj);
  } }
  call done := IsGraySetEmpty(tid);
  call CollectorRootScanBarrierEnd(tid);
}
```

```
assert tid == GcTid;
Color := ...;
done := (forall v:int :: memAddr(v) ==>
                         Color[v] != GRAY);
```

```
WriteBarrier(linear tid:Tid, y:idx) {
  var rootVal:int := ReadRoot(tid, y);
  if (memAddr(rootVal)) {
    if (ReadMutatorPhase(tid) == MARK) {
      call GraySetInsertIfWhite(tid, rootVal);
  } }
}
```

```
assert mutatorTidWhole(tid)
    && rootAddr(y) && tidOwns(tid, y);
if (     memAddr(root[y])
    && Color[root[y]] == WHITE
    && mutatorPhase[tid] == MARK) {
           Color[val] := GRAY;          }
```

```
WriteFieldRaw(linear tid:Tid, x:idx, f:fld, y:idx) {
  var valx:int := ReadRoot(tid, x);
  var valy:int := ReadRoot(tid, y);
  call WriteFieldGeneral(tid, valx, f, valy);
}
```

```
assert mutatorTidWhole(tid)
    && rootAddr(x) && tidOwns(tid, x)
    && rootAddr(y) && tidOwns(tid, y)
    && fieldIndex(f)
    && memAddr(root[x])
    && memAddrAbs(rootAbs[x]);
memAbs[rootAbs[x]][f] := rootAbs[y];
mem[root[x]][f] := root[y];
```

Figure 24: Verified garbage collector phases 5, 6 (pseudocode excerpts in solid boxes; atomic action specs in dashed boxes)

collectors [13, 12, 14, 4] whose tri-color invariants are more subtle). By itself, though, Dijkstra's collector is not a modern or performant collector. First, it becomes incorrect in the presence of more than one program thread (mutator). Second, it requires that the write-barrier be run not only on updates of heap pointers, but also on modifications of root pointers, i.e., on modifications of the runtime stacks and the registers; modern high-performance collectors avoid this overhead.

Therefore, our algorithm (shown inside the solid boxes in Figure 24) extends and modifies Dijkstra's collector to make it work with parallel programs and to not require a write-barrier on root modifications. Like Dijkstra's collector, our algorithm first *marks* all objects reachable from roots (registers and stacks), shown in Figure 24's Mark procedure, and then *sweeps* away all unreached objects, shown in Figure 24's Sweep procedure. As in Dijkstra's collector, our algorithm employs a tri-color abstraction to describe the trace of the reachable objects. Objects are said to be *white* if the collector has not seen them yet during the trace. Objects that the collector encounters become gray and remain gray until the collector scans their children. Once all the children of an object are noted (meaning that none of them are white), the object becomes black. The collector works by choosing a node from the set of gray objects (GraySetChoose, called from MarkAllGrays in Figure 24), *shading* all its white children to gray (GraySetInsertChildIfWhite), and then removing the object from the gray set by making the object black (GraySetRemove). The shading operation grays a node if it is white, and does nothing otherwise. The trace terminates when all roots point to black objects (according to ScanRoots) and there are no more gray objects (according to IsGraySetEmpty, called by ScanRoots). Termination is guaranteed because objects can only get darker. Correctness is guaranteed using an invariant that a black object never points to a white object during the trace (black objects can only point to gray objects or black objects). At the end of the trace, objects pointed by the roots must be black, and since no gray objects remain, black objects only point to black objects, so the entire set of objects reachable from the roots must be black.

Concurrent mutator operations on objects (ReadField and WriteField in Figure 24) could potentially break the no-black-to-white invariant, because a mutator's WriteField operation could potentially redirect a pointer of a black object to point to a white object. Therefore, coordination between the program and the concurrent collector is required: before each raw pointer update (WriteFieldRaw), the WriteField procedure executes a *write-barrier* (WriteBarrier). Before pointer field $x.f$ is set to reference an object y, WriteBarrier shades y, ensuring that even if x is black, a pointer from x to y will not violate the no-white-to-black invariant.

The write barrier should shade objects only while the collector is in its mark phase, not when the collector is sweeping or is idle, and the collector may only switch between phases (mark, sweep, or idle) when no mutator is in the middle of a WriteField or Alloc operation. To achieve this (and thereby support correct and efficient support for multiple mutator threads), we extend Dijkstra's collector with explicit tracking of phases, via a handshaking mechanism [13, 12]. A shared variable, collectorPhase, contains the current collector phase. The collector initiates a handshake by incrementing collectorPhase (in Handshake, called by GarbageCollect). Each mutator thread keeps cached copy of collectorPhase, and periodically checks to see if the cached copy mismatches the current collectorPhase, and if so, updates the cached copy with the most recent value (in our algorithm, a mutator's call to the allocator, Alloc, checks this in UpdateMutatorPhase, but the exact location of the check is not critical to correctness). The GarbageCollector waits until all cached copies equal collectorPhase (WaitForMutators in GarbageCollect), and then executes a phase (Mark, Sweep, or, for the idle phase, nothing). Note that each mutator thread can read its own cached phase without acquiring a lock (ReadMutatorPhase in WriteBarrier), leading to efficient WriteBarrier performance.

Dijkstra's collector requires a write barrier on modifications to roots as well as modifications to objects. We eliminate this overhead by employing repeated tracing phases until all objects referenced by roots are black. A tracing phase starts by stopping all mutators and marking their roots. The process of stopping the mutators is similar to a handshake and is done using the CollectorRootScanBarrierStart, CollectorRootScanBarrierWait, and CollectorRootScanBarrierEnd procedures on the collector side and TestRootScanBarrier procedure on the mutator side. At the end of the root scan (before the mutators reawaken), all roots point to gray or black objects. If no gray objects remain (IsGraySetEmpty), then all roots point to black objects, and marking is complete. Otherwise, we trace from gray objects until completion and start a new tracing phase (by stopping the mutators and checking the roots again). In a worst-case theoretical scenario we may need to run many root scans and discover more and more white root descendants to trace each time. But in practice we usually finish after a small number of scans, so we obtain correctness and termination in all scenarios and we obtain good performance in real-world scenarios.

8.2 Collector Verification in Boogie

We have implemented and verified our algorithm in Boogie, including initialization (Initialize), the GC (GarbageCollect), the allocator (Alloc), and the mutator operations (ReadField, WriteField, and Eq), and all the lower-level operations required to implement them (some of these appear in Figure 24; others are omitted from the figure to save space). To make the verification as realistic as possible, our Boogie code implements everything in terms of individual CPU operations, such as load, store, atomic increment/decrement, and CAS (compare-and-swap); in contrast to some previous work [26], we do not assume any built-in higher-level operations. To ease verification, we make some simplifications: we use a naive allocator (sequential search for free space), we assume a sequentially consistent memory model, and we assume that all objects have the same number of fields. (Except for the assumption of sequential consistency, none of these substantially alter the nature of the proof.)

Overall, our implementation consists of about 2100 lines of Boogie code. The GC verification takes 60 seconds on the same PC used for microbenchmarks. The bulk of this time, 54 seconds, is taken by the verification of the refinement checks from Section 4.2.2. The linear type checking, the yield safety checks, and the commutativity checks take the rest of the time and are insignificant in comparison.

Our verification takes advantage of all techniques in CIVL: refinement, assertions, reduction, and linearity. Refinement gives us extremely simple high-level action specifications for Initialize, ReadField, WriteField, Eq, and Alloc, shown in their entirety in Figure 24's dashed boxes. (Initialize and GarbageCollect have empty actions; the GarbageCollector itself is just an internal implementation detail inside Initialize, which serves only to set up the global GC invariant needed by the other high-level actions.) Crucially, ReadField, WriteField, Eq, and Alloc appear atomic to mutators, even though internally, WriteField and Alloc involve many interleaved operations on shared GC data structures. Figure 24 shows only shows phases 5 and 6, the two most abstract phases of refinement; phases 1-4 fill in the implementation details, such as implementing the set of gray objects as an explicit stack (an array of elements with a pointer to the stack top, in phase 4), handshaking (phase 3), locks (phase 2), and wrapping the primitive CPU operations in left/right/non-moving atomic actions (phase 1). Ultimately, the phases are built on trusted CPU-level atomic actions, such as reading and writing roots directly:

```
procedure PrimitiveWriteRoot(i:idx, v:int)
  atomic [assert rootAddr(i); root[i] := v;]

procedure PrimitiveReadRoot(i:idx)
  returns (v:int)
  atomic [assert rootAddr(i); v := root[i];]
```

We write the highest-level action specifications in terms of an abstract view of memory, as in earlier work on sequential garbage collector verification [44, 28]. (Abstract memory is infinite and eternal: once allocated, an abstract object lives forever. Deallocation is an underlying implementation detail, not exposed in the abstract interface.) Our abstract view describes a machine as consisting of just three variables: abstract memory memAbs:[obj][fld]obj, mapping object identifiers and fields to other objects, abstract root values rootAbs:[idx]obj, mapping root names to objects, and allocSet:[obj]bool, the set of objects allocated so far. At this high layer of abstraction, we use Boogie's hiding to hide all other variables (such as the concrete root set, "root", the concrete memory, "mem", and the colors, "Color", used by lower-level procedures).

All operations are relative to root names of type idx. ReadField, for example, reads an object field from an object pointed to by root x into a root y. The predicates rootAddr and tidOwns establish that x and y are valid root names, owned by a particular mutator tid. (We assume that each root is private to a single mutator stack or register file; sharing between mutator threads takes place through shared pointers to objects.) The predicates fieldIndex(f) and memAddrAbs(o) establish that x.f is a valid field of a valid object. Allocation establishes memAddrAbs(o) for newly allocated objects so that they may be used by subsequent ReadField and WriteField operations. It also establishes o's unique identity by ensuring that it did not previously belong to the allocated object set.

In addition to atomic action specifications, the verification establishes invariants using assertion reasoning (omitted from Figure 24 to save space). For example, Initialize establishes a global mapping toAbs:[int]obj from physical memory mem and abstract memory memAbs:

```
(forall x:int, f:fld ::
      memAddr(x)
   && toAbs[x] != nil
   && fieldIndex(f)
  ==>    toAbs[mem[x][f]]
     == memAbs[toAbs[x]][f])
```

and Mark maintains the no-black-to-white invariant:

```
(forall x:int, f:fld ::
      memAddr(x)
   && Black(Color[x])
   && fieldIndex(f)
   && memAddr(mem[x][f])
  ==> Gray(Color[mem[x][f]])
   || Black(Color[mem[x][f]]))
```

Finally, linearity plays a key role in establishing mutual exclusion. The GC thread has its own thread id gcTid, and each mutator has its own thread id. The Initialize procedure consumes gcTid (written here as "consume") and borrows all the mutator thread ids (written as "linear", as in Section 2), so that it's clear that no other concurrent actions are allowed during initialization. This allows all the internal initialization actions to be both-movers, without requiring any explicit locking. Initialize must consume gcTid because it passes gcTid to the newly spawned GarbageCollector thread; since gcTid is consumed, it's impossible to call Initialize twice in an attempt to spawn two parallel GC threads (which naturally expresses how the algorithm is only safe for a single GC thread).

Because CIVL's linearity is based on sets of values, we can represent thread identifiers as sets that can be subdivided into subsets (similar to how fractional permissions may be divided into fractions). During root scanning, each mutator thread places a fraction of its thread id in a global variable, and reclaims the fraction from the global variable after root scanning completes; a collector invariant tracks that the global variable contains non-empty fractions from all mutators during root scanning. Thus, during root scanning, CIVL's rules for linearity prove that no interference occurs between the collector and any mutator operations that require the whole mutator thread id ("mutatorTidWhole", used in Figure 24's ReadField, WriteField, Alloc, and most other mutator operations).

8.3 Discussion

We now put atomicity refinement techniques from the literature and CIVL in context. The refinement proof spans six levels of abstraction. Each of refinement proof relating two consecutive levels is made feasible by a different blend of the techniques in CIVL.

The topmost-level description of the garbage collector provides an idealized, abstract view of memory. At this level, none of the lowest-level implementation variables are visible – variable hiding has been used to project them away. In the top few levels of the garbage collector proof, invariant-based non-interference reasoning was our primary tool, while reduction simplified verification by enabling us to use coarser atomic actions and fewer location invariants. Linear variables were used throughout the proof to model the distinct thread identifiers for the garbage collector thread and mutator threads, but were most instrumental in encoding single-threaded execution in the initialization phase of the program. For these top few levels of our proof, rely-guarantee and separation-logic-based approaches would have also performed well, as demonstrated by the garbage collector proof of Liang et al. [42], where the atomicity of actions in the lower levels our proof is *assumed* but not verified. An important distinguishing capability in CIVL is being able to use location invariants rather than pure rely-guarantee reasoning. This helped interactive proof at the top levels significantly. For the mark phase of the garbage collector, we made critical use of different invariants at different locations in procedure bodies. While the same non-interference argument could have been encoded in rely-guarantee reasoning, as we had done ourselves in an earlier version of our proof, it would have required the use of several additional auxiliary shared variables. Invariants, rely and guarantee conditions referring to such auxiliary variables throughout the program made interactive invariant reasoning more difficult to manage.

In the lower levels of the garbage collector proof, where correctness of concurrent data structures and synchronization primitives were proven, we made relatively little use of location invariants, and made heavier use of linear variables and reduction. We also used variable hiding heavily to hide low-level implementation variables. For lower-level refinement tasks, for instance, when verifying the correctness of a lock-protected concurrently-accessed stack, ownership arguments, separation logic, or QED-style atomicity would have been sufficient. But, at the higher levels of our proof, where non-interference reasoning via invariants and linear variables was indispensable, atomicity alone, or ownership or separation logic arguments alone would have run into difficulty.

While existing techniques in the literature have as their "sweet spot" a few of the refinement proofs in our garbage collector proof, they run into difficulty in others. More critically, they do not facilitate layering refinement proofs, which is required for stepwise refinement. Using a realistic top-down proof as CIVL's design driver led us to combine in one tool and consistent theory, the verification techniques of linearity, reduction and non-interference reasoning in the service of a modular refinement proof directed by the syntactic structure of the imperative concurrent program.

9 Related work

Our work is the first to provide tool and theory to support automated, modular whole-program refinement through multiple layers, as distinct from existing work on single-layer atomicity refinement between procedure implementations and specifications. While many of the verification techniques used within CIVL appear in the literature, CIVL is the first to make sound, joint use of them to decompose the refinement task following the syntactic structure of a program. In the following, we first contrast CIVL with refinement verification techniques, and then with tools and techniques for reasoning about concurrent programs in general.

9.1 Refinement-oriented verification

Atomic action specifications have been explored by the CALVIN [22, 25] verifier previously. CIVL makes a distinction between preemptive and cooperative semantics, and carries out refinement verification on a procedure body with cooperative semantics as enabled by movers types and reduction. CALVIN attempts to verify refinement directly on the preemptive semantics, making only limited use of movers at the lowest-level representation. CALVIN, unlike CIVL, does not support location invariants and linear variables but incorporates rely-guarantee reasoning. The same non-interference reasoning can be carried out using location invariants or rely-guarantee reasoning, and CIVL supports both. However, in certain cases, rely-guarantee reasoning requires use of auxiliary (shared) variables and makes interactive proofs difficult as was the case in our GC proof. We find location invariants to be a powerful verification device.

QED [17] is a simplifier for concurrent programs and is close in spirit to the refinement-oriented approach of CIVL. A key distinction between CIVL and QED is the fact that a proof step in QED is a small rewrite in the concurrent program that must be justified by potentially expensive reduction and invariant reasoning. In QED, procedures can be proven atomic only one procedure at a time, and only by transforming their bodies by reduction to be yield free. The number of small proof steps directly affect both programmer and computer effort. By contrast, CIVL supports large proof steps, in each of which the bodies of several procedures are automatically replaced by atomic actions, thereby lowering the cost of both interaction and automation. The non-interference reasoning in QED is even more limited than CALVIN. QED supports only global invariants and does not support rely-guarantee reasoning or linear variables.

Liang et al. [42] present a method for verifying that procedure bodies refine atomic specifications The key verification approach is rely-guarantee reasoning and the refinement (simulation) relation between a procedure and its specification is constrained so it is preserved under parallel composition. No tool support is provided. Authors present a (paper) GC proof, which is limited in scope compared to ours, as their proof corresponds to a few layers of our proof. In particular, the GC is not refined down to individual atomic memory accesses. Since this work uses different languages to describe the high-level and low-level programs, it is not immediately possible to carry out a multi-level stepwise refinement proof.

Turon and Wand [50] use ownership disciplines and separation logic to verify refinement of atomic specifications by concurrent data structure implementations. Rely-guarantee reasoning is supported to provide compositionality and non-interference arguments. This work targets a single refinement step between atomic specifications for methods and their implementations. No tool support for this verification method is provided.

Verifying linearizability of concurrent data structures (see, e.g., [16, 31]) can be viewed as an instance of one-level of refinement in our setting. CIVL can be used for mechanical verification of linearizability, as we did for the Treiber stack. Tools and techniques specific to verifying linearizability cannot be easily generalized for stepwise refinement proofs through multiple levels.

Refinement proofs between implementations and specifications of protocols have been investigated using the TLA+ [39] specification language. Compositional proofs between specifications and implementations consisting of modules [1] have been investigated in this context. Modular refinement proofs for hardware systems have been investigated extensively (e.g., [32, 15]) using the SMV [45] and Mocha [3] model checking tools. To verify a concurrent, shared-memory program using such tools, one must encode the program semantics as a state-transition system and express verification goals in terms of this system. For concurrent, shared-memory software, CIVL enables reasoning on the structured, imperative multithreaded program text rather than a logic description of the program's state-transition relation.

9.2 Reasoning about concurrency

In this section, we discuss foundational techniques for combating the complexity of concurrent program verification. CIVL and refinement techniques discussed in the previous section have common ideas with tools and formalisms discussed in this section, however, the latter primarily target verification of a *single* program rather than refinement. Refinement in CIVL is orthogonal to these techniques, which can be aided by CIVL's ability to connect a complex concurrent program to a simpler abstraction.

VCC [9] is a tool for verifying concurrent C programs. Chalice [40] is a language and modular verification tool for concurrent programs. VCC does not support refinement and Chalice does so only for sequential programs. VCC and Chalice base their invariant reasoning on objects, object ownership, and type invariants. Invariant reasoning in CIVL is more primitive and based on predicates in yield statements. Although the approach in VCC and Chalice is more convenient when applicable, CIVL's approach is more flexible. VCC and Chalice can reason sequentially about objects exclusively owned by a thread; CIVL accomplishes the same using linear variables. Neither VCC nor Chalice support movers and reduction reasoning.

Concurrent separation logic [47] reasons about concurrency without explicitly checking for non-interference between threads. Recently, tools based on this logic that blend in explicit non-interference reasoning (but without support for reduction and mover reasoning) have been developed [20, 51]. CIVL's combination of interference checking and linear variables is an extreme example of this trend, is very general and technique-agnostic. We supply very primitive abstractions and let programmers mix and match these abstractions freely to encode the non-interference reasoning style of their choice.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, Jan. 1993.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[3] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 521–525, 1998.

[4] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA*, 2003.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[6] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[8] J. Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.

[9] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, 2009.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[11] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11), Nov. 1978.

[12] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, 1994.

[13] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL*, 1993.

[14] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In *PLDI*, 2000.

[15] A. T. Eiríksson. The formal design of 1m-gate asics. *Form. Methods Syst. Des.*, 16(1):7–22, Jan. 2000.

[16] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 296–311. Springer Berlin Heidelberg, 2010.

[17] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[18] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 27–37, 2005.

[19] A. Farzan, Z. Kincaid, and A. Podelski. Proofs that count. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 151–164, 2014.

[20] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

[21] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[22] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.

[23] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 213–224, 2003.

[24] R. Floyd. Assigning meaning to programs. In *Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[25] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[26] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *CAV*, 1996.

[27] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[28] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, 2009.

[29] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Verified concurrent garbage collector. `http://singularity.codeplex.com/SourceControl/latest#base/Imported/Bartok/runtime/verified/GCs/concur/GC.bpl`.

[30] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[31] T. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR 2013—Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 242–256. Springer Berlin Heidelberg, 2013.

[32] T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '99, pages 494–499, Piscataway, NJ, USA, 1999. IEEE Press.

[33] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[35] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[36] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[37] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, feb 2014.

[38] S. K. Lahiri, S. Qadeer, and D. Walker. Linear maps. In *PLPV*, pages 3–14, 2011.

[39] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2004.

[40] K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.

[41] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[42] H. Liang, X. Feng, and M. Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Trans. Program. Lang. Syst.*, 36(1):3:1–3:55, Mar. 2014.

[43] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[44] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *PLDI*, 2007.

[45] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[46] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[47] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[48] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

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

[50] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 247–258, New York, NY, USA, 2011. ACM.

[51] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[52] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.

[53] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, Apr. 1971.

[54] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.