Automated and modular refinement reasoning for concurrent programs

Chris Hawblitzel[1], Erez Petrank[2], Shaz Qadeer[1], and Serdar Tasiran[3]

[1] Microsoft
[2] Technion
[3] 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 [43], 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 [11] and verification of cache-coherence protocols and distributed algorithms [32]. 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 [30]. 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 [20, 28] 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 express the per-procedure verification obligations 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 code implementing a procedure 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.[4] 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 [24] on an abstraction of P in which each atomic action of P is represented by only its mover type [35, 17]. The mover types themselves are verified separately and automatically using an automated theorem prover [9].

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 post-condition. 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 [38, 29] 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

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

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 [42] that enables logical encoding of thread identifiers, permissions [7], and disjoint memory [31].

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 [6, 13–15, 19, 27]. 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, detailed in a technical report [23], 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 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. This is a novel technique that combines automata-based and logic-based reasoning.
– 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 an example (Figure 1) inspired by the write barrier in our concurrent garbage collector (GC). In a concurrent GC, each object in the heap has a color: `UNALLOC`, `WHITE`, `GRAY`, or `BLACK`. The GC traverses reachable objects, marking the reached objects

GRAY and then BLACK. At the end of the traversal, reached objects are BLACK, unreached objects are WHITE, and the GC deallocates the WHITE objects. The threads in the system must cooperate with the GC to ensure that the collection creates no dangling pointers (i.e., if object A is reachable and A points to object B, then B should not be deallocated). Therefore, before a mutator thread mutates an object A to point to an object B, the thread executes a write barrier to check the color of B. If B is WHITE, the write barrier darkens B's color to GRAY to ensure that the GC does not deallocate B. WB implements the write barrier. The write barrier is only invoked on allocated objects, thus, colors cannot be UNALLOC when WB is called. 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 object encountered by a mutator. If Color <= WHITE, WB calls the more expensive procedure WBSlow to re-examine and possibly update Color while holding the lock. The annotation yield Color >= cNoLock is a local invariant expected to be preserved by the environment of WB. CIVL simplifies reasoning about WBSlow by allowing us to express its specification as the following atomic action:

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

This specification indicates that regardless of how the environment interferes with its execution, to its caller it appears as if WBSlow atomically executes the code above.

```
var Color: int; // UNALLOC=0, WHITE=1,
                // GRAY=2, BLACK=3

procedure WB(linear tid:Tid)
atomic [if (Color == WHITE) Color := GRAY];
requires Color >= WHITE;
ensures Color >= GRAY;
{
  var cNoLock:int;
  yield Color >= WHITE;
  cNoLock := GetColorNoLock(tid);
  yield Color >= cNoLock;
  if (cNoLock <= WHITE)
    call WBSlow(tid);
  yield Color >= GRAY;
}

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 [...];
```

Fig. 1: Write barrier

Per-procedure simulation, non-interference via invariants. The verification of WB illustrates a combination of techniques. We first explain how WB's post-condition is verified. To see that this task is not trivial, consider a scenario in which WB, not holding a lock, reads Color and sets cNoLock to GRAY and then yields. Another thread sets Color to WHITE. WB resumes, but because the local variable cNoLock is GRAY, does nothing and exits with Color being WHITE, violating WB's postcondition. But, in the GC this scenario is not possible. The yield predicate (location invariant) Color >= cNoLock expresses the fact that other threads can only modify Color to a higher (darker) value. CIVL verifies the correctness of this location invariant and rules out this undesirable scenario. Using this location invariant, `WB`'s pre-condition, and `WBSlow`'s atomic specification, CIVL is able to verify `WB`'s post-condition.

In Figure 1, we suppose for illustration's sake that `WB` and `WBSlow` have slightly different atomic specifications, one testing for `Color == WHITE` and the other for `Color <= WHITE`. In this case, verifying that the implementation of `WB` refines its atomic specification relies on `Color` not being `UNALLOC`. Otherwise, `WBSlow` would set `Color` to `GRAY` whereas `WB` would leave it unmodified, leading to a refinement violation. `WB`'s precondition `Color >= WHITE` and the location invariant `Color >= cNoLock` imply that `Color` is never `UNALLOC` during the execution of `WB`. Given this constraint, CIVL checks atomicity refinement for `WB` by verifying the existence of a particular simulation-relation. Each control path through `WB` is analyzed as a sequence of code fragments, from one `yield` statement to the next. For each control path through a procedure, exactly one code fragment must be simulated by the atomic action specification while others do not modify global state. This refinement proof for `WB` makes use of (1) correct modeling of environment interference by the pre- and post-conditions, and the yield predicate, and (2) the atomic action specification for the called procedure `WBSlow`. The CIVL verifier automatically computes a logical verification condition capturing the 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.

Preemptive vs cooperative semantics. The verification of `WBSlow` highlights another important feature 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 [17]. 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.

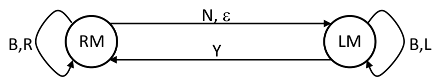

Fig. 2: Yield sufficiency automaton

Given verified mover types for actions, CIVL verifies the correctness of the placement of `yield` statements using a novel approach. A *yield sufficiency automaton* (Figure 2) encodes all sequences of atomic actions (of **R**ight, **L**eft, **B**oth and **N**on-mover types) 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` statements. 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 3. CIVL verifies that this automaton is simulated by the yield sufficiency automaton using an existing algorithm for computing simulation relations [24].

The use of commutativity reasoning is optional in CIVL, but beneficial in our experience. Commutativity reasoning may be avoided by annotating atomic action specifications with the mover type `atomic` and inserting a yield statement before every invocation of an atomic action. In our experience with CIVL, using more yield statements, each with an accompanying location invariant, 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.[5] Second, the computational cost of the pairwise mover reasoning is replaced by the cost of pairwise non-interference checks between yield predicates and concurrently executing atomic actions.

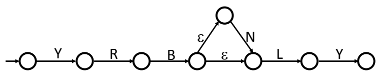

Fig. 3: Abstraction of `WBSlow`

Linear variables. In Figure 1, thread identifier (`tid`) variables are declared `linear` to indicate that two threads cannot possess the same thread identifier simultaneously. We now explain this feature of CIVL in more detail using the program in Figure 4. This example 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 from its value in a different thread.

```
type Tid;
procedure Allocate()
  returns (linear tid:Tid);

var a:[Tid]int;

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

Fig. 4: Encoding thread identifiers

[5] Location invariants that cannot refer to the state of other threads are known to be incomplete, both in theory and in practice.

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 [42]. 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 and commutativity. Linearity is general enough to support much more than just fixed thread identifiers: CIVL also uses it to express separation of memory (as is done commonly in separation logic proofs [39]; see [31]) and to express permissions [7] 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.

3 Verification

In this section, we present our verification method on a core concurrent programming language called CIVL (Figure 5). Due to lack of space, we can only provide an overview of the design of the CIVL verifier. The full formalization of the language and detailed rules for all verification judgments is available in a technical report [23].

$$
\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & \mathit{skip} \mid \mathit{yield}\ e \mid \mathit{call}\ A \mid \\
 & & \mathit{call}\ P \mid \mathit{async}\ P \mid \\
 & & \mathit{ablock}\ \{e\}\ s \mid s;\ s \mid \\
 & & \mathit{if}\ \mathit{le}\ \mathit{then}\ s\ \mathit{else}\ s \mid \\
 & & \mathit{while}\ \{e\}\ \mathit{le}\ \mathit{do}\ s \\
F \in \mathit{Frame} & ::= & (P, L, s) \\
T \in \mathit{Thread} & ::= & (\mathit{TL}, \overrightarrow{F}) \\
\mathit{Prog} \in \mathit{Program} & ::= & (\mathit{ps}, \mathit{as}, G, \overrightarrow{T})
\end{array}
$$

Fig. 5: Syntax

A CIVL program *Prog* contains procedures *ps*, atomic actions *as*, global state G, and threads $\overrightarrow{T}$. Each thread T in $\overrightarrow{T}$ contains thread-local state TL and stack frames $\overrightarrow{F}$. Each stack frame F in $\overrightarrow{F}$ contains a procedure name P, procedure-local state L, and a statement s representing the code in P that remains to be executed. Thus, *Prog* contains all information to represent not only the static program written by the programmer but also the entire state of the program as it executes. The statements in CIVL contain the usual constructs such as sequencing, conditional control flow, and looping. In addition, it contains invocation of procedures (*call* P), execution of atomic actions (*call* A), and thread creation (*async* P). Each atomic action has a single-state *gate predicate* and a two-state *transition relation*. If a thread executes an atomic action in a state (disjoint union of global, thread-local, and procedure-local state) where its gate predicate does not hold, the program fails; otherwise, the state is modified according to its transition relation. The execution of *Prog* is modeled as the usual *preemptive* semantics in which a nondeterministically chosen thread may execute any number of steps. *Prog* is *unsafe* if some execution fails the gate of an atomic action; otherwise, *Prog* is *safe*.

Suppose a program $Prog^{hi}$ 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 $Prog^{hi}$ to a realizable implementation $Prog^{lo}$, these coarse atomic actions must be refined down to lower-level actions. During refinement, a high-level atomic action A is implemented by a procedure P, which is itself implemented using lower-level atomic actions. In CIVL, the programmer can simultaneously refine many atomic actions by specifying a partial function RS from procedures to atomic actions; $Prog^{hi}$ is obtained from $Prog^{lo}$ by replacing each occurrence of *call* P for $P \in dom(RS)$ with *call* $RS(P)$. The main contribution of this paper is a verification method that allows us to validate such a refinement from $Prog^{hi}$ to $Prog^{lo}$ (or abstraction from $Prog^{lo}$ to $Prog^{hi}$) so that safety of $Prog^{hi}$ implies the safety of $Prog^{lo}$ as well.

While abstracting $Prog^{lo}$ to $Prog^{hi}$, it is often inconvenient to reason about $Prog^{lo}$ using its preemptive semantics, which allows potential interference at *every* control location in a thread from concurrently-executing threads. To make reasoning more convenient, CIVL provides the statement *yield* e, an annotation used to specify a cooperative semantics for the program. In this semantics, a thread executes continuously until it reaches a yield statement, at which point a different thread may be scheduled. To ensure that any reasoning performed on cooperative semantics is also sound for preemptive semantics, CIVL exploits commutativity reasoning. It allows the programmer to specify the commutativity type of atomic actions in the program—B for both mover, R for right mover, L for left mover, and N for non mover [17]. The CIVL verifier checks the correctness of these commutativity types by verifying each atomic action pairwise against every atomic action in the program. While it is sound to put a yield statement before and after every atomic action, the programmer may omit certain yield statements, e.g., a yield after a right mover or a yield before a left mover. In general, the *Yield Sufficiency Automaton* from Figure 2 encodes all sequences of atomic actions and yield statements for which reasoning about cooperative semantics is sound. Given the commutativity types of atomic actions and the program code annotated with yield statements, the CIVL verifier checks modularly for each procedure that its implementation is connected to the yield sufficiency automaton via a simulation relation [24].

In addition to introducing a control location where interference is allowed to occur, a yield statement *yield* e also provides an invariant e to constrain the environment interference. The invariant e is similar to the location invariant in the method of Owicki and Gries [38]. It is expected to hold when the executing thread reaches the yield statement (sequential correctness) and also be preserved by concurrently-executing threads (non-interference). Each procedure is equipped with a precondition, a postcondition, and a set of (potentially) modified thread-local variables. CIVL uses these procedure annotations to verify the sequential correctness of location invariants for each procedure separately. To verify non-interference, it would suffice to check that each location invariant is preserved by each atomic action in the program. CIVL increases the precision of this check by allowing each location invariant to be preserved across an atomic block, introduced as the statement *ablock* $\{e\}$ s. The invariant e annotating the atomic block is expected to hold when this statement begins execution and is verified as part of sequential correctness. The CIVL type checker checks that the statement s inside this atomic block does not have any yield statement or other atomic blocks inside it. Thus, non-interference of a location invariant e' against *ablock* $\{e\}$ s is achieved by proving the Floyd-Hoare triple $\{e \wedge e'\}s\{e'\}$.

Having verified sequential correctness and non-interference for location invariants, it remains to verify refinement, i.e., if $RS(P) = A$, then the atomic action A is correctly refined by the procedure P. This requirement means that any path from entry to exit of P must contain exactly one atomic block that implements the action A; all other atomic blocks on the path must leave global and thread-local variables unchanged. To perform this check, the CIVL verifier introduces the following fresh local variables in P: (1) a *Boolean* variable b initialized to *false* to track whether an atomic block along the current execution has modified a global or thread-local variable, (2) variables to capture snapshot of global and thread-local variables at the beginning of each atomic block. By updating these auxiliary variables appropriately, the refinement check is reduced to a collection of assertions introduced into the body of P at the end of atomic blocks and at the exit of P.

Often, commutativity and non-interference checks require knowledge about distinctness of local program variables in different threads. For example, in Figure 1, to prove that `AcquireLock` commutes to the right of `ReleaseLock`, the verifier must know that the input parameter `tid` to these atomic actions is different if they are being executed by different threads. A similar situation arises in Figure 4, when attempting to prove that the location invariant `t == a[tid]` is preserved by the atomic action `a[tid] := t + 1`. Information about distinctness of program variables in different threads is difficult to provide as a location invariant whose scope is local to the context of the unique executing thread. As an alternative, we exploit reasoning based on a linear type system [42]. The programmer declares certain variables as linear at input and output interfaces of procedures and actions. Using this interface information, the CIVL type system computes a set of *available* linear variables at each control location in a procedure. The availability of a variable may change at an assignment or a procedure call, e.g., if `y` is available just before `x := y`, then `y` is not available and `x` is available just afterwards. The CIVL type checker guarantees that the values contained in available linear variables, across all threads at their respective control locations, are distinct from each other. This fact is introduced as a logical assumption by the verifier when performing commutativity and non-interference checks.

The interaction between the linear type system and logical reasoning in CIVL is more general than the description above. In CIVL, the programmer may specify an arbitrary function *Perm* from a value to a set of values; the set *Perm*(v) is the set of *permissions* associated with v. The example described in the previous paragraph corresponds to the special case when $Perm(v) = \{v\}$. The CIVL type checker enforces a generalization of the distinctness invariant that the permission sets corresponding to the values in available variables across all threads are mutually disjoint.

3.1 Safety guarantee

We can combine the verification techniques described above to verify the safety of a program $Prog^{lo}$. Specifically, we can guarantee that $Prog^{lo}$ is safe (i.e., all atomic actions will satisfy their gates when run) if the following conditions hold:

1. $Prog^{hi}$ is safe when executed with preemptive semantics.
2. $Prog^{lo}$ is a valid refinement of $Prog^{hi}$, according to the rules for refinement in CIVL. Specifically, for any atomic action A in $Prog^{hi}$ implemented by a procedure P in $Prog^{lo}$, any path from entry to exit of P must contain exactly one atomic block that implements the action A; all other atomic blocks on the path must leave the global and thread-local state unchanged. Furthermore, all calls to A in $Prog^{hi}$ are replaced by calls to P in $Prog^{lo}$.
3. The invariants of $Prog^{lo}$ satisfy sequential correctness and non-interference with respect to cooperative semantics.
4. $Prog^{lo}$ is *well-typed* with respect to linearity. Specifically, $Prog^{lo}$ does not try to duplicate any linear variables, and linear variables passed to procedures calls and atomic actions are available as expected by the type checker.
5. The atomic actions in $Prog^{lo}$ satisfy the pairwise commutativity checks.
6. The yield statements in $Prog^{lo}$ are sufficient, according to the yield sufficiency automaton in Figure 2.
7. Any infinite execution of $Prog^{lo}$ must visit a yield statement infinitely often.

By themselves, conditions 1-4 guarantee that $Prog^{lo}$ will be safe when executed with cooperative semantics. Conditions 5-7 then additionally ensure that $Prog^{lo}$ will be safe when executed with preemptive semantics. The technical report [23], which includes formal definitions of all the conditions for an extension of the language in Figure 5, formalizes this safety guarantee into a soundness theorem by establishing a simulation relation between $Prog^{lo}$ and $Prog^{hi}$. Since the theorem connects the safety of one program's preemptive semantics to another program's preemptive semantics, multiple applications of the theorem can be chained together to establish the safety of a low-level program: the lowest level $Prog^0$ is safe because $Prog^1$ is safe, $Prog^1$ is safe because $Prog^2$ is safe, and so on.

4 Modules

The technical report[23] describes a simple module system built on CIVL that allows separate verification of modules, allowing programmers to check a large program by breaking it into smaller pieces and checking the pieces independently. A key challenge for modular verification in CIVL is the checking of non-interference and commutativity. Naively, these are whole-program judgments, quadratically checking all pairs of actions or all pairs of yields and atomic blocks from an entire program. To check these judgments on a per-module basis rather for a whole program, we observe that commutativity and non-interference are trivially satisfied 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 location invariant 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. Statements in M's procedures can only read and write M's own global variables, and M's actions and location invariants can only refer to M's own global variables. (On the other hand, procedure assertions that are not checked for non-interference, such as the e in *ablock* $\{e\}$ s, may mention global variables from other modules, since these assertions can neither interfere with other modules' location invariants nor be interfered with by other modules' statements.)

Note that ownership can change across refinement layers. For example, a library module implementing locks may define a variable to represent the abstract state of a lock; after the lock module is verified at a low layer, another module can take ownership of the lock variable in a higher layer (see [23] for a detailed example of ownership transfer across three layers, from a lock module to a datatype module to a client module).

5 Implementation

We have implemented the method described in this section as a conservative extension of the Boogie [4] 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 [5]. Annotated atomic code blocks are also inferred automatically. Non-interference checks are collected as inlined procedures invoked at appropriate places within the code of a procedure for increased precision.

We automated the simulation relation check used for yield sufficiency in Section 3 by adapting an algorithm by Henzinger et al.[24] for computing the similarity relation of labeled graphs. 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.

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.

6 Experience

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 [14], the ticket algorithm [15], Treiber stack [27], work-stealing queue [6], device cache [13], and lock-protected increment [19]. 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.

6.1 Garbage collector

We have used CIVL to design and verify a realistic concurrent mark-sweep garbage collection (GC) algorithm (available at [22]). In particular, although our algorithm is based on an earlier algorithm by Dijkstra et al [10], it extends the earlier algorithm with various modern optimizations and embellishments to improve generality and performance. These extensions include lower write barrier overhead, phase-based synchronization and handshaking, and coordination between the GC and mutator threads during root scanning; our use of linearity aids the proof of root scanning, while our rely-guarantee encoding aids management of colors inside the write barrier (which is similar to the barrier in Section 2). Furthermore, our encoding of the algorithm in CIVL spans a wide range of abstraction, from low-level memory operations all the way up to high-level specifications; we used six layers of refinement to help hide low-level details from the high-level portions of the verification.

We believe that CIVL's combination of features makes practical, for the first time, verification across such a wide range of abstraction:

– The GC's lowest layers relied primarily on reduction to prove that operations on concurrent data structures and synchronization operations appear atomic to higher layers.

– The GC's higher layers relied primarily on invariant-based non-interference reasoning. This reasoning was simplified because reduction already made lower-layer operations atomic, reducing the amount of interference between higher-layer operations. In addition, the use of location invariants made certain layers of the proof more manageable compared to an earlier effort verifying the same GC where we used rely-guarantee reasoning and auxiliary variables to reason about non-interference.
– 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 expressing mutual exclusion during initialization and during root scanning. In initialization and root scanning, the mutator threads temporarily donate a fraction of their linear permissions to the GC thread. The distinctness invariant from Section 3 guarantees that the mutator threads and GC threads cannot simultaneously possess the same linear permissions; we leverage this guarantee to prove non-interference of mutator and GC actions during initialization and root scanning.

civl's support for refinement also enabled concise specifications of the GC's correctness: a correct GC must implement Allocate, ReadField, and WriteField actions that appear to act atomically, even though the implementations of these operations actually execute concurrently with the GC thread and with other program threads. The specification states that Allocate atomically adds new objects to the heap, while ReadField and WriteField read and write heap object fields. Although the GC's Mark and Sweep code constitutes most of the GC code, they are hidden in the high-level specification; they have detailed correctness specifications in the middle layers of the proof, but the most important point at the high level is that their work not interfere with Allocate, ReadField, and WriteField. In particular, Mark must coordinate with WriteField's write barrier, and Sweep must not remove objects reachable by ReadField and WriteField.

Overall, our GC implementation consists of about 2100 lines of Boogie code. The verification takes 60 seconds on the same PC used for microbenchmarks. The bulk of this time, 54 seconds, is taken by the verification of sequential correctness and non-interference. The checks for linear variables, yield sufficiency, and commutativity take the rest of the time and are insignificant in comparison.

7 Related work

Our work is the first to provide a 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. civl combines a number of techniques in a novel manner to decompose the refinement task following the syntactic structure of a program. Below, we first contrast civl with refinement verification techniques, and then with tools and techniques for reasoning about concurrent programs in general.

7.1 Refinement-oriented verification

Atomic action specifications have been explored by the CALVIN [18, 21] verifier. CIVL 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. CIVL supports both location invariants or rely-guarantee reasoning, and either technique can be used to prove non-interference. 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.

QED [13] 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. [34] 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 [40] 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., [12, 25]) 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+ [32] specification language. Compositional refinement proofs [1] have also been investigated in this context. Modular refinement proofs for hardware systems have been investigated extensively (e.g., [26, 11]) using the SMV [36] 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.

7.2 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 [8] is a tool for verifying concurrent C programs. Chalice [33] 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 [37] 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 [16, 41]. 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. 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.
5. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.
6. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.
7. J. Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.
8. 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.
9. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
10. 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.
11. A. T. Eiríksson. The formal design of 1M-gate ASICs. *Form. Methods Syst. Des.*, 16(1):7–22, Jan. 2000.
12. 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.
13. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.
14. 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.
15. 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.
16. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
17. 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.
18. 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.
19. 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.
20. R. Floyd. Assigning meaning to programs. In *Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
21. S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

22. 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`.
23. C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. Technical Report MSR-TR-2015-8, Microsoft Research, 2015. `http://research.microsoft.com/apps/pubs/?id=238907`.
24. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
25. 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.
26. T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proc. 1999 IEEE/ACM Intl. Conf. on Computer-aided Design*, ICCAD '99, pages 494–499. IEEE Press, 1999.
27. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
28. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
29. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.
30. 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.
31. S. K. Lahiri, S. Qadeer, and D. Walker. Linear maps. In *PLPV*, pages 3–14, 2011.
32. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional, 2004.
33. K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
34. 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.
35. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
36. K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.
37. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
38. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
39. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
40. A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 247–258, New York, NY, USA, 2011. ACM.
41. V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.
42. P. Wadler. Linear types can change the world! In *Programming Concepts and Methods.* North, 1990.
43. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, Apr. 1971.