The Civl Verifier

Bernhard Kragl
Amazon Web Services and *IST Austria*

Shaz Qadeer
Facebook

***Abstract*—Civl is a static verifier for concurrent programs designed around the conceptual framework of layered refinement, which views the task of verifying a program as a sequence of program simplification steps each justified by its own invariant. Civl verifies a layered concurrent program that compactly expresses all the programs in this sequence and the supporting invariants. This paper presents the design and implementation of the Civl verifier.**

I. Introduction

Correctness of critical specifications of concurrent systems rests upon invariants about the global system state. The classical approach to static verification is to represent the entire organizational structure—processes, threads, procedures, looping, branching, sequencing—of a concurrent system as a flat transition relation that encodes its operational semantics. Further reasoning is performed on this transition relation. This approach leads to massively complex invariants that are hard to specify for the programmer and difficult to verify via automated tools.

$$
\begin{array}{lll}
 & a\colon x := n & \\
b\colon & \text{acquire}(l) & \text{acquire}(l) \\
c\colon & t_1 := x & t_2 := x \\
d\colon & x := t_1 + 1 & x := t_2 + 1 \\
e\colon & \text{release}(l) & \text{release}(l) \\
 & f\colon \text{assert } x = n + 2 &
\end{array}
$$

Fig. 1. Parallel increment (version 0).

We motivate our work using the program in Figure 1. This program starts with a single thread that initializes a global variable x to a constant n, creates two threads that run in parallel each incrementing x by 1 while holding the lock l, waits for the two threads to finish, and then asserts that $x = n + 2$. The goal of verification is to prove this assertion for all values of n and all executions of the program.

The classical approach to verification of concurrent programs models the verification problem in Figure 1 as a transition system shown in Figure 2, comprising an initial predicate *Init*, a transition predicate *Next*, and a safety predicate *Safe*. To prove that all reachable states of the transition system satisfy the predicate *Safe*, an inductive invariant *Inv* must be invented such that $\mathit{Init} \Rightarrow \mathit{Inv}$, $\mathit{Inv} \wedge \mathit{Next} \Rightarrow \mathit{Inv}'$, and $\mathit{Inv} \Rightarrow \mathit{Safe}$.

This research was performed while Bernhard Kragl was at IST Austria, supported in part by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award).

Init: $pc = pc_1 = pc_2 = a$

Next:

$$
\begin{array}{l}
\quad pc = a \wedge pc' = pc_1 = pc_2 = b \wedge x' = n \wedge eq(l, t_1, t_2) \\
\vee\ pc_1 = b \wedge pc_1' = c \wedge l = \bigcirc \wedge l' = ① \wedge eq(pc, pc_2, x, t_1, t_2) \\
\vee\ pc_1 = c \wedge pc_1' = d \wedge t_1' = x \wedge eq(pc, pc_2, l, x, t_2) \\
\vee\ pc_1 = d \wedge pc_1' = e \wedge x' = t_1 + 1 \wedge eq(pc, pc_2, l, t_1, t_2) \\
\vee\ pc_1 = e \wedge pc_1' = f \wedge l' = \bigcirc \wedge eq(pc, pc_2, x, t_1, t_2) \\
\vee\ pc_2 = b \wedge pc_2' = c \wedge l = \bigcirc \wedge l' = ② \wedge eq(pc, pc_1, x, t_1, t_2) \\
\vee\ pc_2 = c \wedge pc_2' = d \wedge t_2' = x \wedge eq(pc, pc_1, l, x, t_1) \\
\vee\ pc_2 = d \wedge pc_2' = e \wedge x' = t_2 + 1 \wedge eq(pc, pc_1, l, t_1, t_2) \\
\vee\ pc_2 = e \wedge pc_2' = f \wedge l' = \bigcirc \wedge eq(pc, pc_1, x, t_1, t_2) \\
\vee\ pc_1 = pc_2 = f \wedge pc' = f \wedge eq(pc_1, pc_2, l, x, t_1, t_2)
\end{array}
$$

Safe: $(pc = f \Rightarrow x = n + 2) \wedge$
$(pc_1 \in \{c, d, e\} \Rightarrow l = ①) \wedge (pc_2 \in \{c, d, e\} \Rightarrow l = ②)$

Fig. 2. Transition relation of the program in Figure 1. The lock l can be either available (value $\bigcirc$), or held by the first or second thread (values ① and ②). The predicate *eq* denotes unmodified variables, e.g., $eq(l)$ means $l' = l$.

This approach is clearly problematic for several reasons. First, the encoding as a transition system flattens and eliminates the syntactic structure of the program. Forcing the programmer to think about the inductive invariant at the level of this encoding significantly reduces productivity. Second, the inductive invariant is likely to have as much case analysis as the encoded transition relation, making it even more tedious and unproductive for the programmer to specify it. For example, the inductive invariant for our example program is larger than its transition relation. This trivial parallel increment program is just the tip of the iceberg; the task of specification and verification explodes in complexity if we turn our attention to realistic implementations of large concurrent systems.

There are two broad approaches to the problem of inductive invariants for concurrent systems. One approach is automatic generation of inductive invariants [1], [2], [3] eliminating the need to specify them manually. Another approach is to specify them via annotations on the structured program itself [4], [5] reducing the cognitive burden on the programmer. Civl falls into this latter class of techniques; its contribution is to allow more proofs to be expressed on the structured program.

Civl proposes an alternative proof strategy which encourages the programmer to think in terms of a sequence of program versions that increasingly simplify the original program. Denoting the program in Figure 1 as version 0, we show three progressively simpler versions in Figure 3.

The simplification from version 0 to version 1 is based on mover types [6], [7]. Acquiring of lock l is a right mover, release of lock l is a left mover, and accesses to the shared variable x protected by the lock l are left and right movers.

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23

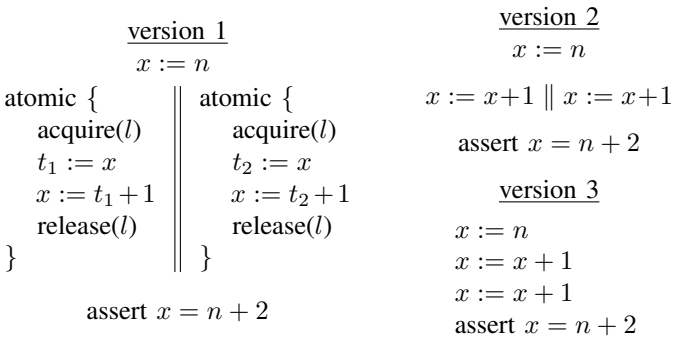

Fig. 3. Simplifying parallel increment.

Consequently, the code fragment executed by each child thread can be treated as an atomic block which executes in one step.

The simplification from version 1 to version 2 summarizes each atomic block with an atomic increment of x, while hiding global variable l and local variables t_1 and t_2. This summarization is possible because each atomic block leaves the value of l unchanged.

Finally, the simplification from version 2 to version 3 applies mover types again. Since each atomic increment is both a left and right mover, the two parallel increments can be converted into a sequence of two increments. Version 3 can be verified trivially by constructing a sequential verification condition and using an SMT solver to discharge it.

There are several advantages of the Civl approach. First, the transition relation of the program is never exposed to the programmer who specifies program versions using the familiar syntax of structured concurrent programs. Second, although an invariant may be needed to justify a program transformation in general, each invariant is simpler because it justifies only one transformation. Finally, invariants, even when they are needed, are supplied by annotating the structured program itself.

Section II presents a high-level overview of layered refinement, the collection of techniques underlying the Civl approach. Taken together, these techniques increase proof productivity by allowing the correctness argument to be expressed as a single layered concurrent program [8]. This section is targeted to an expert in the theory of concurrency verification and may be skipped on a first reading of the paper. Section III presents the modeling and specification features available to a Civl user through concrete examples.

Since the first published description of Civl [9], we have reimplemented the verifier completely. Section IV describes the current architecture of the Civl implementation as a conservative extension of the Boogie verifier.

The main contribution of Civl is a methodology supported by automated reasoning for implementing verified concurrent systems. We present two arguments that Civl improves the state of the art in constructing verified programs. First, Civl clearly allows new proofs of concurrent systems to be expressed. Second, these proofs have been accomplished on many programs by many researchers including several who were not involved in the design and implementation of Civl. Section V presents this accumulated experience.

II. Layered Refinement

Civl advocates layered refinement over structured concurrent programs. Instead of proving the safety of a program in one shot, the new approach allows the programmer to specify a chain of increasingly simpler programs starting from the original program. Each link of the chain, from program P to program Q, represents a single simplification that may be viewed as an abstraction from P to Q or a refinement from Q to P. The correctness of the program is established piecemeal by focusing on the simpler invariant required for each refinement step separately. Most importantly, all the layers and the supporting invariants are specified as a structured and layered concurrent program [8], thus hiding the low-level transition relation from the programmer.

Layered concurrent programs introduce a succinct presentation for multi-layer refinement proofs, which offer two major advantages for interactive proof construction. First, through a syntax for expressing "data layering" (i.e., which variables live on which layers) and "control layering" (i.e., which operations live on which layers), it is easy for the user to write, refine, and maintain a proof outline. Second, a layered concurrent program expresses only the changes in the program from one layer to the next. Thus, layered concurrent programs can result in much smaller proofs, especially for large programs.

While traditional approaches view refinement as a mechanism to specify behavior of concurrent programs, Civl views refinement as a tactic to simplify verification of safety properties. Consequently, the simulation relation justifying the refinement step in Civl is computed but never revealed to the programmer who focuses only on the program layers and the connecting invariants. The viability of the layered refinement approach depends on the existence of program simplification tactics that are easy to use by the programmer and whose justification can be checked automatically. Civl incorporates a number of such tactics described below.

Creating atomic blocks. The Civl programming model comprises concurrently-executing and dynamically-created tasks operating over global memory, each access to which must be encapsulated inside an indivisible atomic action. Global variables model either shared memory or communication channels. Civl uses a theory of commutative atomic actions [6], [7] to create sequential code blocks that appear to execute atomically, despite accesses to global state by multiple atomic actions in the code block.

Creating atomic actions. An atomic code block might be internally complex, due to sequencing, branching, looping, and recursion. Civl summarizes such a code block with an atomic action that hides all the internal details in favor of a declarative specification. Thus, atomic actions in Civl are used to model both low-level execution primitives and high-level summary specifications. To support such diverse usage, an atomic action in Civl generalizes a guarded command [10] to include a specification of failure [11] (in addition to blocking or successful execution) and the creation of asynchronous activity in the form of pending asyncs [12].

Synchronizing asynchrony. Civl supports elimination of pending asyncs from the atomic actions in a program via a tactic known as inductive sequentialization [13]. Introduction and elimination of pending asyncs in atomic actions together enable a program simplification that provides the appearance of executing in one step a collection of atomic computations executing asynchronously. This tactic amplifies the use of commutative atomic actions to allow summarization of both synchronous and asynchronous computation.

Civl allows introduction and hiding of global and local variables to change the state representation of the program. This change often results in a program whose atomic actions become commutative and thus the other tactics mentioned above become applicable. Variable introduction is performed as part of the tactic that creates atomic blocks; calls to special atomic actions assign meaning to the introduced variables. Variable hiding is performed as part of the tactic that creates atomic actions from atomic blocks; the created atomic action does not refer to the hidden variables.

Variable introduction and hiding in Civl has two other benefits. First, variable introduction naturally allows the user to introduce an arbitrary safety specification for the program. Second, it becomes unnecessary to support the notion of ghost state present in most provers for concurrent programs. Changing the state representation of the program often addresses the need for ghost state. Also, a variable may be introduced and hidden at the same layer for those special cases when ghost state is needed purely for invariant specification.

The tactic that creates atomic actions often needs constraints on the reachable states of the program. These constraints are supplied via yield invariants [14] which are named and parameterized invariants that can be reused and suitably instantiated across multiple program locations where interference may happen. Yield invariants combine the precision and flexibility of location invariants [4] with the compactness and modularity of rely-guarantee specifications [5]. Civl supports local reasoning with permissions that are redistributed by atomic actions and otherwise passed around the program without duplication [14]. Permissions are useful in proving locally both that yield invariants are interference-free and that atomic actions satisfy desired commutativity properties.

Civl supports the verification of arbitrary safety properties. Civl's notion of correctness is that the lowest-layer program is free of assertion failures. Arbitrary safety properties are expressible as assertions because auxiliary state (e.g., history variables) can be introduced into the program in addition to program state.

The client of a system constructed with layered refinement only needs to check that the established high-level specification captures the desired property. The details of a layered proof are not trusted since they are checked by Civl. However, the introduction of auxiliary state into the system at the lowest layer, sometimes needed to express a specification, is trusted.

III. Programming and Proving in Civl

In this section we illustrate the input language and the verification features of Civl. The presentation is necessarily brief and selective. Detailed documentation is available at our website civl-verifier.github.io.

Syntax. Civl is built on top of Boogie [15], a language and verifier for sequential programs. Boogie provides standard features for imperative programming such as assignments, sequencing, branching, looping, and procedures. Additionally, it provides specification features such as assert and assume statements, loop invariants, preconditions, postconditions, and axioms. The expression language of Boogie is first-order logic with built-in theories such as uninterpreted functions, integers, bitvectors, datatypes, and arrays. Civl adds the keywords `async` (asynchronous procedure call), `par` (parallel procedure call), and `yield` (yield point) to express concurrent behaviors. All other syntactic extensions are implemented using generic *attributes* which attach to abstract syntax tree nodes of a Boogie program. Attributes are of the form `{:attr e1, e2, ...}`, where `attr` is the attribute name and `e1, e2, ...` are parameter expressions of the attribute.

Atomic actions. Every access to a global variable has to be encapsulated into an atomic action. An atomic action consists of a *gate*, a one-state predicate that specifies the condition under which the action can execute or otherwise fail, and a *transition relation*, a two-state predicate that specifies the possible state updates of the action. Atomic actions are capable of specifying uniformly both low-level operations (like writing to a memory location or sending a message on a channel) and high-level operations (like acquiring a lock or reaching consensus in a distributed system). For example, the left column in Figure 4 shows atomic actions which acquire and release a lock, modeled by the global variable `l`. The Boogie procedures are identified as atomic actions by the `:right`/`:left` annotations which also declare their mover types; actions that are non-movers are annotated with `:atomic`. The action `AcquireSpec` blocks until `l` equals `None()` (denoting the availability of the lock) and then updates `l` to `Some(tid)` (denoting that the lock is held by the current thread with thread id `tid`). Conversely, `ReleaseSpec` asserts that the current thread holds the lock (the `assert` statement specifies the gate) and updates `l` to `None()`.

Program layers. In a Civl proof, the user explicitly organizes the program into layers using *layer annotations*. Variables and atomic actions have a *layer range*. In Figure 4, variable `l` is introduced at layer 1 and hidden at layer 2, and action `AcquireSpec` only exists at layer 2.

Concurrent computations are expressed by *yielding procedures*. The yielding procedure `Acquire` in Figure 4 acquires a lock by repeatedly invoking the compare-and-swap operation `CAS_b` to atomically set the global Boolean variable `b` from `false` to `true`. A yielding procedure is subject to interference from other concurrent threads at any point during its execution. However, `Acquire` is declared to *refine* the atomic action `AcquireSpec` at layer 1. This means that Civl checks that

```
var {:layer 1,2} l: Option Tid;

procedure {:right} {:layer 2,2}
AcquireSpec({:linear "tid"} tid: Tid)
modifies l;
{
  assume l == None();
  l := Some(tid);
}

procedure {:left} {:layer 2,2}
ReleaseSpec({:linear "tid"} tid: Tid)
modifies l;
{
  assert l == Some(tid);
  l := None();
}
```

```
var {:layer 0,1} b: bool;

procedure {:yields} {:layer 1}
  {:refines "AcquireSpec"}
  {:yield_preserves "LockInv"}
Acquire({:layer 1}{:linear "tid"} tid: Tid)
{
  var t: bool;

  while (true)
    invariant {:layer 1}{:yields}
      {:yield_loop "LockInv"} true;
  {
    call t := CAS_b(false, true);
    if (t) {
      call set_l(Some(tid));
      break;
    }
  }
}
```

```
procedure {:intro} {:layer 1}
set_l(v: Option Tid)
modifies l;
{ l := v; }

procedure {:yields} {:layer 2}
  {:refines "ClientSpec"}
  {:yield_preserves "LockInv"}
Client({:layer 1,2} {:hide}
       {:linear "tid"} tid: Tid)
{
  call Acquire(tid);
  ...
  call Release(tid);
}

procedure {:atomic} {:layer 3,3}
ClientSpec()
{ ... }
```

Fig. 4. A layered program, showing a lock implementation and its client. Left: Atomic actions for acquiring and releasing a lock. Middle: A spinlock implementation that refines the atomic action specification. Right: Introduction action for proving the lock refinement and a client of the lock.

`Acquire` "behaves like" `AcquireSpec`, and thus clients of the former can ignore the details of its implementation and instead reason with the atomic behavior of the latter. `Acquire` uses the global Boolean variable `b`, while `AcquireSpec` uses the global lock variable `l`. The connection between these two different representations is established by the *introduction action* `set_l`, which sets `l` from `None()` to `Some(tid)` when `b` is set from `false` to `true`. Finally, the yielding procedure `Client` protects a critical section with calls to `Acquire` and `Release` and declares that it refines the action `ClientSpec` at layer 2.

The layer annotation of a yielding procedure denotes its *disappearing layer*. The procedure exists (with changing bodies) on all layers below and up to its disappearing layer. For example, `Acquire` exists on layer 0 and 1, and `Client` exists on layer 0, 1, and 2. Intuitively, a procedure is replaced with its refined atomic action above its disappearing layer.

Figure 4 encodes four program layers. Layer 0 is the most concrete program. It contains procedure `Client` which calls procedure `Acquire`, and `Acquire` implements a spinlock using calls to `CAS_b`; `b` is the only global variable, and `Client` and `Acquire` have no input parameters. Layer 1 introduces the global variable `l` and the local input parameters `tid`, along with the introduction action `set_l` (the call to `set_l` does not exist at layer 0). At layer 2, `Acquire` is gone and the body of `Client` is rewritten to make calls to the actions `AcquireSpec` and `ReleaseSpec`; `b` is hidden and `l` is the only global variable. At layer 3, `Client` is also gone, and any potential calls to `Client` are replaced by its atomic summary `ClientSpec`; global variable `l` and the parameter `tid` do not exist anymore.

Layering provides a form of modularity. At layer 2 we do not care about how the lock is implemented, and at layer 3 we do not care that a lock was used at all. The applied proof tactics (variable introduction, variable hiding, and atomic blocks) simplify the necessary invariants on every layer.

Yield sufficiency. Civl partitions the bodies of yielding procedures into *yield-to-yield fragments*. The following code locations are *yield points*: procedure entry and exit, loop headers annotated with `{:yields}`, and explicit `yield` statements. Context switches are only considered at yield points, and the code between two yield points is a yield-to-yield fragment. At layer 1, in `Acquire` every loop iteration (i.e., call to `CAS_b`) is a yield-to-yield fragment, and in `Client` there is a yield before and after every call. At layer 2, something interesting happens. The body of `Client` does not call any procedures anymore (the calls are to atomic actions now), and thus `Client` has only a single yield-to-yield fragment. Civl justifies this simplification using *reduction* [6], [7]. Concretely, using the fact that `AcquireSpec` is a *right mover* and `ReleaseSpec` is a *left mover*. In general, every yield-to-yield fragment is checked to be a sequence of right movers, followed by at most one non-mover, followed by a sequence of left movers.

Refinement. To justify the summarization of a yielding procedure at layer n by an atomic action, Civl checks that in every execution of the procedure, the effect of the refined action happens in exactly one yield-to-yield fragment and that other yield-to-yield fragments leave the layer-$(n+1)$ state unchanged. In `Acquire`, every loop iteration where `CAS_b` fails leaves `l` unchanged, while the (final) iteration where `CAS_b` succeeds also updates `l` to `Some(tid)` and thus produces the effect of `AcquireSpec`.

Invariants. Civl performs refinement checking modularly, by considering every yield-to-yield fragment in isolation. This usually requires certain properties to hold at yield points, notwithstanding any interference from other concurrent threads. Civl supports location invariants [4] and yield invariants [14], which are checked to be interference-free across all yield-to-yield fragments in the program. Yield invariants are named and parameterized invariants that can be reused and suitably instantiated across multiple yield points. The following code shows the yield invariant `LockInv`.

```
procedure {:yield_invariant} {:layer 1} LockInv();
requires b <==> (l != None());
```

In `Acquire` (Figure 4), `LockInv` is attached to the procedure entry and exit using the `:yield_preserves` annotation, and to the loop header using the `:yield_loop` annotation. We give examples of parameterized yield invariants below.

Permissions. Certain invariants, like those connecting local variables from different scopes, can be tedious to express and propagate. Civl addresses this problem using *linear permissions*. Program variables can be declared as *linear*, from which Civl calculates the *available* variables at every control location, assigns every available variable a set of *permissions*, and ensures that there is no duplication across these permission sets. Civl allows the user to customize the type of permissions and the assignment of permissions to variables.

The lock specification in Figure 4 uses linearity to express unique thread identifiers. The type declaration

```
type {:linear "tid"} Tid;
```

specifies the permissions for the *linear domain* `tid` to be of type `Tid`, the type of thread identifiers. This means that every variable that is linear under domain `tid` gets assigned a set of `Tid` values. The assignment is specified using *collector* functions. Civl uses the following default collector in the absence of a user-specified collector.

```
function {:linear "tid"} TidCol(x: Tid) : [Tid]bool
{ MapConst(false)[x := true] }
```

We use a map from `Tid` to `bool` to model a set. The polymorphic map constructor `MapConst` applied to `false` returns a map set to false everywhere representing an empty set. `TidCol` assigns linear variables of type `Tid` (like the input parameter `tid` of `AcquireSpec` and `ReleaseSpec`) the single value the variable contains as its permission. Consider an instance of `AcquireSpec` and an instance of `ReleaseSpec` with parameters `tid1` and `tid2`, respectively. By linearity, Civl gets to assume that the multiset $\texttt{TidCol(tid1)} \uplus \texttt{TidCol(tid2)} = \{\texttt{tid1}, \texttt{tid2}\}$ does not contain any duplicates, which implies $\texttt{tid1} \neq \texttt{tid2}$. This assumption is used to show that the `AcquireSpec` instance commutes to the right of the `ReleaseSpec` instance, an important part of the proof that `AcquireSpec` and `ReleaseSpec` satisfy their mover types.

Figure 5 presents an example inspired by barrier synchronization to demonstrate how permissions are useful in proving invariants. The program has two global variables, `barrier` and `count`, to represent the set of identifiers inside the barrier and the number of threads outside the barrier, respectively. The atomic actions `EnterBarrier` and `ExitBarrier` encode entering and exiting the barrier by a thread, respectively. The yield invariant `ThreadInv` is parameterized by a thread identifier `j` and indicates that `j` is in the barrier. Typically, a thread with identifier `i` would enter the barrier by calling `EnterBarrier(i)`, yield to other threads by calling `ThreadInv(i)`, and then exit the barrier by calling `ExitBarrier(i)`. The linearity of parameter `j` of `ThreadInv` and parameter `i` of `ExitBarrier` allows us to assume that `j` and `i` are distinct, and therefore `ThreadInv` is preserved by `ExitBarrier`. Preservation by `EnterBarrier` is trivial since this action only adds elements to `barrier`.

Permission redistribution. Now consider the following yield invariant `BarrierInv` that indicates that the sum of the size of `barrier` and `count` is equal to `N`, the total number of threads.

```
var {:layer 0,1} barrier: [Tid]bool;
var {:layer 0,1} count: int;

procedure {:atomic} {:layer 1} EnterBarrier(
  {:linear "tid"} i: Tid)
modifies barrier;
{
    barrier[i] := true;
    count := count - 1;
}

procedure {:atomic} {:layer 1} ExitBarrier(
  {:linear "tid"} i: Tid)
modifies barrier;
{
    assert barrier[i];
    barrier[i] := false;
    count := count + 1;
}

procedure {:yield_invariant} {:layer 1} ThreadInv(
  {:linear "tid"} j: Tid);
requires barrier[j];
```

Fig. 5. Using permissions to prove invariants.

```
procedure {:yield_invariant} {:layer 1} BarrierInv();
requires Size(barrier) + count == N;
```

This invariant cannot be proved on the code in Figure 5. The action `EnterBarrier` does not preserve `BarrierInv` whenever `barrier[i]` already holds upon entry. This condition, of course, cannot happen in the program, since a thread only calls `EnterBarrier` when it is outside the barrier. But this constraint is not encoded in the current specification. An attempt to encode this constraint would be to make the global variable `barrier` linear. However, this strategy would force us to drop the linear annotation on parameter `i` of `ExitBarrier` which would then make `ThreadInv` unprovable.

To solve this programming problem, we present a more sophisticated use of permissions that depends on custom collectors and new linearity annotations on local variables. The datatype declaration

```
type {:linear "perm"} {:datatype} Perm;
function {:constructor} Left(i: Tid): Perm;
function {:constructor} Right(i: Tid): Perm;
```

specifies the permissions for a new linear domain `perm`. The datatype `Perm` has two constructors `Left` and `Right`; each constructor wraps a thread identifier to create a `Perm` value. The collectors for `perm` are shown below.

```
function {:linear "perm"} TidCol(x: Tid) : [Perm]bool
{ MapConst(false)[Left(x) := true][Right(x) := true] }

function {:linear "perm"} TidSetCol(xs: [Tid]bool)
: [Perm]bool
{ (lambda p: Perm :: is#Left(p) && xs[i#Left(p)]) }
```

The collector `TidCol` defines the permissions stored in a single thread identifier `x` as the set comprising `Left(x)` and `Right(x)`. The collector `TidSetCol` collects the permissions in a set of thread identifiers `xs` by collecting `Left(x)` for each element `x` in `xs`. Additionally, there is the following default collector for type `Perm`.

```
function {:linear "perm"} PermCol(x: Perm) : [Perm]bool
{ MapConst(false)[x := true] }
```

Figure 6 shows the revised code for our example which now uses the linear domain `perm` throughout. The global

```
var {:layer 0,1} {:linear "perm"} barrier: [Tid]bool;
var {:layer 0,1} count: int;

procedure {:atomic} {:layer 1} EnterBarrier(
  {:linear_in "perm"} i: Tid)
returns ({:linear "perm"} p: Perm)
modifies barrier;
{
    barrier[i] := true;
    count := count - 1;
    p := Right(i);
}

procedure {:atomic} {:layer 1} ExitBarrier(
  {:linear_in "perm"} p: Perm, {:linear_out "perm"} i: Tid)
modifies barrier;
{
    assert p == Right(i) && barrier[i];
    barrier[i] := false;
    count := count + 1;
}

procedure {:yield_invariant} {:layer 1} ThreadInv(
  {:linear "perm"} p: Perm, j: Tid);
requires p == Right(j) && barrier[j];
```

Fig. 6. Permission redistribution in atomic actions.

variable `barrier` is linear and consequently a store of permissions. The signatures and implementation of `EnterBarrier`, `ExitBarrier`, and `ThreadInv` have also changed.

We now present the intuition behind the revised implementation. `EnterBarrier` splits the permissions `{Left(i), Right(i)}` contained in its input parameter `i` into `Left(i)` which is put into `barrier` and `Right(i)` which is returned via the output parameter `p`. The `linear_in` annotation on `i` indicates that the permissions in `i` are consumed by the call and are therefore unavailable afterwards. The permission `p` and the unavailable thread identifier `i` are used to call `ThreadInv`. Finally, when `ExitBarrier` is called with `p` and `i` and `i` is removed from `barrier`, the permission `Left(i)` is also removed from `barrier`. This permission becomes available to be joined with `Right(i)` contained in `p` so that the full permission set `{Left(i), Right(i)}` is put into `i` which becomes available after the call. This protocol is indicated by the `linear_in` annotation on `p` and the `linear_out` annotation on `i`.

This example shows that permissions can be redistributed without duplication by an atomic action among global variables and its parameters. This ability to soundly redistribute permissions allows us to compactly express and prove coordination protocols.

Asynchrony. Asynchronous invocations—calls that create a new concurrent thread of computation without the caller waiting for the operation to complete—are challenging to specify and verify. Civl provides the inductive sequentialization [13] proof rule to sidestep the arduous task of inventing complex inductive invariants that capture all possible interleavings of an asynchronous program.

Consider the action `ASYNC_SUM` in Figure 7. It uses an output variable `PAs` that represents *pending asyncs*, asynchronous operations that are spawned by `ASYNC_SUM` but executed asynchronously at some later time. Concretely, `ASYNC_SUM` creates the multiset of pending asyncs `set_of_ADD(1, n)` = $\{$`ADD(1)`, `ADD(2)`, ..., `ADD(n)`$\}$, which could be refined to a

```
procedure {:atomic}{:layer 1}{:IS "SUM","INV"}{:elim "ADD"}
ASYNC_SUM (n: int)
returns ({:pending_async "ADD"} PAs:[PA]int)
modifies x;
{
  assert n >= 0;
  PAs := set_of_ADD(1, n);
}

procedure {:atomic}{:layer 2} SUM (n: int)
modifies x;
{
  assert n >= 0;
  x := x + (n * (n+1)) div 2;
}

procedure {:left}{:layer 1} ADD (i: int)
modifies x;
{ x := x + i; }

procedure {:IS_invariant}{:layer 1} INV (n: int)
returns ({:pending_async "ADD"} PAs:[PA]int,
         {:choice} choice:PA)
modifies x;
{
  var i: int;
  assert n >= 0;
  assume 0 <= i && i <= n;
  x := x + (i * (i+1)) div 2;
  PAs := set_of_ADD(i+1, n);
  choice := ADD(i+1);
}
```

Fig. 7. Sequentialization of the asynchronously computed sum from 1 to n. We are omitting annotations that support automated reasoning with quantifiers.

procedure that asynchronously invokes `ADD` in a `while` loop.

The annotations on `ASYNC_SUM` tell Civl instead to convert it into `SUM`, by *eliminating* from it the pending asyncs to `ADD` using the *invariant action* `INV`. `SUM` adds to `x` the value $\frac{n(n+1)}{2}$, which is the cumulative effect of the asynchronous `ADD` operations. The key is that `INV` only talks about a single interleaving of the `ADD` operations: `ADD(1); ADD(2); ...; ADD(n)`. It represents any prefix of this single interleaving as follows. It (1) nondeterministically picks `i` between `0` and `n` denoting the number of finished `ADD`'s, (2) increases `x` by $\frac{i(i+1)}{2}$ to capture the effect of executing `ADD(1)` to `ADD(i)`, (3) creates pending asyncs for `ADD(i+1)` to `ADD(n)`, and (4) specifies that the next pending async we wish to execute in our sequential order is `ADD(i+1)`. `INV` represents `ASYNC_SUM` with `i` = `0`, `SUM` with `i` = `n`, and the induction order from `i` to `i+1` is specified by the user through the output variable `choice`. The justification for this sequential reduction is that `ADD` is a left mover, and thus can always be commuted to the desired location in the sequentialization.

IV. Implementation

Civl is implemented as a conservative extension of the Boogie verifier. The extensions to the syntax (Section III) and the verification engine do not affect ordinary Boogie programs. The Boogie verifier itself is implemented as a pipeline with a sequence of phases—parsing, type checking, verification condition generation, solver invocation, and error reporting. For every procedure, a verification condition in SMT-LIB format is passed to an SMT solver running in a separate process. If an error is discovered, a diagnostic error trace is calculated by examining the model returned by the solver.

The implementation of Civl adds two more phases into the pipeline of the Boogie verifier. Initially, the Civl attributes are parsed together with the rest of the Boogie program and the standard Boogie type checker is run. Then, the *Civl type checker* validates the Civl attributes and converts them into internal data structures. Next, the *Civl processor* compiles all proof obligations related to concurrency down to sequential Boogie procedures. Finally, the existing Boogie pipeline for converting procedures into verification conditions takes over.

Civl type checker. The type checker has three main parts.

First, a *layer analysis* [8] checks that the layer annotations are consistent. This analysis ensures that all program layers encoded by the input layered program are well-formed, e.g., that variables accessed and procedures/actions called on some layer actually exist on that layer. It also ensures the soundness of our refinement check. For example, in Figure 4 we could not refine `Client` at layer 1, because its callee `Acquire` first needs to be converted to the action `AcquireSpec`, which happens from layer 1 to layer 2. For sound variable introduction, only introduction actions and invariants are allowed to access global variables at their introduction layer. For example, at layer 1 only `set_l` and `LockInv` refer to `l`, whereas `AcquireSpec` only refers to it at layer 2.

Second, a *yield sufficiency analysis* [7] checks, for each layer separately, that it is safe to consider context switches only at yield points. This check is implemented by computing a simulation relation [16] between a labeled control-flow graph and a specification automaton that encodes all sequences of mover types allowed by Lipton's reduction theorem [6]. The specification automaton is shown in panel ① of Figure 8. Panel ② shows the labeled graph for procedure `Acqurie` at layer 1. Node n_0 represents the loop head. Since the loop is yielding, the edge to the loop condition n_1 is labeled Y. At n_1 we either exit the loop and thus the entire procedure on the private edge to n_3, or we execute the non-mover `CAS_b` on the edge to n_2 labeled N. At n_2, corresponding to the if condition, we either execute the introduction action `set_l` and break from the loop, or we loop back to the loop head n_0, both of which are private edges. Panels ③ and ④ show that the calls to the yielding procedures `Acquire` and `Release` are labeled with Y at layer 1 but with the mover type of their respective refined atomic action at layer 2. For simplicity, Civl does not allow a yield-to-yield fragment that starts within a loop to wrap around the loop head, and thus checks that every loop that contains a Y edge is a yielding loop.

Third, a *linear flow analysis* [14] computes the available linear variables at each control location of a procedure, and ensures that calls to procedures, atomic actions, and yield invariants satisfy their linear interfaces. The following code snippet refers to Figure 6.

```
// i available, p unavailable
call p := EnterBarrier(i);
// i unavailable, p available
call ThreadInv(p, i);
// i unavailable, p available
call ExitBarrier(p, i);
// i available, p unavailable
```

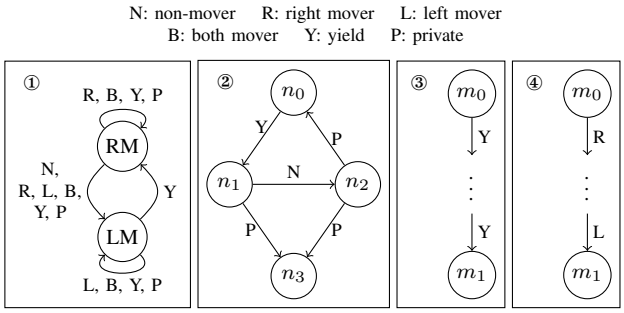

Fig. 8. Labeled control-flow graphs for yield sufficiency analysis of Figure 4. ① Specification automation. ② `Acquire` at layer 1. ③ `Client` at layer 1. ④ `Client` at layer 2.

`EnterBarrier` requires `i` to be available and consumes it, making `p` available in return. The unavailable `i` can be used in places where it is not required to be linear, in particular the calls to `ThreadInv` and `ExitBarrier`. After `ExitBarrier` which consumes `p`, variable `i` is available again.

Civl processor. To target Boogie's verification-condition generator, Civl eliminates layers, concurrency, and linearity from the input layered concurrent program by creating a collection of sequential *checker procedures*. There are two advantages to this approach. First, modular decomposition into checker procedures improves scalability by creating small verification problems. Second, verification failures in checker procedures are processed to create targeted error messages. In the following we explain the categories of checker procedures Civl generates. We do not have the space to present detailed encodings; we suggest that interested readers use the command-line flag `-civlDesugaredFile` to inspect the plain Boogie program generated by the Civl processor.

A common functionality required by multiple checker procedures is the computation of a logical transition relation from the code representation of an atomic action. For each code path, Civl computes a path constraint from its static single assignment form, and then iteratively eliminates intermediate copies of variables by finding and inlining definitions. Variables that cannot be eliminated are existentially quantified. The transition relation is the disjunction over all path formulas.

Permission redistribution among linear variables occurs through assignment, parameter passing, and mutation in atomic actions. The first two sources of redistribution are tracked by the syntactic flow analysis in the Civl type checker. For the third source, a checker procedure for each atomic action ensures that no permission duplication occurs due to its execution. This semantic check involves user-supplied collector functions. For example, the checker procedure for `ExitBarrier` from Figure 6 validates the postcondition

```
TidSetCol(barrier) ⊎ TidCol(i) ⊆
    TidSetCol(old(barrier)) ⊎ PermCol(old(p)),
```

stating that the permissions flowing into the action through `barrier` and `p` must be a subset of the permissions flowing out through `barrier` and `i`. The resulting non-duplication guarantee among linear variables is injected into all the following checks as a free assumption.

```
procedure CommutativityChecker(tid_1: Tid, tid_2: Tid)
requires tid_1 != tid_2;   // derived from linearity
requires l == Some(tid_2); // gate of ReleaseSpec
modifies b, l;
{
  call AcquireSpec(tid_1); // inlined
  call ReleaseSpec(tid_2); // inlined
  // trans. rel. of ReleaseSpec(tid_2); AcquireSpec(tid_1)
  assert l == Some(tid_1);
}
```

Fig. 9. Commutativity checker for `AcquireSpec` and `ReleaseSpec`.

The mover type of each atomic action is verified by pairwise checks against every atomic action with an overlapping layer range. Each such check is encoded by multiple checker procedures to account for commutativity of both failing and successful behaviors. For example, the commutativity check between `AcquireSpec` and `ReleaseSpec` is shown in Figure 9. Recall that this check succeeds because the first call blocks due to the constraint we get from linearity. In addition, each left mover and introduction action is separately checked to have a failing or successful behavior from each initial state.

Invariants are verified separately for each layer n, resulting in a checker procedure for each yielding procedure with disappearing layer at least n. Civl constructs the checker procedure from the code of the yielding procedure as follows. First, calls to invariants and introduction actions at layers other than n are dropped and calls to yielding procedures with disappearing layers lower than n are rewritten to calls of their respective refined actions. Next, asynchronous and parallel calls (of which ordinary calls are a special case) are translated. An asynchronous call to a yielding procedure is translated into an assertion of the precondition of the procedure. An asynchronous call to an action is either synchronized or converted into a pending async [12]. A parallel call may contain arms that are actions, yield invariants, or yielding procedures. Each such call is rewritten into a sequence comprising calls to actions and parallel calls whose arms are either yield invariants or yielding procedures. For example, `par` $A \mid P \mid I \mid B \mid C \mid Q \mid D$ with actions A, B, C and D, procedures P and Q, and invariant I, is rewritten to `call` A; `par` $P \mid I$; `call` B; `call` C; `par` Q; `call` D. All calls to atomic actions are inlined. Any parallel call remaining at this point is a yield where interference is possible. Next, each yield is instrumented to record a snapshot of the global variables immediately after the yield. This snapshot is used to assert the preservation of all invariants in the program at the end of a yield-to-yield fragment. Finally, each parallel call (with arms that are yielding procedures or yield invariants) comprising a yield is itself desugared as follows: (1) assert preconditions of yielding procedures and yield invariants, (2) havoc all global variables, (3) assume postconditions of yielding procedures and yield invariants. The soundness of this translation of concurrent code to sequential code is ensured by the yield sufficiency analysis of the Civl type checker. A side condition for asynchronous calls forbids global state updates between an asynchronous call to a yielding procedure and the next yield point. Additionally, there are restrictions on the sequence of arms in a parallel call. For example, any left mover must occur before any right mover, and there cannot be both a yielding procedure and a non-mover in the sequence.

At the disappearing layer n of every yielding procedure, a checker procedure verifies refinement of the specified atomic action by tracking two local Boolean variables, `pc` and `ok`, each initialized to false. The variable `pc` is set to true as soon as a yield-to-yield fragment modifies any layer-$(n+1)$ state; before any such modification it is asserted that `pc` is false. The variable `ok` is set to true as soon as a yield-to-yield fragment modifies the layer-$(n+1)$ state according to a transition admitted by the refined action; `ok` is asserted to be true when the procedure returns. Overall, we check that layer-$(n+1)$ state is modified at most once, and that a behavior of the refined action occurs at least once.

Each invocation of the inductive sequentialization [13] rule results in a collection of checker procedures, one each for the base and conclusion case and one for the inductive step corresponding to each eliminated pending async.

V. Experience

Civl has been used in many efforts to develop verified concurrent systems, both by the authors of Civl and by other researchers. These efforts include a concurrent garbage collector [9], a Paxos implementation [13], and implementations of concurrent data structures: the FastTrack data-race detector [17], Chase-Lev deque [18], and Java weakly-consistent objects [19]. Civl has also been used to prototype techniques for verification under TSO semantics [20]. Civl is fast enough to be used for interactive development. Even on our large benchmarks, verification time is a few seconds.

Our experience suggests that Civl's specification mechanisms—layering, commutativity, yield invariants—are natural for users. These features aid discovery of provable implementations by encouraging the user to think about different layers of abstraction, the primitives for each layer, and suitable organization of the reasoning technique at each layer. In addition, layers enable partitioning of work among multiple developers each working on the proof of a particular layer with agreed-upon interfaces between layers.

We present more details about two major case studies to provide anecdotal evidence for the improvements in developing verified concurrent systems enabled by Civl.

Concurrent Garbage Collector. An author of this paper together with other researchers used Civl to develop a verified concurrent garbage collector and object allocator that improves upon the mark-and-sweep garbage collector by Dijkstra el al. [21] in two ways. First, the new collector supports more than one mutator running in parallel with the collector. Second, it requires a write-barrier only on updates of heap pointers but not on root modifications. The Civl implementation is realistic, given in terms of individual CPU operations. The refined specification comprises high-level atomic actions for object allocation and access, that provide the illusion of unbounded memory in which individual objects are not reused.

The proof is done via a sequence of 6 program transformations connecting 7 program layers. Layer 0 is described in terms of individual atomic CPU operations. Layer $0 \rightarrow 1$ introduces locks and atomic actions for read/write accesses. Layer $1 \rightarrow 2$ uses the locks and protected accesses to construct higher-level atomic operations that are used in the barrier synchronization algorithm for root scanning and in the mark-sweep algorithm. The collector operates in three phases—idle, mark, and sweep. Layer $2 \rightarrow 3$ reasons about the coordination between the collector and the mutators to make phase changes safely. The mark algorithm performs a depth-first search of the heap starting from the roots. The stack in this search comprises "gray" objects. Layer $3 \rightarrow 4$ changes the representation of the gray objects to a set. Layer $4 \rightarrow 5$ reasons about the root scanning algorithm that internally uses barrier synchronization to create an atomic action that scans all roots in one step. Reasoning about the write barrier also happens during this transformation. Layer $5 \rightarrow 6$ reasons about the mark-sweep algorithm using the atomic actions for scanning roots, maintaining the set of gray objects, and changing object colors. The garbage collector is hidden entirely, leaving the client with atomic actions for allocating objects, reading and writing object fields, and checking object equality.

This proof was constructed and reported in 2015 [9]. Since then, Civl has been rewritten but the proof has been maintained and improved. The current artifact is 2031 LOC and verifies in 25s on a standard Mac. The biggest improvement happened with the introduction of yield invariants [14] which reduced the verification time by a factor of 10.

Paxos. The Paxos protocol [22] establishes consensus among a set of unreliable nodes in an asynchronous network without a central coordinator. This protocol lies at the core of any system with replicated state. It is difficult to both understand and implement. The authors of this paper together with other researchers constructed a verified implementation [13] of single-decree Paxos, which establishes consensus on a single value. The verified implementation only uses primitive atomic actions, like reading or writing a single memory address, and sending or receiving a single message.

The proof is constructed via a sequence of 2 program transformations done over 3 layers. Layer 0 implements event handlers using primitive atomic actions for sending and receiving network messages, and for updates to the local state and decision variable at each Paxos node. The transformation from layer 0 to layer 1 converts event handlers to atomic actions at the granularity typically used to describe protocols in papers. At the same time, this transformation changes the state representation to make it easier to apply the next transformation. The invariant justifying this transformation simply connects the two state representations. The transformation from layer 1 to layer 2 uses inductive sequentialization [13] to create a single atomic action where consensus is reached in one step by nondeterministically setting decisions at each node consistently. The invariant justifying this transformation captures the intuition of the protocol. It has 4 conjuncts and is considerably simpler than the invariants in other published proofs of the Paxos protocol. For example, the proof [23] using Ivy has 5 other supporting invariants in addition to the 4 used in the Civl proof. The current artifact for the Civl proof is 1116 LOC and verifies in 7s on a standard Mac.

VI. Related Work

In this section we compare Civl to other *reusable tools* that have *support for concurrency*.

TLA+ [24] and Event-B [25] are two classic tools for refinement reasoning over transition systems. Ivy [26] verifies transition systems using a restricted modeling and specification language (notably without functions and arbitrary quantification) that makes verification conditions decidable. While Ivy requires manual effort to encode distributed systems concepts in this restricted language, Civl requires manual effort to automate quantifier reasoning. Ivy also has a synchronous, reactive programming language from which it can extract asynchronous, distributed implementations [27]. This programming model, which cannot express fine-grained concurrency, can be encoded in Civl by threading a linear parameter through atomic actions and procedures. Ivy provides liveness reasoning and information hiding via modules.

Iris [28] is a Coq-based formalization of a program logic suitable for reasoning about fine-grained concurrent programs with higher-order ghost state. The focus in Iris is to clarify and simplify concurrent separation logics around a few primitive concepts in order to provide a suitable foundation for developing reasoning mechanisms for concurrent programs. Compared to Iris, Civl is less flexible but provides more automation on a programming notation that supports standard models of concurrent programming. ReLoC [29] is a logic built on top of Iris for interactively proving contextual refinement judgments.

Chalice [30] verifies monitor invariants, in addition to absence of data races and deadlocks, on a small Java-like concurrent programming language. VeriFast [31] supports separation logic specifications, resource invariants, and higher-order ghost state on concurrent C and Java programs. Prusti [32] uses the guarantees of the Rust type system to simplify the manual annotation effort. VerCors [33] builds on separation logic specifications and provides verification features for several concurrent programming idioms, e.g., based on histories and process algebra. VCC [34] is a verifier for concurrent C programs. VCC allows the programmer to construct a custom verification methodology via extensive support for the introduction of ghost types and values. Noninterference is accomplished via a network of type-level global invariants which together must satisfy certain stability and admissibility conditions. Similar to Civl, these tools use SMT solvers as the reasoning engine, exploit programmer interaction, and support modular reasoning. Civl provides features not present in these tools such as layered refinement and yield invariants.

Anchor [35], a successor to Calvin-R [36], is a lightweight verifier for a small Java-like programming language. Anchor allows the programmer to compactly specify conditional mover types for read and write accesses of shared object fields.

It is less modular than Civl and other tools discussed here; inlining is used extensively to deal with procedure calls.

Armada [37] is a language and verifier that implements layers, mover types, and explicit noninterference reasoning. Armada is inspired by Civl but also supports weak memory and extensibility via new simplification tactics. While Civl represents all program layers in a single layered concurrent program, Armada connects explicitly written programs using proof scripts that invoke mechanized theorems.

VII. Conclusion

The Civl static verifier aids the development of verified concurrent systems through language-integrated proof structuring mechanisms, an array of program-simplifying proof tactics, and modular and automatable verification conditions. The modeling features provided in Civl are general; they can be specialized to many different domains by building custom linguistic support and automation. For example, it is possible to use Civl as the verification backend for domain-specific languages suitable for developing implementations of distributed protocols, concurrent data structures, or even system-level hardware implementations. Overall, Civl opens many new opportunities in development of programming tools for concurrent systems.

Civl's capabilities to generate verification conditions for checking commutativity, refinement, and noninterference can be leveraged individually by a verifier. It is also conceivable to design a programming language that supports layering and atomic actions natively, and uses Civl as a backend for verification. This language would generate executable code from the lowest-layer program which invokes atomic actions whose implementation is provided by the language runtime.

Our experience suggests that progress on the following important challenges should increase the applicability and usability of Civl. First, Civl's verification conditions have quantifiers which can results in unpredictable verification times. Domain-specific techniques for automatic quantifier instantiation or language mechanisms for conveniently specifying instances would help. Second, Civl supports linear maps [38] for reasoning about disjoint but flat memory. Extension to support reasoning about nested linear maps would make it easier to encode standard heap programming models. Third, layered programs in Civl are challenging to comprehend, edit, and refactor; tools to help with these tasks would be helpful. A module system for factoring out libraries and their layered proofs would aid the development of large verified systems.

References

[1] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *POPL*, 2011.

[2] A. Farzan, Z. Kincaid, and A. Podelski, "Proof spaces for unbounded parallelism," in *POPL*, 2015.

[3] A. Farzan and A. Vandikas, "Reductions for safety proofs," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2020.

[4] S. S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, 1976.

[5] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, 1983.

[6] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, 1975.

[7] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI*, 2003.

[8] B. Kragl and S. Qadeer, "Layered concurrent programs," in *CAV*, 2018.

[9] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran, "Automated and modular refinement reasoning for concurrent programs," in *CAV*, 2015.

[10] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, 1975.

[11] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," in *POPL*, 2009.

[12] B. Kragl, S. Qadeer, and T. A. Henzinger, "Synchronizing the asynchronous," in *CONCUR*, 2018.

[13] B. Kragl, C. Enea, T. A. Henzinger, S. O. Mutluergil, and S. Qadeer, "Inductive sequentialization of asynchronous programs," in *PLDI*, 2020.

[14] B. Kragl, S. Qadeer, and T. A. Henzinger, "Refinement for structured concurrent programs," in *CAV*, 2020.

[15] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2005.

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

[17] J. R. Wilcox, C. Flanagan, and S. N. Freund, "VerifiedFT: a verified, high-performance precise dynamic race detector," in *PPoPP*, 2018.

[18] S. O. Mutluergil and S. Tasiran, "A mechanized refinement proof of the Chase-Lev deque using a proof system," *Computing*, vol. 101, no. 1, 2019.

[19] S. Krishna, M. Emmi, C. Enea, and D. Jovanovic, "Verifying visibility-based weak consistency," in *ESOP*, 2020.

[20] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran, "Reasoning about TSO programs using reduction and abstraction," in *CAV*, 2018.

[21] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, 1978.

[22] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 1998.

[23] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made EPR: decidable reasoning about distributed protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017.

[24] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, 2002.

[25] J. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 6, 2010.

[26] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *PLDI*, 2016.

[27] K. L. McMillan and O. Padon, "Ivy: A multi-modal verification tool for distributed algorithms," in *CAV*, 2020.

[28] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *J. Funct. Program.*, vol. 28, 2018.

[29] D. Frumin, R. Krebbers, and L. Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency," in *LICS*, 2018.

[30] K. R. M. Leino and P. Müller, "A basis for verifying multi-threaded programs," in *ESOP*, 2009.

[31] B. Jacobs and F. Piessens, "Expressive modular fine-grained concurrency specification," in *POPL*, 2011.

[32] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019.

[33] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, "The VerCors tool set: Verification of parallel and concurrent software," in *IFM*, 2017.

[34] E. Cohen, M. Moskal, W. Schulte, and S. Tobies, "Local verification of global invariants in concurrent programs," in *CAV*, 2010.

[35] C. Flanagan and S. N. Freund, "The Anchor verifier for blocking and non-blocking concurrent software," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020.

[36] S. N. Freund and S. Qadeer, "Checking concise specifications for multithreaded software," *J. Object Technol.*, vol. 3, no. 6, 2004.

[37] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao, "Armada: low-effort verification of high-performance concurrent programs," in *PLDI*, 2020.

[38] S. K. Lahiri, S. Qadeer, and D. Walker, "Linear maps," in *PLPV*, 2011.