Armada: Automated Verification of Concurrent Code with Sound Semantic Extensibility

JACOB R. LORCH, Microsoft Research, USA
YIXUAN CHEN, Yale University, USA
MANOS KAPRITSOS and HAOJUN MA, University of Michigan, USA
BRYAN PARNO, Carnegie Mellon University, USA
SHAZ QADEER, Calibra, USA
UPAMANYU SHARMA, Massachusetts Institute of Technology, USA
JAMES R. WILCOX, Certora, USA
XUEYUAN ZHAO, Carnegie Mellon University, USA

Safely writing high-performance concurrent programs is notoriously difficult. To aid developers, we introduce Armada, a language and tool designed to formally verify such programs with relatively little effort. Via a C-like language and a small-step, state-machine-based semantics, Armada gives developers the flexibility to choose arbitrary memory layout and synchronization primitives so that they are never constrained in their pursuit of performance. To reduce developer effort, Armada leverages SMT-powered automation and a library of powerful reasoning techniques, including rely-guarantee, TSO elimination, reduction, and pointer analysis. All of these techniques are proven sound, and Armada can be soundly extended with additional strategies over time. Using Armada, we verify five concurrent case studies and show that we can achieve performance equivalent to that of unverified code.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Concurrent programming languages;**

Additional Key Words and Phrases: Refinement, weak memory models, x86-TSO

ACM Reference format:
Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2022. Armada: Automated Verification of Concurrent Code with Sound Semantic Extensibility. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 12 (May 2022), 39 pages.
https://doi.org/10.1145/3502491

This work was supported in part by the National Science Foundation under grant no. FMitF-2018915, the National Science Foundation and VMware under grant no. CNS-1700521, a grant from the Alfred P. Sloan Foundation, and a Google Faculty Fellowship.

Authors' addresses: J. R. Lorch, Microsoft Research, 1 Microsoft Way, Redmond, WA 98052, USA; email: lorch@microsoft.com; Y. Chen, Yale University, 51 Prospect Street, New Haven, CT 06511, USA; email: yixuan.chen@yale.edu; M. Kapritsos and H. Ma, Department of Computer Science and Engineering, 2260 Hayward St, Ann Arbor, MI, 48109, USA; emails: {manosk, mahaojun}@umich.edu; B. Parno and X. Zhao, School of Computer Science and Carnegie Institute of Technology, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA, 15213, USA; emails: parno@cmu.edu, xueyuanz@alumni.cmu.edu; S. Qadeer, Meta Inc., 1288 123rd Ave NE, Bellevue, WA 98005, USA; email: shaz.qadeer@gmail.com; U. Sharma, 32 Vassar St G978A, Cambridge, MA 02139, USA; email: upamanyu@mit.edu; J. R. Wilcox, Paul G. Allen Center, Box 352350, 185 E Stevens Way NE, Seattle, WA 98195-2350, USA; email: james@certora.com.

0164-0925/2022/05-ART12 $15.00
https://doi.org/10.1145/3502491

1 INTRODUCTION

Ever since processor speeds plateaued in the early 2000s, building high-performance systems has increasingly relied on concurrency. Writing concurrent programs, however, is notoriously error prone, as programmers must consider all possible thread interleavings. If a bug manifests on only one such interleaving, it is extremely hard to detect using traditional testing techniques, let alone to reproduce and repair. Formal verification provides an alternative: a way to guarantee that the program is completely free of such bugs.

This article presents Armada, a methodology, language, and tool that enables low-effort verification of high-performance, concurrent code. Armada's contribution rests on three pillars: *flexibility* for high performance; *automation* to reduce manual effort; and an expressive, low-level framework that allows for *sound semantic extensibility*. These three pillars let us achieve automated verification, with semantic extensibility, of concurrent C-like imperative code executed in a weak memory model (x86-TSO [38]).

Prior work (see Section 8) has achieved some of these, but not simultaneously. For example, Iris [27] supports powerful and sound semantic extensibility but focuses less on automation and C-like imperative code. Conversely, CIVL [21], for instance, supports automation and imperative code without sound extensibility. Instead, it relies on paper proofs when using techniques such as reduction, and the CIVL team is continuously introducing new trusted tactics as they encounter new use cases [40]. Recent work building larger verified concurrent systems [6, 7, 18] supports sound extensibility but sacrifices flexibility and, thus, some potential for performance optimization, to reduce the burden of proof writing.

In contrast, Armada achieves all three properties, which we now expand and discuss in greater detail:

Flexibility. To support high-performance code, Armada lets developers choose any memory layout and any synchronization primitives they need for high performance. Fixing on any one strategy for concurrency or memory management will inevitably rule out clever optimizations that developers come up with in practice. Hence, Armada uses a common low-level semantic framework that allows arbitrary flexibility akin to the flexibility provided by a C-like language. For example, it supports pointers to fields of objects and to elements of arrays, lock-free data structures, optimistic racy reads, and cache-friendly memory layouts. We enable such flexibility by using a small-step state-machine semantics rather than one that preserves structured program syntax but limits *a priori* the set of programs that can be verified.

Automation. However, actually writing programs as state machines is unpleasantly tedious. Hence, Armada introduces a higher-level syntax that lets developers write imperative programs that are automatically translated into state-machine semantics. To prove these programs correct, the developer then writes a series of increasingly simplified programs and proves that each is a sound abstraction of the previous program, eventually arriving at a simple, high-level specification for the system. To create these proofs, the Armada developer simply annotates each level with the proof strategy necessary to support the refinement proof connecting it to the previous level. Armada then analyzes both levels and automatically generates a lemma demonstrating that refinement holds. Typically, this lemma uses one of the libraries we have developed to support eight common concurrent-systems reasoning patterns (e.g., logical reasoning about memory regions, rely-guarantee, TSO elimination, and reduction). These lemmas are then verified by an SMT-powered theorem prover. Explicitly manifesting Armada's lemmas lets developers perform *lemma customization*, that is, augmentations to lemmas in the rare cases in which the automatically generated lemmas are insufficient.

Sound semantic extensibility. Each of Armada's proof-strategy libraries, and each proof generated by our tool, is mechanically proven to be correct. Insisting on verifying these proofs gives us the confidence to extend Armada with arbitrary reasoning principles, including newly proposed approaches, without worrying that in the process we may undermine the soundness of our system. Note that inventing new reasoning principles is an explicit non-goal for Armada. Instead, we expect Armada's flexible design to support new reasoning principles as they arise.

Our current implementation of Armada uses Dafny [29] as a general-purpose theorem prover. Dafny's SMT-based [12] automated reasoning simplifies development of our proof libraries and developers' lemma customizations. However, Armada's broad structure and approach are compatible with any general-purpose theorem prover. We extend Dafny with a backend that produces C code compatible with ClightTSO [46], which can then be compiled to an executable by CompCertTSO in a way that preserves Armada's guarantees.

We evaluate Armada on five case studies and show that it handles complex heap and concurrency reasoning with relatively little developer-supplied proof annotation. We also show that Armada programs can achieve performance comparable to that of unverified code.

In summary, this article makes the following contributions.

- A flexible language for developing high-performance, verified, concurrent systems code.
- A mechanically verified, extensible semantic framework that already supports a collection of eight verified libraries for performing refinement-based proofs, including region-based pointer reasoning, rely-guarantee, TSO elimination, and reduction.
- A practical tool that uses these techniques to enable reasoning about complex concurrent programs with modest developer effort.

This article is an expanded version of an earlier paper [33]. It contains more details about Armada, discusses a later version of the implementation (see Section 5), and describes an additional case study (see Section 6.3).

2 OVERVIEW

As shown in Figure 1, to use Armada, a developer writes an implementation program in the Armada language. The developer also writes an imperative specification, which need not be performant or executable, in that language. This specification should be easy to read and understand so that others can determine (e.g., through inspection) whether it meets their expectations. Given these two programs, Armada's goal is to prove that all finite behaviors of the implementation are permitted by the specification, that is, that the implementation *refines* the specification. The developer defines what this means via a *refinement relation* ($\mathcal{R}$). For instance, if the state contains a console log, the refinement relation might be that the log in the implementation is a prefix of that in the spec.

Because of the large semantic gap between the implementation and specification, we do not attempt to directly prove refinement. Instead, the developer writes a series of N Armada programs to bridge the gap between the implementation (level 0) and the specification (level $N+1$). Each pair of adjacent levels $i, i+1$ in this series should be similar enough to facilitate automatic generation of a refinement proof that respects $\mathcal{R}$; the developer supplies a short proof *recipe* that gives Armada enough information to automatically generate such a proof. Given the pairwise proofs, Armada leverages refinement transitivity to prove that the implementation indeed refines the specification.

We formally express refinement properties and their proofs in the Dafny language [29]. To formally describe what refinement means, Armada translates each program into its small-step state-machine semantics, expressed in Dafny. For instance, we represent the state of a program as a Dafny datatype and the set of its legal transitions as a Dafny predicate over pairs of states. To formally prove refinement between a pair of levels, we generate a Dafny lemma whose conclusion

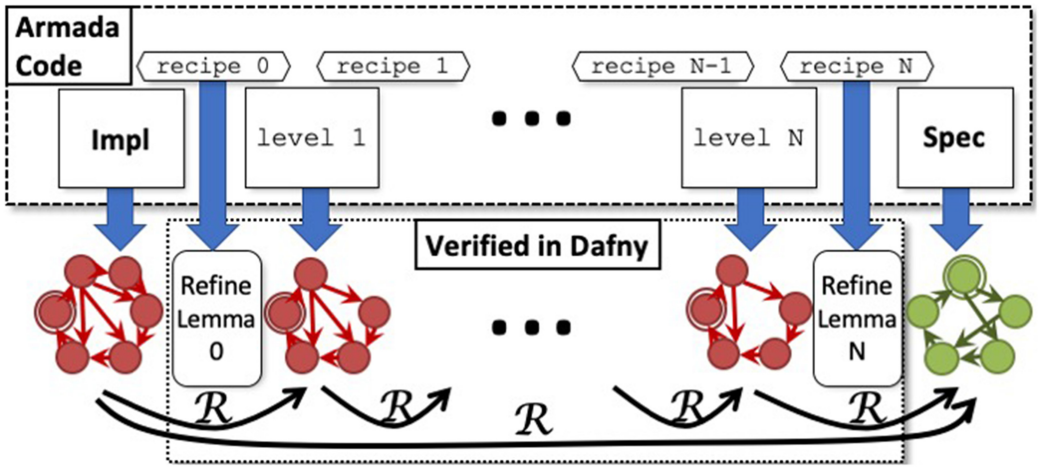

Fig. 1. **Armada Overview** The Armada developer writes a low-level implementation in Armada designed for performance. The developer then defines a series of levels, each of which abstracts the program at the previous level, eventually reaching a small, simple specification. Each refinement is justified by a simple refinement recipe specifying which refinement strategy to use. As shown via blue arrows, Armada automatically translates each program into a state machine and generates refinement proofs demonstrating that the refinement relation $\mathcal{R}$ holds between each pair of levels. Finally, it uses transitivity to show that $\mathcal{R}$ holds between the implementation and the spec.

indicates a refinement relation between their state machines. We use Dafny to verify all proof material we generate so that ultimately the only aspect of Armada we must trust is its translation of the implementation and specification into state machines.

2.1 Example Specification and Implementation

To introduce Armada, we describe its use on an example program that searches for a good, but not necessarily optimal, solution to an instance of the traveling salesman problem.

The specification, shown in Figure 2, demands that the implementation output a valid solution (i.e., one that visits every city exactly once), and it implicitly requires the program not to crash. Armada specifications can use powerful declarations as statements. Here, the `somehow` statement expresses that somehow the program updates `s` so that `valid_soln(s)` holds.

The example implementation, also shown in Figure 2, creates 100 threads. Each thread searches through 10,000 random solutions. If a thread finds a solution shorter than the best length found so far, it updates the global variables storing the best length and solution. The main routine joins the threads and prints the best solution found.

Note that this example has a benign race: the access to the shared variable `best_len` in the first `if (len < best_len)`. It is benign because the worst consequence of reading a stale value is that the thread unnecessarily acquires the mutex.

2.2 Example Proof Strategy

We bridge the gap between the implementation and specification by introducing intermediate levels that gradually simplify the implementation. Figure 3 depicts the program, called `ArbitraryGuard`, at level 1 in our example proof. This program is like the implementation except that it *arbitrarily* chooses whether to acquire the lock by using `*` in place of the guard condition `len < best_len`.

```
level Specification {
  method main() {
    var s:Solution;
    somehow modifies s ensures valid_soln(s);
    output_solution(s);
  }
}

level Implementation {
  // Global variables
  var best_solution:Solution;
  var best_len:uint32 := 0xFFFFFFFF;
  var mutex:Mutex;

  method worker() { // Search for good solution
    var i:int32 := 0, s:Solution, len:uint32;
    while i < 10000 {
      choose_random_solution(&s);
      len = get_solution_length(&s);
      if (len < best_len) {
        lock(&mutex);
        if (len < best_len) {
          best_len := len;
          copy_solution(&best_solution, &s);
        }
        unlock(&mutex);
      }
      i := i + 1;
    }
  }

  method main() {
    var i:int32 := 0;
    var a:uint64[100];
    initialize_mutex(&mutex);
    while i < 100 {
      a[i] := create_thread worker();
      i := i + 1;
    }
    i := 0;
    while i < 100 {
      join a[i];
      i := i + 1;
    }
    print_solution(&best_solution);
  }
}
```

Fig. 2. The Armada spec and implementation for our running example, which searches for a not-necessarily-optimal solution to a traveling salesman problem.

```
level ArbitraryGuard {
  ...
  len = get_solution_length(&s);
  if (*) {   // arbitrary choice as guard
    lock(&mutex);
    if (len < best_len) {
      best_len := len;
      copy_solution(&best_solution, &s);
    }
    unlock(&mutex);
  }
  ...
}
```

Fig. 3. Version of our example program in which the first guard condition is relaxed to an arbitrary choice.

Our transformation of the `Implementation` program to the `ArbitraryGuard` program is an example of *weakening*, in which a statement is replaced by one whose behaviors are a superset of the original. Or, more precisely, a state-transition relation is replaced by a superset of that relation. The two levels' programs thus exhibit *weakening correspondence*, that is, it is possible to map each

```
proof ImplementationRefinesArbitraryGuard {
  refinement Implementation ArbitraryGuard
  weakening
}
```

Fig. 4. In this recipe for a refinement proof, the first line indicates what should be proved (that the `Implementation`-level program refines the `ArbitraryGuard`-level program) and the second line indicates which strategy (in this case, weakening) generates the proof.

```
level BestLenSequential {
  ...
  if (len < best_len) {
    best_len ::= len; // immediately visible to
                      // all threads
    copy_solution(&best_solution, &s);
  }
  ...
}
```

Fig. 5. Version of the example program in which the assignment to `best_len` is now sequentially consistent.

```
proof ArbitraryGuardRefinesBestLenSequential {
  refinement ArbitraryGuard BestLenSequential
  tso_elim best_len "s.s.globals.mutex.holder == tid"
}
```

Fig. 6. This recipe proves that the `ArbitraryGuard`-level program refines the `BestLenSequential`-level program. It uses TSO elimination based on strategy-specific parameters. In this case, the first parameter (`best_len`) indicates which location's updates differ between levels and the second parameter is an ownership predicate.

low-level program step to an equivalent or weaker one in the high-level program. The proof that `Implementation` refines `ArbitraryGuard` is straightforward but tedious to write. Instead, the developer simply writes a recipe for this proof, shown in Figure 4. This recipe instructs Armada to generate a refinement proof using the weakening correspondence between the program pair.

Having removed the racy read of `best_len`, we can now demonstrate an ownership invariant: that threads only access that variable while they hold the mutex and no two threads ever hold the mutex. This allows a further transformation of the program to the one shown in Figure 5. This replaces the assignment `best_len := len` with `best_len ::= len`, signifying the use of sequentially consistent memory semantics for the update rather than x86-TSO semantics [38]. Since strong consistency is easier to reason about than weak-memory semantics, proofs for further levels will be easier.

Just as for weakening, Armada generates a proof of refinement between programs whose only transformation is a replacement of assignments to a variable with sequentially consistent assignments. For such a proof, the developer's recipe supplies the variable name and the ownership predicate, as shown in Figure 6.

If the developer mistakenly requests a TSO-elimination proof for a pair of levels that do not permit it (e.g., if the first level still has the racy read and, thus, does not own the location when it accesses it), then Armada will either generate an error message indicating the problem or generate an invalid proof. In the latter case, running the proof through the theorem prover (i.e., Dafny verifier) will produce an error message. For instance, it might indicate which statement may access the variable without satisfying the ownership predicate or which statement might cause two threads to simultaneously satisfy the ownership predicate.

3 SEMANTICS AND LANGUAGE DESIGN

Armada is committed to allowing developers to adopt any memory layout and synchronization primitives needed for high performance. This affects the design of the Armada language and our choice of semantics.

The Armada language (Section 3.1) allows the developer to write specification, code, and proofs in terms of programs, and the core language exposes low-level primitives (e.g., fixed-width integers or specific hardware-based atomic instructions) so that the developer is not locked into a particular abstraction and can reason about the performance of the code without an elaborate mental model of what the compiler might do. This also simplifies the Armada compiler.

To facilitate simpler, cleaner specifications and proofs, Armada also includes high-level and abstract features that are not compilable. For example, Armada supports mathematical integers, and it allows arbitrary sequences of instructions to be performed atomically (given suitable proofs).

The semantics of an Armada program (Section 3.2), however, are expressed in terms of a small-step state machine, which provides a "lowest common denominator" for reasoning via a rich and diverse set of proof strategies (Section 4). It also avoids baking in assumptions that facilitate one particular strategy but preclude others.

3.1 The Armada Language

As shown in Figure 1, developers express implementations, proof steps, and specifications all as programs in the Armada language. This provides a natural way of describing refinement: an implementation refines a specification if all of its externally visible behaviors are valid behaviors of the specification. The developer helps prove refinement by bridging the gap between implementation and specification via intermediate-level programs.

We restrict the implementation level to the core Armada features (Section 3.1.1), which can be compiled directly to corresponding low-level C code. The compiler will reject programs outside this core. Programs at all other levels, including the specification, can use the entirety of Armada (Section 3.1.2), summarized in Figure 7. Developers connect these levels using a refinement relation (Section 3.1.3). To let Armada programs use external libraries and special hardware features, we also support developer-defined external methods (Section 3.1.4).

3.1.1 Core Armada. The core of Armada supports features commonly used in high-performance C implementations. It has as primitive types signed and unsigned integers of 8, 16, 32, and 64 bits, and pointers. It supports arbitrary nesting of **`struct`**s and single-dimensional arrays, including **`struct`**s of arrays and arrays of **`struct`**s. It lets pointers point not only to whole objects but also to fields of **`struct`**s and elements of arrays. It does not yet support `unions`.

For control flow, it supports method calls, **`return`**, **`goto`**, **`if`**, and **`while`**, along with **`break`** and **`continue`**.

It supports allocation of objects (**`malloc`**) and arrays of objects (**`calloc`**), and freeing them (**`dealloc`**). It supports creating threads (**`create_thread`**) and waiting for their completion (**`join`**).

For synchronization between threads, it supports read-modify-write instructions such as **`atomic_exchange`** and **`compare_and_swap`**.

Each statement may have at most one shared-location access, since the hardware does not support atomically performing multiple shared-location accesses. To aid our compiler and proof generators in identifying non-shared locations, one can mark a local variable as `noaddr`, meaning that it is illegal to take its address and, consequently, it can be assumed to be non-shared.

3.1.2 Proof and Specification Support. The full Armada language offers rich expressivity to allow natural descriptions of specifications. Furthermore, all program levels between the

Types

```
T ::= uint8 | uint16 | uint32 | uint64
    | int8 | int16 | int32 | int64                    (primitive types)
    | ptr<T>                                         (pointers)
    | T[N]                                           (arrays)
    | struct {var ⟨field⟩:T; ...}                    (structs)
    | int | (T, ..., T) | T → T                      (mathematical types)
    | x : T "|" e                                    (subset types)
    | ...
```

Expressions

```
e ::= ⟨literal⟩ | ⟨variable⟩
    | ⟨uop⟩ e | e1 ⟨bop⟩ e2                          (unary/binary operators)
    | &e | *e | null                                 (pointer manipulation)
    | e.⟨field⟩                                      (struct manipulation)
    | e1[e2]                                         (indexing)
    | *                                              (non-deterministic value)
    | old(e)                                         (old value of e in two-state predicate)
    | allocated(e) | allocated_array(e)              (validity)
    | $me | $sb_empty | $state                       (meta variables)
    | global_view(e)                                 (global view evaluation)
    | if_undefined(e, e)                             (safe expression evaluation)
    | ...
```

Statements

```
⟨LHS⟩ ::= ⟨variable⟩ | *e | e.⟨field⟩ | e[e]
⟨RHS⟩ ::= e | ⟨method⟩(e, ...)
        | malloc(T) | calloc(T, e)                   (allocation)
        | create_thread ⟨method⟩(e, ...)             (threads)
        | compare_and_swap(e, e, e)
        | atomic_exchange(e, e)                      (read-modify-write action)
⟨spec⟩ ::= undefined_unless e
        | modifies e | ensures e
     S ::= var ⟨variable⟩:T [:= ⟨RHS⟩];
        | ⟨LHS⟩, ... := ⟨RHS⟩, ...;                  (assignment)
        | ⟨LHS⟩, ... ::= ⟨RHS⟩, ...;                 (TSO-bypassing assignment)
        | if e S1 else S2 | while e1 [invariant e2] S
        | break; | continue; | assert e; | S1 S2
        | dealloc e; | join e; | label ⟨label⟩: S
        | goto ⟨label⟩
        | somehow ⟨spec⟩*;                           (declarative atomic action)
        | explicit_yield {S} | yield;                (atomicity)
        | assume e; S                                (enablement condition)
```

Fig. 7. Armada language syntax.

implementation and specification are abstract constructs that exist solely to facilitate the proof; thus, they also use this full expressivity. Here, we briefly describe interesting features of the language.

Meta variables provide access to machine states in an Armada program. Variable **`$me`** evaluates to the current thread's ID, **`$sb_empty`** evaluates to true when the current thread's store buffer (Section 3.2.1) is empty, and **`$state`** allows access to miscellaneous fields of the current total state.

Meta functions let one express properties of the state that cannot be described by normal expressions. One uses **`global_view`**`(e)` to evaluate `e` as if the store buffer were empty, that is, from the perspective of all other threads. One uses **`if_undefined`**`(e1, e2)` to turn an expression that may cause undefined behavior into one that cannot; for example, **`if_undefined`**`(*p, 0)` is `*p` if it is legal to dereference `p` and `0` otherwise.

Atomic blocks are modeled as executing to completion without interruption by other threads. The semantics of an atomic block prevents thread interruption but not termination; a behavior may terminate in the middle of an atomic block. This allows us to prove that a block of statements can be treated as atomic without having to prove that no statement in the block exhibits undefined behavior (see Section 3.2.3).

Following CIVL [21], we permit some program counters within otherwise atomic blocks to be marked as *yield points*. Hence, the semantics of an **explicit_yield** block is that a thread t within such a block cannot be interrupted by another thread unless t's program counter is at a yield point (marked by a **yield** statement). This permits modeling atomic sequences that span loop iterations without having to treat the entire loop as atomic. Section 4.2.1 shows the utility of such sequences, and Flanagan et al. describe further uses in proofs of atomicity via purity [16].

Enablement conditions can be attached to a statement that cannot execute unless all of its conditions are met. One adds an enablement condition C to a statement by preceding that statement with the statement **assume** C. To make this easier for developers to understand, we provide an alias of **wait_until** for the keyword **assume**; the developer can then think of the program as blocking at that point until the condition becomes true. If similar conditions need to be assumed at multiple sites, developers can define a *universal step constraint*, which is an enablement condition on every instruction in the program. As an example, to add a requirement that the variable `w` must be positive before any statement can execute, one writes `universal_step_constraint WPositive { w >= 0 }`. Such a constraint is a top-level declaration, not a statement; thus, its syntax does not appear in Figure 7.

TSO-bypassing assignment statements perform an update with sequentially consistent semantics. Normal assignments (using `:=`) follow x86-TSO semantics (Section 3.2.1), but assignments using `::=` are immediately visible to other threads.

Assert statements crash the program if their predicates do not hold.

Somehow statements allow the declarative expression of arbitrary atomic actions. A **somehow** statement can have **undefined_unless** clauses (preconditions), **modifies** clauses (framing), and **ensures** clauses (postconditions). The semantics of a **somehow** statement is that it has undefined behavior if any of its preconditions are violated and that it modifies the lvalues in its framing clauses arbitrarily subject to the constraint that each two-state postcondition predicate holds between the old and new states. The two-state predicate in an **ensures** clause uses the standard Dafny [29] style of referring to components of the pre-state within **old** expressions. For instance, to say that the variable `x` must be at least twice what it was before, one could use `ensures x >= old(x) * 2`.

Ghost variables represent the state that is not part of the machine state and has sequentially consistent semantics. Ghost variables can be of any type supported by the theorem prover, not just those that can be compiled to C. Ghost types supported by Armada include mathematical integers; datatypes; sequences; and finite and infinite sets, multisets, and maps.

Ghost types cannot appear in a **struct** or array, to simplify heap modeling. This restriction does not significantly reduce expressivity, since ghost variables support powerful datatype and sequence constructs. The downside of this restriction is that developers cannot introduce a ghost state directly into an object at higher levels. A reasonable alternative is to introduce a ghost variable mapping object pointers to the extra state. That is, instead of replacing a struct `S` with a struct `S'` that includes a field `F` of ghost type `T`, one might instead add a ghost variable `m` of type `map<ptr<S>,T>`, and maintain the invariant that `m(p)` contains the value that should have been in `(*p).F`.

The ghost state is permitted at all levels, but it may only affect control flow and the non-ghost state in the abstract levels above the implementation level. The ghost state is permitted to affect

```
var snapshot;
if (!precondition_satisfied()) {
  ManifestUndefinedBehavior();
}
havoc_write_set();
snapshot := read_read_set();
while (* || !post_condition_satisfied()) {
  if (snapshot != read_read_set()) {
    ManifestUndefinedBehavior();
  }
  havoc_write_set();
  snapshot := read_read_set();
}
```

Fig. 8. This is the default model for external methods, in which the read set is the list of locations in `reads` clauses and the write set is the list of locations in **`modifies`** clauses. `read_read_set` reads all of the locations in the read set, `havoc_write_set` non-deterministically writes to all of the write set locations, and `ManifestUndefinedBehavior` triggers undefined behavior and terminates execution (see Section 3.2.3).

control flow at abstract levels since all states in an abstract program, including the program counter, are effectively ghost states.

3.1.3 Refinement Relations. Armada aims to prove that the implementation *refines* the specification. The developer defines, via a *refinement relation* $\mathcal{R}$, what refinement means. Formally, $\mathcal{R} \subseteq S_0 \times S_{N+1}$, where S_i is the set of states of the level-i program, level 0 is the implementation, and level $N + 1$ is the spec. A pair $\langle s_0, s_{N+1} \rangle$ is in $\mathcal{R}$ if the state is permitted to be s_0 whenever according to the specification it may be s_{N+1}. An implementation refines the specification if every finite behavior of the implementation has, with the addition of stuttering steps, a corresponding equal-length behavior of the specification where corresponding state pairs are in $\mathcal{R}$.

The developer writes $\mathcal{R}$ as an expression parameterized over the low-level and high-level states. Hence, we can also use $\mathcal{R}$ to define what refinement means between programs at consecutive levels in the overall refinement proof, that is, to define $\mathcal{R}_{i,i+1}$ for arbitrary level i. To allow composition into an overall proof, $\mathcal{R}$ must be transitive: $\forall i, s_i, s_{i+1}, s_{i+2} \;.\; \langle s_i, s_{i+1} \rangle \in \mathcal{R}_{i,i+1} \wedge \langle s_{i+1}, s_{i+2} \rangle \in \mathcal{R}_{i+1,i+2} \Rightarrow \langle s_i, s_{i+2} \rangle \in \mathcal{R}_{i,i+2}$.

A typical refinement proof involves the developer finding a relation R and demonstrating that each behavior of the implementation corresponds to a behavior of the specification where corresponding state pairs are in R. Note that such a relation R is merely a proof tool and is not trusted. To prove refinement, one must prove that $R \subseteq \mathcal{R}$.

3.1.4 External Methods. Since we do not expect Armada programs to run in a vacuum, Armada supports declaring and calling *external methods*. An external method models a runtime, library, or operating-system function; or a hardware instruction the compiler supports, such as test-and-set. For example, the developer could model a runtime-supplied print routine via:

```
method {:extern} PrintInteger(n:uint32) {
  somehow modifies log ensures log == old(log) + [n];
}
```

In a sequential program, we could model an external call via a straightforward Hoare-style signature. However, in a concurrent setting, this could be unsound if, for example, the external library were not thread-safe. Hence, we allow the Armada developer to supply a more detailed, concurrency-aware model of the external call as a "body" for the method. This model is not, of course, compiled; rather, it specifies the effects of the external call on Armada's underlying state-machine model. Since it is not compiled, it may use the uncompilable subset of Armada.

If the developer does not supply a model for an external method, we model it via the Armada code snippet in Figure 8. That is, we model the method as making arbitrary and repeated changes to

Types

$x, y, z \in \text{Var}$	Variables	$p \in \text{Ptr}$	Pointers
$v \in \text{Val}$	Values	$e \in \text{Expr}$	Expressions
$l \in \text{Loc}$	Locations	$\text{pc} \in \text{PC}$	Program counter
$\text{tid} \in \text{Tid} \triangleq \mathbb{N}$	Thread ID	$b \in \text{Entry} \triangleq \text{Loc} \times \text{Val}$	Store buffer entry
$f \in \text{Frame} \triangleq \text{Var} \rightarrow \text{Val}$	Stack frame	$T \in \text{Thread} \triangleq \text{PC} \times \text{list Frame} \times \text{list Entry}$	Thread state
$H \in \text{Heap} \triangleq \text{Ptr} \rightarrow \text{Val}$	Heap state	$\text{Mem} \triangleq (\text{Var} \rightarrow \text{Val}) \times \text{Heap}$	Shared Memory
$G \in \text{Ghost} \triangleq \text{Var} \rightarrow \text{Val}$	Ghost state	$S \in \text{State} \triangleq (\text{Tid} \rightarrow \text{Thread}) \times \text{Mem} \times \text{Ghost}$	Total state
$C \in \text{Command}$	Comands	$\text{prog} \in \text{Prog} \triangleq \text{PC} \rightarrow \text{Command}$	Program
$\text{update} :: (\text{tid} \times \text{Entry} \times \text{State}) \rightarrow \text{State}$	memory update	$\text{eval} :: (\text{tid} \times \text{Expr} \times \text{State}) \rightarrow \text{Val}$	(r-)expression evaluation
$\text{UB} :: (\text{Command} \times \text{State}) \rightarrow \text{bool}$	Undefined behaviors	$\xrightarrow[\text{prog}]{\text{tid}} :: (\text{prog} \times \text{tid} \times \text{State} \times \text{State}) \rightarrow \text{bool}$	small-step state transition

Buffer Application, Local View, and Update

Seq-Cons

$$\frac{\vec{b} = [b_0] + \vec{b'}}{\text{apply}(\text{tid}, \vec{b}, S) = \text{apply}(\text{tid}, \vec{b'}, \text{update}(\text{tid}, b_0, S))}$$

Seq-Nil

$$\text{apply}(\text{tid}, [], S) = S$$

Local-View

$$\frac{S = (\text{TMap}, M, G) \qquad \text{TMap}[\text{tid}] = (\text{pc}, f, \vec{b})}{S|_{\text{tid}} = \text{apply}(\text{tid}, \vec{b}, S)}$$

Fig. 9. Operational semantics, Part I.

its write set (as specified in a **`modifies`** clause); as having undefined behavior if a concurrent thread ever changes its read set (as specified in a `reads` clause); and as returning when its postcondition is satisfied, but not necessarily as soon as it is satisfied.

3.2 Small-Step State-Machine Semantics

To create a soundly extensible semantic framework, Armada translates an Armada program into a state machine that models its small-step semantics. We represent the state of a program as a Dafny datatype that contains the set of threads, the heap, static variables, ghost state, and whether and how the program terminated. The thread state includes the program counter, the stack, and the x86-TSO store buffer (Section 3.2.1). We represent steps of the state machine (i.e., the set of legal transitions) as a Dafny predicate over pairs of states. Examples of steps include assignment, method calls and returns, and evaluating the guard of an **`if`** or **`while`**.

Figures 9 and 10 present a simplified operational semantics covering core concepts of the model, eliding details of expression evaluation, memory updates, and undefined-behavior conditions. The transition rules presented in Figure 10 demonstrate peculiarities of the x86-TSO model (Tau, TSO-Write, Fence, Xchg) and illustrate the expressiveness of the specification language (TSO-Bypassing-Write, Assume). In the subsequent subsections, we highlight four interesting elements of our semantics: they encode the complex x86-TSO model (Section 3.2.1), they are program specific (Section 3.2.2), they model undefined behavior as a terminating state (Section 3.2.3), and they model the heap as immutable (Section 3.2.4).

3.2.1 x86 Total-Store Order (TSO). We model memory using x86-TSO semantics [38]. Specifically, a thread's write is not immediately visible to other threads; rather, it enters a *store buffer*, a first-in-first-out (FIFO) queue. To model this, our state includes a global memory of type Mem and, for each thread, a store buffer of type list Entry. A thread's local view ($S|_{\text{tid}}$) of memory is what would result from applying its store buffer, in FIFO order, to the global memory.

Transitions

TAU

$$\frac{\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \quad \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, [b_0] + \vec{b}) \quad \text{update}(\text{tid}, b_0, S) = (\text{TMap}', M', G)}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}[\text{tid} \mapsto (\text{pc}, \vec{f}, \vec{b})], M', G)}$$

TSO-WRITE

$$\frac{\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \quad \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, \vec{b}) \quad \text{prog}[\text{tid}] = \texttt{l} := \texttt{e} \quad \text{eval}(\text{tid}, e, S|_{\text{tid}}) = v}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}[\text{tid} \mapsto (\text{next}(\text{pc}), \vec{f}, \vec{b} + (l, v))], M, G)}$$

TSO-BYPASSING-WRITE

$$\frac{\begin{array}{c}\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \quad \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, \vec{b}) \\ \text{prog}[\text{tid}] = \texttt{l} ::= \texttt{e} \quad \text{eval}(\text{tid}, e, S|_{\text{tid}}) = v \quad \text{update}(\text{tid}, (l, v), S) = (\text{TMap}', M', G') \quad \text{TMap}'[\text{tid}] = (\text{pc}, \vec{f}', \vec{b})\end{array}}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}'[\text{tid} \mapsto (\text{next}(\text{pc}), \vec{f}', \vec{b})], M', G')}$$

FENCE

$$\frac{\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \quad \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, []) \quad \text{prog}[\text{tid}] = \texttt{fence}}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}[\text{tid} \mapsto (\text{next}(\text{pc}), \vec{f}, [])], M, G)}$$

CAS-SUCCESS

$$\frac{\begin{array}{c}\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \\ \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, []) \quad \text{prog}[\text{tid}] = \texttt{l}_2 := \texttt{CAS}(\texttt{l}_1, \texttt{e}_{\texttt{new}}, \texttt{e}_{\texttt{old}}) \quad \text{eval}(\text{tid}, l_1, S) = \text{eval}(\text{tid}, e_{\texttt{old}}, S) \\ \text{update}(\text{tid}, (l_1, \text{eval}(\text{tid}, e_{\texttt{new}}, S)), \text{update}(\text{tid}, (l_2, \text{eval}(\text{tid}, l_1, S)), S)) = (\text{TMap}', M', G') \quad \text{TMap}'[\text{tid}] = (\text{pc}, \vec{f}', [])\end{array}}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}'[\text{tid} \mapsto (\text{next}(\text{pc}), \vec{f}', [])], M', G')}$$

CAS-FAILURE

$$\frac{\begin{array}{c}\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \\ \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, []) \quad \text{prog}[\text{tid}] = \texttt{l}_2 := \texttt{CAS}(\texttt{l}_1, \texttt{e}_{\texttt{new}}, \texttt{e}_{\texttt{old}}) \quad \text{eval}(\text{tid}, l_1, S) \neq \text{eval}(\text{tid}, e_{\texttt{old}}, S) \\ \text{update}(\text{tid}, (l_2, \text{eval}(\text{tid}, l_1, S)), S) = (\text{TMap}', M', G') \quad \text{TMap}'[\text{tid}] = (\text{pc}, \vec{f}', [])\end{array}}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}'[\text{tid} \mapsto (\text{next}(\text{pc}), \vec{f}', [])], M', G')}$$

XCHG

$$\frac{\begin{array}{c}\neg\text{UB}(\text{prog}[\text{tid}], S) \quad S = (\text{TMap}, M, G) \quad \text{TMap}[\text{tid}] = (\text{pc}, \vec{f}, []) \quad \text{prog}[\text{tid}] = \texttt{l}_2 := \texttt{xchg}(\texttt{l}_1, \texttt{e}) \\ \text{update}(\text{tid}, (l_1, \text{eval}(\text{tid}, e, S)), \text{update}(\text{tid}, (l_2, \text{eval}(\text{tid}, l_1, S)), S)) = (\text{TMap}', M', G') \quad \text{TMap}'[\text{tid}] = (\text{pc}, \vec{f}', [])\end{array}}{S \xrightarrow[\text{prog}]{\text{tid}} (\text{TMap}'[\text{tid} \mapsto (\text{next}(\text{pc}), \vec{f}', [])], M', G')}$$

ASSUME

$$\frac{\neg\text{UB}(\text{prog}[\text{tid}], S) \quad \text{prog}[\text{tid}] = \texttt{assume e; C} \quad \text{eval}(\text{tid}, e, S) = \text{true} \quad S \xrightarrow[\text{prog}[\text{tid} \mapsto C]]{\text{tid}} S'}{S \xrightarrow[\text{prog}]{\text{tid}} S'}$$

Fig. 10. Operational semantics, Part II.

We formalize the behavior of x86-TSO using the TSO-WRITING rule. A write becomes globally visible when the processor asynchronously drains it from a store buffer. This asynchronicity is modeled by the non-deterministic triggering of the TAU rule. The developer can use a memory-fence instruction **`fence`** to flush the store buffer. The FENCE rule formalizes this behavior by imposing an enabling condition that the store buffer is empty. This alternative model is equivalent to flushing due to the presence of the TAU rule.

We have found it useful to include some ghost state in each store buffer entry: the PC of the instruction that added the entry. This ghost state simplifies writing proofs without changing the semantics. The additional ghost information is not presented in Figure 9, as it does not affect executions of state machines.

3.2.2 Program-Specific Semantics. To aid in automated verification of state-machine properties, we tailor each state machine to the program rather than make it generic to all programs. Such specificity ensures that the verification condition for a specific step relation includes only facts about that step.

A program-specific semantics for any specific instruction is simpler than a generic semantics because it only has to specify the behavior of the semantics in the case of that instruction. Thus, the formula for the program-specific semantics is less complex and smaller. Furthermore, program-specific semantics bypasses the trouble of variable binding in typical deep embedding semantics frameworks. Instead of modeling the variable binding mechanism manually, the state machine translator defers the task to the host langauge by generating specific abstract datatypes for global and stack variables. As a result, the eval and update functions can access and modify the corresponding fields in the abstract datatype with ease.

Specificity also aids reasoning by case analysis by restricting the space of program counters, heap types, and step types. A generic semantics may have to quantify over types-for example, the semantics of + depends on whether it is adding **`int32`**s or **`int64`**s—whereas a program-specific semantics never has to do that. Specifically, the program-counter type is an enumerated type that includes only PC values in the program. The state's heap allows only built-in types and user-defined **`struct`** types that appear in the program text.

Furthermore, the state-machine step (transition) type is an enumerated type that includes only the specific steps in the program. Each step type has a function that describes its specific semantics. In other words, the transition rules shown in Figure 10 are specialized to individual instructions in the program. For instance, there is no generic function for executing an update statement (rule TSO-Write). Instead, for each update statement, there is a program-specific step function with the specific lvalue and rvalue from the statement.

The end result is an SMT-friendly semantics. That is, Dafny automatically discharges many proofs with little or no help.

3.2.3 Undefined Behavior as Termination. Our semantics has three terminating states. These occur when the program exits normally, when asserting a false predicate, and when invoking undefined behavior. The latter means executing a statement under conditions we do not model, for example, an access to a freed pointer or a division by zero. Our decision to model undefined behavior as termination follows CIVL [21] and simplifies our specifications by removing a great deal of non-determinism. It also simplifies reasoning about behaviors, for example, by letting developers state invariants that do not necessarily hold after such an undefined action occurs. However, this decision means that, as in CIVL, our refinement proofs are meaningless if (1) the spec ever exhibits undefined behavior, or (2) the refinement relation $\mathcal{R}$ allows the low-level program to exhibit undefined behavior when the high-level program does not. We prevent condition (2) by adding to the developer-specified $\mathcal{R}$ the conjunct "if the low-level program exhibits undefined behavior, then the high-level program does." Preventing condition (1) currently relies on the careful attention of the specification writer (or reader).

We have found it useful to model any instruction that can cause undefined behavior as two different state machine transitions: one that causes undefined behavior and one that avoids it. That is, in the list of state machine transition cases, we have two cases for each instruction that can cause undefined behavior. For instance, if one instruction is `r := n / d`, we have one transition case

for when d is zero and one for when it is non-zero. This simplifies proof generation since, as we will discuss in Section 4, we often break a proof about the state machine into one proof for each state machine transition. Having a separate case for undefined-behavior steps simplifies the proof for each case, since each such proof has to consider only one type of outcome.

3.2.4 Immutable Heap Structure. To permit pointers to fields of **`struct`**s and to array elements, we model the heap as a forest of pointable-to objects. The roots of the forest are (1) allocated objects and (2) global and local variables whose addresses are taken in the program text. An array object has its elements as children and a **`struct`** object has its fields as children. To simplify reasoning, we model the heap as unchanging throughout the program's lifetime; that is, allocation is modeled not as creating an object but as finding an object and marking its pointers as valid; freeing an object marks all of its pointers as freed. This design does not prevent us from using dynamic data structures such as linked lists, as the heap model still supports the dynamic allocation of new objects and connecting them through pointers.

To make this sound, we restrict allowable operations to ones whose compiled behaviors lie within our model. Some operations, such as dereferencing a pointer to freed memory or comparing another pointer to such a pointer, trigger undefined behavior. We disallow all other operations whose behavior could diverge from our model. For instance, we disallow programs that cast pointers to other types or that perform mathematical operations on pointers.

Due to their common use in C array idioms, we do permit comparison between pointers to elements of the same array, and adding to (or subtracting from) a pointer to an array element. That is, we model pointer comparison and offsetting but treat them as having undefined behavior if they stray outside the bounds of a single array. Unfortunately, this prevents the C idiom of treating a pointer to the nth element of an n-element array as a sentinel for the end of the array. Instead, the programmer must use a pointer to the $n-1$st element.

4 REFINEMENT FRAMEWORK

Armada's goals rely on our extensible framework for automatic generation of refinement proofs. The framework consists of:

Strategies. A *strategy* is a proof generator designed for a particular type of correspondence between a low-level and a high-level program. An example correspondence is *weakening*; two programs exhibit it if they match except for statements in which the high-level version admits a superset of behaviors of the low-level version.

Library. Our *library* of generic lemmas is useful in proving refinements between programs. Often, they are specific to a certain correspondence.

Recipes. The developer generates a refinement proof between two program levels by specifying a *recipe*. A recipe specifies which strategy should generate the proof, and the names of the two program levels. A recipe can also include strategy-specific annotations; for instance, the variable-hiding strategy described in Section 4.2.8 takes as additional arguments the list of hidden variables. Figure 4 shows an example.

Verification experts can extend the framework with new strategies and library lemmas. Developers can leverage these new strategies via recipes. Armada ensures sound extensibility because for a proof to be considered valid, all of its lemmas and all of the lemmas in the library must be verified by Dafny. Hence, arbitrarily complex extensions can be accommodated. For instance, we need not worry about unsoundness or incorrect implementation of the Cohen-Lamport reduction logic we use in Section 4.2.1 or the rely-guarantee logic we use in Section 4.2.2.

Many of the proofs we must generate are so elaborate that automated verification would take too long to verify them or, worse, only verify them sometimes, leading to proof instability. Thus, we break each proof into modest-sized lemmas, with each lemma small enough to be amenable to automated verification. Generally, our approach is to use one lemma for each step or, in the case of two-step properties, each pair of steps. Furthermore, we enforce local reasoning by selectively revealing definitions of state machines so that only the related elements are visible to the prover (Section 4.1.2). Generating such a large number of lemmas would be unreasonably tedious for a human proof writer; thus, it is fortunate that we can use an automatic proof generator. This lets us obtain stable proofs with little human cost.

4.1 Aspects Common to All Strategies

Each strategy can leverage a set of Armada tools. For instance, we provide machinery to prove that developer-supplied inductive invariants are inductive and to produce a refinement function that maps low-level states to high-level states.

The most important generic proof technique we provide is *non-determinism encapsulation*. State-transition relations are non-deterministic because some program statements are non-deterministic; for example, a method call will set uninitialized stack variables to arbitrary values. Reasoning about such general relations is challenging. Thus, the state-machine translator encapsulates all non-deterministic parameters of each step in a *step object* and expresses the transition relation as a deterministic function NextState that computes state $i + 1$ from state i and step object i. For instance, if a method `M` has an uninitialized stack variable `x`, then each step object corresponding to a call to `M` has a field `newframe_x` that stores `x`'s initial value. This way, the proof can reason about the low-level program using an *annotated behavior*, which consists of a sequence of states and a sequence of step objects. The relationship between pairs of adjacent states in such an annotated behavior is a deterministic function, making reasoning easier.

4.1.1 Regions. To simplify proofs about pointers, we use *region-based* reasoning, in which memory locations (i.e., addresses) are assigned abstract *region ids*. Proving that two pointers are in different regions shows they are not aliased.

We carefully design our region reasoning to be automation friendly and compatible with any Armada strategy. To assign regions to memory locations rather than rely on developer-supplied annotations, we use Steensgaard's pointer analysis algorithm [45]. Our implementation of Steensgaard's algorithm begins by assigning distinct regions to all memory locations, then merges the regions of any two variables assigned to each other.

We perform region reasoning purely in Armada-generated proofs without requiring changes to the program or the state machine semantics. Hence, in the future, we can add more complex pointer analysis as needed.

We assign regions to memory locations through a proof construct called the *region map*. This is a map from addresses to a program-dependent `RegionId` datatype. For each pointer variable (i.e., variable with type **`ptr`**<T>), we generate a region invariant for that variable stating that any non-null address it holds is in the designated region and prove this invariant inductively.

To employ region-based reasoning, the developer simply adds **`use_regions`** to a recipe. Armada then performs the static analysis described earlier, generates the pointer invariants, and generates lemmas to inductively prove the invariants. If regions are overkill and the proof only requires an invariant that all addresses of in-scope variables are valid and distinct, the developer instead adds **`use_address_invariant`**.

4.1.2 Selective State-Machine Revelation. Armada generates a state machine tailored to the program to facilitate automated verification (Section 3.2.2). Sometimes, a specialized state machine

still contains too much information to verify in a timely fashion. However, lemmas generated by Armada usually reason about only a single step, and the rest of the state machine is irrelevant to the verification. For example, when proving that a particular step obeys the ownership predicate in TSO elimination (Section 4.2.3), one does not care about any other steps.

To aid in automated verification, Armada localizes proof contexts to particular step(s). It hides state machine definitions using the opaque Dafny attribute and generates revelation lemmas specialized to individual steps. Because Armada generates individual lemmas to reason about specific step(s), it can deterministically decide which revelation lemma(s) to invoke.

Compared with alternative approaches such as using triggers and reliance on SMT solvers' heuristics, selective revelation enforces better local reasoning and ensures that verification scales with program size.

4.1.3 Lemma Customization. Occasionally, verification fails for programs that correspond properly, because an automatically generated lemma has insufficient annotation to guide Dafny. For instance, the developer may weaken `y := x & 1` to `y := x % 2`, which is valid but requires bit-vector reasoning. Thus, Armada lets the developer arbitrarily supplement an automatically generated lemma with additional developer-supplied lemmas (or lemma invocations).

When the developer specifies that a certain customization should be inserted into a specific lemma, Armada chooses an appropriate location in the lemma to insert it. That is, it inserts it after any generic lemma invocations that might be useful to the customization and after any revelations of relevant opaque definitions (Section 4.1.2).

It is useful for strategies to use predictable lemma names so that lemma-customization annotations can refer to them in a way that will not change with minor changes to the programs. The developer can label statements involved in tricky reasoning; our strategies will generate lemmas about those statements with predictable names using those labels. For instance, if the developer writes the statement **`label`** `BitAnd: y := x & 1` in the method `Foo` and wants to weaken it to `y := x % 2`, then the developer can specify that extra lemma material belongs in the lemma `lemma_LiftNext_Update_Foo_BitAnd` that is generated by the weakening proof strategy. The developer may add an extra lemma invocation about bit-vectors and modulo on the variable `x`, which will be automatically inserted in `lemma_LiftNext_Update_Foo_BitAnd`. It will be inserted after any revelations of relevant opaque definitions and after any generic lemmas invoked by the weakening proof strategy.

Figure 11 shows, as an example, one of two lemma customizations used in a barrier implementation (see Section 6.1). In this case, the developer discovers that a specific automatically generated lemma does not verify. Thus, the developer writes a helper lemma demonstrating that if the barrier variable is non-zero in a thread's local view then its store buffer must contain one or the barrier variable must contain one. They then use the keyword **`extra`** to tell Armada to invoke certain extra proof text (an invocation of the helper lemma) within the automatically generated lemma. Armada inserts that text verbatim after various helpful lemma calls the strategy always includes, such as `lemma_GetThreadLocalViewAlwaysCommutesWithConvert`. That way, the lemma customization can use the postconditions of those helpful lemmas.

Armada's lemma customization contrasts with static checkers such as CIVL [21]. The constraints on program correspondence imposed by a static checker must be restrictive enough to ensure soundness. If they are more restrictive than necessary, a developer cannot appeal to more complex reasoning to convince the checker to accept the correspondence. A static checker could, in theory, make up for this by providing a way for developers to supply proofs of specific properties. Our technique is even more general because it lets developers augment *any* lemma generated by the proof-generation strategy, not just ones that the strategy's authors considered as potentially in need of augmentation.

```
// Lemma customization written by developer in recipe:
include_file "extra.dfy"
import_module L1RefinesL2Helpers
extra lemma_BehaviorSatisfiesLocalInvariant_worker_YYN_YieldedRevisited_6_Then_From_worker_6_T_To_worker_7
      "L1RefinesL2Helpers.lemma_BarrierNonzeroInLocalViewImpliesStoreBuffer1OrBarrier1(s19.s);"

// Proof code generated automatically:
lemma lemma_BehaviorSatisfiesLocalInvariant_worker_YYN_YieldedRevisited_6_Then_From_worker_6_T_To_worker_7(
  tid: Armada_ThreadHandle,
  s0: LPlusState, s1: LPlusState, step1: L.Armada_Step, s2: LPlusState,
  // [details elided]
  )
  requires RequiresClauses(LProcName_worker, s0, tid)
  requires YieldPredicate(s0, s1, tid)
  requires LPlus_ValidStep(s1, step1, tid) && s2 == LPlus_GetNextState(s1, step1, tid)
  // [lots of automatically generated content elided]
  requires YieldPredicate(s24, s25, tid)
  requires LPlus_ValidStep(s25, step25, tid) && s26 == LPlus_GetNextState(s25, step25, tid)
  requires L.Armada_ValidStep_WhileTrue_worker_6(s25.s, tid) &&
           s26.s == L.Armada_GetNextState_WhileTrue_worker_6(s25.s, tid)
  ensures LocalInv(s25, tid)
{
  lemma_StoreBufferAppendHasEffectOfAppendedEntryAlways_L();
  lemma_GetThreadLocalViewAlwaysCommutesWithConvert();
  // Lemma customization automatically inserted below, after general lemma invocations above:
  L1RefinesL2Helpers.lemma_BarrierNonzeroInLocalViewImpliesStoreBuffer1OrBarrier1(s19.s);
}
```

Fig. 11. Sample use of lemma customization.

4.1.4 Intermediate Proof Levels. To simplify the task of proving refinement between a low-level program L and a high-level program H, we use a multi-layered proof approach. That is, we generate intermediate state machines L^+, L^A, and H^A and prove that L refines L^+, L^+ refines L^A, L^A refines H^A, and H^A refines H. We then invoke a lemma showing transitivity of refinement to demonstrate that L refines H. We now describe each of these intermediate state machines in turn.

The first intermediate state machine, L^+, is L augmented with auxiliary state. For instance, one piece of state in L^+ but not L is the ID of the initial thread running `main`. It is often convenient for a proof to refer to this ID, and this obviates the developer explicitly storing it in a ghost variable. Another type of auxiliary state is the region map discussed in Section 4.1.1.

The second intermediate state machine, L^A, introduces atomic substeps to L^+. Recall that the trusted state-machine semantics models block atomicity by forbidding a thread T' from executing while a different thread T is inside an atomic block. It does this by modeling a state machine step as a sequence of substeps that cannot end in the middle of an atomic block (unless the program terminates there, e.g., due to undefined behavior). However, it still models T's execution of the n instructions of an atomic block as n separate state-machine substeps. In L^A, however, we model those n instructions as a single state-machine substep. This reduces the number of states and transitions that need to be reasoned about, simplifying proofs using this state machine.

A key advantage of using an atomic-substep state machine is that one can establish invariants by showing that each atomic block preserves them, rather than having to show that each individual instruction preserves them. Indeed, it allows one to use simple invariant predicates since those predicates do not have to account for states in the middle of atomic blocks. For instance, suppose that one introduces a ghost variable `g` with the intent of having some invariant relation between it and a real variable `x`. One can do this by adding an assignment to `g` immediately following each update to `x` and having this assignment be within the same atomic block. In this case, one may want to prove that the intended relation between `g` and `x` is an invariant without having to specify ungainly exceptions such as "except if the PC is at one of the points just before the assignment of `g`." Using an atomic-substep state machine allows this.

One caveat in the use of atomic substeps is that, unlike the set of instruction steps, the set of atomic substeps is not necessarily finite. For instance, the code block

```
explicit_yield {
  while (insufficient(x)) {
    x := x + 1;
  }
}
```

leads to infinitely many atomic substeps: one that increments x zero times, one that increments x once, one that increments x twice, and so on. Having an infinite set of atomic substeps is problematic since for many proofs we automatically generate one lemma per step, which would result in an infinite number of lemmas.

For this reason, our atomic-substep state-machine generator sometimes breaks an atomic substep into pieces and models each piece as a separate state-machine step. Breaking atomic substeps into pieces means that if we want to prove an invariant, we have to show that it holds not only at all yield points but also at all non-yielding break points. We use as break points every loop head and method entry point that may be visited twice without an intervening yield point; this ensures that the set of steps is finite. For example, the head of the `while (insufficient(x))` loop in the previous paragraph would be a break point since it can be visited twice without yielding.

The final intermediate state machine is H^A, the atomic-substep state machine version of the high-level state machine H. We construct it from H analogously to how we construct L^A from L^+.

For any state machine S, S and its atomic analog S^A are equivalent. That is, they describe exactly the same set of steps and, thus, the same set of behaviors. After all, every step of S completes any atomic block it starts; thus, it can be partitioned into sequences corresponding to atomic substeps from S^A. Also, every atomic substep in S^A is composed of a sequence of substeps from S; thus, any step in S^A is a sequence of atomic substeps that can be broken down into a sequence of substeps in S. Because S and S^A are equivalent, they refine each other trivially: for every behavior of one, there exists a behavior of the other (the same behavior) such that corresponding states refine each other (because they are identical and the refinement relation is reflexive).

For this reason, proving refinement in either direction is fairly mechanical. For instance, to prove that L^+ refines L^A, we generate a proof that every step in L^+ can be partitioned into atomic substeps in L^A. (See Section 7 for an example.) Proving that H^A refines H is even simpler: we generate a proof that every atomic substep in H^A can be divided into substeps in H.

The proof we generate that L refines L^+ is also fairly mechanical. Thus, the only part of the proof that is strategy specific is the refinement proof between L^A and H^A. This means that the strategies, which we will discuss next, benefit from having to reason only about atomic substeps and by having the low-level state augmented by the auxiliary state.

4.2 Specific Strategies

Our current implementation has eight strategies for eight different correspondence types. We now describe them.

4.2.1 Reduction. Because of the complexity of reasoning about all possible interleavings of statements in a concurrent program, a powerful simplification is to replace a sequence of statements with an atomic block. A classic technique for achieving this is reduction [32], which shows that one program refines another if the low-level program has a sequence of statements $R_1, R_2, \ldots, R_n, N, L_1, L_2, \ldots, L_m$ while the high-level program replaces those statements with a single atomic action having the same effect. Each R_i (L_i) must be a right (left) mover, that is, a statement that commutes to the right (left) with any step of another thread.

An overly simplistic approach is to consider two programs to exhibit the reduction correspondence if they are equivalent except for a sequence of statements in the low-level program that

Low level

```
lock(&mutex);
while (condition()) {
  do_something();
  unlock(&mutex);

  lock(&mutex);
}
unlock(&mutex);
```

High level

```
explicit_yield {
  lock(&mutex);
  while (condition()) {
    do_something();
    unlock(&mutex);
    yield;
    lock(&mutex);
  }
  unlock(&mutex);
}
```

Fig. 12. Reduction requiring the use of Cohen-Lamport generalization because the atomic block spans loop iterations.

corresponds to an atomic block with those statements as its body in the high-level program. This formulation would prevent us from considering cases in which the atomic blocks span loop iterations (e.g., Figure 12).

Instead, Armada's approach to sound extensibility gives us the confidence to use a generalization of reduction, due to Cohen and Lamport [10], that allows steps that do not necessarily correspond to consecutive statements in the program. It divides the states of the low-level program into a first phase (states following a right mover), a second phase (states preceding a left mover), and no phase (all other states). Programs may never pass directly from the second phase to the first phase, and for every sequence of steps starting and ending in no phase, there must be a step in the high-level program with the same aggregate effect.

Hence, our strategy considers two programs to exhibit the reduction correspondence if they are identical except that some yield points in the low-level program are not yield points in the high-level program. The strategy produces lemmas demonstrating that each Cohen-Lamport restriction is satisfied; for example, one lemma establishes that each step ending in the first phase commutes to the right with each other step. This requires generating many lemmas, one for each pair of steps of the low-level program in which the first step in that pair is a right mover.

Our use of encapsulated nondeterminism (Section 4.1) greatly aids the automatic generation of certain reduction lemmas. Specifically, we use it in each lemma showing that a mover commutes across another step, as follows. Suppose that we want to prove commutativity between a step σ_i by thread i that goes from s_1 to s_2 and a step σ_j from thread j that goes from s_2 to s_3. We must show that there exists an alternate-universe state s_2' such that a step from thread j can take us from s_1 to s_2' and a step from thread i can take us from s_2' to s_3. To demonstrate the existence of such an s_2', we must be able to automatically generate a proof that constructs such an s_2'. Fortunately, our representation of a step encapsulates all non-determinism. Thus, it is straightforward to describe such an s_2' as $\mathsf{NextState}(s_1, \sigma_j)$. This simplifies proof generation significantly, as we do not need code that can construct alternative-universe intermediate states for arbitrary commutations. All we must do is emit lemmas hypothesizing that $\mathsf{NextState}(\mathsf{NextState}(s_1, \sigma_j), \sigma_i) = s_3$, with one lemma for each pair of step types. The automated theorem prover can typically dispatch these lemmas automatically.

4.2.2 Rely-Guarantee Reasoning. Rely-guarantee reasoning [22, 30] is a powerful technique for reasoning about concurrent programs using Hoare logic. Our framework's generality lets us leverage this style of reasoning without relying on it as our *only* means of reasoning. Furthermore, our level-based approach lets developers use such reasoning piecemeal. That is, they do not have to use rely-guarantee reasoning to establish *all* invariants *all at once*. Rather, they can establish some invariants and *cement* them into their program, that is, add them as enabling conditions in one level so that higher levels can simply assume them.

Low level	High level
`t := best_len;`	`t := best_len;`
	`assume t >= ghost_best;`
`if (len < t) { ... }`	`if (len < t) { ... }`

Fig. 13. In assume introduction, the high-level program has an extra enabling condition. The correspondence might be proven by establishing that `best_len` ≥ `ghost_best` is an invariant and that `ghost_best` is monotonically non-increasing.

Two programs exhibit the *assume-introduction* correspondence if they are identical except that the high-level program has additional enabling constraints on one or more statements. The correspondence requires that each added enabling constraint *always holds* in the low-level program at its corresponding program position.

Figure 13 gives an example using a variant of our running traveling-salesman example. In this variant, the correctness condition requires that we find the optimal solution. Thus, it is not reasonable to simply replace the guard with `*` as we did in Figure 3. Instead, we want to justify the racy read of `best_len` by arguing that the result it reads is conservative, that is, that at worst it is an overestimate of the best length so far. We represent this best length with the ghost variable `ghost_best` and somehow establish that `best_len >= ghost_best` is an invariant. We also establish that between steps of a single thread, the variable `ghost_best` cannot increase; this is an example of a rely-guarantee predicate [22]. Together, these establish that `t >= ghost_best` always holds before the evaluation of the guard.

Benefits. The main benefit to using assume-introduction correspondence is that it adds enabling constraints to the program being reasoned about. More enabling constraints means fewer behaviors to be considered while locally reasoning about a step.

Another benefit is that it cements an invariant into the program. That is, it ensures that what is an invariant now will remain so even as further changes are made to the program as the developer abstracts it. For instance, after proving refinement of the example in Figure 13, the developer may produce a next-higher-level program by weakening the assignment `t := best_len` to `t := *`. This usefully eliminates the racy read to the variable `best_len` but has the downside of eliminating the relationship between `t` and the variable `best_len`. However, now that we have cemented the invariant that `t >= ghost_best`, we do not need this relationship any more. Now, instead of reasoning about a program that performs a racy read and then branches based on it, we reason only about a program that chooses an arbitrary value and then blocks forever if that value does not have the appropriate relationship to the rest of the state. Notice, however, that assume-introduction can be used only if this condition is already known to *always hold* in the low-level program at this position. Therefore, assume-introduction never introduces any additional blocking in the low-level program.

Proof generation. The proof generator for this strategy uses rely-guarantee logic, letting the developer supply standard Hoare-style annotations. The developer may annotate each method with preconditions and postconditions, may annotate each loop with loop invariants, and may supply invariants and rely-guarantee predicates.

One can add some such annotations directly to the Armada code in the high-level program using notation borrowed from Dafny. For instance, one can write a precondition on a method with a **`requires`** clause, a postcondition on a method with an **`ensures`** clause, a loop invariant on a loop with an **`invariant`** clause, or a rely-guarantee predicate with a **`yield_predicate`** clause.

However, some annotations cannot be expressed easily in the Armada language and are best expressed using a Dafny expression referencing the state-machine representation of the state. Thus,

one can add an annotation in the proof recipe by writing out that expression, referring to the current state as `s`, the next state (if applicable) as `s'`, and the current thread as `tid`. This lets one describe a condition that always holds (**`chl_invariant`**), a condition that always holds before a specific instruction executes (**`chl_local_invariant`**), a rely-guarantee predicate (**`chl_yield_pred`**), a method pre- or postcondition (**`chl_precondition`** or **`chl_postcondition`**), or a loop invariant (**`chl_loop_modifies`**). For instance, if one wants to write that a precondition for calling method `M` is that every other thread's store buffer is empty, one could write:

```
chl_precondition M OtherStoreBuffersEmpty
  "forall other ::
     other in s.s.threads && other != tid ==>
     |s.s.threads[other].storeBuffer| == 0"
```

We use the word "modifies" in **`chl_loop_modifies`** because it can express not only an invariant that holds at the beginning of each loop body but also a restriction on how the state can change in the course of zero or more loop iterations. One can do this by referencing `s'`, which refers to the state at the end of zero or more loop iterations.

Our strategy generates one lemma for each program path that starts at a method's entry and makes no backward jumps. This is always a finite path set; thus, it only has to generate finitely many lemmas. Each such lemma establishes properties of a state machine that resembles the low-level program's state machine but differs in the following ways. Only one thread ever executes and it starts at the beginning of a method. Calling another method simply causes the state to be havocked subject to its postconditions. Before evaluating the guard of a loop, the state changes arbitrarily subject to the loop invariants. Between program steps, the state changes arbitrarily subject to the rely-guarantee predicates and invariants.

The generated lemmas must establish that each step maintains invariants and rely-guarantee predicates, that method preconditions are satisfied before calls, that method postconditions are satisfied before method exits, and that loop invariants are reestablished before jumping back to loop heads. This requires several lemmas per path: one for each invariant, one to establish preconditions if the path ends in a method call, one to establish maintenance of the loop invariant if the path ends just before a jump back to a loop head, and so on. The strategy uses these lemmas to establish the conditions necessary to invoke a library lemma that proves properties of rely-guarantee logic.

The logic we use is complex; thus, the previous discussion should not be convincing that it is sound or that we have correctly implemented it. Fortunately, as discussed earlier, all Dafny code is verified. Thus, there is no need to trust the proof generator or our libraries.

4.2.3 TSO Elimination. We observe that even in programs using sophisticated lock-free mechanisms, most variables are accessed via a simple ownership discipline (e.g., "always by the same thread" or "only while holding a certain lock") that straightforwardly provides data race freedom (DRF) [2]. It is well understood that x86-TSO behaves indistinguishably from sequential consistency under DRF [5, 24]. Our level-based approach means that developers need not prove that they follow an ownership discipline for *all* variables to get the benefit of reasoning about sequential consistency. In particular, Armada allows a level at which the sophisticated variables use regular assignments and the simple variables use TSO-bypassing assignments. Indeed, developers need not even prove an ownership discipline for all such variables at once; they may find it simpler to reason about those variables one at a time or in batches. At each point, they can focus on proving an ownership discipline just for the specific variable(s) to which they are applying TSO elimination. As with any proof, if the developer makes a mistake (e.g., by not following the ownership discipline), Armada reports a proof failure.

```
var x:int32;
ghost var lockholder:Option<uint64>;
...
tso_elim x "s.s.ghosts.lockholder == Some(tid)"
```

Fig. 14. Variables in a program, followed by invocation, in a recipe, of the TSO-elimination strategy. The part in quotation marks indicates under what condition the thread `tid` owns (has exclusive access to) the variable `x` in state `s`: when the ghost variable `lockholder` refers to that thread.

A pair of programs exhibits the *TSO-elimination* correspondence if all assignments to a set of locations $\mathcal{L}$ in the low-level program are replaced by TSO-bypassing assignments. Furthermore, the developer supplies an *ownership predicate* (as in Figure 14) that specifies which thread (if any) owns each location in $\mathcal{L}$. It must be an invariant that no two threads own the same location at once and no thread can read or write a location in $\mathcal{L}$ unless it owns that location. Any step releasing ownership of a location must ensure the thread's store buffer is empty, for example, by being a fence.

4.2.4 Weakening. As discussed earlier, two programs exhibit the weakening correspondence if they match except for certain statements in which the high-level version admits a superset of behaviors of the low-level version. The strategy generates a lemma for each statement in the low-level program proving that, considered in isolation, it exhibits a subset of behaviors of the corresponding statement of the high-level program.

4.2.5 Non-deterministic Weakening. A special case of weakening is when the high-level version of the state transition is non-deterministic, with that non-determinism expressed as an existentially quantified variable. For example, in Figure 4, the guard on an **`if`** statement is replaced by the `*` expression indicating non-deterministic choice. For simplicity of presentation, that figure shows the recipe invoking the weakening strategy. However, in practice, it would use *non-deterministic weakening*.

Proving non-deterministic weakening requires demonstrating a witness for the existentially quantified variable. Our strategy uses various heuristics to identify this witness and generate the proof accordingly.

4.2.6 Combining. Two programs exhibit the *combining* correspondence if they are identical except that an atomic block in the low-level program is replaced by a single statement in the high-level program that has a superset of its behaviors. This is analogous to weakening in that it replaces what appears to be a single statement (an atomic block) with a statement with a superset of behaviors. However, it differs subtly because our model for an atomic block is not a single step but rather a sequence of steps that cannot be interrupted by other threads.

The key lemma generated by the combining proof generator establishes that all paths from the beginning of the atomic block to the end of the atomic block exhibit behaviors permitted by the high-level statement. This involves breaking the proof into pieces, one for each path prefix that starts at the beginning of the atomic block and does not pass beyond the end of it.

4.2.7 Variable Introduction. A pair of programs exhibits the *variable-introduction* correspondence if they differ only in that the high-level program has variables (and assignments to those variables) that do not appear in the low-level program. The high-level program may read the new variables only in the right-hand side of one of the new assignments.

Our strategy for variable introduction creates refinement proofs for program pairs exhibiting this correspondence. The main use of this is to introduce ghost variables that abstract the concrete

state of the program. Ghost variables are easier to reason about because they can be arbitrary types and because they use sequentially consistent semantics.

Another benefit of ghost variables is that they can obviate concrete variables. Once the developer introduces enough ghost variables and establishes invariants linking the ghost variables to the concrete state, they can weaken the program logic that depends on concrete variables to depend on ghost variables instead. Once program logic no longer depends on a concrete variable, the developer can *hide* it, using the strategy described next.

4.2.8 Variable Hiding. A pair of programs $\langle L, H \rangle$ exhibits the *variable-hiding* correspondence if $\langle H, L \rangle$ exhibits the variable-introduction correspondence. In other words, the high-level program H has fewer variables than the low-level program L, and L only uses those variables in assignments to them. Our variable-hiding strategy creates refinement proofs for program pairs exhibiting this correspondence. As alluded to in Section 4.2.7, this is useful to remove the concrete state — and, thus, program complexity —once ghost variables have taken their place in program logic.

5 IMPLEMENTATION

Our implementation consists of a state-machine translator to translate Armada programs to state-machine descriptions; a framework for proof generation and a set of tools fitting in that framework; and a library of lemmas useful for invocation by proofs of refinement. It is open source and available at https://github.com/microsoft/armada.

5.1 Code Details

Since Armada is similar to Dafny, we implement the state-machine translator using a modified version of Dafny's parser and type-inference engine. After the parser and resolver run, our code performs state-machine translation. In all, our state-machine translator is 13,531 new source lines of code (SLOC [47]) of C#. Each state machine includes common Armada definitions of datatypes and functions; these constitute 584 SLOC of Dafny.

Our proof framework is also written in C#. Its abstract syntax tree (AST) code is a modification of Dafny's AST code. We have an abstract proof generator that deals with general aspects of proof generation (Section 4.1), and we have one subclass of that generator for each strategy. Our proof framework is 5,370 SLOC of C#.

We also extend Dafny with a 1,805-SLOC backend that translates an Armada AST into C code compatible with CompCertTSO [46], a version of CompCert [4] that ensures the emitted code respects x86-TSO semantics.

Our general-purpose proof library is 6,850 SLOC of Dafny.

5.2 Version Differences

This article discusses version 0.2 of Armada, released in January 2021; our earlier paper [33] discusses version 0.1, released in April 2020. Notable ways in which the new version differs from the old version include the following.

- The language has additional features **`atomic_exchange`**, **`compare_and_swap`**, **`global_view`**, **`goto`**, **`if_undefined`**, **`$state`**, and **`universal_step_constraint`** (Section 3.1).
- Each store buffer entry includes the PC of the instruction that added the entry to simplify proofs (Section 3.2.1).
- Instructions that cause undefined behavior are modeled as two different state machine transitions (Section 3.2.3).
- Selective state-machine revelation makes proofs more stable (Section 4.1.2).

Table 1. Example Programs Used to Evaluate Armada

Name	Description	Impl. SLOC	Proof SLOC	# of proof layers
Barrier	Barrier described by Schirmer and Cohen [43] as incompatible with ownership-based proofs	53	263	2
Pointers	Program using multiple pointers	29	13	1
Counter	Shared counter used by Owicki and Gries [39] to illustrate use of auxiliary variables	50	130	9
MCSLock	Mellor-Crummey and Scott (MCS) lock [34]	42	688	6
Queue	Lock-free queue from liblfds library [31, 35]	70	663	8

- Lemma customizations are inserted after useful lemma invocations and opaque-definition revelations (Section 4.1.3).
- Intermediate proof levels include atomic levels L^A and H^A (Section 4.1.4).
- The rely-guarantee reasoning strategy allows four new types of recipe annotation: `chl_local_invariant`, `chl_precondition`, `chl_postcondition`, and `chl_loop_modifies` (Section 4.2.2).
- Variable hiding also permits hiding ghost variables (Section 4.2.8).
- A more compact heap model reduces the size of the common definitions in every state-machine specification, thereby reducing the size of the trusted computing base.
- Generated state machine specifications and proofs are compatible with the latest Dafny release, version 3.2.0.

Proofs generated by version 0.2 are larger than those generated by version 0.1, largely due to recent additions to improve proof stability. Two notable recent contributors to proof stability and size are selective state-machine revelation (Section 4.1.2) and atomic intermediate proof levels (Section 4.1.4).

6 EVALUATION

To show Armada's versatility, we evaluate it on the programs in Table 1. Our evaluations show that we can prove the correctness of programs not amenable to verification via ownership-based methodologies [43], programs with pointer aliasing, shared counter between threads, lock implementations from previous frameworks [17], and libraries of real-world high-performance data structures.

6.1 Barrier

The Barrier program includes a barrier implementation described by Schirmer and Cohen [43]: "each processor has a flag that it exclusively writes (with volatile writes without any flushing) and other processors read, and each processor waits for all processors to set their flags before continuing past the barrier." They give this as an example of what their ownership-based methodology for reasoning about TSO programs cannot support. Like other uses of Owens's publication idiom [37], this barrier is predicated on the allowance of races between writes and reads to the same location.

The key safety property is that each thread does its post-barrier write after all threads do their pre-barrier writes. We cannot use the TSO-elimination strategy since the program has data races; thus, we prove as follows. A first level uses variable introduction to add ghost variables representing initialization progress and which threads have performed their pre-barrier writes. A second level uses rely-guarantee to add an enabling condition on the post-barrier write that all pre-barrier writes are complete. This condition implies the safety property.

One author took ~3 days to write the proof levels, mostly to write invariants and rely-guarantee predicates involving x86-TSO reasoning. Due to the complexity of this reasoning, the original recipe had many mistakes; output from verification failures aided discovery and repair.

The implementation is 53 SLOC. The first proof level uses 10 additional SLOC for new variables and assignments, and 5 SLOC for the recipe; Armada generates 22,715 SLOC of proof. The next level uses 38 additional SLOC for enabling conditions, loop invariants, preconditions, and postconditions; 109 SLOC of Dafny for lemma customization; and 101 further SLOC for the recipe, mostly for invariants and rely-guarantee predicates. Armada generates 98,003 SLOC of proof.

We use this program for an elaborated example of Armada's use in Section 7.

6.2 Pointers

The Pointers program writes via distinct pointers of the same type. The correctness of our refinement depends on the static pointer analysis in our region-based reasoning (Section 4.1.1), proving that these different pointers do not alias. Specifically, we prove that the program assigning values via two pointers refines a program assigning those values in the opposite order. The automatic pointer analysis reveals that the pointers cannot alias and, thus, that the reversed assignments result in the same state. The program is 29 SLOC, the recipe is 13 SLOC, and Armada generates 6,997 SLOC of proof.

6.3 Counter

The Counter program is a variant of a program used by Owicki and Gries [39] to illustrate the use of auxiliary variables to prove correctness of concurrent programs. It creates a global counter and increments it in two concurrent threads. Those threads acquire a lock before updating the counter.

Our proof establishes that, regardless of the execution order, the final counter always equals two. Following Owicki and Gries, we introduce an auxiliary ghost variable representing each thread's contribution to the global counter value; we use the variable-introduction strategy for this key step. The complete proof uses most of our proof strategies in nine levels, including reduction, TSO elimination, and rely-guarantee reasoning. The implementation takes 50 SLOC, the recipes use 130 SLOC, and Armada generates 169,270 SLOC of proof.

6.4 MCSLock

The MCSLock program includes a lock implementation developed by Mellor-Crummey and Scott [34]. It uses compare-and-swap instructions and fences for thread synchronization. It excels at fairness and cache-awareness by having threads spin on their own locations. We use it to demonstrate that our methodology allows modeling locks hand-built out of hardware primitives, as done for CertiKOS [25].

Our proof establishes the safety property that statements between `acquire` and `release` can be reduced to an atomic block. For simplicity, we use a large array to simulate a dynamic linked list at the implementation level. This way, we can focus on proving the ownership of the lock. Our refinement proof uses six transformations, including the following two notable ones. The fifth transformation proves that both `acquire` and `release` properly maintain the ownership represented by ghost variables. For example, `acquire` secures ownership and `release` returns it. We prove this by introducing enabling conditions and annotating the program. The last transformation reduces statements between `acquire` and `release` into a single atomic block through reduction.

The implementation is 42 SLOC. Level 1 keeps 42 SLOC and uses 12 SLOC for its recipe. Level 2 adds 16 SLOC to the program and uses 4 SLOC for its recipe. Level 3 keeps 58 SLOC as level 2 and uses 8 SLOC for its recipe. Level 4 removes 3 SLOC from the program and uses 4 SLOC for

its recipe. Level 5 adds 19 SLOC to the program and uses 145 SLOC for its recipe. Level 6 adds 2 SLOC to the program and uses 21 SLOC for its recipe. Levels 5 and 6 collectively use a further 457 SLOC for proof customization. In comparison, the authors of CertiKOS verify an MCS lock via concurrent certified abstraction layers [25] using 3.2K LOC to prove the safety property.

6.5 Queue

The Queue program includes a lock-free queue from the liblfds library [31, 35], used at AT&T, Red Hat, and Xen. We use it to show that Armada can handle a practical, high-performance lock-free data structure.

Proof. Our goal is to prove that the enqueue and dequeue methods behave like abstract versions in which enqueue adds to the back of a sequence and dequeue removes the first entry of that sequence, as long as at most one thread of each type is active. Our proof introduces an abstract queue, uses an inductive invariant and weakening to show that logging using the implementation queue is equivalent to logging using the abstract queue, then hides the implementation. This leaves a simpler enqueue method that appends to a sequence and a dequeue method that removes and returns its first element.

It took ~6 person-days to write the proof levels. Most of this work involved identifying the inductive invariant to support weakening of the logging using implementation variables to logging using the abstract queue.

The implementation is 70 SLOC. We use eight proof transformations, the fourth of which does the key weakening described in the previous paragraph. The first three proof transformations introduce the abstract queue using recipes with a total of 12 SLOC. The fourth transformation uses a recipe with 67 SLOC, including proof customization, and an external file with 522 SLOC to define an inductive invariant and helpful lemmas. The final four levels hide the implementation variables using recipes with a total of 16 SLOC, leading to a final layer with 46 SLOC. From all of our recipes, Armada generates 197,968 SLOC of proof.

Performance. We measure performance in Docker on a machine with an Intel Xeon E5-2687W CPU running at 3.10 GHz with 8 cores and 32 GB of memory. We use GCC 6.3.0 with `-O2` and CompCertTSO 1.13.8255. We use liblfds version 7.1.1 [31]. We run (1,000 times) its built-in benchmark for evaluating queue performance, using queue size 512. The benchmark runs one thread that repeatedly enqueues and another thread that repeatedly dequeues from a shared queue, with 5 million operations of warmup followed by 50 million operations from which throughput is computed.

Our Armada port of liblfds's lock-free queue uses modulo operators, instead of bitmask operators, to avoid invoking bit-vector reasoning. To account for this, we also measure liblfds-modulo, a variant we write with the same modifications.

To account for the maturity difference between CompCertTSO and modern compilers, we also report results for the Armada code compiled with GCC. Such compilation is not sound, since GCC does not necessarily conform to x86-TSO. We include these results only to give an idea of how much performance loss is due to using CompCertTSO. To constrain GCC's optimizations and thereby make the comparison somewhat reasonable, we insert the same barriers liblfds uses before giving GCC our generated ClightTSO code.

Figure 15 shows our results. The Armada version compiled with CompCertTSO achieves 70% of the throughput of the liblfds version compiled with GCC. Most of this performance loss is due to the use of modulo operations rather than bitmasks and the use of a 2013-era compiler rather than a modern one. After all, when we remove these factors, we achieve virtually identical performance (99% of throughput). This is not surprising since the code is virtually identical.

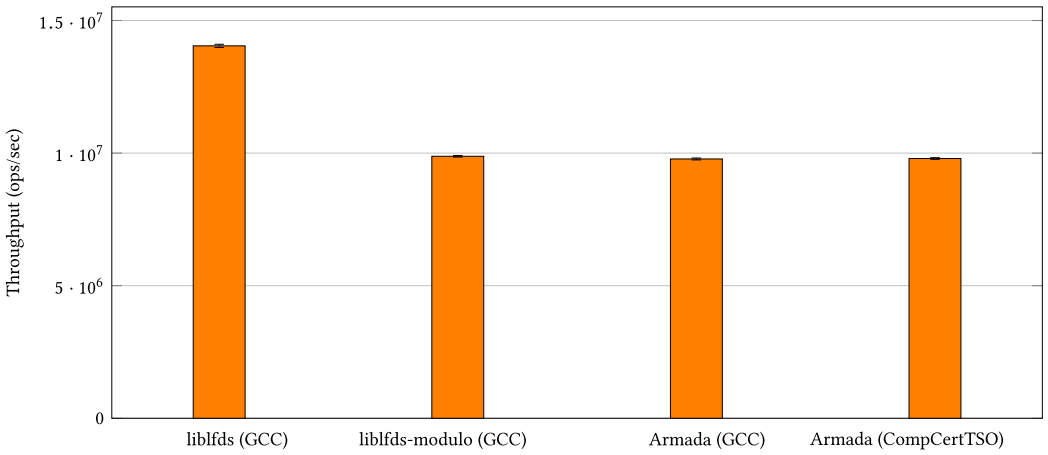

Fig. 15. These are performance results for liblfds's lock-free queue versus the corresponding code written in Armada. The Armada version and our variant liblfds-modulo use modulo rather than bitmask operations. Each data point is the mean of 1,000 trials; error bars indicate 95% confidence intervals.

```
structs SharedStructs
{
  ghost var log:seq<uint32> := [];

  refinement_constraint @"
     || (ls.stop_reason == hs.stop_reason &&
         ls.ghosts.log == hs.ghosts.log)
     || (ls.stop_reason.Armada_NotStopped? &&
         ls.ghosts.log <= hs.ghosts.log)
  "
}
```

Fig. 16. Refinement relation $\mathcal{R}$ for barrier program.

7 SAMPLE PROOF

To more concretely illustrate the use of Armada to prove program properties, we now present a detailed discussion of the barrier program in Section 6.1 and the proof that it refines a version of the program annotated with extra ghost variables. Those ghost variables keep track of which threads have had their barrier entries initialized, whether all threads have had their barrier entries initialized, and which threads have passed the barrier.

As described in Section 3.1.3, the first thing the developer writes is the refinement relation $\mathcal{R}$. This constrains what it means for an implementation to refine a specification. Here, we use a log prefix relationship: an implementation state refines a specification state if the log of the implementation's externally visible events so far is a prefix of that of the specification. In other words, the implementation's externally visible behavior must always be a prefix of a behavior permitted by the specification.

Figure 16 gives the Armada code describing this relation. The ghost variable `log` describes a sequence of all of the values printed by the program. Because we consider the program stopping to be an externally visible event, our refinement relation requires more than just a prefix relationship for `log`: if the implementation has stopped, then its `log` must match that of the specification and it must have stopped for the same reason.

The next step for the developer is to write the implementation, as shown in Figure 17. The implementation level is marked as `:concrete` to indicate that this level is intended to be compiled

```
level {:concrete} Impl using SharedStructs
{
  noaddr var barrier:uint32[10];

  method {:extern} print_uint32(i:uint32)
  {
    log := log + [i];
  }

  method worker(id:uint32)
  {
    noaddr var waiting_for_barrier:int8 := 1;
    noaddr var i:uint32;

    print_uint32(id);

    barrier[id] := 1;
    while waiting_for_barrier != 0
    {
      waiting_for_barrier := 0;
      i := 0;
      while i < 10
      {
        if barrier[i] == 0
        {
          waiting_for_barrier := 1;
        }
        i := i + 1;
      }
    }

    print_uint32(id + 100);
  }

  method main()
  {
    noaddr var i:uint32;
    noaddr var tids:uint64[10];

    i := 0;
    while i < 10
    {
      barrier[i] := 0;
      i := i + 1;
    }

    fence;

    i := 0;
    while i < 10
    {
      tids[i] := create_thread worker(i);
      i := i + 1;
    }

    i := 0;
    while i < 10
    {
      join tids[i];
      i := i + 1;
    }
  }
}
```

Fig. 17. Implementation of barrier program.

and run. The Armada tool ensures that the level uses only compilable features of the language. As an exception, the `print_uint32` method is an external method, meant to be provided by a trusted library. We model it as appending the printed value to the log, and implement it in C with:

```
void print_uint32(uint32 i)
{
  printf("%lu\n", i);
}
```

The implementation of our barrier program works as follows. The `main` thread initializes the `barrier` array with zeros, then uses a memory-fence instruction **`fence`** to flush these initial zeros to all cores. It then creates `worker` threads, each of which writes to the console, sets its array element to 1, loops waiting for all elements of the array to be non-zero, then writes more to the console. Since the workers rely solely on x86-TSO semantics, they perform only normal assignments and no fence instructions.

Figure 18 shows the first level of the proof, level 1. Compared with the implementation level 0, it introduces several new ghost variables useful in proving the safety property:

- The `barrier_initialized` array indicates, for each thread, whether its barrier entry has been initialized to 0.
- The `all_initialized` Boolean indicates whether all threads have had their barrier entries initialized to 0.
- The `threads_past_barrier` array indicates, for each thread, whether it has passed the barrier.

To prove this, the developer simply writes:

```
proof ImplRefinesL1
{
  refinement Impl L1
  var_intro barrier_initialized, all_initialized, threads_past_barrier
}
```

The Armada tool takes care of generating the proof.

The generated proof works as follows. As discussed in Section 4.1.4, it contains intermediate state machines L^+, L^A, and H^A and a proof that L refines L^+, L^+ refines L^A, L^A refines H^A, and H^A refines H.

To prove that L refines L^+, the proof makes use of a library `GenericArmadaPlus.i.dfy` that defines the predicate `RequirementsForSpecRefinesPlusSpec` and the lemma `lemma_SpecRefinesPlusSpec`. The former describes the conditions under which the latter can establish that one specification refines the other because they're related by "plus." Figure 19 gives more detail about these elements of the generic proof library. Using this library in this case is quite straightforward, since the requirements can be easily proven by automated verification with little annotation. Figure 20 shows the Dafny code generated to invoke the library and thereby prove refinement.

We prove that L^+ refines L^A similarly. The library `LiftToAtomic.i.dfy` defines `RequirementsForLiftingToAtomic` and `lemma_SpecRefinesAtomicSpec`. The former describes the conditions under which the latter can establish refinement.

The trickiest part of the proof is that L^A refines H^A. It is tricky because the high-level program contains instructions (the introduced assignments) that do not exist in the low-level program. The library here requires us to prove that the atomic substeps in L^A are *introducibly liftable* to atomic substeps in H^A. That is, whenever we take an atomic substep in L^A, we can always either execute a corresponding substep in H^A (lifting the substep from the low level to the high level) or execute a substep that exists only in H^A (executing an introduced assignment).

Our formal specification for the introducible liftability condition is in Figure 21. It states that if: (1) the invariant `inv` is satisfied in low-level state `ls`, (2) the lifting relation `relation` holds between `ls` and high-level state `hs`, and (3) one takes an atomic substep `lpath` in L^A from `ls` to `ls'` (defined as `lasf.path_next(ls, lpath, tid)`), then one of two conditions must hold. The first condition is that there exists a corresponding step `hpath` in H^A such that `relation(ls', hs')` holds, where `hs'` is the result of taking step `hpath` from `hs` (i.e., `hasf.path_next(hs, hpath, tid)`). The other

```
level L1 using SharedStructs
{
  noaddr var barrier:uint32[10];
  ghost var barrier_initialized:seq<bool> :=
    [false, false, false, false, false, false, false, false, false, false];
  ghost var all_initialized:bool := false;
  ghost var threads_past_barrier:seq<bool> := [];

  method {:extern} print_uint32(i:uint32)
  {
    log := log + [i];
  }

  method worker(id:uint32)
  {
    noaddr var waiting_for_barrier:int8 := 1;
    noaddr var i:uint32;

    print_uint32(id);

    threads_past_barrier := if 0 <= id as int < |threads_past_barrier| then
      threads_past_barrier[id as int := true] else threads_past_barrier;
    barrier[id] := 1;
    while waiting_for_barrier != 0
    {
      waiting_for_barrier := 0;
      i := 0;
      while i < 10
      {
        if barrier[i] == 0
        {
          waiting_for_barrier := 1;
        }
        i := i + 1;
      }
    }

    print_uint32(id + 100);
  }

  method main()
  {
    noaddr var i:uint32;
    noaddr var tids:uint64[10];

    i := 0;
    while i < 10
    {
      atomic {
        barrier[i] := 0;
        barrier_initialized :=
          if 0 <= i as int < |barrier_initialized| then
            barrier_initialized[i as int := true]
          else
            barrier_initialized;
      }
      i := i + 1;
    }
    fence;

  label post_fence:
    threads_past_barrier := [false, false, false, false, false, false, false, false, false, false];
    all_initialized := true;

    i := 0;
    while i < 10
    {
      tids[i] := create_thread worker(i);
      i := i + 1;
    }

    i := 0;
    while i < 10
    {
      join tids[i];
      i := i + 1;
    }
  }
}
```

Fig. 18. Level 1 of barrier proof, introducing ghost variables.

```
predicate InitsMatch<LState(!new), HState(!new), OneStep(!new), PC>(
  lasf: Armada_SpecFunctions<LState, OneStep, PC>,
  hasf: Armada_SpecFunctions<HState, OneStep, PC>,
  convert: HState->LState
  )
{
  forall ls :: lasf.init(ls) ==> exists hs :: hasf.init(hs) && ls == convert(hs)
}

predicate NextsMatch<LState(!new), HState(!new), OneStep(!new), PC>(
  lasf: Armada_SpecFunctions<LState, OneStep, PC>,
  hasf: Armada_SpecFunctions<HState, OneStep, PC>,
  convert: HState->LState
  )
{
  forall ls, hs, step, tid :: lasf.step_valid(ls, step, tid) && ls == convert(hs) ==>
    hasf.step_valid(hs, step, tid) && lasf.step_next(ls, step, tid) == convert(hasf.step_next(hs, step, tid))
}

predicate TausMatch<LState(!new), HState(!new), OneStep(!new), PC>(
  lasf: Armada_SpecFunctions<LState, OneStep, PC>,
  hasf: Armada_SpecFunctions<HState, OneStep, PC>,
  convert: HState->LState
  )
{
  forall step :: lasf.is_step_tau(step) <==> hasf.is_step_tau(step)
}

predicate ThreadPCsMatch<LState(!new), HState(!new), OneStep(!new), PC>(
  lasf: Armada_SpecFunctions<LState, OneStep, PC>,
  hasf: Armada_SpecFunctions<HState, OneStep, PC>,
  convert: HState->LState
  )
{
  forall ls, hs, tid :: ls == convert(hs) ==> lasf.get_thread_pc(ls, tid) == hasf.get_thread_pc(hs, tid)
}

// ...

predicate RequirementsForSpecRefinesPlusSpec<LState(!new), HState(!new), OneStep(!new), PC(!new)>(
  lasf: Armada_SpecFunctions<LState, OneStep, PC>,
  hasf: Armada_SpecFunctions<HState, OneStep, PC>,
  convert: HState->LState
  )
{
  && InitsMatch(lasf, hasf, convert)
  && NextsMatch(lasf, hasf, convert)
  && TausMatch(lasf, hasf, convert)
  && ThreadPCsMatch(lasf, hasf, convert)
  && NonyieldingPCsMatch(lasf, hasf, convert)
  && StateOKsMatch(lasf, hasf, convert)
}

lemma lemma_SpecRefinesPlusSpec<LState(!new), HState(!new), OneStep(!new), PC>(
  lasf: Armada_SpecFunctions<LState, OneStep, PC>,
  hasf: Armada_SpecFunctions<HState, OneStep, PC>,
  convert: HState->LState
  ) returns (
  refinement_relation: RefinementRelation<LState, HState>
  )
  requires RequirementsForSpecRefinesPlusSpec(lasf, hasf, convert)
  ensures  SpecRefinesSpec(Armada_SpecFunctionsToSpec(lasf),
                           Armada_SpecFunctionsToSpec(hasf), refinement_relation)
  ensures  refinement_relation == iset ls, hs | ls == convert(hs) :: RefinementPair(ls, hs)
{
  refinement_relation := iset ls, hs | ls == convert(hs) :: RefinementPair(ls, hs);
  var hspec := Armada_SpecFunctionsToSpec(hasf);
  forall lb | BehaviorSatisfiesSpec(lb, Armada_SpecFunctionsToSpec(lasf))
    ensures exists hb :: BehaviorRefinesBehavior(lb, hb, refinement_relation) && BehaviorSatisfiesSpec(hb, hspec)
  {
    var hb := lemma_LiftBehavior(lasf, hasf, convert, refinement_relation, lb);
    assert BehaviorRefinesBehavior(lb, hb, refinement_relation) && BehaviorSatisfiesSpec(hb, hspec);
  }
}
```

Fig. 19. Part of the generic proof library establishing conditions under which a level L can be proven to refine a level L^+.

condition is that there exists an "introduced" step `hpath` in H^A such that `relation(ls, hs')` holds and `hs'` is less than `hs` by some given progress measure `progress_measure`.

The invariant `inv`, the state relation `relation`, and the progress measure `progress_measure` are arbitrary and customizable. That is, the caller of the library may set these as desired so long as `inv` can be proven to be an invariant of the program and `relation` implies $\mathcal{R}$. In our case, the

```
function ConvertTotalState_LPlusL(lps: LPlusState): LState
{
  lps.s
}

lemma lemma_EstablishRequirementsForLSpecRefinesLPlusSpec()
  ensures RequirementsForSpecRefinesPlusSpec(L.Armada_GetSpecFunctions(),
                                             LPlus_GetSpecFunctions(), ConvertTotalState_LPlusL)
{
  var lasf := L.Armada_GetSpecFunctions();
  var hasf := LPlus_GetSpecFunctions();
  var convert := ConvertTotalState_LPlusL;
  forall ls | lasf.init(ls)
    ensures exists hs :: hasf.init(hs) && ls == convert(hs)
  {
    var hs := MakeLPlusInitialState(ls);
    assert hasf.init(hs) && ls == convert(hs);
  }
}

lemma lemma_LSpecRefinesLPlusSpec() returns (refinement_relation: RefinementRelation<LState, LPlusState>)
  ensures SpecRefinesSpec(Armada_SpecFunctionsToSpec(L.Armada_GetSpecFunctions()),
                          Armada_SpecFunctionsToSpec(LPlus_GetSpecFunctions()), refinement_relation)
  ensures refinement_relation == iset ls: LState, lps: LPlusState | ls == lps.s :: RefinementPair(ls, lps)
{
  lemma_EstablishRequirementsForLSpecRefinesLPlusSpec();
  refinement_relation := lemma_SpecRefinesPlusSpec(L.Armada_GetSpecFunctions(),
                                                   LPlus_GetSpecFunctions(), ConvertTotalState_LPlusL);
}
```

Fig. 20. Invocation of the library lemma to prove that the barrier implementation level L refines the corresponding L^+.

developer has not expressed any custom invariants; thus, the invariant `inv` that we generate is just the default "no thread has a null thread ID." The `relation` that we generate is `LiftingRelation`, as given in Figure 22. Here, most of the code is boilerplate that we put in every variable-introduction proof. The exception is the predicate `IsReturnSite_H`, which is customized to describe all of the return sites of this particular H^A program. For `progress_measure`, we always generate the code in Figure 23.

Proving the introducible liftability condition requires, for each step in L^A, a lemma ensuring that one of the two conditions holds. Figure 24 shows one such lemma for a particularly tricky point in L^A. Here, we consider the case that the level L^A program is about to execute the `i := 0` instruction immediately following the **`fence`**. There are three possibilities to consider. If the program at level H^A is also immediately following the **`fence`**, then we show that we can introduce an assignment to `threads_past_barrier` and that this decreases the progress measure. If the program at level H^A is just past that assignment, we instead show that we can introduce an assignment to `all_initialized` and that this decreases the progress measure. The only other case is that the program at level H^A has done both assignments, in which case we show that we can lift the assignment `i := 0` in L^A to the corresponding assignment in H^A. This lemma, and the lemmas it depends on, are cumbersome and tedious to write. Thus, it is fortunate that we generate it all automatically.

Once we have proven that L^A refines H^A, we prove that H^A refines H. The approach here is similar to showing that L^+ refines L^A: we must satisfy the requirements of a library dedicated to this type of proof.

Finally, we put it all together into a final lemma establishing that L (in this case, the barrier implementation) refines H (the level 1 program that introduces the three variables). This is done by invoking all of the lemmas we have built, plus one final lemma that establishes four-way transitivity of refinement. This final lemma, shown in Figure 25, demonstrates what we set out to prove: that L refines H using the given refinement relation $\mathcal{R}$.

8 RELATED WORK

Concurrent separation logic [36] is based on unique ownership of heap-allocated memory via locking. Recognizing the need to support flexible synchronization, many program logics inspired

```
predicate AtomicPathIntroduced<LState, LPath, LPC, HState, HPath, HPC>(
  lasf: AtomicSpecFunctions<LState, LPath, LPC>,
  hasf: AtomicSpecFunctions<HState, HPath, HPC>,
  relation: (LState, HState)->bool,
  progress_measure: (HState, LPath, Armada_ThreadHandle)->(int, int),
  ls: LState,
  lpath: LPath,
  tid: Armada_ThreadHandle,
  hs: HState,
  hpath: HPath
  )
{
  var lty := lasf.path_type(lpath);
  var hty := hasf.path_type(hpath);
  var hs' := hasf.path_next(hs, hpath, tid);
  && hasf.path_valid(hs, hpath, tid)
  && relation(ls, hs')
  && hasf.state_ok(hs')
  && ProgressMade(progress_measure(hs, lpath, tid), progress_measure(hs', lpath, tid))
  && match lty
      case AtomicPathType_Tau => hty.AtomicPathType_Tau?
      case AtomicPathType_YY => hty.AtomicPathType_YY? || hty.AtomicPathType_Tau?
      case AtomicPathType_YS => hty.AtomicPathType_YY? || hty.AtomicPathType_Tau?
      case AtomicPathType_YR => hty.AtomicPathType_YY? || hty.AtomicPathType_Tau?
      case AtomicPathType_RY => hty.AtomicPathType_RR?
      case AtomicPathType_RS => hty.AtomicPathType_RR?
      case AtomicPathType_RR => hty.AtomicPathType_RR?
}

predicate LiftAtomicPathSuccessful<LState, LPath, LPC, HState, HPath, HPC>(
  lasf: AtomicSpecFunctions<LState, LPath, LPC>,
  hasf: AtomicSpecFunctions<HState, HPath, HPC>,
  inv: LState->bool,
  relation: (LState, HState)->bool,
  ls: LState,
  lpath: LPath,
  tid: Armada_ThreadHandle,
  hs: HState,
  hpath: HPath
  )
{
  var ls' := lasf.path_next(ls, lpath, tid);
  var hs' := hasf.path_next(hs, hpath, tid);
  && inv(ls')
  && hasf.path_valid(hs, hpath, tid)
  && relation(ls', hs')
  && hasf.path_type(hpath) == lasf.path_type(lpath)
  && lasf.state_ok(ls') == hasf.state_ok(hs')
}

// Introducible liftability condition:

forall ls, lpath, hs, tid ::
  && inv(ls)
  && relation(ls, hs)
  && lasf.path_valid(ls, lpath, tid)
  ==> exists hpath :: LiftAtomicPathSuccessful(lasf, hasf, inv, relation, ls, lpath, tid, hs, hpath)
                    || AtomicPathIntroduced(lasf, hasf, relation, progress_measure, ls, lpath, tid, hs, hpath)
```

Fig. 21. Introducible liftability condition.

by concurrent separation logic have been developed to increase expressiveness [11, 13, 14, 23, 28]. We are taking an alternative approach of refinement over small-step operational semantics that provides considerable flexibility at the cost of low-level modeling whose overhead we hope to overcome via proof automation.

CCAL and concurrent CertiKOS [18, 19] propose *certified concurrent abstraction layers*. CSPEC [6] also uses layering to verify concurrent programs. Layering means that a system implementation is divided into layers, each built on top of the other, with each layer verified to conform to an API and specification assuming that the layer below conforms to its API and specification. Composition rules in CCAL ensure end-to-end termination-sensitive contextual refinement properties when the implementation layers are composed together. Armada does not (yet) support layers: all components of a program's implementation must be included in level 0. Thus, Armada currently does not allow independent verification of one module whose specification is then used by another module. Also, Armada proves properties about programs only while CCAL supports general

```
predicate IsReturnSite_H(pc: H.Armada_PC)
{
  pc.Armada_PC_worker_1? || pc.Armada_PC_worker_End?
}

predicate VarIntroThreadInvariant(hs: HState, tid: Armada_ThreadHandle)
  requires tid in hs.threads
{
  var t := hs.threads[tid];
  var pc := t.pc;
  && StackMatchesMethod_H(t.top, PCToMethod_H(pc))
  && (forall eframe ::
      eframe in t.stack ==>
        IsReturnSite_H(eframe.return_pc))
  && (!hs.stop_reason.Armada_NotStopped? || !H.Armada_IsNonyieldingPC(pc) || HAtomic_IsRecurrentPC(pc))
}

predicate LiftingRelation(ls: LPlusState, hs: HState)
{
  && ls.s == ConvertTotalState_HL(hs)
  && forall tid ::
      tid in hs.threads ==>
        VarIntroThreadInvariant(hs, tid)
}
```

Fig. 22. Lifting relation used to establish that the implementation refines level 1 where variables are introduced.

```
function ProgressMeasure(hs: HState, lpath: LAtomic_Path, tid: Armada_ThreadHandle): (v: (int, int))
  ensures v.0 >= 0
  ensures v.1 >= 0
{
  if tid in hs.threads then
    var t := hs.threads[tid];
    (MaxPCInstructionCount_H() - PCToInstructionCount_H(t.pc), 0)
  else
    (0, 0)
}
```

Fig. 23. Definition of progress measure.

composition, such as the combination of a verified operating system, thread library, and program. On the other hand, CCAL uses a strong memory model disallowing all data races, while Armada uses the x86-TSO memory model and, thus, can verify programs with benign races and lock-free data structures. As we shall discuss in Section 9, Armada's level-based approach can be considered as a complement to abstraction layers.

Recent work [7] uses the Iris framework [27] to reason about a concurrent file system. Like CertiKOS and Cspec, this approach involves a significant amount of manual effort, making developers write their code in a particular style that may limit both performance optimization opportunities and the ability to port existing code.

QED [15] is the first verifier for functional correctness of concurrent programs to incorporate reduction for program transformation and to observe that weakening atomic actions can eliminate conflicts and enable further reduction arguments. CIVL [21] extends and incorporates these ideas into a refinement-oriented program verifier based on the framework of layered concurrent programs [26]. (Layers in CIVL correspond to levels in Armada, not layers in CertiKOS and Cspec.) Armada improves upon CIVL by providing a flexible framework for soundly introducing new mechanically verified program transformation rules; CIVL's rules are proven correct only on paper. CSim^2 [42] is a verification framework that brings Hoare Logic reasoning and program refinement together. Like the assume-introduction strategy (Section 4.2.2) in Armada, CSim^2 uses rely-guarantee reasoning to achieve modular verification of concurrent programs. However, Armada supports many other reasoning strategies, such as reduction and region-based reasoning. Armada is also extensible to new techniques that may be invented in the future.

```
lemma lemma_IntroduceOrLiftAtomicPath_From_main_6_To_main_7(
  lasf: AtomicSpecFunctions<LPlusState, LAtomic_Path, L.Armada_PC>,
  hasf: AtomicSpecFunctions<HState, HAtomic_Path, H.Armada_PC>,
  ls: LPlusState, lpath: LAtomic_Path, tid: Armada_ThreadHandle, hs: HState
  )
  returns (hpath: HAtomic_Path)
  requires lasf == LAtomic_GetSpecFunctions()
  requires hasf == HAtomic_GetSpecFunctions()
  requires InductiveInv(ls)
  requires LiftingRelation(ls, hs)
  requires LAtomic_ValidPath(ls, lpath, tid)
  requires lpath.LAtomic_Path_From_main_6_To_main_7?
  ensures LiftAtomicPathSuccessful(lasf, hasf, InductiveInv, LiftingRelation, ls, lpath, tid, hs, hpath)
          || AtomicPathIntroduced(lasf, hasf, LiftingRelation, ProgressMeasure, ls, lpath, tid, hs, hpath)
{
  lemma_LAtomic_PathHasPCEffect_From_main_6_To_main_7(ls, lpath, tid);
  assert tid in ls.s.threads;
  assert tid in hs.threads;
  var lpc := ls.s.threads[tid].pc;
  var hpc := hs.threads[tid].pc;
  assert lpc.Armada_PC_main_6?;
  assert VarIntroThreadInvariant(hs, tid);
  assert !L.Armada_IsNonyieldingPC(lpc);
  if hpc.Armada_PC_main_post_fence? {
    hpath := lemma_IntroduceAtomicPath_From_main_post_fence_To_main_8(lasf, hasf, ls, lpath, tid, hs);
    assert AtomicPathIntroduced(lasf, hasf, LiftingRelation, ProgressMeasure, ls, lpath, tid, hs, hpath);
    return;
  }
  if hpc.Armada_PC_main_8? {
    hpath := lemma_IntroduceAtomicPath_From_main_8_To_main_9(lasf, hasf, ls, lpath, tid, hs);
    assert AtomicPathIntroduced(lasf, hasf, LiftingRelation, ProgressMeasure, ls, lpath, tid, hs, hpath);
    return;
  }
  if hpc.Armada_PC_main_9? {
    hpath := lemma_LiftAtomicPath_From_main_6_To_main_7(lasf, hasf, ls, lpath, tid, hs);
    assert LiftAtomicPathSuccessful(lasf, hasf, InductiveInv, LiftingRelation, ls, lpath, tid, hs, hpath);
    return;
  }
  assert false;
}
```

Fig. 24. Proof establishing that a barrier implementation step can be introducibly lifted to level when the low-level program has just completed the **fence** instruction.

```
lemma lemma_ProveRefinement()
  ensures SpecRefinesSpec(L.Armada_Spec(), H.Armada_Spec(), GetLHRefinementRelation())
{
  var rr1 := lemma_LSpecRefinesLPlusSpec();
  var rr2 := lemma_LiftToAtomic();
  var rr3 := lemma_LiftLAtomicToHAtomic();
  var rr4 := lemma_LiftFromAtomic();
  lemma_SpecRefinesSpecQuadrupleConvolution(
    L.Armada_Spec(), Armada_SpecFunctionsToSpec(LPlus_GetSpecFunctions()),
    AtomicSpec(LAtomic_GetSpecFunctions()), AtomicSpec(HAtomic_GetSpecFunctions()),
    H.Armada_Spec(), rr1, rr2, rr3, rr4, GetLHRefinementRelation());
}
```

Fig. 25. Final lemma establishing that L (the barrier implementation) refines H (the level 1 program that introduces the three ghost variables).

Unlike Armada, VCC [9] is designed to verify existing concurrent C code *in situ*. Rather than using a simple, low-level state machine model based on x86-TSO memory semantics, as in Armada, VCC assumes a sequentially consistent memory model and bakes in a notion of ownership and object invariants for reasoning about aliasing. We soundly reconstruct some of these techniques in our reduction strategy (Section 4.2.1). Later VCC-related work [43] proposes moving to a TSO memory model but maintaining ownership information in the ghost state. This brings them closer to Armada's memory model, but because they rely on ownership as the primary concurrency reasoning tool, they cannot handle correct programs with benign races, for example, instances in which correctness relies on the fact that an address is written to only by a single writer, and the values written are monotonic. Our Barrier case study (Section 6.1) is one such example. It is unclear whether this proposed memory model was incorporated into the VCC tool.

9 DISCUSSION AND FUTURE WORK

In this section, we discuss the limitations of the current design and prototype of Armada and suggest items for future work.

Armada currently supports the x86-TSO memory model [38]. Thus, it is not directly applicable to other architectures, such as ARM and Power. Apart from the prevalence of the x86 architecture [20], we believe that x86-TSO is a good first step as it illustrates how to account for weak memory models while still being simple enough to keep the proof complexity manageable. ARM and Power have even weaker memory models than x86-TSO, which will likely result in more complex invariants as the developer tries to establish refinement to a stronger semantics, like our TSO-bypassing assignments. While we expect the invariants to be more complicated, the principle behind such proofs is still the same: the developer must show some type of ownership of a variable in order to show refinement to the stronger semantics.

Armada currently supports verification of a whole program only in contrast to CCAL's contextual-refinement-based layer system. We have plans to improve verification modularity by supporting modular verification against layer APIs, similar to those in CCAL. In CCAL, $L \vdash_{\mathcal{R}} M : H$ denotes that module M running on top of underlay L faithfully implements the specification in overlay H. CCAL further supports composition of layers with composition rules such as the vertical composition rule.

$$\textsc{Vertical Composition}\quad \frac{L_1 \vdash_{\mathcal{R}_1} M_1 : L_2 \qquad L_2 \vdash_{\mathcal{R}_2} M_2 : L_3}{L_1 \vdash_{\mathcal{R}_1 \circ \mathcal{R}_2} M_1 \oplus M_2 : L_3}$$

CCAL advocates for dividing programs into tiny modules such that the verification of individual modules is straightforward, while complex behaviors of a whole system emerge as all of the layers compose together. For example, CertiKOS divides an allocation table implementation into layers, one implementing a getter/setter interface, one implementing page allocation logic, and one implementing a locking mechanism. To do this in Armada, one would include the entire implementation in level 0, then abstract parts of the program one at a time in the same order as CCAL.

Furthermore, it is worth pointing out that Armada's level-based approach can be seen as a special case of CCAL's layer calculus. If a specification is expressed in the form of a program, then Armada's refinement between lower-level L and higher-level H with respect to refinement relation $\mathcal{R}$ can be expressed in the layer calculus as $L \vdash_{\mathcal{R}} \emptyset : H$. That is, without introducing any additional implementation in the higher layer, the specification can nevertheless be transformed between the underlay and overlay interfaces. Indeed, the authors of concurrent CertiKOS sometimes use such $\emptyset$-implementation layers when a complex layer implementation cannot be further divided into smaller pieces [19, 25]. This phenomenon manifests primarily in verification of locking mechanisms, in which fine-grained concurrency prevents division into smaller modules. Proofs of refinement for these modules are tedious and complicated. With support of modular verification, Armada may help here with its rich set of concurrency-reasoning techniques and its level-based refinement framework.

Armada uses Dafny to verify all proof material that we generate. As such, the trusted computing base (TCB) of Armada includes not only the compiler and the code for extracting state machines from the implementation and specification but also the Dafny toolchain. This toolchain includes Dafny, Boogie [3], Z3 [12], and our script for invoking Dafny.

Armada uses the CompCertTSO compiler, whose semantics is similar, but not identical, to Armada's. In particular, CompCertTSO represents memory as a collection of blocks, while Armada adopts a hierarchical forest representation. Additionally, in CompCertTSO, the program is modeled as a composition of a number of state machines—one for each thread—alongside a TSO state

machine that models global memory. Armada, on the other hand, models the program as a single-state machine that includes all threads and the global memory. We currently assume that the CompCertTSO model refines our own. It is future work to formally prove this by demonstrating an injective mapping between the memory locations and state transitions of the two models.

The current prototype of Armada uses Dafny's finite sequences to reason about behaviors. As such, it can prove safety but not liveness properties. This is not a fundamental limitation, however. Instead of sequences, one can use Dafny's **imap**s (infinite maps), to emulate an infinite sequence by mapping an integer i to the i^{th} state in a behavior. Even with such a change in place, though, we still expect liveness properties to be challenging to establish. Such properties have traditionally been difficult to prove, even in non-concurrent systems. In the Armada context, the developer would have to further contend with the complexity incurred by concurrency as well as having to show that **wait_until** statements eventually return.

Armada currently supports state transitions involving only the current state, not future states. Hence, Armada can encode *history variables* but not *prophecy variables* [1]. Note that Armada proofs manifest the entire behavior as a finite sequence of states, thus allowing the proof to reason about future states, if needed. To fully support prophecy variables, however, Armada must be expanded to allow invariants to be expressed over an entire behavior, rather than a single state, as the current prototype does.

Since we consider properties of single behaviors only, we cannot verify hyperproperties [8]. However, we can verify safety properties that *imply* hyperproperties, such as the unwinding conditions that Nickel uses to prove noninterference [41, 44].

Finally, the current prototype of Armada requires developers to manually define all layers, which requires an unfortunate amount of copy-paste overhead. We believe that much of this overhead can be avoided by reducing the number of layers and by making each layer more concise. To reduce the number of layers, one can introduce composite transformations, such as those used in CIVL [21]. In CIVL, each layer transition defines five mini-transitions in a certain order. First, the tool performs reduction, followed by variable introduction. Then, it proves invariants and does variable hiding. Finally, it establishes new atomic action specifications. Such composite layer definitions could reduce the number of layers that the developer has to write. To further reduce the developer's effort, layers can be defined more concisely by expressing them as a delta—for example, a patch—from the previous layer. This would allow layers to be introduced with very little developer effort while also enabling changes at an early layer to automatically propagate to all subsequent layers.

10 CONCLUSION

Via a common, low-level semantic framework, Armada supports a panoply of powerful strategies for automated reasoning about memory and concurrency, even while giving developers the flexibility needed for performant code. Armada's strategies can be soundly extended as new reasoning principles are developed. Our evaluation on five case studies demonstrates that Armada is a practical tool that can handle a diverse set of complex concurrency primitives as well as real-world, high-performance data structures.

ACKNOWLEDGMENTS

The authors are grateful to Ronghui Gu, the shepherd of our PLDI'20 paper of which this is an extended version, and to the anonymous PLDI'20 reviewers and TOPLAS referees for their valuable feedback that greatly improved the paper. We also thank Tej Chajed, Chris Hawblitzel, and Nikhil Swamy for reading early drafts of the paper and providing useful suggestions, and Rustan Leino for early discussions and for helpful Dafny advice and support.

REFERENCES

[1] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (May 1991), 253–284.

[2] Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—a new definition. In *Proceedings of International Symposium on Computer Architecture (ISCA'90)*. 2–14.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of Formal Methods for Components and Objects (FMCO'05)*. 364–387.

[4] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal verification of a C compiler front-end. In *Proceedings of International Symposium on Formal Methods (FM'06)*. 460–475.

[5] Gérard Boudol and Gustavo Petri. 2009. Relaxed memory models: An operational approach. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'09)*. 392–403.

[6] Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI'18)*. 306–322.

[7] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP'19)*. 243–258.

[8] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[9] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*. 23–42.

[10] Ernie Cohen and Leslie Lamport. 1998. Reduction in TLA. In *Concurrency Theory (CONCUR'98)*. 317–331.

[11] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'14)*. 207–231.

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. 337–340.

[13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'13)*. 287–300.

[14] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *Proceedings of European Symposium on Programming (ESOP)*. 363–377.

[15] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'09)*. 2–15.

[16] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2004. Exploiting purity for atomicity. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. 221–231.

[17] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'15)*. 595–608.

[18] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 653–669.

[19] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. 646–661.

[20] Tom's Hardware. 2021. Market share of the x86 architecture. Retrieved March 15, 2022 from https://www.tomshardware.com/news/amds-cpu-market-share-and-revenue-jump-as-apples-m1-arm-chips-rise.

[21] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of Computer Aided Verification (CAV)*. 449–465.

[22] C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems* 5, 4 (Oct. 1983), 596–619.

[23] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28, e20 (2018).

[24] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'17)*. 175–189.

[25] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and liveness of MCS lock—layer by layer. In *Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'17)*. 273–297.

[26] Bernhard Kragl and Shaz Qadeer. 2018. Layered concurrent programs. In *Proceedings of International Conference on Computer Aided Verification (CAV'18)*. 79–102.
[27] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *Proceedings of European Symposium on Programming (ESOP'17)*. 696–723.
[28] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: Compositional abstractions for concurrent data structures. *Proceedings of the ACM Symposium on Programming Languages* 2 *(POPL'18)*. 37:1–37:31.
[29] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Proceedings of Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. 348–370.
[30] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'12)*. 455–468.
[31] LibLFDS. 2019. LFDS 7.11 queue implementation. Retrieved March 15, 2022 from https://github.com/liblfds/liblfds7.1.1/tree/master/liblfds7.1.1/liblfds71\1/src/lfds711_queue_bounded_singleproducer_singleconsumer. (Nov. 2019).
[32] Richard J. Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Communications of the ACM* 18, 12 (Dec. 1975), 717–721.
[33] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. 197–210.
[34] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (Feb. 1991), 21–65.
[35] Maged M. Michael and Michael L. Scott. 2006. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC'06)*. 267–275.
[36] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1–3 (2007), 271–307.
[37] Scott Owens. 2010. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proceedings of European Conference on Object-Oriented Programming*. 478–503.
[38] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs'09)*. 391–407.
[39] Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM* 19, 5 (May 1976), 279–285.
[40] Shaz Qadeer. 2019. Private communication. (2019).
[41] John Rushby. 1992. Noninterference, Transitivity, and Channel-control Security Policies. Technical Report CSL-92-02, SRI International. (1992).
[42] David Sanan, Yongwang Zhao, Shang-Wei Lin, and Liu Yang. 2021. CSim2: Compositional top-down verification of concurrent systems using rely-guarantee. *ACM Transactions on Programming Languages and Systems* 43, 1, Article 2 (Feb. 2021), 46 pages.
[43] Norbert Schirmer and Ernie Cohen. 2010. From total store order to sequential consistency: A practical reduction theorem. In *Proceedings of Interactive Theorem Proving (ITP'10)*. 403–418.
[44] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 287–305.
[45] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'96)*. 32–41.
[46] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'11)*. 43–54.
[47] David A. Wheeler. 2004. SLOCCount. Software distribution. (2004). Retrieved March 15, 2022 from http://www.dwheeler.com/sloccount/.

Received April 2021; revised October 2021; accepted November 2021