Verifying Concurrent Programs: Refinement, Synchronization, Sequentialization

by

Bernhard Kragl

September 4, 2020

A thesis presented to the
Graduate School
of the
Institute of Science and Technology Austria, Klosterneuburg, Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Institute of Science and Technology

The thesis of Bernhard Kragl, titled *Verifying Concurrent Programs: Refinement, Synchronization, Sequentialization*, is approved by:

Supervisor: Thomas A. Henzinger, IST Austria, Klosterneuburg, Austria

Signature: _____

Committee Member: Krishnendu Chatterjee, IST Austria, Klosterneuburg, Austria

Signature: _____

Committee Member: Shaz Qadeer, Novi, Seattle, USA

Signature: _____

Committee Member: Georg Weissenbacher, TU Wien, Vienna, Austria

Signature: _____

Defense Chair: Björn Hof, IST Austria, Klosterneuburg, Austria

Signature: _____

signed page is on file

IST Austria Thesis, ISSN: 2663-337X

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: _____

Bernhard Kragl
September 4, 2020
signed page is on file

Abstract

Designing and verifying concurrent programs is a notoriously challenging, time consuming, and error prone task, even for experts. This is due to the sheer number of possible interleavings of a concurrent program, all of which have to be tracked and accounted for in a formal proof. Inventing an inductive invariant that captures all interleavings of a low-level implementation is theoretically possible, but practically intractable. We develop a refinement-based verification framework that provides mechanisms to simplify proof construction by decomposing the verification task into smaller subtasks.

In a first line of work, we present a foundation for refinement reasoning over *structured concurrent programs*. We introduce *layered concurrent programs* as a compact notation to represent multi-layer refinement proofs. A layered concurrent program specifies a sequence of connected concurrent programs, from most concrete to most abstract, such that common parts of different programs are written exactly once. Each program in this sequence is expressed as structured concurrent program, i.e., a program over (potentially recursive) procedures, imperative control flow, gated atomic actions, structured parallelism, and asynchronous concurrency. This is in contrast to existing refinement-based verifiers, which represent concurrent systems as flat transition relations. We present a powerful refinement proof rule that decomposes refinement checking over structured programs into modular verification conditions. Refinement checking is supported by a new form of modular, parameterized invariants, called *yield invariants*, and a *linear permission* system to enhance local reasoning.

In a second line of work, we present two new reduction-based program transformations that target asynchronous programs. These transformations reduce the number of interleavings that need to be considered, thus reducing the complexity of invariants. *Synchronization* simplifies the verification of asynchronous programs by introducing the fiction, for proof purposes, that asynchronous operations complete synchronously. Synchronization summarizes an asynchronous computation as immediate atomic effect. *Inductive sequentialization* establishes sequential reductions that captures every behavior of the original program up to reordering of coarse-grained commutative actions. A sequential reduction of a concurrent program is easy to reason about since it corresponds to a simple execution of the program in an idealized synchronous environment, where processes act in a fixed order and at the same speed.

Our approach is implemented the Civl verifier, which has been successfully used for the verification of several complex concurrent programs. In our methodology, the overall correctness of a program is established piecemeal by focusing on the invariant required for each refinement step separately. While the programmer does the creative work of specifying the chain of programs and the inductive invariant justifying each link in the chain, the tool automatically constructs the verification conditions underlying each refinement step.

Acknowledgments

This thesis would not have been possible without the generous help, support, and guidance of many people. I am grateful to all those who contributed, directly or indirectly, to the research presented in this thesis.

To my advisor Tom Henzinger, for guiding me through the twisty roads of graduate school. Early on, Tom gave me great freedom and encouragement to pursue problems I find interesting. Talking to Tom always revealed how convoluted and unrefined my ideas still were. His ability to, seemingly without effort, see the big picture, distill wild thoughts to their essence, and suggest directions to get unstuck never failed to impress me. Tom taught me to keep going after failing, and to celebrate successes; but not for too long, since research is never done.

To Shaz Qadeer, for spending endless hours in Skype calls, teaching me everything about concurrency, and much more. What started as a summer internship at Microsoft Research turned into an ongoing fruitful collaboration, which is the topic of this thesis. Shaz taught me how to be both enthusiastic and cynic about research, especially about our own work, and when to be perfectionist or pragmatic.

To Constantin Enea and Suha Mutluergil, for the fun stays in Paris, our collaboration, and the many valuable discussions from which I learned a lot.

To Krishnendu Chatterjee and Georg Weissenbacher, for serving on my thesis committee and valuable advice along the way.

To all group members at IST Austria over the years, for keeping the office a lively and nurturing environment, the insightful discussions and useful feedback, and the heated coffee-break debates.

To my "research friends", whom I always look forward to meet again at the next conference, workshop, or summer school. To my "civilian friends", for pulling me out of the research bubble from time to time.

Finally, to my family; my parents Angela and Friedrich, my sister Birgit, and above all, my wife Sandra, for their endless love, support, and patience, which makes it all worthwhile.

About the Author

Bernhard Kragl completed a BSc and MSc in Computer Sciences at the Vienna University of Technology, after which he joined Tom Henzinger's group at IST Austria as a PhD student. His research interests are in programming languages and formal methods for the development of reliable software systems. His thesis work focuses on reasoning techniques for concurrent and distributed systems that ease the verification burden for programmers.

List of Publications

This dissertation is a collection of the following publications.

- **Chapter 2**
  Bernhard Kragl and Shaz Qadeer. Layered concurrent programs. In *CAV*, 2018. doi:10.1007/978-3-319-96145-3_5
- **Chapter 3**
  Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for structured concurrent programs. In *CAV*, 2020. doi:10.1007/978-3-030-53288-8_14
- **Chapter 4**
  Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the asynchronous. In *CONCUR*, 2018. doi:10.4230/LIPIcs.CONCUR.2018.21
- **Chapter 5**
  Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *PLDI*, 2020. doi:10.1145/3385412.3385980

Table of Contents

List of Tables

List of Figures

1 Introduction

Concurrency is ubiquitous in today's computing landscape. Geographically distributed data centers rely on fault-tolerant distributed algorithms to ensure consistency; massively-parallel supercomputers empower large-scale scientific computing; mobile and web applications use event-driven asynchronous programming to achieve a fast and responsive user experience; controllers of embedded systems need to react to asynchronous events from the outside world; just to name a few examples. Despite their widespread use, designing and implementing concurrent programs remains a notoriously challenging and error-prone task. This is due to the sheer number of behaviors a concurrent program can exhibit. Concurrent operations can execute in many different orders, e.g., because of the unpredictability of scheduling threads on a multiprocessor or the delivery of messages in a distributed network. In short, there are many inherent sources of nondeterminism, and keeping track of all possible executions is an overwhelming mental task, especially as systems evolve. This thesis is concerned with formal methods to help designers and programmers build more reliable concurrent systems by offering techniques, methodologies, and tools that assist in the rigorous analysis and verification of these systems.

1.1 Specifications

Before we can even say *if* a system behaves correctly, we need to say *what it means* for the system to behave correctly; we need a *specification*. Specifications vary greatly in shape and form. For example, we could be interested in shallow generic properties (e.g., memory safety or data-race freedom), rich functional properties, temporal properties, hyperproperties (e.g., privacy properties), quantitative properties, etc. In this thesis we are interested in rich functional safety[1] properties expressed in mathematical logic. For example, in a system consisting of a finite set of processes P, the formula

$$\forall\, p_1, p_2 \in P.\ p_1.\text{hasDecided} \wedge p_2.\text{hasDecided} \implies p_1.\text{decidedValue} = p_2.\text{decidedValue}$$

states that the processes never reach an inconsistent decision. If any two processes p_1 and p_2 have reached a decision, then they must have decided on the same value.

We acknowledge that obtaining and maintaining good specifications is a hard problem by itself. We do not directly contribute to this problem. However, the refinement approach we follow (see below) facilitates a dual view of programs as both implementations and specifications. Intermediate programs derived by our approach can be understood as natural specifications of subcomponents.

[1] Intuitively, a safety property states that "something bad never happens" [84].

1.2 The Landscape of Verification Approaches

There are many different approaches to gain confidence in the correct functioning of a computer system, spanning a spectrum between cost (i.e., time, resources, expertise, etc.) and benefit (i.e., strength of provided guarantees). We provide a (necessarily rough and incomplete) overview to position our work. Detailed discussions and comparisons to related work can be found in every subsequent chapter (in particular, Section 2.1.1, Section 3.1.1, Section 4.8, and Section 5.6).

Testing. A seemingly straight-forward technique is to run a system and see if it behaves as expected. In principle, testing can even be done by treating the system as a black box. For example, the Jepsen project [3] tests real binaries on real clusters by injecting faults into the cluster.

> Since 2013, Jepsen has analyzed over two dozen databases, coordination services, and queues—and we've found replica divergence, data loss, stale reads, read skew, lock conflicts, and much more.
>
> `https://jepsen.io/analyses` (accessed June 4, 2020)

A recent line of work [28, 83, 96] provides a theoretical explanation for this effectiveness.

A major challenge in testing is capturing and optimizing *test coverage*, i.e., "how much" of a program has been exercised by tests. For realistic programs, the space of program behaviors is unbounded in many dimensions, e.g., the data domain, number of processes or threads, and length of executions. Thus, testing can never truly exercise all program behaviors. So when are we done testing, and did we gain enough confidence?

Concurrent programs are particularly challenging to test. First, local state of concurrent computations must be efficiently gathered, without perturbing the program under test (e.g., [46]). Similar problems arise in *monitoring* (aka *runtime verification)*. Second, tests for concurrent programs are inevitably nondeterministic. The same input can lead to many different behaviors because of uncontrollable timing, making certain bugs extremely hard to reproduce.[2] *Systematic testing* [90, 39] addresses this problem by gaining control over the scheduler and driving executions towards "interesting" behaviors.

Static analysis. Instead of actually executing a program, a static analyzer processes the source code of a program and produces a list of potential bugs. Technically, many static analyses are cast in the *abstract interpretation* framework [33], as fixpoint computations over suitable abstract domains. As an example, the Infer static analyzer supports data-race detection in concurrent programs [15]. Static analyzers typically have to trade off precision against annotation effort and efficiency.

Deductive verification. This thesis is concerned with *formal proofs*, rigorous mathematical arguments that show that *all* possible behaviors of a concurrent program are correct. This very strong guarantees come at the price of requiring higher expertise (e.g., familiarity with mathematical logic) and manual effort.

[2] The term *heisenbug*, a pun on the name of Werner Heisenberg that alludes to the observer effect of quantum mechanics, is often used in the context of concurrent programs.

Deductive verification usually employs a *program logic* to connect programs with assertions, and a *proof system* to derive valid conclusions about programs. Verifiers are typically either implemented on top of interactive theorem provers like Coq or Isabelle where proofs are written down explicitly (e.g., [116, 64, 104]), or tools that emit logical verification conditions based on user annotations (like invariants) that are checked using a theorem prover (e.g., [14, 77, 88]). Our work follows the latter approach.

Model checking. *Model checking* originated as a technique to systematically explore the state space of systems modeled as finite state graphs (i.e., Kripke structures) [30, 101]. The idea was to replace proof construction by algorithmic search. Pioneering tools include Spin [61] and SMV [85]. Approaches to address the combinatorial *state explosion problem* include compositional techniques [7, 8, 31] which decompose the verification problem into smaller subproblems, and partial-order methods [50] which reduce the state space by exploiting commutativity.

(Concurrent) software systems give rise to an infinite state space. For certain restricted classes of systems and properties, the verification problem stays decidable, e.g., based on the theory of well-structured transition systems [45]. *Abstraction refinement*-based verifiers like SLAM [13] and Blast [58] gradually refine a system abstraction (e.g., predicate abstraction [51]) by analyzing counterexamples. Threader [54] and Weaver [44] follow a similar approach for concurrent programs given as a parallel composition of a fixed number of threads. Parameterized verification involves challenging formulas that rely on quantification and set cardinality reasoning [111]. *Bounded model checking* compromises on completeness by restricting the set of explored executions. For example, context-switch bounding [99] or delay bounding [43] operates under the hypothesis that bugs can be effectively discovered by focusing on certain schedules [89, 38].

In general, there is no clear separation between the above areas, and many approaches borrow or combine ideas from different areas.

1.3 Deductive Verification of Concurrent Programs

In this section we review some fundamental notions of deductive verification, including *inductive invariants*, *reduction*, and *refinement*.

1.3.1 Inductive Invariants

Imagine a program to be modeled as a transition system $(Var, Init, Next, Safe)$, where Var is the set of program variables, $Init$ it the initial-state predicate over Var, $Next$ is the transition predicate over $Var \cup Var'$ that describes all program transitions from one state to the next, and $Safe$ is a safety predicate over Var. We want to ensure that no program execution can reach an unsafe state, i.e., for all sequences of state $s_1, \ldots, s_n$ with $s_1 \models Init$ and $s_i, s'_{i+1} \models Next$ for all pairs of consecutive states, $s_n \models Safe$ should hold. A fundamental approach to achieve this is by means of finding an *inductive invariant*—a predicate Inv over Var such that (1) $Init \implies Inv$, (2) $Inv \wedge Next \implies Inv'$, and (3) $Inv \implies Safe$. The invariant holds in the initial state, is preserved across transitions, and does not hold in any unsafe state. Thus, Inv overapproximates the set of reachable states and separates them from the unsafe states.

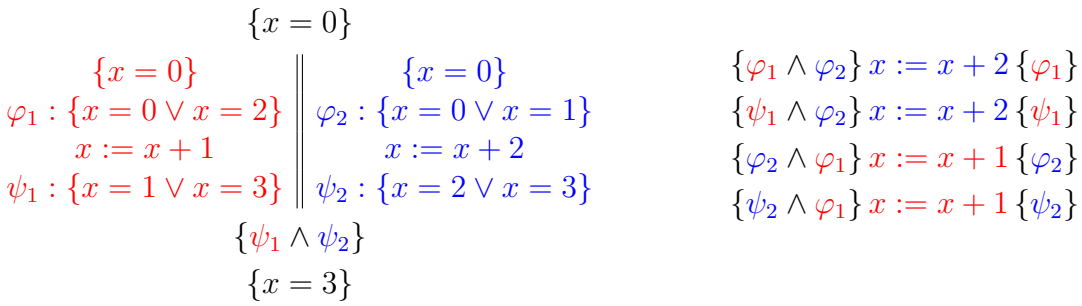


Figure 1.1: Owicki-Gries proof of a simple concurrent program.

While elegant and intuitive in theory, modeling programs as transition systems has severe limitations for verification in practice. The transition predicate *Next* has to encode the program's control flow, causing a case distinction over all possible program steps that could be enabled in any given state. The need for case distinction is then carried over to *Inv*, usually amplified and prone to complexity. Thus, transition relations are appropriate to assign semantics to programs, but not suitable as the object of interaction for program proofs.

In the world of sequential programs, Floyd [48] showed that program proofs can be constructed by annotating each control location of a program with an *inductive assertion*. Today we call the resulting proof system *Floyd-Hoare logic* [60], which forms the basis of modern program verifiers [14]. The judgments in Floyd-Hoare logic are often written as $\{\varphi\}\, c\, \{\psi\}$, with the intended meaning that if a command c executes from a state that satisfies the *precondition* φ and terminates, then the state after executing c satisfies the *postcondition* ψ. Can we have a proof rule for concurrent programs?

Owicki and Gries [95] offered the following rule for the parallel composition command.

$$\frac{\Psi_1 : \{\varphi_1\}\, c_1\, \{\psi_1\} \quad \Psi_2 : \{\varphi_2\}\, c_2\, \{\psi_2\} \quad \Psi_1, \Psi_2 \text{ interference-free}}{\{\varphi_1 \wedge \varphi_2\}\, c_1 \parallel c_2\, \{\psi_1 \wedge \psi_2\}}$$

To prove a property of the parallel composition of c_1 and c_2, both commands can be proved separately. However, the resulting proofs Ψ_1 and Ψ_2 must be *interference-free*, which means that no assertion used in Ψ_1 can be invalidated by a command in c_2, and similarly for Ψ_2 and c_1. For example, let us prove

$$\{x = 0\}\, x := x + 1 \parallel x := x + 2\, \{x = 3\},$$

where two threads increment the shared integer variable x by 1 and 2, respectively. Both assignments to x are assumed to be atomic. Figure 1.1 shows a proof outline. In the left thread, the precondition $x = 0$ is weakened to $x = 0 \vee x = 2$, resulting in the postcondition $x = 1 \vee x = 3$ after $x := x + 1$. In the right thread, $x = 0$ is weakened to $x = 0 \vee x = 1$, resulting in the postcondition $x = 2 \vee x = 3$ after $x := x + 2$. Together, ψ_1 and ψ_2 imply $x = 3$. The prescribed *non-interference* conditions are shown on the right of Figure 1.1. For example, $\varphi_1 \wedge \varphi_2$ implies $x = 0$, and thus after $x := x + 2$ we have $x = 2$, which still satisfies φ_1.

It might came as a surprise that the weakenings to φ_1 and φ_2 allowed us to recover the precise postcondition $x = 3$ from ψ_1 and ψ_2. Let us slightly modify the example such that both threads increment x by 1 and prove

$$\{x = 0\}\, x := x + 1 \parallel x := x + 1\, \{x = 2\}.$$

$$
\begin{array}{c}
\{x = 0\} \\
[done_1 := false; done_2 := false] \\
\{\neg done_1 \wedge x = (\text{if } done_2 \text{ then } 1 \text{ else } 0)\} \\
[x := x + 1; done_1 := true] \\
\{done_1 \wedge x = (\text{if } done_2 \text{ then } 2 \text{ else } 1)\}
\end{array}
\quad \Big\| \quad
\begin{array}{c}
\{\neg done_2 \wedge x = (\text{if } done_1 \text{ then } 1 \text{ else } 0)\} \\
[x := x + 1; done_2 := true] \\
\{done_2 \wedge x = (\text{if } done_1 \text{ then } 2 \text{ else } 1)\}
\end{array}
\\
\{x = 2\}
$$

Figure 1.2: Necessity of auxiliary variables in Owicki-Gries proofs.

Is it possible to adapt the proof in Figure 1.1? It is illustrative to try, but the answer is no. Just referring to x in assertions is not enough. The assertions will be either too weak to imply the postcondition $x = 2$, or too strong to be interference-free. The solution to this problem is to add *auxiliary variables* to the program, which can be updated together with regular program commands to store information that can then be used in assertions. These auxiliary variables are often called *ghost state*, because they can *see* (i.e,. depend on) the real program variables, but not vice versa. Figure 1.2 shows a proof outline that uses the auxiliary Boolean variable $done_1$ to remember if the left thread is before or after its update to x, and a similar auxiliary variable $done_2$ for the right thread. These variables are updated from *false* to *true* together (i.e., atomically) with the respective increment to x, indicated by $[\ldots]$. The reader can confirm that the assertions are valid, interference-free, and imply the postcondition $x = 2$.

By flooding a program with enough auxiliary variables—essentially capturing the program counter of every thread—the Owicki-Gries rule is complete [94]. That is, relative to a strong enough assertion language, any valid Floyd-Hoare judgment can be proved. This is good news in theory, but bad news in practice. As we see in Figure 1.2, the assertions contain excessive case distinction, which becomes a major issue in more sophisticated programs. Additionally, the assertions in one thread need to talk about the local state of the other thread. Proof rules like rely-guarantee [63] aim for so called *thread-modular* proofs, but such proofs do not always exist.

Having identified the invention of inductive invariants as the main challenge for verifying concurrent programs, there are two ways forward.

In the first direction, we can work on automating invariant discovery, e.g., [54, 44]. A common theme in many works on automation is the use of logical tools, like interpolation, to find a suitable assertion language in which to search for proofs. Despite gradual progress, the state-of-the-art in invariant inference for concurrent programs does not yet scale to rich correctness properties of realistic systems.

In the second direction—which is the main topic of this thesis—we can work on techniques and methodologies that aid interactive proof construction. Here, the focus is on providing ways to decompose and structure a proof, splitting the mental activity into manageable pieces, and ultimately making for a more pleasant and productive verification experience.

1.3.2 Shared-Memory vs. Message-Passing Concurrency

Concurrent programming is often separated into two camps: *shared-memory concurrency*, concerned with concurrent data structures accessed by multiple threads of a multiprocessors, and *message-passing concurrency*, concerned with distributed processes that communicate by exchanging messages via a network. Of course, a logical shared memory can actually be distributed, and message passing can be performed on the same node. The point is that both paradigms provide different structuring mechanisms. In particular, the reduction of shared state in message passing is embraced by languages like Erlang and Go. However, in a formal proof the state of communication channels remains globally shared and subject to interference.

To reason about message-passing programs, the proof rule of Owicki and Gries was adapted to the model of *communicating sequential processes (CSPs)* [79, 10, 9], and later to asynchronous message passing [102], flush channels [24], and causally-ordered delivery [106]. Essentially, these proof systems bake in the semantics of communication primitives (i.e., send/receive commands) and adequately specialize the necessary noninterference obligations.

In our work, we do not treat message-passing primitives specially. They are modeled as regular shared-state operations. The benefit is that (1) implementations can mix both message-passing and shared-memory programming, and (2) proofs of the message-passing part can be significantly simplified by not writing invariants directly on the low-level network state, but first transforming the state representation to a more abstract one.

1.3.3 Reduction

Concurrent programs are usually not written for the sake of concurrency. Despite many possible interleavings, we expect many interleavings to be "equivalent" in the sense of producing the same observable effect. For example, two typical applications of concurrency are (1) speeding up a computational task, and (2) replicating state to deal with faults. In both cases, concurrency is not needed to produce a desired behavior. Instead, concurrency has to be suitably controlled (via synchronization) to only produce "useful" behaviors. Indeed, most consistency conditions for concurrent systems, like *linearizability* [59], define the admissible concurrent interleavings in terms of an underlying sequential specification.

Recall the example from Figure 1.1. This example only has two possible interleavings, $x := x + 1; x := x + 2$ and $x := x + 2; x := x + 1$. Do we really need to reason about both of them, and—essentially—let the assertions enumerate the possible intermediate states where $x = 2$ and $x = 1$, respectively? Since addition is *commutative*, the execution order in this example does not matter. Thus, we can pick either one of the interleavings, prove it, and argue that the other interleaving is "equivalent". In general, to exploit commutativity arguments to reduce the reasoning task to a subset of interleavings, we need to address three problems:

1. How to establish the commutativity of operations?
2. How to specify which subset of interleavings to consider?
3. How to justify that all other interleavings are implicitly covered?

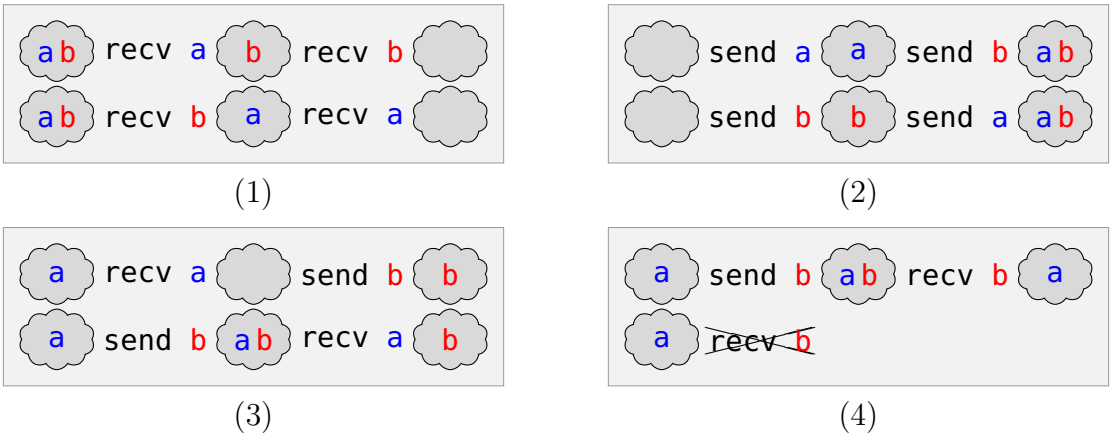


Figure 1.3: Over a bag channel, `receive` is a right mover and `send` is a left mover, but not vice versa. Clouds represent the (unordered) content of a channel, and a and b are values received from and sent to the channel by different threads. (1) receive a moves right of receive b; (2) send b moves left of send a; (3) send b moves left of receive a (receive a moves right of send b); (4) send b does not move right of receive b (receive b does not move left of send b).

In a seminal paper, Lipton [80] introduced the notion of *right movers* and *left movers*. In Lipton's original definition, an operation is a right mover, if in every execution of a given program, the operation can be commuted to the right (i.e., later in time) of any operation performed by another thread, without changing the value of any program variable in the final state. Analogously, an operation is a left mover, if it can be commuted to the left (i.e., earlier in time) of any operation performed by another thread. He noted [80]:

> Essentially, a right mover is a statement that performs a "seize" while a left mover is a statement that performs a "release" of a "resource."

Of a somewhat different flavor, we illustrate in Figure 1.3 that over a channel with bag semantics—modeling a network where messages can get reordered—the send operation is a left mover and the receive operation is a right mover.

Lipton's reduction replaces a sequence $c_1; \ldots; c_n$ in the program with the indivisible statement $[c_1; \ldots; c_n]$, provided, for some i, $c_1, \ldots, c_{i-1}$ are right movers and $c_{i+1}, \ldots, c_n$ are left movers (c_i is unconstrained). Lipton's original study focused on classifying the "wait" primitive $P(a) = [\text{assume } a > 0; a := a - 1]$ and the "signal" primitive $V(a) = [a := a + 1]$ over *semaphore* variables a as right and left movers, respectively, and showing that a reduced program halts if and only if the original program halts.

In [47], the idea of movers was picked up to define a type system that proves the atomicity of methods in a concurrent object-oriented programming language. This type system is based on annotations that declare if reads and writes to fields are protected by locks. The work on the QED verifier [41] introduced the notion of *gated atomic actions*, which describe an atomic operation not only as a set of transitions (i.e., possible state updates), but additionally contain a *gate* that states a condition that must be satisfied when the operation executes (like an assertion). Gates can capture contextual information, which makes it useful to establish commutativity properties of atomic action by looking at them in isolation. Concretely, instead of a priori classifying *specific* operations as movers, *mover types* are established by a pairwise logical commutativity analysis over the

set of gated atomic actions of a program. Using gated atomic actions as the technical substrate, Elmas et al. [41] identified *abstraction* as a symbiotic counterpart to reduction, which enables iterative program simplification. Abstracting a gated atomic action (i.e., strengthening its gate or weakening its transition relation) can strengthen its mover type, making reduction applicable. Reduction forms new coarser-grained atomic actions, which can be abstracted to make reduction applicable again. For example, a plain write operation is not commutative. However, stating in the gate that the write operation is executed while holding a lock makes it a mover.

Reduction is in the service of invariant discovery, because inductive invariants for a reduced program can be much simpler than for the original program, more than compensating for the reduction argument. In this thesis we present new reduction-based proof rules for asynchronous programs and distributed systems.

Reduction bears resemblance to the notion of *robustness* [17, 20, 22]. However, instead of simplifying a program, these robustness results interpret the same program under two different semantics. Concretely, under certain conditions a program running on a weaker semantics can be verified assuming a stronger semantics, or alternatively, a property of a program verified under a strong semantics is preserved when the program runs under a weaker semantics.

1.3.4 Refinement

Program development by *stepwise refinement* [117] is the idea of developing a program by starting with a high-level abstract specification, and gradually refining it down to a concrete low-level implementation. Alternatively, a low-level implementation can be gradually abstracted to prove a high-level specification. Or both top-down and bottom-up design are combined. Formal verification techniques based on stepwise refinement have long been advocated, in theory, for the construction of verified concurrent programs (e.g., [11, 103, 37]).

Going back to transition system models, consider a concrete transition system C and an abstract transition system A. A standard way to show that C correctly implements A is the specification of a *refinement mapping* from the concrete states of C to the abstract states of A.[3] In a so called *forward-simulation* argument, the refinement mapping is shown to map every concrete step of C to an abstract step of A, thus mapping every concrete behavior to some allowed abstract behavior. In general, the existence of refinement mappings relies on two types of auxiliary variables [4]: *history variables* that "remember" something about the past of an execution, and *prophecy variables* that "predict" the future of an execution. Lynch and Vaandrager [82] present further simulation techniques.

The work in this thesis is in the context of the Civl verifier, originally described by Hawblitzl et al. [56]. As a refinement-based verifier, Civl advocates the verification of concurrent programs across multiple *layers* of refinement. A core, distinguishing design feature of Civl is that each layer in a refinement proof remains a *structured program*, i.e., a program with procedures, imperative control flow, structured parallelism, and asynchronous concurrency. This is in contrast to previous refinement-based verifiers, like TLA+ [76] or Event-B [5], which use a representation of concurrent systems as flat transition system. The benefits of using a structured program representation include

[3] Given the directionality, it would perhaps be more appropriate to talk about an *abstraction mapping*.

(1) naturally bridging the gap to real implementations, and (2) preserving the structure that is present in program syntax. In Civl, proof steps correspond to a small, simplifying program transformation, which make atomic actions more and more coarse-grained and abstract, and the program less and less concurrent. The invariants necessary to justify each step are comparatively simple, and the overall decomposition makes proofs easier to construct and reuse. The implementation of Civl translates the verification problem into a set of verification conditions that are automatically discharged by a theorem prover.

1.4 Contributions and Outline

This dissertation contributes new techniques and methodologies to simplify the construction of formal correctness proof of concurrent programs. Broadly speaking, this contributions fall into two classes. First, Chapter 2 and Chapter 3 present the foundations of building a verifier using a structured-program representation. Second, Chapter 4 and Chapter 5 present new reduction-based proof rules to enable simpler proofs of asynchronous programs.

Each chapter corresponds to a self-contained conference paper, which outlines the challenges of a particular piece in our framework, presents a technical development, describes its implementation, demonstrates its usefulness on case studies, and compares to related work. To allow each chapter to focus on a specific aspect, the notation and formalization is not unified across chapters.

In summary, this dissertation makes the following main contributions.

Layered concurrent programs. We present *layered concurrent programs* (Chapter 2), a compact formalism to represent all programs in a multi-layer refinement proof as one syntactic unit, avoiding excessive duplication of program parts that remain unchanged across refinement steps.

Yield invariants. We introduce *yield invariants* (Chapter 3), a new specification idiom that encapsulates inductive invariants as named, parameterized, and reusable entities, which can be invoked specific to a call site (similar to procedures).

Refinement along program structure. We present a powerful refinement rule (Chapter 3), that decomposes the verification problem over structured concurrent programs into modular verification conditions. This proof rule integrates yield invariants with a system of *linear permissions* to enhance local reasoning. It supports both global and local *variable introduction* and *hiding*, and modular abstraction of recursive procedures.

Synchronization. We present a reduction principle called *synchronization* (Chapter 4), which converts asynchronous calls into synchronous calls. The verification task is simplified, because instead of reasoning about an effect to occur asynchronously at a later time in an execution, we can reason about the effect to happen immediately.

Inductive sequentialization. We present a reduction principle called *inductive synchronization* (Chapter 5), which reduces reasoning about distributed systems to a single representative execution. We show that even complicated protocols like Paxos admit simple sequential reductions.

Pending asyncs. We extend gated atomic actions with the notion of *pending asyncs*. Using pending asyncs, gated atomic actions do not only represent the effect of updating shared global variables, but also the effect of creating asynchronous computation. Pending asyncs were first introduced in the work on synchronization. The refinement rule in Chapter 3 describes the "creation" of pending asyncs, while Chapter 4 and Chapter 5 describe techniques to "eliminate" pending asyncs form atomic actions.

CIVL **verifier.** All techniques presented in this dissertation are implemented in the CIVL verifier, which is publicly available as part of Boogie [2, 1]. In particular, the theory in Chapter 2 and Chapter 3 was the basis of a new design and implementation of CIVL, and the techniques in Chapter 4 and Chapter 5 are available as new proof tactics that integrate symbiotically with the already existing tactics. Our implementation was used to show the applicability and usefulness of our methodology in multiple case studies.

2 Layered Concurrent Programs

Abstract. We present layered concurrent programs, a compact and expressive notation for specifying refinement proofs of concurrent programs. A layered concurrent program specifies a sequence of connected concurrent programs, from most concrete to most abstract, such that common parts of different programs are written exactly once. These programs are expressed in the ordinary syntax of imperative concurrent programs using gated atomic actions, sequencing, choice, and (recursive) procedure calls. Each concurrent program is automatically extracted from the layered program. We reduce refinement to the safety of a sequence of concurrent checker programs, one each to justify the connection between every two consecutive concurrent programs. These checker programs are also automatically extracted from the layered program. Layered concurrent programs have been implemented in the Civl verifier which has been successfully used for the verification of several complex concurrent programs.

2.1 Introduction

Refinement is an approach to program correctness in which a program is expressed at multiple levels of abstraction. For example, we could have a sequence of programs $\mathcal{P}_1, \ldots, \mathcal{P}_h, \mathcal{P}_{h+1}$ where $\mathcal{P}_1$ is the most concrete and the $\mathcal{P}_{h+1}$ is the most abstract. Program $\mathcal{P}_1$ can be compiled and executed efficiently, $\mathcal{P}_{h+1}$ is obviously correct, and the correctness of $\mathcal{P}_i$ is guaranteed by the correctness of $\mathcal{P}_{i+1}$ for all $i \in [1, h]$. These three properties together ensure that $\mathcal{P}_1$ is both efficient and correct. To use the refinement approach, the programmer must come up with each version $\mathcal{P}_i$ of the program and a proof that the correctness of $\mathcal{P}_{i+1}$ implies the correctness of $\mathcal{P}_i$. This proof typically establishes a connection from every behavior of $\mathcal{P}_i$ to some behavior of $\mathcal{P}_{i+1}$.

Refinement is an attractive approach to the verified construction of complex programs for a number of reasons. First, instead of constructing a single monolithic proof of $\mathcal{P}_1$, the programmer constructs a collection of localized proofs establishing the connection between $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ for each $i \in [1, h]$. Each localized proof is considerably simpler than the overall proof because it only needs to reason about the (relatively small) difference between adjacent programs. Second, different localized proofs can be performed using different reasoning methods, e.g., interactive deduction, automated testing, or even informal reasoning. Finally, refinement naturally supports a bidirectional approach to correctness—

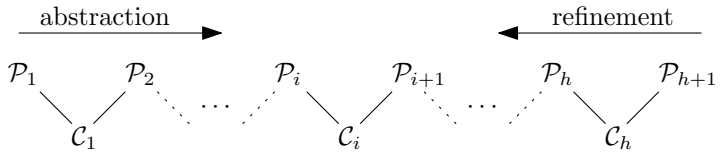


Figure 2.1: Concurrent programs $\mathcal{P}_i$ and connecting checker programs $\mathcal{C}_i$ represented by a layered concurrent program $\mathcal{LP}$.

bottom-up verification of a concrete program via successive abstraction or top-down derivation from an abstract program via successive concretization.

This paper explores the use of refinement to reason about concurrent programs. Most refinement-oriented approaches model a concurrent program as a flat transition system, a representation that is useful for abstract programs but becomes increasingly cumbersome for a concrete implementation. To realize the goal of verified construction of efficient and implementable concurrent programs, we must be able to uniformly and compactly represent both highly-detailed and highly-abstract concurrent programs. This paper introduces layered concurrent programs as such a representation.

A layered concurrent program $\mathcal{LP}$ represents a sequence $\mathcal{P}_1, \ldots, \mathcal{P}_h, \mathcal{P}_{h+1}$ of concurrent programs such that common parts of different programs are written exactly once. These programs are expressed not as flat transition systems but in the ordinary syntax of imperative concurrent programs using gated atomic actions [41], sequencing, choice, and (recursive) procedure calls. Our programming language is accompanied by a type system that allows each $\mathcal{P}_i$ to be automatically extracted from $\mathcal{LP}$. Finally, refinement between $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ is encoded as the safety of a checker program $\mathcal{C}_i$ which is also automatically extracted from $\mathcal{LP}$. Thus, the verification of $\mathcal{P}_1$ is split into the verification of h concurrent checker programs $\mathcal{C}_1, \ldots, \mathcal{C}_h$ such that $\mathcal{C}_i$ connects $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ (Figure 2.1).

We highlight two crucial aspects of our approach. First, while the programs $\mathcal{P}_i$ have an interleaved (i.e., preemptive) semantics, we verify the checker programs $\mathcal{C}_i$ under a cooperative semantics in which preemptions occur only at procedure calls. Our type system [47] based on the theory of right and left movers [80] ensures that the cooperative behaviors of $\mathcal{C}_i$ cover all preemptive behaviors of $\mathcal{P}_i$. Second, establishing the safety of checker programs is not tied to any particular verification technique. Any applicable technique can be used. In particular, different layers can be verified using different techniques, allowing for great flexibility in verification options.

2.1.1 Related Work

This paper formalizes, clarifies, and extends the most important aspect of the design of Civl [56], a deductive verifier for layered concurrent programs. Hawblitzel et al. [57] present a partial explanation of Civl by formalizing the connection between two concurrent programs as sound program transformations. In this paper, we provide the first formal account for layered concurrent programs to represent all concurrent programs in a multi-layered refinement proof, thereby establishing a new foundation for the verified construction of concurrent programs.

Civl is the successor to the Qed [41] verifier which combined a type system for mover types with logical reasoning based on verification conditions. Qed enabled the specification of a layered proof but required each layer to be expressed in a separate file leading to code

duplication. Layered programs reduce redundant work in a layered proof by enabling each piece of code to be written exactly once. QED also introduced the idea of abstracting an atomic action to enable attaching a stronger mover type to it. This idea is incorporated naturally in layered programs by allowing a concrete atomic action to be wrapped in a procedure whose specification is a more abstract atomic action with a more precise mover type.

Event-B [5] is a modeling language that supports refinement of systems expressed as interleaved composition of events, each specified as a top-level transition relation. Verification of Event-B specifications is supported by the Rodin [6] toolset which has been used to model and verify several systems of industrial significance. TLA+ [76] also specifies systems as a flat transition system, enables refinement proofs, and is more general because it supports liveness specifications. Our approach to refinement is different from Event-B and TLA+ for several reasons. First, Event-B and TLA+ model different versions of the program as separate flat transition systems whereas our work models them as different layers of a single layered concurrent program, exploiting the standard structuring mechanisms of imperative programs. Second, Event-B and TLA+ connect the concrete program to the abstract program via an explicitly specified refinement mapping. Thus, the guarantee provided by the refinement proof is contingent upon trusting both the abstract program and the refinement mapping. In our approach, once the abstract program is proved to be free of failures, the trusted part of the specification is confined to the gates of atomic actions in the concrete program. Furthermore, the programmer never explicitly specifies a refinement mapping and is only engaged in proving the correctness of checker programs.

The methodology of refinement mappings has been used for compositional verification of hardware designs [86, 87]. The focus in this work is to decompose a large refinement proof connecting two versions of a hardware design into a collection of smaller proofs. A variety of techniques including compositional reasoning (converting a large problem to several small problems) and customized abstractions (for converting infinite-state to finite-state problems) are used to create small and finite-state verification problems for a model checker. This work is mostly orthogonal to our contribution of layered programs. Rather, it could be considered an approach to decompose the verification of each (potentially large) checker program encoded by a layered concurrent program.

2.2 Concurrent Programs

In this section we introduce a concurrent programming language. The syntax of our programming language is summarized in Figure 2.2.

Preliminaries. Let *Val* be a set of *values* containing the Booleans. The set of *variables* *Var* is partitioned into *global variables* *GVar* and *local variables* *LVar*. A *store* σ is a mapping from variables to values, a *gate* ρ is a set of stores, and a *transition relation* τ is a binary relation between stores.

Atomic actions. A fundamental notion in our approach is that of an atomic action. An atomic action captures an indivisible operation on the program state together with its precondition, providing a universal representation for both low-level machine operations

$$
\begin{array}{rcll}
 & & \mathit{Val} & \supseteq \ \mathbb{B} \\
v & \in & \mathit{Var} & = \ \mathit{GVar} \cup \mathit{LVar} \\
I, O, L & \subseteq & \mathit{LVar} & \\
\sigma & \in & \mathit{Store} & = \ \mathit{Var} \to \mathit{Val} \\
\rho & \in & \mathit{Gate} & = \ 2^{\mathit{Store}} \\
\tau & \in & \mathit{Trans} & = \ 2^{\mathit{Store} \times \mathit{Store}} \\
A & \in & \mathit{Action} & \\
P, Q & \in & \mathit{Proc} & \\
\iota, o & \in & \mathit{IOMap} & = \ \mathit{LVar} \rightharpoonup \mathit{LVar}
\end{array}
\qquad
\begin{array}{rcl}
gs & \in & 2^{\mathit{GVar}} \\
as & \in & A \mapsto (I, O, \rho, \tau) \\
ps & \in & P \mapsto (I, O, L, s) \\
m & \in & \mathit{Proc} \cup \mathit{Action} \\
\mathcal{I} & \in & 2^{\mathit{Store}}
\end{array}
$$

$$\mathcal{P} \in \mathit{Prog} ::= (gs, as, ps, m, \mathcal{I})$$

$$s \in \mathit{Stmt} ::= \texttt{skip} \mid s\,;s \mid \texttt{if}\ \rho\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{pcall}\ \overline{(A, \iota, o)}\ \overline{(P, \iota, o)}\ \overline{(A, \iota, o)}$$

Figure 2.2: Concurrent programs

(e.g., reading a variable from memory) and high-level abstractions (e.g., atomic procedure summaries). Most importantly for reasoning purposes, our programming language confines all accesses to global variables to atomic actions. Formally, an *atomic action* is a tuple (I, O, ρ, τ). The semantics of an atomic action in an execution is to first evaluate the gate ρ in the current state. If the gate evaluates to *false* the execution *fails*, otherwise the program state is updated according to the transition relation τ. *Input variables* in I can be read by ρ and τ, and *output variables* in O can be written by τ.

Remark 1. Atomic actions subsume many standard statements. In particular, (nondeterministic) assignments, assertions, and assumptions. The following table shows some examples for programs over variables x and y.

command	ρ	τ
$x := x + y$	*true*	$x' = x + y \wedge y' = y$
havoc x	*true*	$y' = y$
assert $x < y$	$x < y$	$x' = x \wedge y' = y$
assume $x < y$	*true*	$x < y \wedge x' = x \wedge y' = y$

Procedures. A *procedure* is a tuple (I, O, L, s) where I, O, L are the *input*, *output*, and *local variables* of the procedure, and s is a *statement* composed from skip, sequencing, if, and parallel call statements. Since only atomic actions can refer to global variables, the variables accessed in if conditions are restricted to the inputs, outputs, and locals of the enclosing procedure. The meaning of skip, sequencing, and if is as expected and we focus on parallel calls.

Pcalls. A *parallel call* (*pcall*, for short) $\texttt{pcall}\ \overline{(A, \iota, o)}\ \overline{(P, \iota, o)}\ \overline{(A, \iota, o)}$ consists of a sequence of invocations of atomic actions and procedures. We refer to the invocations as the *arms* of the pcall. In particular (A, ι, o) is an *atomic-action arm* and (P, ι, o) is a *procedure arm*. An atomic-action arm executes the called atomic action, and a procedure arm creates a child thread that executes the statement of the called procedure. The parent thread is blocked until all arms of the pcall finish. In the standard semantics the order of arms does not matter, but our verification technique will allow us to consider

the atomic action arms before and after the procedure arms to execute in the specified order. Parameter passing is expressed using partial mappings ι, o between local variables; ι maps formal inputs of the callee to actual inputs of the caller, and o maps actual outputs of the caller to formal outputs of the callee. Since we do not want to introduce races on local variables, the outputs of all arms must be disjoint and the output of one arm cannot be an input to another arm. Finally, notice that our general notion of a pcall subsumes sequential statements (single atomic-action arm), synchronous procedure calls (single procedure arm), and unbounded thread creation (recursive procedure arm).

Concurrent programs. A *concurrent program* $\mathcal{P}$ is a tuple $(gs, as, ps, m, \mathcal{I})$, where gs is a finite set of global variables used by the program, as is a finite mapping from *action names* A to atomic actions, ps is a finite mapping from *procedure names* P to procedures, m is either a procedure or action name that denotes the entry point for program executions, and $\mathcal{I}$ is a set of initial stores. For convenience we will liberally use action and procedure names to refer to the corresponding atomic actions and procedures.

Semantics. Let $\mathcal{P} = (gs, as, ps, m, \mathcal{I})$ be a fixed concurrent program. A *state* consists of a global store assigning values to the global variables and a pool of *threads*, each consisting of a local store assigning values to local variables and a statement that remains to be executed. An *execution* is a sequence of states, where from each state to the next some thread is selected to execute one step. Every step that switches the executing thread is called a *preemption* (also called a context switch). We distinguish between two semantics that differ in (1) preemption points, and (2) the order of executing the arms of a pcall.

In *preemptive semantics*, a preemption is allowed anywhere and the arms of a pcall are arbitrarily interleaved. In *cooperative semantics*, a preemption is allowed only at the call and return of a procedure, and the arms of a pcall are executed as follows. First, the leading atomic-action arms are executed from left to right without preemption, then all procedure arms are executed arbitrarily interleaved, and finally the trailing atomic-action arms are executed, again from left to right without preemption. In other words, a preemption is only allowed when a procedure arm of a pcall creates a new thread and when a thread terminates.

For $\mathcal{P}$ we only consider executions that start with a single thread that execute m from a store in $\mathcal{I}$. $\mathcal{P}$ is called *safe* if there is no failing execution, i.e., an execution that executes an atomic action whose gate evaluates to *false*. We write $Safe(\mathcal{P})$ if $\mathcal{P}$ is safe under preemptive semantics, and $CSafe(\mathcal{P})$ if $\mathcal{P}$ is safe under cooperative semantics.

2.2.1 Running Example

In this section, we introduce a sequence of three concurrent programs (Figure 2.3) to illustrate features of our concurrent programming language and the layered approach to program correctness. Consider the program $\mathcal{P}_1^{lock}$ in Figure 2.3(a). The program uses a single global Boolean variable `b` which is accessed by the two atomic actions `CAS` and `RESET`. The compare-and-swap action `CAS` atomically reads the current value of `b` and either sets `b` from *false* to *true* and returns *true*, or leaves `b` *true* and returns *false*. The `RESET` action sets `b` to *false* and has a gate (represented as an assertion) that states that the action must only be called when `b` is *true*. Using these actions, the procedures `Enter` and `Leave` implement a spinlock as follows. `Enter` calls the `CAS` action and retries

(a) $\mathcal{P}_1^{lock}$

```
var b : bool


proc Main()
  if (*)
    pcall Worker(), Main()

proc Worker()

  pcall Alloc()
  pcall Enter()
  pcall Leave()

proc Alloc() : ()
  skip


proc Enter()
  var success : bool
  pcall success := CAS()
  if (success)
    skip
  else
    pcall Enter()

proc Leave()
  pcall RESET()
  skip

atomic CAS() : (success : bool)
  if (b) success := false
  else   success, b := true, true

atomic RESET()
  assert b
  b := false
```

(b) $\mathcal{P}_2^{lock}$

```
var lock : int
var linear slots : set<int>

proc Main()
  if (*)
    pcall Worker(), Main()

proc Worker()
  var linear tid : int
  pcall tid := ALLOC()
  pcall ACQUIRE(tid)
  pcall RELEASE(tid)

right ALLOC() : (linear tid : int)
  assume tid != 0 && tid in slots
  slots := slots - tid

right ACQUIRE(linear tid : int)
  assert tid != 0
  assume lock == 0
  lock := tid




left RELEASE(linear tid : int)
  assert tid != 0 && lock == tid
  lock := 0
```

(c) $\mathcal{P}_3^{lock}$

```


both SKIP()
  skip
```

Figure 2.3: Lock example

(through recursion on itself) until it succeeds to set `b` from *false* to *true*. `Leave` just calls the `RESET` action which sets `b` back to *false* and thus allows another thread executing `Enter` to stop spinning. Finally, the procedures `Main` and `Worker` serve as a simple client. `Main` uses a pcall inside a nondeterministic if statement to create an unbounded number of concurrent worker threads, which just acquire the lock by calling `Enter` and then release the lock again by calling `Leave`. The call to the empty procedure `Alloc` is an artifact of our extraction from a layered concurrent program and can be removed as an optimization.

Proving $\mathcal{P}_1^{lock}$ safe amounts to showing that `RESET` is never called with `b` set to *false*, which expresses that $\mathcal{P}_1^{lock}$ follows a locking discipline of releasing only previously acquired locks. Doing this proof directly on $\mathcal{P}_1^{lock}$ has two drawbacks. First, the proof must relate the possible values of `b` with the program counters of all running threads. In general, this approach requires sound introduction of ghost code and results in complicated case distinctions in program invariants. Second, the proof is not reusable across different lock implementations. The correctness of the client does not specifically depend on using a spinlock over a Boolean variable, and thus the proof should not as well. We show how our refinement-based approach addresses both problems.

Program $\mathcal{P}_2^{lock}$ in Figure 2.3(b) is an abstraction of $\mathcal{P}_1^{lock}$ that introduces an abstract lock specification. The global variable `b` is replaced by `lock` which ranges over integer thread identifiers (`0` is a dedicated value indicating that the lock is available). The procedures `Alloc`, `Enter` and `Leave` are replaced by the atomic actions `ALLOC`, `ACQUIRE` and `RELEASE`, respectively. `ALLOC` allocates unique and non-zero thread identifiers using a

set of integers `slot` to store the identifiers not allocated so far. `ACQUIRE` blocks executions where the lock is not available (`assume lock == 0`) and sets `lock` to the identifier of the acquiring thread. `RELEASE` asserts that the releasing thread holds the lock and sets `lock` to `0`. Thus, the connection between $\mathcal{P}_1^{lock}$ and $\mathcal{P}_2^{lock}$ is given by the invariant `b <==> lock != 0` which justifies that `Enter` refines `ACQUIRE` and `Leave` refines `RELEASE`. The potential safety violation in $\mathcal{P}_1^{lock}$ by the gate of `RESET` is preserved in $\mathcal{P}_2^{lock}$ by the gate of `RELEASE`. In fact, the safety of $\mathcal{P}_2^{lock}$ expresses the stronger locking discipline that the lock can only be released by the thread that acquired it.

Reasoning in terms of `ACQUIRE` and `RELEASE` instead of `Enter` and `Leave` is more general, but it is also simpler! Figure 2.3(b) declares atomic actions with a *mover type* [47], `right` for *right mover*, and `left` for *left mover*. A right mover executed by a thread commutes to the right of any action executed by a different thread. Similarly, a left mover executed by thread commutes to the left of any action executed by a different thread. A sequence of right movers followed by at most one non-mover followed by a sequence of left movers in a thread can be considered atomic [80]. The reason is that any interleaved execution can be rearranged (by commuting atomic actions), such that these actions execute consecutively. For $\mathcal{P}_2^{lock}$ this means that `Worker` is atomic and thus the gate of `RELEASE` can be discharged by pure sequential reasoning; `ALLOC` guarantees `tid != 0` and after executing `ACQUIRE` we have `lock == tid`. As a result, we finally obtain that the atomic action `SKIP` in $\mathcal{P}_3^{lock}$ (Figure 2.3(c)) is a sound abstraction of procedure `Main` in $\mathcal{P}_2^{lock}$. Hence, we showed that program $\mathcal{P}_1^{lock}$ is safe by soundly abstracting it to $\mathcal{P}_3^{lock}$, a program that is trivially safe.

The correctness of `right` and `left` annotations on `ACQUIRE` and `RELEASE`, respectively, depends on pair-wise commutativity checks among atomic actions in $\mathcal{P}_2^{lock}$. These commutativity checks will fail unless we exploit the fact that every thread identifier allocated by `Worker` using the `ALLOC` action is unique. For instance, to show that `ACQUIRE` executed by a thread commutes to the right of `RELEASE` executed by a different thread, it must be known that the parameters `tid` to these actions are distinct from each other. The *`linear`* annotation on the local variables named `tid` and the global variable `slots` (which is a set of integers) is used to communicate this information.

The overall invariant encoded by the *`linear`* annotation is that the set of values stored in `slots` and in local linear variables of active stack frames across all threads are pairwise disjoint. This invariant is guaranteed by a combination of a linear type system [113] and logical reasoning on the code of all atomic actions. The linear type system ensures using a flow analysis that a value stored in a linear variable in an active stack frame is not copied into another linear variable via an assignment. Each atomic action must ensure that its state update preserves the disjointness invariant for linear variables. For actions `ACQUIRE` and `RELEASE`, which do not modify any linear variables, this reasoning is trivial. However, action `ALLOC` modifies `slots` and updates the linear output parameter `tid`. Its correctness depends on the (semantic) fact that the value put into `tid` is removed from `slots`; this reasoning can be done using automated theorem provers.

2.3 Layered Concurrent Programs

A layered concurrent program represents a sequence of concurrent programs that are connected to each other. That is, the programs derived from a layered concurrent program share syntactic structure, but differ in the granularity of the atomic actions and the set

of variables they are expressed over. In a layered concurrent program, we associate layer numbers and layer ranges with variables (both global and local), atomic actions, and procedures. These layer numbers control the introduction and hiding of program variables and the summarization of compound operations into atomic actions, and thus provide the scaffolding of a refinement relation. Concretely, this section shows how the concurrent programs $\mathcal{P}_1^{lock}$, $\mathcal{P}_2^{lock}$, and $\mathcal{P}_3^{lock}$ (Figure 2.3) and their connections can all be expressed in a single layered concurrent program. In Section 2.4, we discuss how to check refinement between the successive concurrent programs encoded in a layered concurrent program.

Syntax. The syntax of layered concurrent programs is summarized in Figure 2.4. Let $\mathbb{N}$ be the set of non-negative integers and $\mathbb{I}$ the set of nonempty *intervals* $[a, b]$. We refer to integers as *layer numbers* and intervals as *layer ranges*. A *layered concurrent program* $\mathcal{LP}$ is a tuple $(GS, AS, IS, PS, m, \mathcal{I})$ which, similarly to concurrent programs, consists of global variables, atomic actions, and procedures, with the following differences.

1. GS maps global variables to layer ranges. For $GS(v) = [a, b]$ we say that v is introduced at layer a and available up to layer b.

2. AS assigns a layer range r to atomic actions denoting the layers at which an action exists.

3. IS (with a disjoint domain from AS) distinguishes a special type of atomic actions called *introduction actions*. Introduction actions have a single layer number n and are responsible for assigning meaning to the variables introduced at layer n. Correspondingly, statements in layered concurrent programs are extended with an `icall` statement for calling introduction actions.

4. PS assigns a layer number n, a layer number mapping for local variables ns, and an atomic action A to procedures. We call n the *disappearing layer* and A the *refined atomic action*. For every local variable v, $ns(v)$ is the *introduction layer* of v.

   The $\texttt{pcall}_\alpha$ statement in a layered concurrent program differs from the `pcall` statement in concurrent programs in two ways. First, it can only have procedure arms. Second, it has a parameter α which is either ε (*unannotated pcall*) or the index of one of its arms (*annotated pcall*). We usually omit writing ε in unannotated pcalls.

5. m is a procedure name.

The *top layer* h of a layered concurrent program is the disappearing layer of m.

Intuition behind layer numbers. Recall that a layered concurrent program $\mathcal{LP}$ should represent a sequence of $h + 1$ concurrent programs $\mathcal{P}_1, \cdots, \mathcal{P}_{h+1}$ that are connected by a sequence of h checker programs $\mathcal{C}_1, \cdots, \mathcal{C}_h$ (cf. Figure 2.1). Before we provide formal definitions, let us get some intuition on two core mechanisms: global variable introduction and procedure abstraction/refinement.

Let v be a global variable with layer range $[a, b]$. The meaning of this layer range is that the "first" program that contains v is $\mathcal{C}_a$, the checker program connecting $\mathcal{P}_a$ and $\mathcal{P}_{a+1}$. In particular, v is not yet part of $\mathcal{P}_a$. In $\mathcal{C}_a$ the introduction actions at layer a can

$$[a,b] = \{x \mid a,b,x \in \mathbb{N} \wedge a \leq x \leq b\}$$

$$\begin{array}{rcl} n,\alpha & \in & \mathbb{N} \\ r & \in & \mathbb{I} = \{[a,b] \mid a \leq b\} \\ ns & \in & \mathit{LVar} \rightharpoonup \mathbb{N} \end{array}$$

$$\begin{array}{rcl} \mathit{GS} & \in & \mathit{GVar} \rightharpoonup \mathbb{I} \\ \mathit{AS} & \in & A \mapsto (I, O, \rho, \tau, r) \\ \mathit{IS} & \in & A \mapsto (I, O, \rho, \tau, n) \\ \mathit{PS} & \in & P \mapsto (I, O, L, s, n, ns, A) \\ m & \in & \mathit{Proc} \\ \mathcal{I} & \in & 2^{\mathit{Store}} \end{array}$$

$$\mathcal{LP} \in \mathit{LayeredProg} ::= (\mathit{GS}, \mathit{AS}, \mathit{IS}, \mathit{PS}, m, \mathcal{I})$$

$$s \in \mathit{Stmt} ::= \cdots \mid \texttt{icall}\ (A, \iota, o) \mid \texttt{pcall}_{\alpha}\ \overline{(P_i, \iota_i, o_i)}_{i \in [1,k]} \quad (\alpha \in \{\varepsilon\} \cup [1,k])$$

Figure 2.4: Layered Concurrent Programs

modify v and thus assign its meaning in terms of all other available variables. Then v is part of $\mathcal{P}_{a+1}$ and all programs up to and including $\mathcal{P}_b$. The "last" program containing v is $\mathcal{C}_b$. In other words, when going from a program $\mathcal{P}_i$ to $\mathcal{P}_{i+1}$ the variables with upper bound i disappear and the variables with lower bound i are introduced; the checker program $\mathcal{C}_i$ has access to both and establishes their relationship.

Let P be a procedure with disappearing layer n and refined atomic action A. The meaning of the disappearing layer is that P exists in all programs from $\mathcal{P}_1$ up to and including $\mathcal{P}_n$. In $\mathcal{P}_{n+1}$ and above every invocation of P is replaced by an invocation of A. To ensure that this replacement is sound, the checker program $\mathcal{C}_n$ performs a refinement check that ensures that every execution of P behaves like A. Observe that the body of procedure P itself changes from $\mathcal{P}_1$ to $\mathcal{P}_n$ according to the disappearing layer of the procedures it calls.

With the above intuition in mind it is clear that the layer annotations in a layered concurrent program cannot be arbitrary. For example, if procedure P calls a procedure Q, then Q cannot have a higher disappearing layer than P, for Q could introduce further behaviors into the program after P was replaced by A, and those behaviors are not captured by A.

2.3.1 Type Checker

We describe the constraints that need to be satisfied for a layered concurrent program to be well-formed. A full formalization as a type checker with top-level judgment $\vdash \mathcal{LP}$ is given in Figure 2.5. For completeness, the type checker includes standard constraints (e.g., variable scoping, parameter passing, etc.) that we are not going to discuss.

(Atomic action)/(Introduction action). Global variables can only be accessed by atomic actions and introduction actions. For a global variable v with layer range $[a,b]$, introduction actions with layer number a are allowed to modify v (for sound variable introduction), and atomic actions with a layer range contained in $[a+1,b]$ have access to v. Introduction actions must be nonblocking, which means that every state that satisfies the gate must have a possible transition to take. This ensures that introduction actions only assign meaning to introduced variables but do not exclude any program behavior.

(Program)

$$\frac{\begin{array}{l}\mathrm{dom}(AS) \cap \mathrm{dom}(IS) = \varnothing \\ PS(m) = (\_, \_, \_, \_, h, \_, A_m) \\ AS(A_m) = (\_, \_, \_, \_, r) \\ h + 1 \in r \\ \forall\ A \in \mathrm{dom}(AS) : (GS, AS) \vdash A \\ \forall\ A \in \mathrm{dom}(IS) : (GS, IS) \vdash A \\ \forall\ P \in \mathrm{dom}(PS) : (AS, IS, PS) \vdash P\end{array}}{\vdash (GS, AS, IS, PS, m, \mathcal{I})}$$

(Atomic action)

$$\frac{\begin{array}{l}AS(A) = (I, O, \rho, \tau, r) \\ \mathit{Disjoint}(I, O) \\ \forall\ v \in \mathit{ReadVars}(\rho, \tau) : v \in I \vee r \subseteq \widehat{GS}(v) \\ \forall\ v \in \mathit{WriteVars}(\tau) : v \in O \vee r \subseteq \widehat{GS}(v)\end{array}}{(GS, AS) \vdash A}$$

(Introduction action)

$$\frac{\begin{array}{l}IS(A) = (I, O, \rho, \tau, n) \\ \mathit{Disjoint}(I, O) \\ \forall\ v \in \mathit{ReadVars}(\rho, \tau) : v \in I \vee n \in GS(v) \\ \forall\ v \in \mathit{WriteVars}(\tau) : v \in O \vee GS(v) = [n, \_] \\ \mathit{Nonblocking}(\rho, \tau)\end{array}}{(GS, IS) \vdash A}$$

(Procedure)

$$\frac{\begin{array}{l}PS(P) = (I, O, L, s, n, ns, A) \\ AS(A) = (I, O, \_, \_, \_) \\ \mathit{Disjoint}(I, O, L) \\ \forall\ v \in I \cup O \cup L : ns(v) \leq n \\ (AS, IS, PS), P \vdash s\end{array}}{(AS, IS, PS) \vdash P}$$

(Skip)

$$\frac{}{(AS, IS, PS), P \vdash \texttt{skip}}$$

(Sequence)

$$\frac{(AS, IS, PS), P \vdash s_1 \quad (AS, IS, PS), P \vdash s_2}{(AS, IS, PS), P \vdash s_1\,;\, s_2}$$

(If)

$$\frac{\begin{array}{l}PS(P) = (I, \_, L, \_, \_, ns, \_) \\ \forall\ x \in \mathit{ReadVars}(\rho) : x \in I \cup L \wedge ns(x) = 0 \\ (AS, IS, PS), P \vdash s_1 \quad (AS, IS, PS), P \vdash s_2\end{array}}{(AS, IS, PS), P \vdash \texttt{if}\ \rho\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

(Parameter passing)

$$\frac{\begin{array}{ll}\mathrm{dom}(\iota) = I' & \mathrm{dom}(o) \subseteq O \cup L \\ \mathrm{img}(\iota) \subseteq I \cup O \cup L & \mathrm{img}(o) \subseteq O'\end{array}}{\mathit{ValidIO}(\iota, o, I, O, L, I', O')}$$

(Introduction call)

$$\frac{\begin{array}{l}PS(P) = (I_P, O_P, L_P, \_, n_P, ns_P, \_) \\ IS(A) = (I_A, O_A, \_, \tau, n_A) \\ \mathit{ValidIO}(\iota, o, I_P, O_P, L_P, I_A, O_A) \\ n_A = n_P \\ \forall\ v \in \mathrm{dom}(o) : ns_P(v) = n_P\end{array}}{(AS, IS, PS), P \vdash \texttt{icall}\ (A, \iota, o)}$$

(Parallel call)

$$\frac{\begin{array}{l}\forall\ i \neq j : \mathrm{dom}(o_i) \cap \mathrm{dom}(o_j) = \varnothing \\ \qquad\qquad \mathrm{dom}(o_i) \cap \mathrm{img}(\iota_j) = \varnothing \\ \forall\ i : PS(P) = (I_P, O_P, L_P, \_, n_P, ns_P, \_) \\ \qquad PS(Q_i) = (I_i, O_i, \_, \_, n_i, ns_i, A_i) \\ \qquad AS(A_i) = (\_, \_, \_, \_, r_i) \\ \qquad \mathit{ValidIO}(\iota_i, o_i, I_P, O_P, L_P, I_i, O_i) \\ \qquad \forall\ v \in \mathrm{dom}(\iota_i) : ns_P(\iota_i(v)) \leq ns_i(v) \\ \qquad \forall\ v \in \mathrm{dom}(o_i) : ns_i(o_i(v)) \leq ns_P(v) \\ \qquad n_i \leq n_P \qquad [n_i + 1, n_P] \subseteq r_i \\ \qquad i = \alpha \implies n_i = n_P \wedge O_P \subseteq \mathrm{dom}(o_i) \\ \qquad i \neq \alpha \wedge n_i = n_P \implies \mathrm{dom}(o_i) \subseteq L_P \\ \exists\ i : n_1 \leq \cdots \leq n_i \geq \cdots \geq n_k\end{array}}{(AS, IS, PS), P \vdash \texttt{pcall}_\alpha\ \overline{(Q_i, \iota_i, o_i)}_{i \in [1,k]}}$$

$\widehat{GS}(v) = [a+1, b]$ for $GS(v) = [a, b]$

$\mathit{ReadVars}(\rho) = \{v \mid \exists\ \sigma, a : \rho(\sigma) \neq \rho(\sigma[v \mapsto a])\} \cup$

$\mathit{ReadVars}(\tau) = \{v \mid \exists\ \sigma, \sigma', a : (\sigma, \sigma') \in \tau \wedge (\sigma[v \mapsto a], \sigma') \notin \tau\}$

$\mathit{ReadVars}(\rho, \tau) = \mathit{ReadVars}(\rho) \cup \mathit{ReadVars}(\tau)$

$\mathit{WriteVars}(\tau) = \{v \mid \exists\ \sigma, \sigma' : (\sigma, \sigma') \in \tau \wedge \sigma(v) \neq \sigma'(v)\}$

$\mathit{Nonblocking}(\rho, \tau) = \forall\ \sigma \in \rho : \exists\ \sigma' : (\sigma, \sigma') \in \tau$

Figure 2.5: Type checking rules for layered concurrent programs

(If). Procedure bodies change from layer to layer because calls to procedures become calls to atomic actions. But the control-flow structure within a procedure is preserved across layers. Therefore (local) variables accessed in an if condition must be available on all layers to ensure that the if statement is well-defined on every layer.

(Introduction call). Let A be an introduction action with layer number n. Since A modifies global variables introduced at layer n, icalls to A are only allowed from procedures with disappearing layer n. Similarly, the formal output parameters of an icall to A must have introduction layer n. The icall is only preserved in $\mathcal{C}_n$.

(Parallel call). All arms in a pcall must be procedure arms invoking a procedure with a disappearing layer less than or equal to the disappearing layer of the caller. Furthermore, above the disappearing layer of the callee its refined atomic action must be available up to the disappearing layer of the caller. Parameter passing can only be well-defined if the actual inputs exist before the formal inputs, and the formal outputs exist before the actual outputs. The sequence of disappearing layers of the procedures in a pcall must be monotonically increasing and then decreasing, such that the resulting pcall in the extracted programs consists of procedure arms surrounded by atomic-action arms on every layer.

Annotated pcalls are only used for invocations to procedures with the same disappearing layer n as the caller. In particular, during refinement checking in $\mathcal{C}_n$ only the arm with index α is allowed to modify the global state, which must be according to the refined atomic action of the caller. The remaining arms must leave the global state unchanged.

2.3.2 Concurrent Program Extraction

Let $\mathcal{LP} = (GS, AS, IS, PS, m, \mathcal{I})$ be a layered concurrent program such that $PS(m) = (\_, \_, \_, \_, h, \_, A_m)$. We show how to extract the programs $\mathcal{P}_1, \cdots, \mathcal{P}_{h+1}$ by defining a function $\Gamma_\ell(\mathcal{LP})$ such that $\mathcal{P}_\ell = \Gamma_\ell(\mathcal{LP})$ for every $\ell \in [1, h+1]$. For a local variable layer mapping ns we define the set of local variables with layer number less then ℓ as $ns|_\ell = \{v \mid ns(v) < \ell\}$. Now the extraction function Γ_ℓ is defined as

$$\Gamma_\ell(\mathcal{LP}) = (gs, as, ps, m', \mathcal{I}),$$

where

$$\begin{aligned}
gs &= \{v \mid GS(v) = [a, b] \wedge \ell \in [a+1, b]\},\\
as &= \{A \mapsto (I, O, \rho, \tau) \mid AS(A) = (I, O, \rho, \tau, r) \wedge \ell \in r\},\\
ps &= \{P \mapsto (I \cap ns|_\ell, O \cap ns|_\ell, L \cap ns|_\ell, \Gamma^P_\ell(s)) \mid PS(P) = (I, O, L, s, n, ns, _) \wedge \ell \leq n\},\\
m' &= \begin{cases} m & \text{if } \ell \in [1, h] \\ A_m & \text{if } \ell = h+1 \end{cases},
\end{aligned}$$

and the extraction of a statement in the body of procedure P is given by

$$
\begin{aligned}
&\Gamma_\ell^P(\texttt{skip}) && = \texttt{skip},\\
&\Gamma_\ell^P(s_1\,;s_2) && = \Gamma_\ell^P(s_1)\,;\Gamma_\ell^P(s_2),\\
&\Gamma_\ell^P(\texttt{if}\ \rho\ \texttt{then}\ s_1\ \texttt{else}\ s_2) && = \texttt{if}\ \rho\ \texttt{then}\ \Gamma_\ell^P(s_1)\ \texttt{else}\ \Gamma_\ell^P(s_2),\\
&\Gamma_\ell^P(\texttt{icall}\ (A,\iota,o)) && = \texttt{skip},\\
&\Gamma_\ell^P(\texttt{pcall}_\alpha\ \overline{(Q,\iota,o)}) && = \texttt{pcall}\ \overline{(X,\iota|_{ns_Q|_\ell},o|_{ns_P|_\ell})},\\
& && \quad \text{for}\ \begin{array}{l} PS(P)=(\_,\_,\_,\_,\_,ns_P,\_)\\ PS(Q)=(\_,\_,\_,\_,n,ns_Q,A)\end{array}\ \text{and}\ X=\begin{cases} Q & \text{if } \ell\le n\\ A & \text{if } \ell>n\end{cases}.
\end{aligned}
$$

Thus $\mathcal{P}_\ell$ includes the global and local variables that were introduced before ℓ and the atomic actions with ℓ in their layer range. Furthermore, it does not contain introduction actions and correspondingly all icall statements are removed. Every arm of a pcall statement, depending on the disappearing layer n of the called procedure Q, either remains a procedure arm to Q, or is replaced by an atomic-action arm to A, the atomic action refined by Q. The input and output mappings are restricted to the local variables at layer ℓ. The set of initial stores of $\mathcal{P}_\ell$ is the same as for $\mathcal{LP}$, since stores range over all program variables.

In our programming language, loops are subsumed by the more general mechanism of recursive procedure calls. Observe that $\mathcal{P}_\ell$ can indeed have recursive procedure calls, because our type checking rules (Figure 2.5) allow a pcall to invoke a procedure with the same disappearing layer as the caller.

2.3.3 Running Example

We return to our lock example from Section 2.2.1. Figure 2.6 shows its implementation as the layered concurrent program $\mathcal{LP}^{lock}$. Layer annotations are indicated using an `@` symbol. For example, the global variable `b` has layer range $[0,1]$, all occurrences of local variable `tid` have introduction layer 1, the atomic action `ACQUIRE` has layer range $[2,2]$, and the introduction action `iSetLock` has layer number 1.

First, observe that $\mathcal{LP}^{lock}$ is well-formed, i.e., $\vdash \mathcal{LP}^{lock}$. Then it is an easy exercise to verify that $\Gamma_\ell(\mathcal{LP}^{lock}) = \mathcal{P}_\ell^{lock}$ for $\ell \in [1,3]$. Let us focus on procedure `Worker`. In $\mathcal{P}_1^{lock}$ (Figure 2.3(a)) `tid` does not exist, and correspondingly `Alloc`, `Enter`, and `Leave` do not have input respectively output parameters. Furthermore, the icall in the body of `Alloc` is replaced with `skip`. In $\mathcal{P}_2^{lock}$ (Figure 2.3(b)) we have `tid` and the calls to `Alloc`, `Enter`, and `Leave` are replaced with their respective refined atomic actions `ALLOC`, `ACQUIRE`, and `RELEASE`. The only annotated pcall in $\mathcal{LP}^{lock}$ is the recursive call to `Enter`.

In addition to representing the concurrent programs in Figure 2.3, the program $\mathcal{LP}^{lock}$ also encodes the connection between them via introduction actions and calls. The introduction action `iSetLock` updates `lock` to maintain the relationship between `lock` and `b`, expressed by the predicate `InvLock`. It is called in `Enter` in case the CAS operation successfully set `b` to *true*, and in `Leave` when `b` is set to *false*. The introduction action `iIncr` implements linear thread identifiers using the integer variables `pos` which points to the next value that can be allocated. For every allocation, the current value of `pos` is returned as the new thread identifier and `pos` is incremented.

The variable `slots` is introduced at layer 1 to represent the set of unallocated identifiers. It contains all integers no less than `pos`, an invariant that is expressed by the predicate

$\mathcal{LP}^{lock}$

```
var b@[0,1] : bool
var lock@[1,2] : int
var pos@[1,1] : int
var linear slots@[1,2] : set<int>

predicate InvLock
  b <==> lock != 0

predicate InvAlloc
  pos > 0 && slots == [pos,infinity)

init InvLock && InvAlloc

both SKIP@3 ()
  skip

proc Main@2()
refines SKIP
  if (*)
    pcall Worker(), Main()

proc Worker@2()
refines SKIP
  var linear tid@1 : int
  pcall tid := Alloc()
  pcall Enter(tid)
  pcall Leave(tid)

right ALLOC@[2,2]() : (linear tid : int)
  assume tid != 0 && tid in slots
  slots := slots - tid

proc Alloc@1() : (linear tid@1 : int)
refines ALLOC
  icall tid := iIncr()

iaction iIncr@1() : (linear tid : int)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots - tid

right ACQUIRE@[2,2](linear tid : int)
  assert tid != 0
  assume lock == 0
  lock := tid

left RELEASE@[2,2](linear tid : int)
  assert tid != 0 && lock == tid
  lock := 0

proc Enter@1(linear tid@1 : int)
refines ACQUIRE
  var success@0 : bool
  pcall success := Cas()
  if (success)
    icall iSetLock(tid)
  else
    pcall_1 Enter(tid)

proc Leave@1(linear tid@1 : int)
refines RELEASE
  pcall Reset()
  icall iSetLock(0)

iaction iSetLock@1(v : int)
  lock := v

atomic CAS@[1,1]() : (success : bool)
  if (b) success := false
  else   success, b := true, true

atomic RESET@[1,1]()
  assert b
  b := false

proc Cas@0() : (success@0 : bool)
refines CAS

proc Reset@0()
refines RESET
```

Figure 2.6: Lock example (layered concurrent program)

`InvAlloc` and maintained by the code of `iIncr`. The purpose of `slots` is to encode linear allocation of thread identifiers in a way that the body of `iIncr` can be locally shown to preserve the disjointness invariant for linear variables; `slots` plays a similar role in the specification of the atomic action `ALLOC` in $\mathcal{P}_2$. The variable `pos` is both introduced and hidden at layer 1 so that it exists neither in $\mathcal{P}_1^{lock}$ nor $\mathcal{P}_2^{lock}$. However, `pos` is present in the checker program $\mathcal{C}_1$ that connects $\mathcal{P}_1^{lock}$ and $\mathcal{P}_2^{lock}$.

The bodies of procedures `Cas` and `Reset` are not shown in Figure 2.6 because they are not needed. They disappear at layer 0 and are replaced by the atomic actions `CAS` and `RESET`, respectively, in $\mathcal{P}_1^{lock}$.

The degree of compactness afforded by layered programs (as in Figure 2.6) over separate specification of each concurrent program (as in Figure 2.3) increases rapidly with the size of the program and the maximum depth of procedure calls. In our experience, for realistic programs such as a concurrent garbage collector [57] or a data-race detector [115], the saving in code duplication is significant.

2.4 Refinement Checking

Section 2.3 described how a layered concurrent program $\mathcal{LP}$ encodes a sequence of concurrent programs $\mathcal{P}_1, \ldots, \mathcal{P}_h, \mathcal{P}_{h+1}$. In this section, we show how the safety of any concurrent program in the sequence is implied by the safety of its successor, ultimately allowing the safety of $\mathcal{P}_1$ to be established by the safety of $\mathcal{P}_{h+1}$.

There are three ingredients to connecting $\mathcal{P}_\ell$ to $\mathcal{P}_{\ell+1}$ for any $\ell \in [1, h]$—reduction, projection, and abstraction. Reduction allows us to conclude the safety of a concurrent program under preemptive semantics by proving safety only under cooperative semantics.

Theorem 1 (Reduction)**.** *Let $\mathcal{P}$ be a concurrent program. If $MSafe(\mathcal{P})$ and $CSafe(\mathcal{P})$, then $Safe(\mathcal{P})$.*

The judgment $MSafe(\mathcal{P})$ uses logical commutativity reasoning and mover types to ensure that cooperative safety is sufficient for preemptive safety (Section 2.4.1). We use this theorem to justify reasoning about $CSafe(\mathcal{P}_\ell)$ rather than $Safe(\mathcal{P}_\ell)$.

The next step in connecting $\mathcal{P}_\ell$ to $\mathcal{P}_{\ell+1}$ is to introduce computation introduced at layer ℓ into the cooperative semantics of $\mathcal{P}_\ell$. This computation comprises global and local variables together with introduction actions and calls to them. We refer to the resulting program at layer ℓ as $\widetilde{\mathcal{P}}_\ell$.

Theorem 2 (Projection)**.** *Let $\mathcal{LP}$ be a layered concurrent program with top layer h and $\ell \in [1, h]$. If $CSafe(\widetilde{\mathcal{P}}_\ell)$, then $CSafe(\mathcal{P}_\ell)$.*

Since introduction actions are nonblocking and $\widetilde{\mathcal{P}}_\ell$ is safe under cooperative semantics, every cooperative execution of $\mathcal{P}_\ell$ can be obtained by projecting away the computation introduced at layer ℓ. This observation allows us to conclude that every cooperative execution of $\mathcal{P}_\ell$ is also safe.

Finally, we check that the safety of the cooperative semantics of $\widetilde{\mathcal{P}}_\ell$ is ensured by the safety of the preemptive semantics of the next concurrent program $\mathcal{P}_{\ell+1}$. This connection is established by reasoning about the cooperative semantics of a concurrent checker program $\mathcal{C}_\ell$ that is automatically constructed from $\mathcal{LP}$.

Theorem 3 (Abstraction)**.** *Let $\mathcal{LP}$ be a layered concurrent program with top layer h and $\ell \in [1, h]$. If $CSafe(\mathcal{C}_\ell)$ and $Safe(\mathcal{P}_{\ell+1})$, then $CSafe(\widetilde{\mathcal{P}}_\ell)$.*

The checker program $\mathcal{C}_\ell$ is obtained by instrumenting the code of $\widetilde{\mathcal{P}}_\ell$ with extra variables and procedures that enable checking that procedures disappearing at layer ℓ refine their atomic action specifications (Section 2.4.2).

Our refinement check between two consecutive layers is summarized by the following corollary of Theorem 1-3.

Corollary 1. *Let $\mathcal{LP}$ be a layered concurrent program with top layer h and $\ell \in [1, h]$. If $MSafe(\mathcal{P}_\ell)$, $CSafe(\mathcal{C}_\ell)$ and $Safe(\mathcal{P}_{\ell+1})$, then $Safe(\mathcal{P}_\ell)$.*

The soundness of our refinement checking methodology for layered concurrent programs is obtained by repeated application of Corollary 1.

Corollary 2. *Let $\mathcal{LP}$ be a layered concurrent program with top layer h. If $MSafe(\mathcal{P}_\ell)$ and $CSafe(\mathcal{C}_\ell)$ for all $\ell \in [1, h]$ and $Safe(\mathcal{P}_{h+1})$, then $Safe(\mathcal{P}_1)$.*

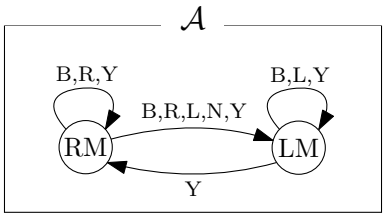


Figure 2.7: Atomicity automaton.

2.4.1 From Preemptive to Cooperative Semantics

We present the judgment $MSafe(\mathcal{P})$ that allows us to reason about a concurrent program $\mathcal{P}$ under cooperative semantics instead of preemptive semantics. Intuitively, we want to use the commutativity of individual atomic actions to rearrange the steps of any execution under preemptive semantics in such a way that it corresponds to an execution under cooperative semantics. We consider mappings $M \in Action \to \{\mathrm{N}, \mathrm{R}, \mathrm{L}, \mathrm{B}\}$ that assign mover types to atomic actions; N for non-mover, R for right mover, L for left mover, and B for both mover. The judgment $MSafe(\mathcal{P})$ requires a mapping M that satisfies two conditions.

First, the atomic actions in $\mathcal{P}$ must satisfy the following logical commutativity conditions [57], which can be discharged by a theorem prover.

- *Commutativity:* If A_1 is a right mover or A_2 is a left mover, then the effect of A_1 followed by A_2 can also be achieved by A_2 followed by A_1.
- *Forward preservation:* If A_1 is a right mover or A_2 is a left mover, then the failure of A_2 after A_1 implies that A_2 must also fail before A_1.
- *Backward preservation:* If A_2 is a left mover (and A_1 is an arbitrary), then the failure of A_1 before A_2 implies that A_1 must also fail after A_2.
- *Nonblocking:* If A is a left mover, then A cannot block.

Second, the sequence of atomic actions in preemptive executions of $\mathcal{P}$ must be such that the desired rearrangement into cooperative executions is possible. Given a preemptive execution, consider, for each thread individually, a labeling of execution steps where atomic action steps are labeled with their mover type and procedure calls and returns are labeled with Y (for yield). The nondeterministic *atomicity automaton* $\mathcal{A}$ in Figure 2.7 defines all allowed sequences. Intuitively, when we map the execution steps of a thread to a run in the automaton, the state RM denotes that we are in the right mover phase in which we can stay until the occurrence of a non-right mover (L or N). Then we can stay in the left mover phase (state LM) by executing left movers, until a preemption point (Y) takes us back to RM. Let $\mathcal{E}$ be the mapping from edge labels to the set of edges that contain the label, e.g., $\mathcal{E}(\mathrm{R}) = \{\mathrm{RM} \to \mathrm{RM}, \mathrm{RM} \to \mathrm{LM}\}$. Thus we have a representation of mover types as sets of edges in $\mathcal{A}$, and we define $\mathcal{E}(A) = \mathcal{E}(M(A))$. Notice that the set representation is closed under relation composition $\circ$ and intersection, and behaves as expected, e.g., $\mathcal{E}(\mathrm{R}) \circ \mathcal{E}(\mathrm{L}) = \mathcal{E}(\mathrm{N})$.

Now we define an intraprocedural control flow analysis that lifts $\mathcal{E}$ to a mapping $\widehat{\mathcal{E}}$ on statements. Intuitively, $x \to y \in \widehat{\mathcal{E}}(s)$ means that every execution of the statement s has a run in $\mathcal{A}$ from x to y. Our analysis does not have to be interprocedural, since

procedure calls and returns are labeled with Y, allowing every possible state transition in $\mathcal{A}$. *MSafe*($\mathcal{P}$) requires $\widehat{\mathcal{E}}(s) \neq \varnothing$ for every procedure body s in $\mathcal{P}$, where $\widehat{\mathcal{E}}$ is defined as follows:

$$\widehat{\mathcal{E}}(\texttt{skip}) = \mathcal{E}(\mathrm{B}) \quad \widehat{\mathcal{E}}(s_1\,;s_2) = \widehat{\mathcal{E}}(s_1) \circ \widehat{\mathcal{E}}(s_2) \quad \widehat{\mathcal{E}}(\texttt{if } \rho \texttt{ then } s_1 \texttt{ else } s_2) = \widehat{\mathcal{E}}(s_1) \cap \widehat{\mathcal{E}}(s_2)$$

$$\widehat{\mathcal{E}}(\texttt{pcall } \overline{A}_1 \overline{P}\ \overline{A}_2) = \begin{cases} \mathcal{E}^*(\overline{A}_1 \overline{A}_2) & \text{if } \overline{P} = \varepsilon \\ \mathcal{E}(\mathrm{L}) \circ \mathcal{E}^*(\overline{A}_1) \circ \mathcal{E}(\mathrm{Y}) \circ \mathcal{E}^*(\overline{A}_2) \circ \mathcal{E}(\mathrm{R}) & \text{if } \overline{P} \neq \varepsilon \end{cases}$$

Skip is a both mover, sequencing composes edges, and if takes the edges possible in both branches. In the arms of a pcall we omit writing the input and output maps because they are irrelevant to the analysis. Let us first focus on the case $\overline{P} = \varepsilon$ with no procedure arms. In the preemptive semantics all arms are arbitrarily interleaved and correspondingly we define the function

$$\mathcal{E}^*(A_1 \cdots A_n) = \bigcap_{\tau \in S_n} \mathcal{E}(A_{\tau(1)}) \circ \cdots \circ \mathcal{E}(A_{\tau(n)})$$

to consider all possible permutations (τ ranges over the symmetric group S_n) and take the edges possible in all permutations. Observe that $\mathcal{E}^*$ evaluates to non-empty in exactly four cases: $\mathcal{E}(\mathrm{N})$ for $\{\mathrm{B}\}^*\mathrm{N}\{\mathrm{B}\}^*$, $\mathcal{E}(\mathrm{B})$ for $\{\mathrm{B}\}^*$, $\mathcal{E}(\mathrm{R})$ for $\{\mathrm{R},\mathrm{B}\}^* \setminus \{\mathrm{B}\}^*$, and $\mathcal{E}(\mathrm{L})$ for $\{\mathrm{L},\mathrm{B}\}^* \setminus \{\mathrm{B}\}^*$. These are the mover-type sequences for which an arbitrary permutation (coming from a preemptive execution) can be rearranged to the order given by the pcall (corresponding to cooperative execution).

In the case $\overline{P} \neq \varepsilon$ there is a preemption point under cooperative semantics between $\overline{A}_1$ and $\overline{A}_2$, the actions in $\overline{A}_1$ are executed in order before the preemption, and the actions in $\overline{A}_2$ are executed in order after the preemption. To ensure that the cooperative execution can simulate an arbitrarily interleaved preemptive execution of the pcall, we must be able to move actions in $\overline{A}_1$ to the left and actions in $\overline{A}_2$ to the right of the preemption point. We enforce this condition by requiring that $\overline{A}_1$ is all left (or both) movers and $\overline{A}_2$ all right (or both) movers, expressed by the leading $\mathcal{E}(\mathrm{L})$ and trailing $\mathcal{E}(\mathrm{R})$ in the edge composition.

2.4.2 Refinement Checker Programs

In this section, we describe the construction of checker programs that justify the formal connection between successive concurrent programs in a layered concurrent program. The description is done by example. In particular, we show the checker program $\mathcal{C}_1^{lock}$ that establishes the connection between $\mathcal{P}_1^{lock}$ and $\mathcal{P}_2^{lock}$ (Figure 2.3) of our running example.

Overview. Cooperative semantics splits any execution of $\mathcal{P}_1^{lock}$ into a sequence of preemption-free execution fragments separated by preemptions. Verification of $\mathcal{C}_1^{lock}$ must ensure that for all such executions, the set of procedures that disappear at layer 1 behave like their atomic action specifications. That is, the procedures `Enter` and `Leave` must behave like their specifications `ACQUIRE` and `RELEASE`, respectively. It is important to note that this goal of checking refinement is easier than verifying that $\mathcal{P}_1^{lock}$ is safe. Refinement checking may succeed even though $\mathcal{P}_1^{lock}$ fails; the guarantee of refinement is that such a failure can be simulated by a failure in $\mathcal{P}_2^{lock}$. The construction of $\mathcal{C}_1^{lock}$ can be understood in two steps. First, the program $\widetilde{\mathcal{P}}_1^{lock}$ shown in Figure 2.8 extends $\mathcal{P}_1^{lock}$ (Figure 2.3(a)) with the variables introduced at layer 1 (globals `lock`, `pos`, `slots` and

$\widetilde{\mathcal{P}}_1^{lock}$

```
var b : bool
var lock : int
var pos : int
var linear slots : set<int>

proc Main()
  if (*)
    pcall Worker(), Main()

proc Worker()
  var linear tid : int
  pcall tid := Alloc()
  pcall Enter(tid)
  pcall Leave(tid)

proc Alloc() : (linear tid : int)
  icall tid := iIncr()

iaction iIncr() : (tid : int)
  assert InvAlloc
  tid := pos
  pos := pos + 1
  slots := slots - tid

proc Enter(linear tid : int)
  var success : bool
  pcall success := CAS()
  if (success)
    icall iSetLock(tid)
  else
    pcall Enter(tid)

proc Leave(linear tid : int)
  pcall RESET()
  icall iSetLock(0)

iaction iSetLock(v : int)
  lock := v

atomic CAS() : (success : bool)
  if (b) success := false
  else   success, b := true, true

atomic RESET()
  assert b
  b := false
```

Figure 2.8: Lock example (variable introduction at layer 1)

locals `tid`) and the corresponding introduction actions (`iIncr` and `iSetLock`). Second, $\mathcal{C}_1^{lock}$ is obtained from $\widetilde{\mathcal{P}}_1^{lock}$ by instrumenting the procedures to encode the refinement check, described in the remainder of this section.

Context for refinement. There are two kinds of procedures, those that continue to exist at layer 2 (such as `Main` and `Worker`) and those that disappear at layer 1 (such as `Enter` and `Leave`). $\mathcal{C}_1^{lock}$ does not need to verify anything about the first kind. These procedures only provide the context for refinement checking and thus all invocation of an atomic action (I, O, ρ, τ) in any atomic-action arm of a pcall is converted into the invocation of a fresh atomic action $(I, O, \mathit{true}, \rho \wedge \tau)$. In other words, the assertions in procedures that continue to exist at layer 2 are converted into assumptions for the refinement checking at layer 1; these assertions are verified during the refinement checking on a higher layer. In our example, `Main` and `Worker` do not have atomic-action arms, although this is possible in general.

Refinement instrumentation. We illustrate the instrumentation of procedures `Enter` and `Leave` in Figure 2.9. The core idea is to track updates by preemption-free execution fragments to the shared variables that continue to exist at layer 2. There are two such variables—`lock` and `slots`. We capture snapshots of `lock` and `slots` in the local variables `_lock` and `_slots` and use these snapshots to check that the updates to `lock` and `slots` behave according to the refined atomic action. In general, any path from the start to the end of the body of a procedure may comprise many preemption-free execution fragments. The checker program must ensure that exactly one of these fragments behaves like the specified atomic action; all other fragments must leave `lock` and `slot` unchanged. To track whether the atomic action has already happened, we use two local Boolean variables—`pc` and `done`. Both variables are initialized to *false*, get updated to *true* during

$\mathcal{C}_1^{lock}$

```
1  macro *CHANGED* is !(lock == _lock && slots == _slots)
2  macro *RELEASE* is lock == 0 && slots == _slots
3  macro *ACQUIRE* is _lock == 0 && lock == tid && slots == _slots
4  ---------------------------------------------------------------------------
5  proc Leave(linear tid)                      # Leave must behave like RELEASE
6    var _lock, _slots, pc, done
7    pc, done := false, false                  # initialize pc and done
8    _lock, _slots := lock, slots              # take snapshot of global variables
9    assume pc || (tid != 0 && lock == tid)    # assume gate of RELEASE
10
11   pcall RESET()
12   icall iSetLock(0)
13
14   assert *CHANGED* ==> (!pc && *RELEASE*)   # state change must be the first and like RELEASE
15   pc := pc || *CHANGED*                     # track if state changed
16   done := done || *RELEASE*                 # track if RELEASE happened
17
18   assert done                               # check that RELEASE happened
19 ---------------------------------------------------------------------------
20 proc Enter(linear tid)                      # Enter must behave like ACQUIRE
21   var success, _lock, _slots, pc, done
22   pc, done := false, false                  # initialize pc and done
23   _lock, _slots := lock, slots              # take snapshot of global variables
24   assume pc || tid != 0                     # assume gate of ACQUIRE
25
26   pcall success := CAS()
27   if (success)
28     icall iSetLock(tid)
29   else
30     assert *CHANGED* ==> (!pc && *ACQUIRE*) # state change must be the first and like ACQUIRE
31     pc := pc || *CHANGED*                   # track if state changed
32     done := done || *ACQUIRE*               # track if ACQUIRE happened
33
34     if (*)                                  # then: check refinement of caller
35       pcall pc := Check_Enter_Enter(tid,    #   check annotated procedure arm
36                                     tid, pc) #     in fresh procedure (defined below)
37       done := true                          #   above call ensures that ACQUIRE happened
38     else                                    # else: check refinement of callee
39       pcall Enter(tid)                      #   explore behavior of callee
40       assume false                          #   block after return (only then is relevant below)
41
42     _lock, _slots := lock, slots            # take snapshot of global variables
43     assume pc || tid != 0                   # assume gate of ACQUIRE
44
45   assert *CHANGED* ==> (!pc && *ACQUIRE*)   # state change must be the first and like ACQUIRE
46   pc := pc || *CHANGED*                     # track if state changed
47   done := done || *ACQUIRE*                 # track if ACQUIRE happened
48
49   assert done                               # check that ACQUIRE happened
50
51 proc Check_Enter_Enter(tid, x, pc) : (pc')  # check annotated pcall from Enter to Enter
52   var _lock, _slots
53   _lock, _slots := lock, slots              # take snapshot of global variables
54   assume pc || tid != 0                     # assume gate of ACQUIRE
55
56   pcall ACQUIRE(x)                          # use ACQUIRE to ''simulate'' call to Enter
57
58   assert *ACQUIRE*                          # check that ACQUIRE happened
59   assert *CHANGED* ==> !pc                  # state change must be the first
60   pc' := pc || *CHANGED*                    # track if state changed
```

Figure 2.9: Instrumented procedures `Enter` and `Leave` (layer 1 checker program)

the execution, and remain at *true* thereafter. The variable `pc` is set to *true* at the end of the first preemption-free execution fragment that modifies the tracked state, which is expressed by the macro `*CHANGED*` on line 1. The variable `done` is set to *true* at the end of the first preemption-free execution fragment that behaves like the refined atomic action. For that, the macros `*RELEASE*` and `*ACQUIRE*` on lines 2 and 3 express the transition relations of `RELEASE` and `ACQUIRE`, respectively. Observe that we have the invariant `pc ==> done`. The reason we need both `pc` and `done` is to handle the case where the refined atomic action may stutter (i.e., leave the state unchanged).

Instrumenting `Leave`. We first look at the instrumentation of `Leave`. Line 8 initializes the snapshot variables. Recall that a preemption inside the code of a procedure is introduced only at a pcall containing a procedure arm. Consequently, the body of `Leave` is preemption-free and we need to check refinement across a single execution fragment. This checking is done by lines 14-16. The assertion on line 14 checks that if any tracked variable has changed since the last snapshot, (1) such a change happens for the first time (`!pc`), and (2) the current value is related to the snapshot value according to the specification of `RELEASE`. Line 15 updates `pc` to track whether any change to the tracked variables has happened so far. Line 16 updates `done` to track whether `RELEASE` has happened so far. The assertion at line 18 checks that `RELEASE` has indeed happened before `Leave` returns. The assumption at line 9 blocks those executions which can be simulated by the failure of `RELEASE`. It achieves this effect by assuming the gate of `RELEASE` in states where `pc` is still *false* (i.e., `RELEASE` has not yet happened). The assumption yields the constraint `lock != 0` which together with the invariant `InvLock` (Figure 2.6) proves that the gate of `RESET` does not fail.

The verification of `Leave` illustrates an important principle of our approach to refinement. The gates of atomic actions invoked by a procedure P disappearing at layer ℓ are verified using a combination of invariants established on $\mathcal{C}_\ell$ and pending assertions at layer $\ell+1$ encoded as the gate of the atomic action refined by P. For `Leave` specifically, `assert b` in `RESET` is propagated to `assert tid != nil && lock == tid` in `RELEASE`. The latter assertion is verified in the checker program $\mathcal{C}_2^{lock}$ when `Worker`, the caller of `RELEASE`, is shown to refine the action `SKIP` which is guaranteed not to fail since its gate is *true*.

Instrumenting `Enter`. The most sophisticated feature in a concurrent program is a pcall. The instrumentation of `Leave` explains the instrumentation of the simplest kind of pcall with only atomic-action arms. We now illustrate the instrumentation of a pcall containing a procedure arm using the procedure `Enter` which refines the atomic action `ACQUIRE` and contains a pcall to `Enter` itself. The instrumentation of this pcall is contained in lines 30-43.

A pcall with a procedure arm is challenging for two reasons. First, the callee disappears at the same layer as the caller so the checker program must reason about refinement for both the caller and the callee. This challenge is addressed by the code in lines 34-40. At line 34, we introduce a nondeterministic choice between two code paths—`then` branch to check refinement of the caller and `else` branch to check refinement of the callee. An explanation for this nondeterministic choice is given in the next two paragraphs. Second, a pcall with a procedure arm introduces a preemption creating multiple preemption-free execution fragments. This challenge is addressed by two pieces of code. First, we check that `lock` and `slots` are updated correctly (lines 30-32) by the preemption-free execution

fragment ending before the pcall. Second, we update the snapshot variables (line 42) to enable the verification of the preemption-free execution fragment beginning after the pcall.

Lines 35-37 in the `then` branch check refinement against the atomic action specification of the caller, exploiting the atomic action specification of the callee. The actual verification is performed in a fresh procedure `Check_Enter_Enter` invoked on line 35. Notice that this procedure depends on both the caller and the callee (indicated in colors), and that it preserves a necessary preemption point. The procedure has input parameters `tid` to receive the input of the caller (for refinement checking) and `x` to receive the input of the callee (to generate the behavior of the callee). Furthermore, `pc` may be updated in `Check_Enter_Enter` and thus passed as both an input and output parameter. In the body of the procedure, the invocation of action `ACQUIRE` on line 56 overapproximates the behavior of the callee. In the layered concurrent program (Figure 2.6), the (recursive) pcall to `Enter` in the body of `Enter` is annotated with 1. This annotation indicates that for any execution passing through this pcall, `ACQUIRE` is deemed to occur during the execution of its unique arm. This is reflected in the checker program by updating `done` to *true* on line 37; the update is justified because of the assertion in `Check_Enter_Enter` at line 58. If the pcall being translated was instead unannotated, line 37 would be omitted.

Lines 39-40 in the `else` branch ensure that using the atomic action specification of the callee on line 56 is justified. Allowing the execution to continue to the callee ensures that the called procedure is invoked in all states allowed by $\mathcal{P}_1$. However, the execution is blocked once the call returns to ensure that downstream code sees the side-effect on `pc` and the snapshot variables.

To summarize, the crux of our instrumentation of procedure arms is to combine refinement checking of caller and callee. We explore the behaviors of the callee to check its refinement. At the same time, we exploit the atomic action specification of the callee to check refinement of the caller.

Instrumenting unannotated procedure arms. Procedure `Enter` illustrates the instrumentation of an annotated procedure arm. The instrumentation of an unannotated procedure arm (both in an annotated or unannotated pcall) is simpler, because we only need to check that the tracked state is not modified. For such an arm to a procedure refining atomic action `Action`, we introduce a procedure `Check_Action` (which is independent of the caller) comprising three instructions: take snapshots, `pcall A`, and `assert !*CHANGED*`.

Pcalls with multiple arms. Our examples show the instrumentation of pcalls with a single arm. Handling multiple arms is straightforward, since each arm is translated independently. Atomic action arms stay unmodified, annotated procedure arms are replaced with the corresponding `Check_Caller_Callee` procedure, and unannotated procedure arms are replaced with the corresponding `Check_Action` procedure.

Output parameters. Our examples illustrate refinement checking for atomic actions that have no output parameters. In general, a procedure and its atomic action specification may return values in output parameters. We handle this generalization but lack of space does not allow us to present the technical details.

2.5 Conclusion

In this paper, we presented layered concurrent programs, a programming notation to succinctly capture a multi-layered refinement proof capable of connecting a deeply-detailed implementation to a highly-abstract specification. We presented an algorithm to extract from the concurrent layered program the individual concurrent programs, from the most concrete to the most abstract. We also presented an algorithm to extract a collection of refinement checker programs that establish the connection among the sequence of concurrent programs encoded by the layered concurrent program. The cooperative safety of the checker programs and the preemptive safety of the most abstract concurrent program suffices to prove the preemptive safety of the most concrete concurrent program.

Layered programs have been implemented in Civl, a deductive verifier for concurrent programs, implemented as a conservative extension to the Boogie verifier [14]. Civl has been used to verify a complex concurrent garbage collector [56] and a state-of-the-art data-race detection algorithm [115]. In addition to these two large benchmarks, around fifty smaller programs (including a ticket lock and a lock-free stack) are available at https://github.com/boogie-org/boogie.

There are several directions for future work. We did not discuss how to verify an individual checker program. Civl uses the Owicki-Gries method [95] and rely-guarantee reasoning [63] to verify checker programs. But researchers are exploring many different techniques for verification of concurrent programs. It would be interesting to investigate whether heterogeneous techniques could be brought to bear on checker programs at different layers.

In this paper, we focused exclusively on verification and did not discuss code generation, an essential aspect of any programming system targeting the construction of verified programs. There is a lot of work to be done in connecting the most concrete program in a concurrent layered program to executable code. Most likely, different execution platforms will impose different obligations on the most concrete program and the general idea of layered concurrent programs would be specialized for different target platforms.

Scalable verification is a challenge as the size of programs being verified increases. Traditionally, scalability has been addressed using modular verification techniques but only for single-layer programs. It would be interesting to explore modularity techniques for concurrent layered programs in the context of a refinement-oriented proof system.

Layered concurrent programs bring new challenges and opportunities to the design of programming languages and development environments. Integrating layers into a programming language requires intuitive syntax to specify layer information and atomic actions. For example, ordered layer names can be more readable and easier to refactor than layer numbers. An integrated development environment could provide different views of the layered concurrent program. For example, it could show the concurrent program, the checker program, and the introduced code at a particular layer. Any updates made in these views should be automatically reflected back into the layered concurrent program.

Acknowledgments. We thank Hana Chockler, Stephen Freund, Thomas A. Henzinger, Viktor Toman, and James R. Wilcox for comments that improved this paper. This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

3 Refinement for Structured Concurrent Programs

Abstract. This paper presents a foundation for refining concurrent programs with structured control flow. The verification problem is decomposed into subproblems that aid interactive program development, proof reuse, and automation. The formalization in this paper is the basis of a new design and implementation of the Civl verifier.

3.1 Introduction

We present a solution to the problem of proving that no execution of a concurrent program leads to a failure. This problem is equivalent to proving an arbitrary safety property on the program. In *deductive verification*, a proof system decomposes this verification problem into a set of *proof obligations* (or *verification conditions*), and discharging these obligations implies the correctness of the program. At its core, any proof system depends on *inductive invariants*, and, in general, these have to be supplied manually. Inventing an inductive invariant is especially challenging for concurrent programs, since it has to capture complicated relationships over the entire program state, across all concurrent computations. Thus, the main practical obstacle to deductive verification is a suitable interaction mode for the programmer to invent and supply the necessary proof hints. This paper develops and implements a systematic conceptual framework for supplying these proof hints on a structured representation of the concurrent program, specifically eliminating the need to write complex invariants on the low-level encoding of the program as a flat transition system.

The Civl verifier [56, 70] addresses the aforementioned challenge by advocating *layered refinement over structured concurrent programs*. Instead of the monolithic approach that requires the programmer to prove the safety of a program $\mathcal{P}$ directly, Civl allows the programmer to specify a chain of increasingly simpler programs $\mathcal{P} = \mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n = \mathcal{P}'$ such that the safety of $\mathcal{P}_i$ implies the safety of $\mathcal{P}_{i-1}$ for all $i \in [1, n]$, thus transferring the safety obligation on $\mathcal{P}$ to $\mathcal{P}'$. The overall correctness of the program is established piecemeal by focusing on the invariant required for each refinement step separately. While the programmer does the creative work of specifying the chain of programs and the inductive invariant justifying each link in the chain, the tool automatically constructs the verification conditions underlying each refinement step.

The core principle of a layered refinement proof in Civl is *iterative program simplification* through two kinds of creative reasoning. First, the programmer must think about the primitive atomic actions used to specify a particular program $\mathcal{P}_i$ in the chain of

programs. These atomic actions must be chosen to have useful commutativity properties which allow the tool to provably eliminate preemptions at many control locations in $\mathcal{P}_i$, thus creating large preemption-free execution fragments. Second, the programmer must think about the justification for the transformation of $\mathcal{P}_i$ into the next program $\mathcal{P}_{i+1}$. This transformation may be complex because (1) some of the variables in $\mathcal{P}_i$ may become irrelevant, (2) new variables may be needed for the primitive atomic actions in $\mathcal{P}_{i+1}$, and (3) the transformation may simplify complex control flow (branching, procedure calls, recursion, etc.) into a single step that executes an atomic action. This paper focuses on the necessary foundation and tool support for this second kind of creative reasoning.

We present our technique on an idealized yet general language RefPL, suitable for expressing structured parallelism, asynchronous computation, atomic actions of arbitrary granularity, and dynamically-scoped preemption-free code fragments. Using the design of RefPL and the formalization of its operational semantics, we present two technical contributions.

Our first contribution is a general proof rule for soundly abstracting a recursive RefPL program $\mathcal{P}$ into another RefPL program $\mathcal{P}'$ that hides subsets of global variables, local variables, procedures, and atomic actions in $\mathcal{P}$. Our proof rule goes beyond Civl in two ways. First, it provides the capability to hide local variables of procedures, specifically parameters, in addition to global variables. This capability allows us to replace a procedure with an atomic action with a smaller interface by hiding the extra parameters. Refinement proofs are simplified because it becomes easy to introduce local snapshots of global variables needed for specifications, pass these snapshots around as parameters to procedures, and finally recover the original interface by hiding these extra parameters. Second, unlike Civl our proof rule is capable of performing refinement proofs on arbitrarily recursive programs. Since hiding low-level details is the core principle of the layered refinement methodology, our proof rule contributes towards increasing the expressiveness of refinement proofs compared to Civl.

Our proof rule depends on invariants that constrain the reachable states of the program. Our second contribution, an aid to our refinement rule but also independently useful, is a new specification idiom called *yield invariants*—named, parameterized, and interference-free invariants that can be called in parallel with ordinary procedures to soundly constrain the interference possible at yields within the called procedure. Since a yield invariant is named, its definition is separate from its invocation, thereby allowing proofs of interference-freedom to be performed once and reused for each call site. Since it is parameterized, it can be specialized to the needs of a call site by passing suitable input parameters.

Reasoning with yield invariants becomes difficult in concurrent programs when the absence of interference must be justified using facts referring to local variables of different procedures executing in different threads. The alternative of using global ghost variables that have the same information as local variables is theoretically possible but impossibly tedious. We observe that local proofs for many of these programming patterns can be achieved by exploiting *permissions* that are redistributed by atomic actions and otherwise passed around the program without duplication via input and output parameters of procedures. To track permissions, we enhance the interface of yield invariants, procedures, and atomic actions with annotations that satisfy a discipline enforced by a combination of linear typing [114] over procedure bodies and logical reasoning over the transitions of atomic actions.

The formalization in this paper is the basis of a new design and implementation of

the Civl verifier. We hope that Civl will serve researchers as a viable platform for experimenting with optimizations and implementation decisions.

To summarize, this paper makes the following contributions:

- It presents a core language RefPL for expressing modular proofs of refinement over structured concurrent programs. The formulation of refinement for RefPL is general and allows the user to encode verification of an arbitrary safety property as refinement verification. Furthermore, RefPL enables the construction of layered proofs [70] of safety via iterated refinement.

- A refinement proof for RefPL is modular and decomposed along program syntax through the use of yield invariants. The interfaces to procedures, actions, and yield invariants exploit a linear typing discipline [114] that enhances local verification through the use of permissions.

- Finally, we present a robust implementation of the refinement rule and yield invariants in the Civl verifier.

3.1.1 Related Work

Formal verification techniques based on stepwise refinement have long been advocated, in theory, for construction of verified programs (e.g., [11, 103, 37]). This paper takes its inspiration from TLA [76] and Event-B [5, 6] which popularized refinement as an approach for reasoning about a concurrent program modeled as a transition system. Recent efforts [55, 26, 53] have developed support for development of verified programs atop the foundation of refinement over transition systems. Our work develops a foundation and tool support for refinement over structured concurrent programs rather than flat transition systems. We are encouraged by broad interest in the use of automatic program simplification [112, 34] to reduce the complexity of reasoning about concurrent programs.

The technique of yield invariants is inspired by interference-free location invariants in the work of Owicki and Gries [95] and the rely specification in rely-guarantee reasoning [63]. Yield invariants attempt to import the reuse of rely specifications to location invariants. We introduce linear interfaces to encode permissions to address the practical concern of unwieldy ghost state. While permissions have been used before for encoding ownership in heap-manipulating programs [88], our encoding of permissions is different, applicable to any shared resource, and targeted specifically at noninterference reasoning.

There are other efforts to build practical verifiers for concurrent programs. Some verifiers focus on automation and target specific programming models and languages [78, 32, 62, 16]. Our verifier is just as automated but capable of targeting a variety of programming models because of the foundation of atomic actions in RefPL. Other verifiers share our focus on expressiveness by providing general and certified metatheory [64] but are less automated; our verifier attempts to increase expressiveness without sacrificing automation. None of these aforementioned verifiers focus on refinement and layered proofs.

Our work bears a superficial resemblance to proof methods [110, 19, 66] for linearizability [59]. Our work targets the general problem of safety verification. Linearizability is a specific safety property to which our method is applicable.

3.2 Overview

In this section, we illustrate our contributions on a set of example programs. Section 3.2.1 presents yield invariants, Section 3.2.2 presents refinement, and Section 3.2.3 presents linear interfaces.

3.2.1 Yield Invariants

Figure 3.1 shows a simple RefPL program. The first column shows a global counter `x`, a procedure `incr_x` that increments `x` twice, and a yield invariant `yield_x` that characterizes the interference from other threads while a thread is executing `incr_x`. The increments of `x` on lines 4 and 6 are separated by a call to the yield invariant `yield_x`. RefPL provides a single call statement for calling any number (including zero) of procedures and yield invariants in parallel. The `preserves` specification on line 3 indicates that `yield_x` is both a precondition (usually indicated by `requires`) and a postcondition (usually indicated by `ensures`). In RefPL, each precondition of a procedure is a call to a yield invariant; all preconditions are called in parallel at procedure entry. Similarly, each postcondition is a call to a yield invariant; all postconditions are called in parallel at procedure exit.

This paper focuses on reasoning about cooperative semantics in which preemptions occur only at entry into a procedure, at a call during its execution, and at exit. The RefPL verifier proves the correctness of `yield_x` and `incr_x` modularly on these cooperative semantics. Specifically, the yield invariant `yield_x` is proved interference-free since the only operations in the program that modify `x` increment it. The procedure `incr_x` is proved by using the precondition of `incr_x` to establish the yield invariant at line 5 and then using the yield invariant to prove the postcondition at exit. This proof of `incr_x` depends on the observation that the input parameter `_x` of `incr_x` is passed as the argument to the three calls to `yield_x`: in the precondition, on line 5, and in the postcondition. The second column shows code similar to what we just discussed, except on global variable `y`, procedure `incr_y`, and yield invariant `yield_y`.

The third column show a procedure `incr_x_y` which uses recursion to create an unbounded number of concurrent threads. `incr_x_y` nondeterministically spawns a copy of itself on lines 20-21, calls procedures to increment `x` and `y` on lines 22-23, and asserts a safety property about `x` and `y` on line 24. Our verification goal is to prove that if a single instance of `incr_x_y` starts in a state that satisfies the initial constraints on `x` and `y`, indicated on lines 1 and 9 respectively, then the assertion on line 24 holds in every copy of `incr_x_y`.

The proof of procedure `incr_x_y` shows the modularity of yield invariants. First, notice that no new yield invariants are needed; the entire proof of `incr_x_y` is achieved by reusing `yield_x` and `yield_y`. Specifically, `yield_x` and `yield_y` are called in parallel with each other at entry, `yield_y` is called in parallel with `incr_x` at line 22, and `yield_x` is called in parallel with `incr_y` at line 23. Second, the arguments to `yield_x` and `yield_y` are specialized to match the constraints in the initial state and the assertions.

```
1  var x: int // ≥ 0
2  procedure incr_x(_x: int)
3  preserves yield_x(_x)
4    x := x + 1
5    call yield_x(_x)
6    x := x + 1
7  invariant yield_x(_x: int)
8    _x ≤ x
9  var y: int // ≥ 0
10 procedure incr_y(_y: int)
11 preserves yield_y(_y)
12   y := y + 1
13   call yield_y(_y)
14   y := y + 1
15 invariant yield_y(_y: int)
16   _y ≤ y
17 procedure incr_x_y()
18 requires yield_x(0)
19 requires yield_y(0)
20   if (*)
21     async incr_x_y()
22   call incr_x(0) || yield_y(0)
23   call incr_y(0) || yield_x(0)
24   assert 0 ≤ x ∧ 0 ≤ y
```

Figure 3.1: Incrementing two separate counters to illustrate yield invariants.

3.2.2 Refining Atomic Actions

Figure 3.2 shows a spin lock implementation and a client that uses the spin lock to atomically increment a shared counter. Procedure `Acquire` (lines 22–28) acquires the lock and procedure `Release` (lines 29–34) releases the lock. Both procedures use a primitive atomic action `CAS` (compare-and-swap) defined on lines 10–14 with two parameters—`old_b` and `new_b`. This action compares the value of a global variable `b` to `old_b`. If they are equal, `b` is set to `new_b` and `true` is returned, otherwise, `b` is not modified and `false` is returned. `Acquire` attempts to set `b` from `false` to `true` repeatedly via recursive call to itself (line 28) until it succeeds. `Release` sets `b` back to `false` from `true`.

Procedure `Incr` (lines 16–21) atomically increments the global variable `count` by acquiring the lock, reading `count` into a local variable `t` by calling `Read` (lines 35–39), writing `t+1` back to `count` by calling `Write` (lines 40–43), and finally releasing the lock. We prove that `Incr` implements an atomic increment via a sequence of two refinement steps.

The first step abstracts the procedures `Acquire`, `Release`, `Read`, and `Write` into atomic actions `AcquireSpec`, `ReleaseSpec`, `ReadSpec`, and `WriteSpec`, respectively. These atomic actions, defined in the third column of Figure 3.2, provide an explicit specification of the locking protocol for accessing the shared variable `count`. The specification of these actions requires the introduction of (1) a local parameter `tid` containing the unique id of the thread executing the code, and (2) a global variable `l` whose value is either `None` when the lock is not held or `Some(tid)` when the lock is held by thread `tid`. The second step uses these atomic actions to abstract `Incr` to an atomic action that increments `count` by 1.

There are two challenges in the first refinement proof. First, the lock implementation is defined using the concrete Boolean variable `b`, whereas the lock specification is defined using the logical lock variable `l`. Second, the implementation of `Acquire` is recursive, which is technically challenging for refinement reasoning. The solution to the first problem is to *introduce* `l` and *hide* `b` during the refinement proof. To introduce `l` into the concrete program, it is updated appropriately when `Acquire` (line 27) and `Release` (line 34) complete successfully. Furthermore, the relationship between the variables `b` and `l` is captured by the yield invariant `LockInv` (lines 7–8) which is used in the precondition and postcondition of `Acquire` and `Release`. The solution to the second problem is a powerful rule for refinement reasoning, described in Section 3.4, which allows the recursive call to `Acquire` on line 28 to be replaced by a call to the specification `AcquireSpec` while modularly proving that the body of `Acquire` refines `AcquireSpec`.

```
1  // Concrete global variables
2  var b: bool // false
3  var count: int
4  // Abstract global variable
5  var l: Option⟨Tid⟩ // None
6  // Supporting invariant
7  invariant LockInv()
8    b ⟺ (l ≠ None)
9  // Primitive actions
10 action CAS(old_b, new_b: bool)
11 returns (success: bool)
12   success := b = old_b
13   if (success)
14     b := new_b
15 // Atomic increment
16 procedure Incr(linear tid: Tid)
17 preserves LockInv()
18   call Acquire(tid)
19   call t := Read(tid) || LockInv()
20   call Write(tid, t+1) || LockInv()
21   call Release(tid)

22 procedure Acquire(
23   linear tid: Tid)
24 refines AcquireSpec
25 preserves LockInv()
26   exec t := CAS(false, true)
27   if (t) l := Some(tid)
28   else call Acquire(tid)
29 procedure Release(
30   linear tid: Tid)
31 refines ReleaseSpec
32 preserves LockInv()
33   exec CAS(true, false)
34   l := None
35 procedure Read(
36   linear tid: Tid)
37 returns (v: int)
38 refines ReadSpec
39   v := count;
40 procedure Write(
41   linear tid: Tid, v: int)
42 refines WriteSpec
43   count := v;

44 action AcquireSpec(
45   linear tid: Tid)
46   assume l = None
47   l := Some(tid)
48 action ReleaseSpec(
49   linear tid: Tid)
50   assert l = Some(tid)
51   l := None
52 action ReadSpec(
53   linear tid: Tid)
54 returns (v: int)
55   assert l = Some(tid)
56   v := count
57 action WriteSpec(
58   linear tid: Tid, v: int)
59   assert l = Some(tid)
60   count := v
```

Figure 3.2: Spin lock to illustrate refinement of atomic actions.

```
procedure Incr(linear tid: Tid)
refines IncrSpec
  exec AcquireSpec(tid)
  exec t := ReadSpec(tid)
  exec WriteSpec(tid, t+1)
  exec ReleaseSpec(tid)

action IncrSpec()
  count := count + 1
```

To set up the second refinement proof, the procedure calls in the body of `Incr` are replaced by invocations of the corresponding abstract atomic actions (as shown on the right here). The rewritten body of `Incr` is preemption-free; a yield may occur only at the beginning or the end. This assumption is justified by a commutativity analysis based on the observation that `AcquireSpec` is a right mover, `ReleaseSpec` is a left mover, and `ReadSpec` and `WriteSpec` are both movers [47]. Proving these mover types requires that the `tid` input parameters of two concurrent actions are distinct, which is specified by the *`linear`* annotation. In addition to encoding distinctness of values, linear variables can be used for encoding disjointness of permissions associated with values. We present an example illustrating permissions in Section 3.2.3 and a detailed technical description in Section 3.4.

For the prove that procedure `Incr` refines the action `IncrSpec`, which increments `count` atomically, we do not need the invariant `LockInv` anymore; in fact we do not need any invariant. Furthermore, the local parameter `tid` and the global variable `l` are no longer needed in the program and can be hidden. Hiding local variables is a novel feature of the refinement method described in this paper. The capability to introduce and subsequently hide global and local variables allows us to chain a sequence of refinement steps, localizing the use of variables to the parts of the proof that need them.

3.2.3 Linear Interfaces

Figure 3.3 shows a synchronization protocol extracted from a verified concurrent garbage collector [56]. There are `N` mutator threads (procedure `Mutator` on line 31) numbered from `1` to `N`, and one collector thread (procedure `Collector` on line 41) with ID `0`. The protocol ensures that no mutator accesses memory (line 40) concurrently while the collector is doing a root scan (line 47) using barrier synchronization. Before the collector runs, it sets the Boolean variable `barrierOn` to `true` (line 43) and waits until the integer variable

```
1  datatype Perm = Left(int) | Right(int)
2  function linear C1(i: int) : Set⟨Perm⟩ =
3    {Left(i), Right(i)}
4  function linear C2(ids: Set⟨int⟩) : Set⟨Perm⟩ =
5    {Left(i) | i ∈ ids}
6  function linear C3(p: Perm) : Set⟨Perm⟩ =
7    {p}
8  const N: int // positive
9  var barrierOn: bool // false
10 var barrierCounter: int // N
11 var linear mutatorsInBarrier: Set⟨int⟩ // ∅
12 // Primitive actions
13 action IsBarrierOn() returns (b: bool)
14   b := barrierOn
15 action EnterBarrier(linear_in i: int)
16 returns (linear_out p: Perm)
17   assert i ∈ [1..N]
18   mutatorsInBarrier := mutatorsInBarrier + {i}
19   barrierCounter := barrierCounter - 1
20   p := Right(i)
21 action WaitForBarrierRelease
22     (linear_in p: Perm, linear_out i: int)
23   assert p = Right(i) ∧ i ∈ mutatorsInBarrier
24   assume ¬barrierOn
25   mutatorsInBarrier := mutatorsInBarrier - {i}
26   barrierCounter := barrierCounter + 1
27 action SetBarrier(b: bool)
28   barrierOn := b
29 action WaitBarrier()
30   assume barrierCounter = 0
31 procedure Mutator(linear i: int)
32 requires i ∈ [1..N]  preserves BarrierInv()
33   var b: bool, p: Perm
34   exec b := IsBarrierOn()
35   if (b)
36     call BarrierInv()
37     exec p := EnterBarrier(i)
38     call BarrierInv() || MutatorInv(p, i)
39     exec WaitForBarrierRelease(p, i)
40   // access memory here
41 procedure Collector(linear i: int)
42 requires i = 0  preserves BarrierInv()
43   exec SetBarrier(true)
44   call BarrierInv() || CollectorInv(i, false)
45   exec WaitBarrier()
46   call BarrierInv() || CollectorInv(i, true)
47   // do root scan here
48   assert mutatorsInBarrier = [1..N]
49   exec SetBarrier(false)
50 // Supporting invariants
51 invariant BarrierInv()
52   mutatorsInBarrier ⊆ [1..N] ∧
53   size(mutatorsInBarrier) + barrierCounter = N
54 invariant MutatorInv(linear p: Perm, i: int)
55   p = Right(i) ∧ i ∈ mutatorsInBarrier
56 invariant CollectorInv(linear i: int, done: bool)
57   i = 0 ∧ barrierOn ∧
58   (done ⟹ mutatorsInBarrier = [1..N])
```

Figure 3.3: Barrier synchronization to illustrate linear interfaces.

`barrierCounter` gets `0` (line 45). Before a mutator accesses memory, it reads `barrierOn` (line 34). If `false`, the mutator goes ahead. Otherwise, it signals to the collector by decrementing `barrierCounter` (line 37) and waits for `barrierOn` to be reset to `false` (line 39).

This example declares both global and local *linear variables* (specified by `linear`, `linear_in`, `linear_out`). Every linear variable—or more precisely, its current value—is assigned a set of *permissions* of type `Perm` according to the *collector functions* `C1`, `C2`, and `C3`. A linear integer `i` holds both `Left(i)` and `Right(i)`, a set of integers holds the corresponding `Left` permissions, and a `Perm` value holds itself. Note that `Perm` is not special; any value can be a permission. For every program location we can compute the set of *available* linear variables. For example, when a mutator enters the barrier (line 37), `i` becomes *unavailable* because the permission `Left(i)` is transferred to the ghost variable `mutatorsInBarrier`. Then `i` becomes available again after exiting the barrier (line 39). Global linear variables (`mutatorsInBarrier` here) are always available. Parameterized by the linear collectors, our linearity framework establishes the generic invariant that all permissions across all available linear variables are disjoint. Now suppose that some mutator `i` is at line 40, where it holds both of its permissions and in particular `Left(i)`, while the collector is at line 48, where `mutatorsInBarrier` holds all `Left` permissions and in particular `Left(i)`. This situation is impossible, since the linearity feature of RefPL ensures that a duplication of permissions is impossible.

The strength of linearity, which leads to a less tedious verification task, is that its invariant connects variables from different scopes, without the need to explicitly state

(and prove) this invariant. The programmer only provides a linearity specification which is checked automatically (see Section 3.4). The resulting guarantees can then be assumed "for free". In contrast, even stating a corresponding invariant requires the introduction of auxiliary global variables and helper invariants to connect them to local variables.

3.3 RefPL: Syntax and Semantics

In this section we present RefPL, a core programming language which is carefully designed to be (1) a minimal yet general modeling language to express concurrent programs, (2) able to express invariants over program executions, and (3) suitable for expressing (refinement-based) program transformations. RefPL focuses on interfaces for modular verification, while abstracting from detailed expression syntax and types.

Syntax. Figure 3.4 (top panel) summarizes the syntax of RefPL. We assume sets of *names* which we use to name actions (A), procedures (P, Q), yield invariants (Y), and statement labels (λ). A set of *variables* is partitioned into *global* and *local variables*, and a *store* σ is a partial map from variables to *values*. We write $\sigma' \subseteq \sigma$ if σ is an extension of σ', $\sigma|_V$ for the restriction of σ to V, $\sigma[\sigma']$ for the store that is like σ' on $\mathrm{dom}(\sigma')$ and otherwise like σ, and $g{\cdot}\ell$ for the combination of a *global* and *local store*. A *program* consists of a finite set of global variables gs, a partial map as from action names to actions, and a partial map ps from procedure names to procedures. Both actions and procedures have an interface of *input variables* I and *output variables* O, and procedures have additional *local variables* L. A *(gated atomic) action* [41, 71] consists of a *gate* ρ and a *transition relation* τ. The gate is a set of stores (i.e., a predicate) over $gs \cup I$. Executing the action in a state that does not satisfy the gate fails the execution. Otherwise, every transition $(\sigma, \sigma', \Omega)$ in τ describes a possible atomic state transition from σ (over $gs \cup I$) to σ' (over $gs \cup O$), together with the creation of new asynchronous threads according to a set of *pending asyncs* Ω; every pending async $(\ell, P) \in \Omega$ is turned into a new thread that executes procedure P with input store ℓ. A *procedure* consists of a *statement* s that is composed of standard control-flow commands and two call commands: `exec` to invoke actions and `call` for the parallel invocation of multiple procedures. Every entry in the invocation sequence of a `call` is called an *arm* of the call, and the *label* λ is used to attach specification information to the call. Parameter passing is expressed using an *input map* ι from the callee's formals I to the caller's actuals $I \cup O \cup L$, and an injective *output map* o from the callee's formals O to the caller's actuals $O \cup L$. Input variables are immutable, since they are not mapped to by output maps and the variables of a procedure are not modified anywhere else. Output and local variables of a procedure are initialized to the default value ☣. In RefPL, loops are modeled using recursion, and conditional statements are modeled using nondeterministic branching ($*$) and actions that assume the branching condition.

Type checking. For a program we require that (1) the action name in an `exec` statement is in $\mathrm{dom}(as)$, (2) the procedure names in a `call` statement are in $\mathrm{dom}(ps)$, and the actual outputs of all arms are disjoint from each other and all actual inputs, and (3) for every pending async (ℓ, P) in the transition relation of an action in $\mathrm{img}(as)$, $P \in \mathrm{dom}(ps)$ and $\mathrm{dom}(\ell)$ contains all inputs of P.

$$A \in \mathit{ActionName} \quad P, Q \in \mathit{ProcName} \quad Y \in \mathit{InvName} \quad \lambda \in \mathit{Label}$$

$$\begin{array}{rcll}
 & & \mathit{Val} & \ni \ \text{☣} \\
v & \in & \mathit{Var} & = \ \mathit{GVar} \cup \mathit{LVar} \\
g & \in & \mathit{GStore} & = \ \mathit{GVar} \rightharpoonup \mathit{Val} \\
\ell & \in & \mathit{LStore} & = \ \mathit{LVar} \rightharpoonup \mathit{Val} \\
\sigma & \in & \mathit{Store} & = \ \mathit{Var} \rightharpoonup \mathit{Val} \\
\rho & \in & \mathit{Gate} & = \ 2^{\mathit{Store}} \\
\tau & \in & \mathit{Trans} & = \ 2^{\mathit{Store} \times \mathit{Store} \times \mathit{PASet}} \\
\Omega & \in & \mathit{PASet} & = \ 2^{\mathit{LStore} \times \mathit{ProcName}} \\
\iota, o & \in & \mathit{IOMap} & = \ \mathit{LVar} \rightharpoonup \mathit{LVar}
\end{array}$$

$$\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & \mid \texttt{skip} \mid s\,;s \mid s * s \\
 & & \mid \texttt{call}_\lambda\ \overline{(P, \iota, o)} \mid \texttt{exec}\ (A, \iota, o) \\
I, O, L & \in & 2^{\mathit{LVar}} \\
\mathit{Action} & ::= & (I, O, \rho, \tau) \\
\mathit{Proc} & ::= & (I, O, L, s) \\
gs & \in & 2^{\mathit{GVar}} \\
as & \in & \mathit{ActionName} \rightharpoonup \mathit{Action} \\
ps & \in & \mathit{ProcName} \rightharpoonup \mathit{Proc} \\
\mathcal{P} \in \mathit{Prog} & ::= & (gs, as, ps)
\end{array}$$

$$\begin{array}{rcl}
\mathit{Inv} & ::= & (I, \rho) \\
\mathit{InvCall} & ::= & (Y, \iota) \\
ys & \in & \mathit{InvName} \rightharpoonup \mathit{Inv} \\
\mathit{pre}, \mathit{post} & \in & \mathit{ProcName} \rightharpoonup 2^{\mathit{InvCall}} \\
\mathit{inv} & \in & \mathit{Label} \rightharpoonup 2^{\mathit{InvCall}} \\
\mathcal{Y} & ::= & (ys, \mathit{pre}, \mathit{post}, \mathit{inv})
\end{array}
\qquad
\begin{array}{rcl}
lg & \in & 2^{\mathit{GVar}} \\
li & \in & (\mathit{ActionName} \cup \mathit{ProcName} \cup \mathit{InvName}) \\
 & & \quad \times \{\triangleright, \triangleleft\} \rightharpoonup 2^{\mathit{LVar}} \\
lo & \in & (\mathit{ActionName} \cup \mathit{ProcName}) \rightharpoonup 2^{\mathit{LVar}} \\
lc & \in & \mathit{Val} \to 2^{\mathit{Val}} \\
\mathcal{L} & ::= & (lg, li, lo, lc)
\end{array}$$

$$\begin{array}{rcl}
\mathit{ref} & \in & \mathit{ProcName} \rightharpoonup \mathit{ActionName} \\
\mathit{mark} & \in & \mathit{Label} \rightharpoonup \{\square, \blacksquare\} \cup \mathbb{N} \\
\mathcal{R} & ::= & (\mathit{ref}, \mathit{mark})
\end{array}$$

$$\begin{array}{rcl}
f & ::= & (P, \ell, s) \\
t & ::= & \mathsf{Lf}\ f \mid \mathsf{Nd}\ f\ \bar{t} \\
\mathcal{T} & ::= & \{t, \ldots, t\} \\
c & ::= & (g, \mathcal{T}) \mid \lightning
\end{array}
\qquad
\begin{array}{rcl}
SC & ::= & \bullet_s \mid SC\,;s \\
TC & ::= & \bullet_t \mid \mathsf{Nd}\ f\ \bar{t}\,TC\bar{t} \\
PC & ::= & \{\,TC\} \uplus \mathcal{T} \\
LC & ::= & PC[\mathsf{Lf}\ (P, \bullet_\ell, SC)]
\end{array}$$

$$\text{for } ps(Q) = (I, O, L, s) \text{ let}$$
$$\mathit{init}(Q, \ell) = (Q, \ell|_I \cup [v \mapsto \text{☣}]_{v \in O \cup L}, s)$$

$$\textbf{(call)} \quad \begin{array}{l}(g, PC[\mathsf{Lf}\ (P, \ell, SC[\texttt{call}_\lambda\ \overline{(Q_i, \iota_i, o_i)}])]) \Rightarrow \\ (g, PC[\mathsf{Nd}\ (P, \ell, SC[\texttt{call}_\lambda\ \overline{(Q_i, \iota_i, o_i)}])\ \overline{\mathsf{Lf}\ \mathit{init}(Q_i, \ell \circ \iota_i)}])\end{array}$$

$$\textbf{(return)} \quad \begin{array}{l}(g, PC[\mathsf{Nd}\ (P, \ell, SC[\texttt{call}_\lambda\ \overline{(Q_i, \iota_i, o_i)}])\ \overline{\mathsf{Lf}\ (Q_i, \ell_i, \texttt{skip})}]) \Rightarrow \\ (g, PC[\mathsf{Lf}\ (P, \ell[\overline{\ell_i \circ o_i^{-1}}], SC[\texttt{skip}])])\end{array}$$

$$\textbf{(exec)}\ \mathit{as}(A) = (\_, \_, \rho, \tau) \quad \tilde{g} \subseteq g \quad (\tilde{g} \cdot (\ell \circ \iota), \hat{g} \cdot \hat{\ell}, \Omega) \in \rho \circ \tau$$
$$g' = g[\hat{g}] \quad \ell' = \ell[\hat{\ell} \circ o^{-1}] \quad \mathcal{T}' = \{\mathsf{Lf}\ \mathit{init}(Q, \ell'') \mid (\ell'', Q) \in \Omega\}$$
$$\overline{(g, PC[\mathsf{Lf}\ (P, \ell, SC[\texttt{exec}\ (A, \iota, o)])]) \Rightarrow (g', PC[\mathsf{Lf}\ (P, \ell', SC[\texttt{skip}])] \uplus \mathcal{T}')}$$

$$\textbf{(fail)}\ \frac{\mathit{as}(A) = (\_, \_, \rho, \_) \quad \neg \exists \tilde{g} \subseteq g : \tilde{g} \cdot (\ell \circ \iota) \in \rho}{(g, LC[\ell][\texttt{exec}\ (A, \iota, o)]) \Rightarrow \lightning} \qquad \textbf{(choice)}\ \frac{s' \in \{s_1, s_2\}}{(g, LC[\ell][s_1 * s_2]) \Rightarrow (g, LC[\ell][s'])}$$

$$\textbf{(skip)}\ (g, LC[\ell][\texttt{skip}\,; s]) \Rightarrow (g, LC[\ell][s]) \qquad \textbf{(stop)}\ (g, \{\mathsf{Lf}\ (\_, \texttt{skip})\} \uplus \mathcal{T}) \Rightarrow (g, \mathcal{T})$$

Figure 3.4: The programming language RefPL: syntax (top panel), proof annotations (middle panel), and operational semantics (bottom panel).

Semantics. Figure 3.4 (bottom panel) presents the operational semantics of RefPL, a transition relation $\Rightarrow$ over *configurations* that consist of a global store over gs and a finite multiset of threads. Each thread is a tree (which generalizes a call stack); a `call` statement creates new leaf nodes (`Lf`) and blocks the caller in an internal node (`Nd`) until all arms of the parallel call finish. Each tree node contains a *frame* (P, ℓ, s) that represents the current state of a procedure P during execution: ℓ is the procedure's current local store and s is a statement that remains to be executed. In the definition of $\Rightarrow$ we use several evaluation contexts that have a unique hole $\bullet$; filling the hole is denoted by $\cdot[\cdot]$. In particular, $SC[s]$ is a statement with s in evaluation position, and $PC[t]$ is a multiset of thread trees where t is a subtree in one of these trees. The operator $\circ$ means function or relation composition.

Atomic actions (invoked through the `exec` command) execute directly in the context of the caller; inline, if you will. If the current store does not satisfy the gate of an executed action, the execution stops in the *failure configuration* $\lightning$. It is important to appreciate the generality of atomic actions. First, they can represent atomic operations at an arbitrary level of granularity, from fine-grained low-level operations (e.g., as implemented in hardware) to coarse-grained summaries (e.g., obtained as part of a layered proof). Second, the notion of pending asyncs subsumes the need for a dedicated asynchronous call statement, and enables advanced proof techniques for asynchronous programs [71, 68]. Finally, all accesses to global variables are confined to atomic actions.

We distinguish between the *preemptive semantics* and the *cooperative semantics* of a program. The preemptive semantics $\Rightarrow$ defines the standard fine-grained behaviors of a concurrent program, where a context switch can happen at any time. A program should be proved correct under its preemptive semantics. However, for reasoning purposes we consider a cooperative semantics, where context switches only happen at procedure calls and returns. We call these locations *yields*. The justification for reducing reasoning about preemptive semantics to cooperative semantics is outside the scope of this paper (Civl uses commutativity reasoning and a reduction argument).

A leaf node `Lf` $(P, \_, s)$ is *yielding*, if it denotes the *entry* or *exit* of procedure P, i.e., if $ps(P) = (\_, \_, \_, s)$ or $s =$ `skip`. A configuration is *yielding* if all leaves are yielding, and *cooperative* if at most one leaf is not yielding. Then the cooperative semantics is given by restricting $\Rightarrow$ to cooperative configurations. Notice that the configuration after an `exec` might be non-yielding. Thus, under cooperative semantics the pending asyncs created by `exec` can only start executing once the caller reaches the next yield. We note that arbitrary yields can be modeled with "empty" parallel calls (i.e., a `call` with no arms).

A *yield-to-yield fragment* $\{P \,|\, \kappa_1\}\, \overline{e}\, \{\kappa_2\}$ of a procedure P is any sequence of `exec` statements $\overline{e}$ that forms a path in P from κ_1 to κ_2, where κ_1 and κ_2 are either `call` statements, $\bot$, or $\top$ ($\kappa_1 = \bot$ for procedure entries; $\kappa_2 = \top$ for procedure exits). For example, procedure `Acquire` in Figure 3.2 has three yield-to-yield fragments: `(A1)` entry/successful `CAS`/`then` branch/exit, `(A2)` entry/failed `CAS`/call in the `else` branch, and `(A3)` call in the `else` branch/exit (i.e., an "empty" fragment). Let $Gate(\overline{e})$ be the set of stores from which executing $\overline{e}$ cannot fail, and let $Trans(\overline{e})$ be the set of tuples $(\sigma, \sigma', \Omega)$ where executing $\overline{e}$ from store σ can result in σ' with all created pending asyncs collected in Ω. We define a reduced transition relation $\Rrightarrow$ over yielding configurations, such that $c \Rrightarrow c'$ if and only if there are cooperative but non-yielding configurations $(c_i)_{1 \le i \le n \wedge n \ge 0}$ with $c \Rightarrow c_1 \Rightarrow \ldots \Rightarrow c_n \Rightarrow c'$. Thus, every step in $\Rrightarrow$ corresponds to the execution of a yield-to-yield fragment under cooperative semantics.

3.4 Abstracting RefPL Programs

This section presents a proof rule for transforming a concurrent program $\mathcal{P}$ into a concurrent program $\mathcal{P}'$ such that there is a simulation between the cooperative executions of $\mathcal{P}$ and $\mathcal{P}'$. The transformation comprises *variable hiding* ($\mathcal{P}'$ has fewer global and local variables than $\mathcal{P}$) and *procedure abstraction* (procedures in $\mathcal{P}$ are summarized to atomic actions in $\mathcal{P}'$). Our proof rule takes as input a *yield specification* $\mathcal{Y}$, a *linearity specification* $\mathcal{L}$, and a *refinement specification* $\mathcal{R}$ (see Figure 3.4), and decomposes the refinement verification problem as follows.

$$\frac{\mathit{Linearity}(\mathcal{P},\mathcal{Y},\mathcal{L}) \quad \mathit{Safety}(\mathcal{P},\mathcal{Y},\mathcal{L}) \quad \mathit{Refinement}(\mathcal{P},\mathcal{Y},\mathcal{L},\mathcal{R},\mathcal{P}')}{\mathcal{Y},\mathcal{L},\mathcal{R} \vdash \mathcal{P} \rightsquigarrow \mathcal{P}'}$$

The yield specification declares yield invariants and attaches them to program locations, and the linearity specification declares linear interfaces and sets up a permission discipline (Section 3.4.1). The *Linearity* judgment (Section 3.4.2) ensures that the linear interfaces of procedures, actions, and invariants in $\mathcal{P}$ are valid, which establishes a linear disjointness property. The *Safety* judgment (Section 3.4.3) ensures that preconditions, postconditions, and invariants in $\mathcal{P}$ are valid and interference-free, which captures reachability information in $\mathcal{P}$. Note that *Linearity* and *Safety* interact, as yield invariants can have a linear interface and safety checking assumes the guarantees of linearity checking. In our proof rule, the guarantees of *Linearity* (Lemma 1) and *Safety* (Lemma 2) establish the context for refinement checking. However, we stress that these guarantees are useful on their own, independent of refinement. The refinement specification (Section 3.4.4) declares how $\mathcal{P}$ is converted to $\mathcal{P}'$, and the *Refinement* judgment ensures that every execution of $\mathcal{P}$ is simulated by an execution of $\mathcal{P}'$ (Theorem 4). In Section 3.5 we show how all of our obligations are implemented in practice.

3.4.1 Yield Invariants and Linear Interfaces

RefPL supports *yield invariants* of the form (I,ρ), where I are input variables and ρ is a gate over $gs \cup I$. In a yield specification $\mathcal{Y} = (ys, pre, post, inv)$, the map ys assigns invariant names to yield invariants, such that invariants can be "invoked" by name—similar to actions and procedures—by supplying an input map ι. We will write φ and ψ for sets of such *invariant calls*, and $\sigma \models \varphi$ to denote that store σ satisfies φ, i.e., $g{\cdot}\ell \models \varphi \iff \forall (Y,\iota) \in \varphi\ \exists \hat{g} \subseteq g : \hat{g}{\cdot}(\ell \circ \iota) \in ys(Y).\rho$. Then invariant calls are assigned to program locations as follows: $pre(P)$ are the *preconditions* that must hold on entry to procedure P, $post(P)$ are the *postconditions* that must hold on exit from procedure P, and $inv(\lambda)$ are the invariants that must hold at calls labeled with λ. These are the yield locations in the cooperative semantics, under which we will show the invariants correct and stable under interference.

RefPL supports *linear permissions* to enhance local reasoning. The core idea of linearity is to identify a subset of *(linear) available variables* among all variables in all frames of a configuration. Every value stored in an available variable is mapped to a set of values called *permissions*, with the desired property that the values in available variables are mapped to disjoint permissions. This disjointness property can then be used as free assumption in other verification conditions.

In a linearity specification $\mathcal{L} = (lg, li, lo, lc)$, the *linear global variables* lg are a subset of gs, which are always available. For every action/procedure/invariant name X, $li(X, \triangleright)$ and $li(X, \triangleleft)$ are subsets of its input variables called *linear-in* and *linear-out*, respectively. The linear-ins expect to receive from an available actual parameter, while the linear-outs ensure that their actual parameter will be available upon return. An input variable can be both linear-in and linear-out (which we assume for all invariants). For every action/procedure name X, its *linear outputs* $lo(X)$ are a subset of its output variables, such that the receiving actual return parameters become available when X returns. For example, in Figure 3.3 the global variable `mutatorsInBarrier` is linear, procedure `Mutator` and yield invariant `CollectorInv` have a linear (linear-in and linear-out) input `i`, action `EnterBarrier` has linear-in input `i` and linear output `p`, and `WaitForBarrierRelease` has a linear-in input `p` and linear-out input `i`. The permissions assigned to an available variable are determined by a *linear collector* function lc, which is a flexible mechanism to encode various permission disciplines. For convenience, we lift lc to collect all permissions of a set of variables V in store σ, i.e., $lc(\sigma, V) = \biguplus_{v \in V} lc(\sigma(v))$. A simple example of a collector function that expresses unique identifiers (as needed in Figure 3.2) would return the singleton set $\{\texttt{tid}\}$ for a thread identifier variable `tid`. Figure 3.3 shows a more advanced usage, where the definition of lc is split across the functions `C1`, `C2`, and `C3` (see Section 3.2.3).

3.4.2 Linearity

Let us assign to every (sub)statement s in $\mathcal{P}$ a *linear type* $\substack{in \\ out}$, written as $s : \substack{in \\ out}$, where in/out is the set of local variables available directly before/after executing s. Based on the linear interfaces in li and lo, the most general linear types can be inferred, but for simplicity we assume all types to be given and define a type checker below. Since linear types annotate each program location with available variables, we can define the collection of linear permissions over a configuration $c = (g, \mathcal{T})$ as $lc(c) = lc(g, lg) \uplus \big(\biguplus_{(P,\ell,s:\substack{in \\ out})} lc(\ell, in)\big)$, where $(P, \ell, s : \substack{in \\ out})$ ranges over all frames in all nodes of $\mathcal{T}$. Then the *linear disjointness property* for a configuration c is $IsSet(lc(c))$, where $IsSet(\cdot)$ states that a multiset does not contain duplicates. We call such a configuration $\mathcal{L}$-*valid*. The $Linearity(\mathcal{P}, \mathcal{Y}, \mathcal{L})$ judgment comprises a semantic check on actions and a syntactic check on procedures, which ensures the preservation of the linear disjointness property as follows.

Lemma 1. *Let c be an $\mathcal{L}$-valid configuration of $\mathcal{P}$. If $c \Rightarrow c'$ then c' is $\mathcal{L}$-valid.*

Essentially, an execution starts with a set of permissions and redistributes these in every step. The permissions can stay the same or decrease, but never increase.

Linear action checking. All state updates (other than parameter passing) are confined to atomic actions. We need to ensure that the outgoing permissions of an action are always a subset of the incoming permissions. Thus, for every $A \in \mathrm{dom}(as)$ with $as(A) = (\_, \_, \rho, \tau)$ we check

$$(g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \rho \circ \tau \wedge inPerm = \big(lc(g, lg) \uplus lc(\ell, li(A, \triangleright))\big) \wedge IsSet(inPerm) \implies \Big(lc(g', lg) \uplus lc(\ell, li(A, \triangleleft)) \uplus lc(\ell', lo(A)) \uplus \big(\biguplus_{(\ell'', P) \in \Omega} lc(\ell'', li(P, \triangleright))\big)\Big) \subseteq inPerm.$$

Starting with a set of permissions in the linear globals and linear-in inputs, the action can redistribute these permissions among the linear globals, its linear-out inputs and

$$\frac{out \subseteq in}{\texttt{skip} : {}^{in}_{out}} \qquad \frac{s_1 : {}^{in}_{out} \quad s_2 : {}^{out}_{out'}}{s_1\,;s_2 : {}^{in}_{out'}} \qquad \frac{s_1 : {}^{in}_{out_1} \quad s_2 : {}^{in}_{out_2}}{s_1 * s_2 : {}^{in}_{out_1 \cap out_2}}$$

$$\frac{\iota(li(A,\triangleright)) \subseteq in \quad out \subseteq \left(in \setminus \iota(li(A,\triangleright))\right) \uplus \iota(li(A,\triangleleft)) \uplus o(lo(A))}{\texttt{exec}\ (A,\iota,o) : {}^{in}_{out}}$$

$$\frac{\left(\biguplus_i \iota_i(li(P_i,\triangleright))\right) \uplus \left(\biguplus_{(Y,\iota)\in inv(\lambda)} \iota(li(Y,\triangleright))\right) \subseteq in \quad out \subseteq (in \setminus \biguplus_i \iota_i(li(P_i,\triangleright))) \uplus (\biguplus_i \iota_i(li(P_i,\triangleleft))) \uplus (\biguplus_i o_i(lo(P_i)))}{\texttt{call}_\lambda\ \overline{(P_i,\iota_i,o_i)} : {}^{in}_{out}}$$

Figure 3.5: Linear type checking.

linear outputs, and the linear-ins of pending asyncs, but permissions cannot appear out of thin air. Notice that this check depends on the user-provided linear collector function *lc*. For example, consider action `EnterBarrier` in Figure 3.3. The linear-in input `i` holds the permissions `Left(i)` and `Right(i)` on entry (cf. collector `C1`). By adding `i` to `mutatorsInBarrier` we hand over the permission `Left(i)` (cf. collector `C2`), and by the assignment to the linear output `p` we hand over the permission `Right(i)` (cf. collector `C3`). Thus, the set of permissions in `mutatorsInBarrier` and `i` before is the same as the permissions in `mutatorsInBarrier` and `p` after executing `EnterBarrier`.

Linear type checking. Now that we can trust the linear interfaces of actions, we need to ensure that the linear types in procedures "add up" w.r.t. control flow and parameter passing. For every $P \in \mathrm{dom}(ps)$ with body $s : {}^{in}_{out}$ we require $in = li(P,\triangleright)$, $out = li(P,\triangleleft) \cup lo(P)$, and a derivation of $s : {}^{in}_{out}$ according to the rules in Figure 3.5, where $\iota(V)$ means $\biguplus_{v\in V} \iota(v)$. For example, in procedure `Mutator` in Figure 3.3 the linear input parameter `i` becomes unavailable at line 37, where it is passed as linear-in. However, this call makes the local variable `p` available, such that it can be passed as linear-in to the call on line 39. This call also passes `i` as linear-out input, which makes `i` available again on line 40.

3.4.3 Safety

In a yielding configuration $(g,\mathcal{T})$, every frame (P,ℓ,s) in $\mathcal{T}$ is associated with a set of invariant calls φ as follows: $\varphi = pre(P)$ if s is the entry of P, $\varphi = post(P)$ if s is `skip` (the exit of P), or $\varphi = inv(\lambda)$ if s is blocked at a call labeled with λ. If $g{\cdot}\ell \models \varphi$ holds in every frame, then we call the configuration $\mathcal{Y}$-*valid*. To show that this property is preserved across the execution of a yield-to-yield fragment (i.e, a step in $\Rrightarrow$), the *Safety*$(\mathcal{P},\mathcal{Y},\mathcal{L})$ judgment is decomposed into two kinds of procedure-modular verification conditions: (1) a *sequential check* which ensures that the next φ in the executing frame is established, and (2) a *noninterference check* which ensures that the φ's in all other frames are preserved. Both checks weave in linearity to enhance local reasoning.

Lemma 2. *Let c be an $\mathcal{L}$-valid, $\mathcal{Y}$-valid configuration of $\mathcal{P}$. If $c \Rrightarrow c'$ then c' is $\mathcal{Y}$-valid.*

Floyd packages. For convenience, let $pre(\kappa)$ be the set of all invariants and preconditions of a `call` statement κ (and $post(\kappa)$ analogously):

$$pre(\texttt{call}_\lambda\ \overline{(Q_i, \iota_i, o_i)}) = inv(\lambda) \cup \left(\bigcup_i \{(Y, \iota_i \circ \iota) \mid (Y, \iota) \in pre(Q_i)\}\right)$$
$$post(\texttt{call}_\lambda\ \overline{(Q_i, \iota_i, o_i)}) = inv(\lambda) \cup \left(\bigcup_i \{(Y, (\iota_i \cup o_i) \circ \iota) \mid (Y, \iota) \in post(Q_i)\}\right)$$

For every yield-to-yield fragment $\{P \mid \kappa_1\}\,\overline{e}\,\{\kappa_2\}$ of $P \in \mathrm{dom}(ps)$ we define a *Floyd package* $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{\psi\}$, which contains the invariants φ and linear available variables ll before, and the invariants ψ after the yield-to-yield fragment:

$$(\varphi, ll) = \begin{cases} (pre(P) & , li(P, \triangleright)) & \text{if } \kappa_1 = \bot \\ (post(\kappa_1) & , out(\kappa_1)) & \text{if } \kappa_1 \neq \bot \end{cases}; \qquad \psi = \begin{cases} post(P) & \text{if } \kappa_2 = \top \\ pre(\kappa_2) & \text{if } \kappa_2 \neq \top \end{cases}.$$

Sequential checking. For every Floyd package $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{\psi\}$ we check

$$\left(\begin{array}{ll} ① & g{\cdot}\ell \models \varphi \\ ② & (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in Trans(\overline{e}) \\ ③ & IsSet(lc(g{\cdot}\ell, lg \cup ll)) \end{array}\right) \implies \left(\begin{array}{ll} ④ & g'{\cdot}\ell' \models \psi \\ ⑤ & \forall (\ell'', P) \in \Omega : g'{\cdot}\ell'' \models pre(P) \end{array}\right).$$

After ② executing $\overline{e}$ from a store with ③ disjoint permissions that ① satisfies φ, it must be the case that ④ ψ and ⑤ the preconditions of all created pending asyncs hold. Notice that we can assume all gates of atomic actions when executing $\overline{e}$. This is the case because yield invariants are not supposed to be strong enough to prove $\mathcal{P}$ safe. Their purpose is to establish the context for refinement checking.

Noninterference checking. For every Floyd package $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{\psi\}$ and every yield invariant $Y \in \mathrm{dom}(ys)$ we check

$$\left(\begin{array}{ll} ① & g{\cdot}\ell \models \varphi \wedge g{\cdot}\ell' \models Y \\ ② & (g{\cdot}\ell, g'{\cdot}\_, \_) \in Trans(\overline{e}) \\ ③ & IsSet(lc(g{\cdot}\ell, lg \cup ll) \uplus lc(\ell', li(Y, \triangleright))) \end{array}\right) \implies ④\ g'{\cdot}\ell' \models Y.$$

After ② executing $\overline{e}$ from a store with ③ disjoint permissions that ① satisfies both φ and Y, it must be the case that ④ Y still holds. A key ingredient that makes our yield invariants powerful is the possibility to pass parameters to them (ℓ' above, which is the same before and after executing $\overline{e}$), together with the possibility to give invariants a linear interface to include them in the disjointness assumption ③. The reuse of named, parameterized invariants that are inductive on their own facilitates ergonomic and modular proofs as well as a reduction in the number of noninterference checks compared to location invariants.

The example in Figure 3.3 uses three yield invariants. `BarrierInv` states a global property on `barrierCounter` and `mutatorsInBarrier`, `MutatorInv` states a property of mutators on line 38, and `CollectorInv` states a property of the collector at lines 44 and 46 (notice the difference in the Boolean parameter). The linear parameters to both `MutatorInv` and `CollectorInv` are essential to prove their noninterference. For example, linearity discharges all noninterference obligations of `CollectorInv` w.r.t. yield-to-yield fragments in procedure `Collector`; there cannot be two different available variables `i` both holding thread identifier 0. `CollectorInv` is also stable across the yield-to-yield fragments in procedure `Mutator`: by linearity, we know that `EnterBarrier` cannot execute if `mutatorsInBarrier` holds all mutator identifiers, and `WaitForBarrierRelease` is

blocked when `barrierOn` is `true`. As an example of a sequential check, observe that the invariants at line 44 together with `barrierCounter = 0` from executing `WaitBarrier` imply the invariants at line 46, in particular that `mutatorsInBarrier` holds all mutator identifiers.

3.4.4 Refinement

Recall that the goal of our proof rule is to transform a program $\mathcal{P} = (gs, as, ps)$ into a program $\mathcal{P}' = (gs', as', ps')$. So far, we showed how the two judgments $\mathit{Linearity}(\mathcal{P}, \mathcal{Y}, \mathcal{L})$ and $\mathit{Safety}(\mathcal{P}, \mathcal{Y}, \mathcal{L})$ establish properties on executions of $\mathcal{P}$, using a linearity specification $\mathcal{L}$ and yield specification $\mathcal{Y}$. In the remainder of this section we show how the $\mathit{Refinement}(\mathcal{P}, \mathcal{Y}, \mathcal{L}, \mathcal{R}, \mathcal{P}')$ judgment ties together $\mathcal{P}$ and $\mathcal{P}'$ using a refinement specification $\mathcal{R}$.

Consider an execution step $c \Rrightarrow c'$ of $\mathcal{P}$. We want to say that there is a representative step $\hat{c} \Rrightarrow \hat{c}'$ in $\mathcal{P}'$. Representative means that $\hat{c}$ and $\hat{c}'$ are abstract representations of c and c', respectively. We capture this notion in an *abstraction mapping* α, which maps every concrete configuration of $\mathcal{P}$ to an abstract configuration of $\mathcal{P}'$. Then the meaning of the judgment $\mathcal{L}, \mathcal{Y}, \mathcal{R} \vdash \mathcal{P} \rightsquigarrow \mathcal{P}'$ derived by our proof rule is expressed in the following theorem.

Theorem 4. *Let c be an $\mathcal{L}$-valid, $\mathcal{Y}$-valid configuration of $\mathcal{P}$. (1) If $c \Rrightarrow \lightning$ then $\alpha(c) \Rrightarrow \lightning$. (2) If $c \Rrightarrow c'$ then either $\alpha(c) = \alpha(c')$, $\alpha(c) \Rrightarrow \alpha(c')$, or $\alpha(c) \Rrightarrow \lightning$.*

The safety of $\mathcal{P}'$ should imply the safety of $\mathcal{P}$. Thus, (1) states that any failure in $\mathcal{P}$ is preserved in $\mathcal{P}'$. And (2) states that every step in $\mathcal{P}$ is matched with a (potentially stuttering) step or failure in $\mathcal{P}'$. Hence, $\mathcal{P}'$ can fail "more often" than $\mathcal{P}$, but otherwise "behaves like" $\mathcal{P}$.

Refinement specification. In a refinement specification $\mathcal{R} = (\mathit{ref}, \mathit{mark})$, the *refinement mapping ref* is a partial map from $\mathrm{dom}(ps)$ to $\mathrm{dom}(as')$. For every procedure $P \in \mathrm{dom}(\mathit{ref})$, we check that P is abstracted by action $A = \mathit{ref}(P)$. Since our refinement checks are procedure-modular, we require $\mathrm{dom}(\mathit{ref})$ to be closed under calls in ps (not including pending asyncs). In general, P executes multiple yield-to-yield fragments and possibly calls other procedures, while A executes in a single atomic step. Thus we need to ensure that exactly one yield-to-yield fragment in P behaves like A, while all other fragments have no visible side effect. We use a *marking function mark* to identify where A should happen in P. For every call statement with label λ, $\mathit{mark}(\lambda)$ is either $\square$ ("before"), $\blacksquare$ ("after"), or the index $i \in \mathbb{N}$ of some arm of the call. This means that we are still before A when the call returns, that we are already after A when reaching the call, or that arm i establishes A, respectively. Naturally, procedure entry and exit are marked with $\square$ and $\blacksquare$, respectively. Then the marks along every path of P must match the regular expression $\square^+\mathbb{N}^?\blacksquare^+$, which distinguishes two cases. (M1) No call is marked with an index $i \in \mathbb{N}$. Then some yield-to-yield fragment switches from $\square$ to $\blacksquare$, which we will check to behave like A. All other yield-to-yield fragments and calls on the path must have no side effect. (M2) Some call is marked with index $i \in \mathbb{N}$. We will check that arm i of this call behaves like A, while all other calls and yield-to-yield fragments on the path must have no side effect. Since we check *mark* per path, there are in general multiple occurrences of (M1) and (M2).

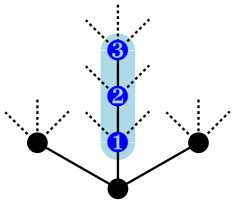
In Figure 3.2, the *ref* mapping is specified using the `refines` keyword. For example, procedure `Acquire` refines the atomic action `AcquireSpec`. The *mark* mapping is not explicitly specified, but we consider the call on line 28 to be marked with 1 (the index of its only arm). Then one path through `Acquire` is marked with $\square\blacksquare$ and the other one with $\square\,1\,\blacksquare$, both matching the regular expression above.

Program rewriting. The program $\mathcal{P} = (gs, as, ps)$ is rewritten into $\mathcal{P}' = (gs', as', ps')$ as follows. First, global variables can be hidden, such that $gs' \subseteq gs$. Second, new atomic actions can be added (for new abstractions of procedures) and unreferenced ones removed, but for $A \in \mathrm{dom}(as) \cap \mathrm{dom}(as')$ we require $as'(A) = as(A)$. Recall that an action can execute in any program that contains the referenced global variables and procedures. Third, $\mathrm{dom}(ps') = \mathrm{dom}(ps)$ and we rewrite every $ps(P) = (I, O, L, s)$ into $ps'(P) = (I', O', L', s')$ as follows. Local variables can be hidden, such that $I' \subseteq I \wedge O' \subseteq O' \wedge L' \subseteq L$. If $P \notin \mathrm{dom}(ref)$, then s' is like s, except that call arms (Q, ι, o) with $ps'(Q) = (I_Q, O_Q, \_, \_)$ turn into $(Q, \iota|_{I_Q}, o|_{O_Q})$, with the requirement $\mathrm{img}(o) \cap (O' \cup L') = \mathrm{img}(o|_{O_Q})$ that formal and actual outputs can only be hidden together. We denote this rewriting of a statement by $\alpha(s)$. If $P \in \mathrm{dom}(ref)$, then $s' = \texttt{exec}\,(ref(P), id(I'), id(O'))$, where $id(\cdot)$ is the identity mapping on a given set of variables. We denote this `exec` statement by $\alpha(P)$. Thus, procedures in $\mathrm{dom}(ref)$ remain in $\mathcal{P}'$, but with their bodies rewritten to a single `exec` to their abstraction. Clearly, the action interface $as' \circ ref(P) = (I', O', \_, \_)$ must match the procedure, and $L' = \varnothing$. Overall, $\mathcal{P}'$ must still typecheck, which ensures, e.g., that the remaining actuals in input/output maps were not hidden.

In the first refinement step of Section 3.2.2, where the procedures in the second column of Figure 3.2 are abstracted to the atomic actions in the third column, the global variable `b` is hidden. In the second refinement step, where procedure `Incr` is abstracted to action `IncrSpec`, the input parameter `tid` and the global variable `l` are hidden. Notice that, in order to chain together these two refinement steps, we performed an auxiliary rewriting step in procedure `Incr` that converted `call` statements to `exec` statements. CIVL automatically performs this transformation as part of a refinement step, justified by a commutativity argument we explained in Section 3.2.2. However, this rewriting is not formalized as part of our refinement rule in this paper.

Skip action. In the following we assume a special action *Skip* that has no inputs and outputs, does not modify global variables, and creates no pending asyncs. Formally, $as(Skip) = (\varnothing, \varnothing, \{\varepsilon\}, \{(\varepsilon, \varepsilon, \varnothing)\})$, where ε is the empty store. Observe that safety verification (i.e., showing that the failure configuration $\lightning$ is unreachable) is a special case of refinement, where all global and local variables are hidden, and all procedures are abstracted to *Skip*.

Abstraction mapping. Figure 3.6 defines the abstraction mapping α. In a given yielding configuration, we restrict the global store to gs' and drop all trees rooted in a node that refines *Skip*. The remaining nodes are traversed recursively, where frames with $P \notin \mathrm{dom}(ref)$ (nodes $\bullet$ on the right) are rewritten as expected. The interesting case is for nodes with $P \in \mathrm{dom}(ref)$, like node ❶ on the right. In this case, ❶ is turned into a leave (cutting off the remaining subtree) whose statement is either $\alpha(P)$ (the single `exec` of $ref(P)$) or `skip`. Intuitively, to match the



Abstraction of configuration

$$\alpha((g, \mathcal{T})) = (g|_{gs'}, \{\alpha(t) \mid t \in \mathcal{T} \wedge root(t) = P \wedge ref(P) \neq Skip\})$$

Abstraction of thread tree

For the definitions of $\alpha(s)$ and $\alpha(P)$, see program rewriting.

$$\ell|_P = \ell|_{I \cup O \cup L} \quad \text{if } ps'(P) = (I, O, L, \_)$$

$$\alpha(\mathsf{Lf}\ (P, \ell, s)) = \mathsf{Lf}\ (P, \ell|_P, \alpha(s)) \qquad \text{if } P \notin \mathrm{dom}(ref)$$

$$\alpha(\mathsf{Nd}\ (P, \ell, s)\ \bar{t}) = \mathsf{Nd}\ (P, \ell|_P, \alpha(s))\ \overline{\alpha(t)} \qquad \text{if } P \notin \mathrm{dom}(ref)$$

$$\alpha(\mathsf{Lf}\ (P, \ell, s)) = \mathsf{Lf}\ (P, \ell|_P, s') \quad s' = \begin{cases} \alpha(P) & \text{if } s \neq \texttt{skip} \\ \texttt{skip} & \text{if } s = \texttt{skip} \end{cases} \qquad \text{if } P \in \mathrm{dom}(ref)$$

$$\alpha(\underbrace{\mathsf{Nd}\ (P, \ell, \_)\ \_}_{t}) = \mathsf{Lf}\ (P, \ell'|_P, s') \quad s', \ell' = \begin{cases} \alpha(P), & \ell \quad \text{if } r(t) = \square \\ \texttt{skip}, & r(t) \quad \text{if } r(t) \neq \square \end{cases} \qquad \text{if } P \in \mathrm{dom}(ref)$$

Early-return computation

$$r(\mathsf{Lf}\ (P, \ell, s)) = \begin{cases} \square & \text{if } s \neq \texttt{skip} \\ \ell & \text{if } s = \texttt{skip} \end{cases}$$

$$r(\mathsf{Nd}\ (P, \ell, SC[\texttt{call}_\lambda\ \overline{(Q, \iota, o)}])\ \bar{t}) = \begin{cases} \square & \text{if } mark(\lambda) = \square \\ \ell & \text{if } mark(\lambda) = \blacksquare \\ \square & \text{if } mark(\lambda) = i \wedge r(t_i) = \square \\ \ell[r(t_i) \circ o_i^{-1}] & \text{if } mark(\lambda) = i \wedge r(t_i) \neq \square \end{cases}$$

Figure 3.6: Abstraction mapping from configurations of $\mathcal{P}$ to configurations of $\mathcal{P}'$.

concrete steps of P (in ❶ and its subnodes), the abstract configuration first stutters at $\alpha(P)$, then transitions to `skip` when the effect of $ref(P)$ happens, and then stutters at `skip` until the return from ❶. The delicate part is to determine if $ref(P)$ happened and to compute the local store for the abstract configuration. This is done by the *early-return function* r. The function recurses on the unique path of marked arms in calls, ❶–❷–❸ in our example, and either returns $\square$ (when "before $ref(P)$") or a local store ℓ (when "after $ref(P)$"). Suppose that ❶,❷,❸ have local stores ℓ_1, ℓ_2, ℓ_3, and that $r(❸) = \ell_3$. Then $r(❷)$ equals ℓ_2 updated with the return parameters from ℓ_3, say ℓ_2', and similarly $r(❶)$ equals ℓ_1 updated with the return parameters from ℓ_2', say ℓ_1', which is the local store for the abstract configuration. Thus, r performs "early" return parameter passing, even though we are still in the middle of executing procedures. To prove Theorem 4, our verification conditions below have to ensure that throughout subsequent concrete execution steps, $r(❶)$ remains ℓ_1'.

Refinement packages. In a procedure $P \in \mathrm{dom}(ref)$, the effect of the abstract action $ref(P)$ can happen either in a yield-to-yield fragment directly in P, or nested inside another called procedure. To handle (potentially recursive) procedure calls during refinement, we decompose the problem into procedure-modular checks. Recall that the marking function *mark* identifies yield-to-yield fragments and call arms in P that should behave like the abstract action $ref(P)$. Conversely, all other yield-to-yield fragments and call arms should have no side effect, which is to say that they should behave like *Skip*. Hence we have a refinement obligation for *every* yield-to-yield fragment and *every* call arm in P, where

refinement is either checked against $\mathit{ref}(P)$ or *Skip*. We capture all these refinement obligations uniformly in *refinement packages* of the form $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{A\}$, where P is the procedure we check refinement for, φ is a set of invariant calls and ll a set of available variables we can assume, $\overline{e}$ is an `exec` sequence denoting the effect we check refinement for, and A is the action we check refinement against.

(R1) Refinement packages for yield-to-yield fragments. For every procedure $P \in \mathrm{dom}(\mathit{ref})$ and yield-to-yield fragment $\{P \mid \kappa_1\}\,\overline{e}\,\{\kappa_2\}$ of P we define the refinement package $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{A\}$ where φ and ll are defined the same as for Floyd packages, and $A = \mathit{ref}(P)$ if $\mathit{mark}(\kappa_1) = \square$ and $\mathit{mark}(\kappa_2) = \blacksquare$, or $A = \mathit{Skip}$ otherwise. This case is rather straightforward. We proved the validity of φ and ll before the fragment, and need to check that the code $\overline{e}$ in the fragment behaves either like $\mathit{ref}(P)$ or `skip`.

(R2) Refinement packages for call arms. For every procedure $P \in \mathrm{dom}(\mathit{ref})$ and $\overline{\texttt{call}_\lambda\,(Q_i, \iota_i, o_i)} : {}^{\mathit{in}}_{\mathit{out}}$ in P, let $\varphi = \mathit{inv}(\lambda)$ and $ll = \mathit{in} \setminus \bigcup_i \iota_i(\mathit{li}(Q_i, \triangleright))$. At a call we know the validity of the invariants attached to the call and the availability of in minus the linear variables passed into the callees. Then for every arm (Q_i, ι_i, o_i), let $A_i = \mathit{ref}(P)$ if $\mathit{mark}(\lambda) = i$ or $A_i = \mathit{Skip}$ otherwise. Now the final missing ingredient for a refinement package $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{A_i\}$ for every arm i is the effect $\overline{e}$ for which we check refinement against A_i. To obtain a modular check, our solution is to use the abstract action specification of the callee Q_i. Formally, $\overline{e} = \texttt{exec}\,(B_i, \iota_i|_I, o_i|_O)$ for $B_i = \mathit{ref}(Q_i)$ with $\mathit{as}'(B_i) = (I, O, \_, \_)$. Recall that this is well-defined, since $\mathrm{dom}(\mathit{ref})$ is closed under calls. Notice that using the specification of a callee while checking the specification of a caller is akin to reasoning with procedure pre- and postconditions, where circular dependencies are resolved via induction on the nesting depth.

Recall (from the end of Section 3.3) that procedure `Acquire` in Figure 3.2 has three yield-to-yield fragments: `(A1)`, `(A2)`, `(A3)`. Each fragment induces an (R1)-type refinement package, where `(A1)` is checked against `AcquireSpec`, while both `(A2)` and `(A3)` are checked against *Skip*. Furthermore, the call on line 28 induces an (R2)-type refinement package against `AcquireSpec`.

Refinement checking. The $\mathit{Refinement}(\mathcal{P}, \mathcal{Y}, \mathcal{L}, \mathcal{R}, \mathcal{P}')$ judgment requires every refinement package $\{P \mid \varphi \mid ll\}\,\overline{e}\,\{A\}$ to be discharged as follows. Let $e = \texttt{exec}\,(A, \mathit{id}(I), \mathit{id}(O))$ for $\mathit{as}'(A) = (I, O, \_, \_)$ be the abstract effect we check refinement against, let $V = \mathit{gs}' \cup I' \cup O'$ for $\mathit{as}' \circ \mathit{ref}(P) = (I', O', \_, \_)$ be the non-hidden variables in the scope of the refinement package, and check

$$\left(\begin{array}{ll} ① & g{\cdot}\ell \models \varphi \\ ② & \mathit{IsSet}(\mathit{lc}(g{\cdot}\ell, \mathit{lg} \cup ll)) \end{array}\right) \implies \left(\begin{array}{ll} ③ & g{\cdot}\ell \in \mathit{Gate}(e) \implies g{\cdot}\ell \in \mathit{Gate}(\overline{e}) \\ ④ & (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \mathit{Gate}(e) \circ \mathit{Trans}(\overline{e}) \implies \\ & \exists g{\cdot}\hat{\ell}, \hat{g}'{\cdot}\hat{\ell}' : (\hat{g}{\cdot}\hat{\ell}, \hat{g}'{\cdot}\hat{\ell}', \Omega|_{\mathit{ref}}) \in \mathit{Trans}(e) \\ & \wedge\, g{\cdot}\ell|_V = \hat{g}{\cdot}\hat{\ell}|_V \wedge g'{\cdot}\ell'|_V = \hat{g}'{\cdot}\hat{\ell}'|_V \end{array}\right)$$

$$\text{where } \Omega|_{\mathit{ref}} = \{(\ell, Q) \in \Omega \mid \mathit{ref}(Q) \neq \mathit{Skip}\}.$$

We assume a store $g{\cdot}\ell$ that satisfies ① invariants and ② linear disjointness according to the refinement package. Then refinement consists of two parts, failure preservation and behavior preservation. First, ③ if $\overline{e}$ can fail in the concrete then e must also fail in the abstract. Second, ④ if e cannot fail in the abstract and $\overline{e}$ can transition to store $g'{\cdot}\ell'$ while creating pending asyncs Ω in the concrete, then there must be a matching transition of e in the abstract. Here matching means that e starts in a store $\hat{g}{\cdot}\hat{\ell}$ that agrees with $g{\cdot}\ell$ on

the non-hidden variables V, ends in a store $\hat{g}' \cdot \hat{\ell}'$ that agrees with $g' \cdot \ell'$ on V, and creates the same pending asyncs except the ones to procedures abstracted to *Skip*.

3.5 Implementation

Civl is a refinement-based verifier for concurrent programs built on top of the widely-used Boogie intermediate verification language. The Boogie [14] verifier provides infrastructure for compiling annotated sequential procedures into logical verification conditions whose validity is checked by a satisfiability-modulo-theories solver. Civl is implemented as an extension of Boogie, which takes as input an annotated layered concurrent program [70] (in a language whose core is RefPL), performs concurrency-specific type checking and static analyses, and then encodes all the verification conditions of its proof rule into a standard sequential Boogie program. Thus, Civl can be understood as a compiler that eliminates concurrency in a RefPL program by translating it down to a collection of sequential procedures, thus reusing the rest of the Boogie pipeline unchanged.

The open-source Civl verifier is a stable tool which is part of the master branch [2] and public release [1] of Boogie. Civl has over 100 regression tests comprising both realistic programs and microbenchmarks. There are many published papers [71, 115, 23, 91, 73] that describe nontrivial examples verified using Civl, most written by researchers other than the developers of Civl. The code in Civl is extensible; entirely new tactics for rewriting concurrent programs have been added to it [71, 68]. Finally, Civl is designed for interactive program development. It is fast and provides several command-line flags to focus verification on parts of the program. Civl has fine-grained error reporting including error traces, which attributes a verification failure to a particular check, local to a small part of the program. This helps the programmer to debug and iteratively improve both implementation and specification.

An early version of the Civl verifier was reported by Hawblitzel et al. [56]. The implementation of the techniques described in this paper has been done as part of the new design and implementation of Civl based on the framework of layered concurrent programs [70]. In the rest of this section, we will continue to use Civl to refer to our new implementation. We now present an overview of the different parts of the verifier.

Type checking. In addition to the standard type checking of a Boogie program, the Civl type checker performs several extra checks. First, it checks that the layer specifications [70] on program elements such as global and local variables, atomic actions, and procedures are correct. Second, it checks using a dataflow analysis that it is sufficient to reason about the safety of cooperative semantics. This analysis exploits mover type [47] annotations on atomic actions to reason that yield-to-yield code fragments satisfy the requirements of Lipton reduction [80]. It also generates logical verification conditions whose validity guarantee the correctness of the mover annotations on atomic actions.

Linearity checking. The Civl linearity checker implements the method described in Section 3.4.2 in two parts. First, it creates for each atomic action a sequential procedure which verifies that the multiset of outgoing permissions is a subset of the multiset of incoming permissions. We use the generalized array theory [36] to encode multisets, and the *IsSet* constraint in particular. Second, it type checks each procedure to compute the

set of available variables at each control location and to verify that linear interfaces of called procedures and atomic actions are used appropriately.

Safety checking. The Civl safety checker implements the method described in Section 3.4.3. Unlike the formal description which enumerates yield-to-yield code fragments, the implementation is efficient, encodes all code fragments in a RefPL procedure into a single sequential procedure with maximal sharing, and adds the safety checks by injecting instrumentation code and assertions into a cloned copy of the original procedure. To express the noninterference check, we add instrumentation variables that take snapshots of global and output variables at every yield. Furthermore, the generalized array theory is used here as well to record the pending asyncs created in a yield-to-yield code fragment, such that their preconditions can be checked.

Refinement checking. The Civl refinement checker implements the method described in Section 3.4.4. Similar to safety checking, the refinement checks are added as instrumentation to procedure copies. At every yield, snapshot variables (similar as for noninterference) are used to refer to the state at the previous yield when asserting the appropriate transition relation. Civl computes a representation of the transition relation of an atomic actions as a logical formula from the user-provided representation as imperative code.

3.6 Conclusions

In this paper, we provide a foundation for refining structured concurrent programs and an implementation in the Civl verifier. The contribution of this paper, and that of Civl in general, is the capability to express *new proofs* with significant advantages for the programmer in terms of proof structuring, annotation effort, and tool performance.

Acknowledgments

Bernhard Kragl and Thomas A. Henzinger were supported by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award).

4 Synchronizing the Asynchronous

Abstract. Synchronous programs are easy to specify because the side effects of an operation are finished by the time the invocation of the operation returns to the caller. Asynchronous programs, on the other hand, are difficult to specify because there are side effects due to pending computation scheduled as a result of the invocation of an operation. They are also difficult to verify because of the large number of possible interleavings of concurrent computation threads. We present *synchronization*, a new proof rule that simplifies the verification of asynchronous programs by introducing the fiction, for proof purposes, that asynchronous operations complete synchronously. Synchronization summarizes an asynchronous computation as immediate atomic effect. Modular verification is enabled via *pending asynchronous calls* in atomic summaries, and a complementary proof rule that eliminates pending asynchronous calls when components and their specifications are composed. We evaluate synchronization in the context of a multi-layer refinement verification methodology on a collection of benchmark programs.

4.1 Introduction

This paper focuses on the deductive verification of *asynchronous concurrent programs*, an important class that includes distributed fault-tolerant protocols, message-passing programs, client-server applications, event-driven mobile applications, workflows, device drivers, and many embedded and cyber-physical systems. A key aspect of such programs is that (long-running) operations complete asynchronously. A process that invokes an operation does not block for the operation to finish. Instead, the result from the completion of the operation is communicated later, e.g., via a callback message. Asynchronous completion not only introduces concurrency and nondeterminism into the program semantics, but also makes the task of specifying the correct behavior of operations difficult. The behavior of a *synchronous* operation can be specified with a precondition and a postcondition because there is no ambiguity about the state just before and just after the operation executes. The behavior of an *asynchronous* operation is harder to specify because multiple operations can be in flight at the same time and partial results from other operations may have already affected the state before the operation finishes.

In this paper, we propose that reasoning about asynchronous computation can be simplified via *synchronization*, a program transformation that generalizes reduction [80, 47]. While reduction allows the creation of a coarse-grained atomic action from a sequence

of fine-grained atomic actions performed by a single thread, synchronization allows the creation of a coarse-grained atomic action from an asynchronous computation executed by a potentially unbounded number of concurrent threads. Synchronization reduces the number of interleavings; it allows us to pretend, for the purposes of proof, that asynchronous calls complete synchronously and atomically, which leads to significantly simpler invariants.

Synchronization, similar to reduction, relies on commutativity properties of low-level atomic actions. Establishing commutativity may be difficult if these atomic actions access shared state that is also accessed by other, interfering concurrent computations. To enable synchronization in the presence of interference, we leverage the observation that commutativity properties among a set of atomic actions can be established by abstracting these actions [41]. In particular, we incorporate synchronization as a program transformation in the verification methodology of *program layers* [70], which allows the programmer to chain together a sequence of increasingly abstract concurrent programs containing atomic actions that are increasingly coarse-grained. Since program layers allow history variables to be introduced, history variables are sufficient for converting an arbitrary safety property into assertions, and the synchronization transformation preserves all assertion failures, our technique is applicable to the proof of arbitrary safety properties of asynchronous programs.

Synchronization, if used naively, leads to summaries that are not modular and hence not reusable. Consider a scenario where a client invokes an operation S of a service, upon whose completion a callback function C is invoked asynchronously. If the code of C is synchronized into S, the summary of S will be cluttered by the effects of C, making reuse across a different client impossible. To solve this problem, we generalize atomic summaries to support *pending asynchronous calls* (*pending asyncs* in short). Using pending asyncs, we can synchronize asynchrony internal to the service, while leaving the asynchronous callback to C as pending in the summary of S, thus enabling the reuse across different clients. Once the summary of S has been absorbed into the client, we need a mechanism to replace the pending async with the effect of the concrete implementation of C. For that we provide a second proof rule to eliminate pending asyncs from specifications.

We integrated our proof rules in the Civl verifier [56] which provided a baseline framework of program layers. We report on our experience verifying a collection of benchmark programs, showing that our technique enables elegant specifications and proofs of asynchronous programs.

4.2 Overview

We start with an overview of our new verification technique based on the two concepts *synchronization* and *pending asyncs*. In our examples we follow the convention of writing procedure names capitalized (e.g., `Acquire`), and atomic action names in all caps (e.g., `ACQUIRE`). We use the notation `[...]` to denote unnamed atomic actions, i.e., the statements inside square brackets are considered to execute indivisibly.

4.2.1 Asynchronous Increments and Decrements

Consider the program in Figure 4.1 (a). The program comprises a single procedure `Main` that uses a global variable `x` and a local variable `i`. Every iteration of the while loop in

(a)
```
global var x

proc Main(n):
  i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    i := i + 1
```

(b)
```
proc Main:
  async Foo
  assert false

proc Foo:
  call Foo
```

(c)
```
global var x

proc Main(n):
  i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    if *: i := i + 1
```

Figure 4.1: Asynchronous increments and decrements

`Main` creates two new threads, one executing an atomic increment `[x := x + 1]`, and one executing an atomic decrement `[x := x - 1]`. Due to asynchronous thread creation, the execution of individual increments and decrements can be interleaved arbitrarily. However, once all threads finish, the value in variable `x` is equal to its initial value. Thus, `Main` *refines* the atomic action `[skip]`, which does nothing.

A standard noninterference-based proof of this program requires an invariant that states that "`x` is equal to its original value, plus the number of finished increment threads, minus the number of finished decrement threads". Stating this invariant requires ghost code that tracks the progress of each thread. In contrast, our *synchronization* proof rule (Section 4.4) allows us to consider both asynchronous calls in `Main` as regular synchronous calls. Then sequential reasoning suffices to prove that the procedure leaves the variable `x` unchanged. Synchronization is justified by the commutativity of atomic actions on shared state. Specifically, both increment and decrement are *left movers* in the context of our program. Thus the asynchronous computation steps in an interleaved execution can be rearranged to obtain a corresponding synchronous execution that preserves final states.

However, commutativity alone is not sufficient! We also need to ensure that synchronization preserves failing behaviors. Consider the program in Figure 4.1 (b) where `Main` asynchronously calls a procedure `Foo` (which calls itself recursively) followed by a failing assertion. The program has failing executions; the assertion can be scheduled any time between steps of `Foo`. If we synchronize the call to `Foo`, however, the nontermination of `Foo` makes the assertion unreachable and thus synchronization must not be allowed. We could require termination of the synchronized program, but this would be unnecessarily restrictive. We propose a weaker condition called *cooperation*, which only requires the possibility to terminate. In other words, it must be impossible for the synchronized program to reach a state where nontermination is inevitable. To illustrate cooperation, consider Figure 4.1 (c), a modification of (a) which nondeterministically increments the loop counter `i`. The program does not terminate because it *may* loop forever, but it cooperates because it *can* always increment `i`. By synchronization we can show analogously to (a) that (c) also refines `[skip]`.

4.2.2 Lock Service

Figure 4.2 (a) shows a simple lock service implementation. A client requests the lock by asynchronously invoking `Acquire`, which is implemented as spinlock using the atomic compare-and-swap (CAS) operation on the global variable `l`. Once successful, the client of the lock service is notified via an asynchronous callback. Summarizing `Acquire` as atomic action via synchronization of the callback is not desirable, because it would drag in the effect of the client into the specification of `Acquire`. Instead, we propose the modular, reusable, and client-independent atomic action specifications `ACQUIRE` and `RELEASE` shown

(a)
```
global var l

proc Acquire (tid):
  b := false
  while !b:
    call b := CAS(l, nil, tid)
  async Callback(tid)

proc Release (tid):
  call [l := nil]
```

(b)
```
global var l

action ACQUIRE (tid):
  assert tid != nil
  assume l == nil
  l := tid
  async Callback(tid)

action RELEASE (tid):
  assert tid != nil && l == tid
  l := nil
```

(c)
```
global var x

proc Callback(tid):
  call [t := x]
  call [x := t + 1]
  async Release(tid)
```

(d)
```
global var x, l

action CALLBACK(tid):
  assert tid != nil
  assert l == tid
  x := x + 1
  l := nil
```

(e)
```
global var x, l

action ACQUIRE' (tid):
  assert tid != nil
  assume l == nil
  l := tid
  x := x + 1
  l := nil
```

(f)
```
global var x

action ACQUIRE'' (tid):
  x := x + 1
```

Figure 4.2: Lock service

in (b). Notice how we represent guarded atomic transitions as program code. But more importantly, observe that the atomic action specification `ACQUIRE` contains a *pending async* to `Callback`. That is, we allow the effect of asynchronous thread creation as part of atomic actions. Now, to make use of such specifications, our technique is complemented with a proof rule to eliminate pending asyncs (Section 4.6), once an atomic action specification for the target is available. For example, consider the callback implementation in (c) that reads and writes a shared variable `x`, and then releases the lock. Since the callback is only supposed to be invoked with the lock held, we strengthen `[t := x]` and `[x := t + 1]` with the gate `assert tid != nil && l == tid`, which makes the operations commutative. Together with `RELEASE` being a left mover, we use synchronization to show that `Callback` refines the atomic action `CALLBACK` in (d). Now that we have an atomic action specification for `Callback`, we use it to eliminate the pending async in `ACQUIRE` and obtain the atomic action `ACQUIRE'` in (e). Notice how the gates of `CALLBACK` are discharged by the code preceding the pending async in `ACQUIRE`. Finally, we can abstract away the lock acquire and release, such that the client of the lock service only sees the atomic action `ACQUIRE''` in (f).

4.2.3 Layered Refinement Proofs

Our proof rules connect a lower-level, more fine-grained program with a higher-level, more coarse-grained program (both a bottom-up and top-down interpretation is possible), and repeated applications lead to a hierarchy of connected programs. However, due to the structure-preserving nature of our rules, in practice (Section 4.7) the programmer only writes a single program with *layer annotations* [70] that encode the program on multiple layers of abstraction. Our verifier automatically extracts the hierarchy of programs and generates the necessary verification conditions to justify their connection.

4.3 An Asynchronous Programming Language

In this section we define a core asynchronous programming language on which we formalize our verification technique, and recall the notion of mover types and reduction.

Variables and stores. Let *Var* be a set of *variables* partitioned into *global variables* *GVar* and *local variables* *LVar*, and $RVar \subseteq LVar$ is a set of *return variables*. A *store* is a mapping $\sigma : Var \to Val$ that assigns a *value* from a domain *Val* to every variable. Similarly, $g : GVar \to Val$ is a *global store* and $\ell : LVar \to Val$ is a *local store*. Let $g{\cdot}\ell$ denote the combination of g and ℓ into a store. To model return values from a procedure with local store ℓ_1 to a caller procedure with local store ℓ_2, we define the resulting store at the caller as $\ell_1 \rhd \ell_2 = \lambda v.\, \text{if } v \in RVar \text{ then } \ell_1(v) \text{ else } \ell_2(v)$.

Atomic actions. We generalize gated actions introduced in [41] with the idea of pending asyncs. An *atomic action* is a pair (ρ, τ), where the *gate* ρ is a set of stores and the *update* τ is a set of *transitions* $(\sigma, \sigma', \Omega)$ where σ,σ' are stores and Ω is a finite multiset of *pending asyncs* (ℓ, P) consisting of a local store and a procedure name. If an atomic action is executed in a store σ with $\sigma \notin \rho$, the program "fails"; otherwise, if $\sigma \in \rho$, a transition $(\sigma, \sigma', \Omega) \in \tau$ atomically updates the store to σ' and creates new threads according to Ω.

Remark 2. Atomic actions subsume many standard programming language statements. In particular, (nondeterministic) assignments, assertions, and assumptions. The following table shows some examples ranging over variables x and y.

command	gate	update
$x := x + y$	*true*	$x' = x + y \wedge y' = y$
havoc x	*true*	$y' = y$
assert $x < y$	$x < y$	$x' = x \wedge y' = y$
assume $x < y$	*true*	$x < y \wedge x' = x \wedge y' = y$

Syntax. A program $\mathcal{P}$ is a finite mapping from *atomic action names* A to atomic actions, and *procedure names* P to *statements* s of the form

$$s ::= \texttt{skip} \mid s; s \mid \texttt{if } le \texttt{ then } s \texttt{ else } s \mid \texttt{call } A \mid \texttt{call } P \mid \texttt{async } P.$$

A program contains a dedicated procedure *Main* that serves as an entry point for executions, and every atomic action name respectively procedure name appearing in a call statement must be properly mapped to an atomic action respectively statement. We will write $\mathcal{P}.A$ and $\mathcal{P}.P$ for $\mathcal{P}(A)$ and $\mathcal{P}(P)$, and $A, P \in \mathcal{P}$ for $A, P \in \text{dom}(\mathcal{P})$. We identify the conditional expression *le* with the set of local stores that satisfy it.

Semantics. A *frame* f is a pair (ℓ, s) of local store ℓ and statement s. A *thread* t is a sequence of frames $\vec{f}$, denoting a call stack. A *state* $(g, \mathcal{T})$ is a pair of global store g and a finite multiset of threads $\mathcal{T}$. By slight abuse of notation we will identify a thread t with the singleton multiset $\{t\}$, and thus write $\mathcal{T} \uplus t$ for adding t to $\mathcal{T}$. Let *statement contexts* *SC*, *frame contexts* *FC*, and *thread contexts* *TC* be defined as follows:

$$SC ::= \bullet_{Stmt} \mid SC; s \qquad FC ::= (\bullet_{LStore}, SC) \qquad TC ::= FC{\cdot}\vec{f}$$

$$(g, TC[\ell][\texttt{skip}; s] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell][s] \uplus \mathcal{T}) \ \textsc{Seq}$$

$$\frac{\mathcal{P}.A = (\rho, \tau) \quad g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \tau \quad \mathcal{T}' = \{(\ell'', \texttt{call } P) \mid (\ell'', P) \in \Omega\}}{(g, TC[\ell][\texttt{call } A] \uplus \mathcal{T}) \Rightarrow (g', TC[\ell'][\texttt{skip}] \uplus \mathcal{T}' \uplus \mathcal{T})} \ \textsc{ActionStep}$$

$$\frac{\mathcal{P}.A = (\rho, \tau) \quad g{\cdot}\ell \notin \rho}{(g, TC[\ell][\texttt{call } A] \uplus \mathcal{T}) \Rightarrow \lightning} \ \textsc{ActionFail}$$

$$\frac{s' = \text{if } (\ell \in le) \text{ then } s_1 \text{ else } s_2}{(g, TC[\ell][\texttt{if } le \texttt{ then } s_1 \texttt{ else } s_2] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell][s'] \uplus \mathcal{T})} \ \textsc{If}$$

$$(g, TC[\ell][\texttt{call } P] \uplus \mathcal{T}) \Rightarrow (g, (\ell, \mathcal{P}.P){\cdot}TC[\ell][\texttt{skip}] \uplus \mathcal{T}) \ \textsc{Call}$$

$$(g, (\ell_1, \texttt{skip}){\cdot}TC[\ell_2][s] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell_1 \triangleright \ell_2][s] \uplus \mathcal{T}) \ \textsc{Return}$$

$$(g, TC[\ell][\texttt{async } P] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell][\texttt{skip}] \uplus (\ell, \texttt{call } P) \uplus \mathcal{T}) \ \textsc{Async}$$

$$(g, (\ell, \texttt{skip}) \uplus \mathcal{T}) \Rightarrow (g, \mathcal{T}) \ \textsc{End}$$

Figure 4.3: Small-step operational semantics

$TC[\ell][s]$ denotes the thread obtained by filling the two unique holes $\bullet_{Stmt}$ and $\bullet_{LStore}$ in TC with statement s and local store ℓ, respectively. Thus, $TC[\ell][s]$ executes s from ℓ as next step. The operational semantics is formalized in Figure 4.3 as a transition relation $\Rightarrow$ between states. An *execution* π is a sequence of states $x_0 \Rightarrow x_1 \Rightarrow \ldots$, and we write $\pi : x_0 \Rightarrow^* x_n$ to denote that π is an execution that starts in x_0 and ends in x_n.

Refinement. Given a program $\mathcal{P}$, we are interested in executions that start with a single thread executing *Main* from some initial store $\sigma = g{\cdot}\ell$, i.e., executions that start in a state $(g, (\ell, \texttt{call } \mathit{Main}))$. In particular, we are interested in executions that either fail or terminate. We define $\mathit{Bad}(\mathcal{P})$ to be the set of initial stores associated with *failing executions*, and $\mathit{Good}(\mathcal{P})$ to be the relation between initial and final stores associated with *terminating executions*:

$$\mathit{Bad}(\mathcal{P}) = \left\{ g{\cdot}\ell \mid \left(g, (\ell, \texttt{call } \mathit{Main})\right) \Rightarrow^* \lightning \right\};$$
$$\mathit{Good}(\mathcal{P}) = \left\{ (g{\cdot}\ell, g') \mid \left(g, (\ell, \texttt{call } \mathit{Main})\right) \Rightarrow^* (g', \varnothing) \right\}.$$

A program $\mathcal{P}_1$ *refines* a program $\mathcal{P}_2$, denoted $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, if (1) $\mathit{Bad}(\mathcal{P}_1) \subseteq \mathit{Bad}(\mathcal{P}_2)$ and (2) $\overline{\mathit{Bad}(\mathcal{P}_2)} \circ \mathit{Good}(\mathcal{P}_1) \subseteq \mathit{Good}(\mathcal{P}_2)$; $\overline{\cdot}$ is set complement, $\circ$ is relation composition. The first condition states that $\mathcal{P}_2$ has to preserve failing executions of $\mathcal{P}_1$. The second condition states that $\mathcal{P}_2$ has to preserve terminating executions of $\mathcal{P}_1$ for initial states that cannot fail. That is, $\mathcal{P}_2$ can fail more often than $\mathcal{P}_1$.

Reduction. Let M be a mapping from atomic action names to *mover types* [47]: B (*both mover*), L (*left mover*), R (*right mover*), N (*non-mover*). Intuitively, an atomic action is a right mover, if it commutes to the right (i.e., later in time) with respect to all other atomic actions in $\mathcal{P}$. A left mover is symmetric, and an atomic action can be both a left and right mover. Reduction has traditionally been applied to multithreaded programs

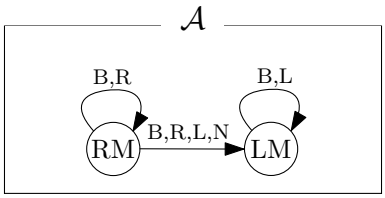


Figure 4.4: Atomicity automaton.

to convert a sequence of atomic actions performed by a single thread into an atomic block. The sequence of mover types of the atomic actions in this block must be a valid run of the nondeterministic *atomicity automaton* $\mathcal{A}$ in Figure 4.4. In this paper, we exploit and extend this work to synchronize asynchronous computation spanning multiple threads.

We define the predicate $\mathit{MoverValid}(\mathcal{P}, M)$ which holds whenever the atomic actions in $\mathcal{P}$ satisfy the mover types indicated by M. Formally, $\mathit{MoverValid}(\mathcal{P}, M)$ holds if for all $A_1, A_2 \in \mathcal{P}$ with $\mathcal{P}.A_1 = (\rho_1, \tau_1)$ and $\mathcal{P}.A_2 = (\rho_2, \tau_2)$, the following conditions hold (generalizing [57] to support pending asyncs).

- **Commutativity:** If $M(A_1) \in \{R, B\}$ or $M(A_2) \in \{L, B\}$, then the effect of executing A_1 followed by A_2 in two different threads can also be achieved by A_2 followed by A_1.

$$\forall g, \bar{g}, g', \ell_1, \ell_1', \ell_2, \ell_2', \Omega_1, \Omega_2 \; \exists \hat{g}, \Omega_1', \Omega_2' : \begin{pmatrix} \wedge & g{\cdot}\ell_1 \in \rho_1 \\ \wedge & g{\cdot}\ell_2 \in \rho_2 \\ \wedge & (g{\cdot}\ell_1, \bar{g}{\cdot}\ell_1', \Omega_1) \in \tau_1 \\ \wedge & (\bar{g}{\cdot}\ell_2, g'{\cdot}\ell_2', \Omega_2) \in \tau_2 \end{pmatrix} \implies \begin{pmatrix} \wedge & (g{\cdot}\ell_2, \hat{g}{\cdot}\ell_2', \Omega_2') \in \tau_2 \\ \wedge & (\hat{g}{\cdot}\ell_1, g'{\cdot}\ell_1', \Omega_1') \in \tau_1 \\ \wedge & \Omega_1 \uplus \Omega_2 = \Omega_1' \uplus \Omega_2' \end{pmatrix}$$

- **Forward preservation:** If $M(A_1) \in \{R, B\}$ or $M(A_2) \in \{L, B\}$, then the failure of A_2 after the execution of A_1 implies that A_2 must also fail before the execution of A_1.

$$\forall g, g', \ell_1, \ell_1', \ell_2, \Omega_1 : (g{\cdot}\ell_1 \in \rho_1 \wedge g{\cdot}\ell_2 \in \rho_2 \wedge (g{\cdot}\ell_1, g'{\cdot}\ell_1', \Omega_1) \in \tau_1) \implies g'{\cdot}\ell_2 \in \rho_2$$

- **Backward preservation:** If $M(A_2) \in \{L, B\}$, then the failure of A_1 before the execution of A_2 implies that A_1 must also fail after the execution of A_2.

$$\forall g, g', \ell_1, \ell_2, \ell_2', \Omega_2 : (g{\cdot}\ell_2 \in \rho_2 \wedge (g{\cdot}\ell_2, g'{\cdot}\ell_2', \Omega_2) \in \tau_2 \wedge g'{\cdot}\ell_1 \in \rho_1) \implies g{\cdot}\ell_1 \in \rho_1$$

- **Nonblocking:** If $M(A_2) \in \{L, B\}$, then A_2 must be nonblocking.

$$\forall \sigma \in \rho_2 \; \exists \sigma', \Omega : (\sigma, \sigma', \Omega) \in \tau_2$$

- **Async freedom:** If $M(A_1) \in \{R, B\}$, then A_1 cannot have pending asynchronous calls.

$$\forall \sigma, \sigma', \Omega : \sigma \in \rho_1 \wedge (\sigma, \sigma', \Omega) \in \tau_1 \implies \Omega = \varnothing$$

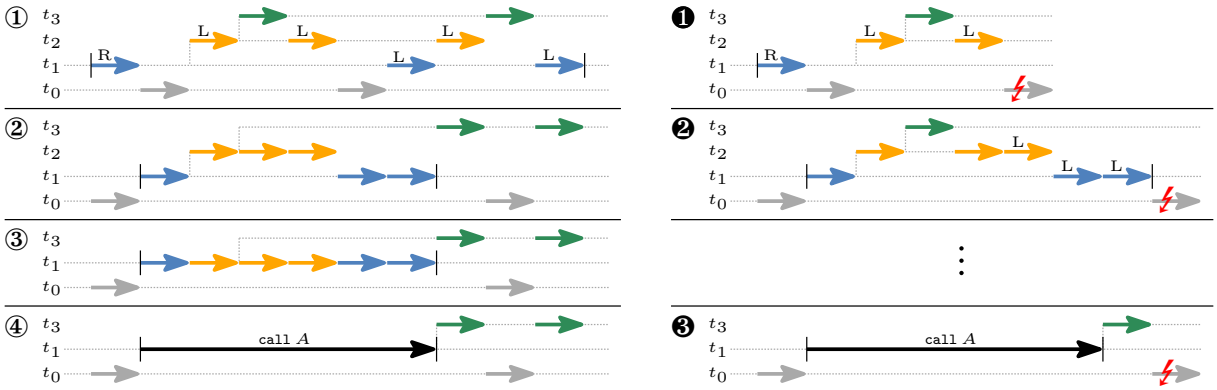


Figure 4.5: Synchronizing asynchronous executions

4.4 Synchronizing Asynchrony

In this section, we formalize the synchronization proof rule which allows us to transform a procedure into an atomic action that summarizes asynchronous effects, either directly or via pending asyncs. Synchronization requires two technical innovations. First, we extend the *commutativity* conditions required for reduction to account for asynchronous thread creation. Second, we impose a new *cooperation* condition necessary for the soundness of our transformation.

Given a procedure Q, a mover typing M, and a set of procedures Σ to synchronize in Q (asynchronous calls to procedures not in Σ are treated as pending asyncs), the SYNCHRONIZE rule transforms procedure Q into an atomic action (ρ, τ) with fresh name A:

$$\boxed{\textsc{Synchronize}}\quad \frac{\mathit{MoverValid}(\mathcal{P}, M) \quad \mathit{Sync}(\mathcal{P}, M, Q, \Sigma) \quad \mathit{Refinement}(\mathcal{P}, Q, \Sigma, \rho, \tau)}{\mathcal{P} \rightsquigarrow \mathcal{P}[Q \mapsto \texttt{call } A] \cup [A \mapsto (\rho, \tau)]} \begin{array}{l} Q \in \mathcal{P} \\ A \notin \mathcal{P} \end{array}$$

We already defined *MoverValid* in the previous section. Now we informally discuss the soundness of SYNCHRONIZE, and formally defined the other two premises *Sync* and *Refinement*. In the next section we show how all premises can be efficiently checked in practice.

Theorem 5. *If* $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ *using the* SYNCHRONIZE *rule, then* $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.

Intuition. The core idea of Theorem 5 is the rewriting of a $\mathcal{P}_1$-execution π_1 into an equivalent $\mathcal{P}_2$-execution π_2. Concretely, (1) if π_1 fails then π_2 must fail, and (2) if π_1 terminates then π_2 must either terminate with the same final state or fail. We illustrate this transformation in Figure 4.5. On the left, ① shows part of an asynchronous execution, initially comprising two threads t_0 and t_1. Thread t_1 executes the transformed procedure Q (the call and return are indicated with black bars), which makes an asynchronous call to spawn t_2, and t_2 asynchronously spawns t_3. Notice that t_2 terminates after three steps. We consider the procedure of t_2 to be in Σ (i.e., to be synchronized), while the procedure of t_3 is not in Σ (i.e., to be treated as pending async). Our goal is to transform execution ① into execution ②, which has the following properties: (1) Q executes without interruption from t_0, (2) t_2 terminates without interruption before t_1 continues, and (3) t_3 only starts after Q returns. To permit this transformation, *Sync* requires that Q, *including* asynchronous

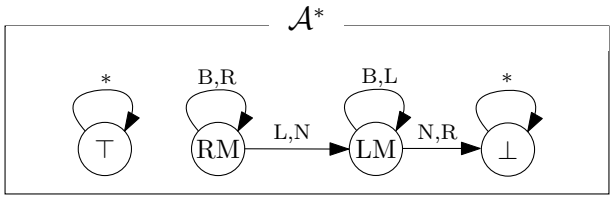


Figure 4.6: Tracking automaton.

calls to procedures in Σ, executes a sequence of right movers, followed by at most one non-mover, followed by a sequence of left movers. Furthermore, the asynchronous calls to procedures in Σ must only execute left movers. The steps of t_1 and t_2 in ① are labeled with mover types that satisfy this conditions. When moving the right mover to the right and the left movers to the left to obtain ②, the commutativity, forward preservation, and backward preservation properties of *MoverValid* guarantee that the executions stay equivalent. Now, as shown in ③, the steps of t_2 can be considered to execute synchronously in its parent t_1. Finally, *Refinement* ensures that the synchronized behavior of Q is summarized by the atomic action A in ④, which captures the creation of t_3 as pending async.

On the right of Figure 4.5, ❶ shows an execution where Q started, but then t_0 failed. Notice that, if all steps of Q before the failure are right movers, these steps can be removed from the execution (by moving them to the right, "past" the failure), and the failure occurs before Q even starts. In ❶, however, Q already executed a left mover. Even if we move the steps of Q together, the partial execution of Q is not summarized by A. However, we know that only left movers can follow in t_1 and t_2. Since left movers are non-blocking and backward preserving, they can be inserted at the end of the execution, right before the failure. The *cooperation* condition (part of *Sync*) ensures that this can be done so that Q is completed, as shown in ❷. Then we can again arrive at an execution where Q is replaced by A (see ❸).

Concurrent tracking semantics. The execution in ① is labeled with mover types that allowed us to rearrange the steps of Q to obtain the execution in ②. To characterize the executions for which such a rearrangement is possible in general, we define the *concurrent tracking semantics* $\overset{M,Q,\Sigma}{\Longrightarrow}$ (Figure 4.7) that is similar to $\Rightarrow$, except that we additionally track a *mover phase* m in frames, which is one of the states of the *tracking automaton* $\mathcal{A}^*$ in Figure 4.6: $\top$ (*no tracking*), RM (*right-mover phase*), LM (*left-mover phase*), $\bot$ (*violation*). Call transitions from $\top$ to RM on a top-level call to Q, or otherwise propagates the mover phase of the caller to the callee. Conversely, Return transitions back to $\top$ when returning from a top-level call to Q, or otherwise propagates the mover phase of the callee to the caller. ActionStep follows a transition in $\mathcal{A}^*$ according to the mover type of the invoked atomic action. In particular, if we are tracking ($m \neq \top$), we stay in RM until a non-right mover (L or N) causes a transition to LM. In LM only left movers should follow, and thus the occurrence of a non-left mover (N or R) causes a transition to the violation state $\bot$. Notice that the async freedom condition of *MoverValid* forces a thread that executes an atomic action with pending asyncs to LM. This is important to ensure that only left movers can follow, which can be moved before the steps of any pending async. Similarly, Async transitions the parent thread of an asynchronous call to LM. The child thread is set to LM if we want to synchronize the call, otherwise it is not tracked. In both ActionStep and Async, if an untracked child thread executes `call` Q, the subsequent application of Call will start to track the child tread separately.

$\xrightarrow{M,Q,\Sigma}$

$$(g,\, TC[\ell][\texttt{skip}; s][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g,\, TC[\ell][s][m] \uplus \mathcal{T})\ \textsc{Seq}$$

$$\frac{\mathcal{P}.A = (\rho, \tau) \quad g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \tau \quad m' = \mathcal{A}^*(m, M(A)) \quad \mathcal{T}' = \{(\ell'', \texttt{call}\ P, \top) \mid (\ell'', P) \in \Omega\}}{(g,\, TC[\ell][\texttt{call}\ A][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g',\, TC[\ell'][\texttt{skip}][m'] \uplus \mathcal{T}' \uplus \mathcal{T})}\ \textsc{ActionStep}$$

$$\frac{\mathcal{P}.A = (\rho, \tau) \quad g{\cdot}\ell \notin \rho}{(g,\, TC[\ell][\texttt{call}\ A][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} \lightning}\ \textsc{ActionFail}$$

$$\frac{s' = \text{if } (\ell \in le) \text{ then } s_1 \text{ else } s_2}{(g,\, TC[\ell][\texttt{if}\ le\ \texttt{then}\ s_1\ \texttt{else}\ s_2][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g,\, TC[\ell][s'][m] \uplus \mathcal{T})}\ \textsc{If}$$

$$\frac{m' = \text{if } (m = \top \wedge P = Q) \text{ then RM else } m}{(g,\, TC[\ell][\texttt{call}\ P][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g,\, (\ell, \mathcal{P}.P, m'){\cdot}\, TC[\ell][\texttt{skip}][m] \uplus \mathcal{T})}\ \textsc{Call}$$

$$\frac{m' = \text{if } (m_2 = \top) \text{ then } \top \text{ else } m_1}{(g,\, (\ell_1, \texttt{skip}, m_1){\cdot}\, TC[\ell_2][s][m_2] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g,\, TC[\ell_1 \triangleright \ell_2][s][m'] \uplus \mathcal{T})}\ \textsc{Return}$$

$$\frac{m' = \text{if } (m \neq \top) \text{ then LM else } \top \quad m'' = \text{if } (m \neq \top \wedge P \in \Sigma) \text{ then LM else } \top}{(g,\, TC[\ell][\texttt{async}\ P][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g,\, TC[\ell][\texttt{skip}][m'] \uplus (\ell, \texttt{call}\ P, m'') \uplus \mathcal{T})}\ \textsc{Async}$$

$$(g,\, (\ell, \texttt{skip}, m) \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, \mathcal{T})\ \textsc{End}$$

$\xrightarrow{\Sigma}$

$$(g,\, TC[\ell][\texttt{skip}; s], \Omega) \xrightarrow{\Sigma} (g,\, TC[\ell][s], \Omega)\ \textsc{Seq}$$

$$\frac{\mathcal{P}.A = (\rho, \tau) \quad g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \tau}{(g,\, TC[\ell][\texttt{call}\ A], \Omega') \xrightarrow{\Sigma} (g',\, TC[\ell'][\texttt{skip}], \Omega \uplus \Omega')}\ \textsc{ActionStep}$$

$$\frac{\mathcal{P}.A = (\rho, \tau) \quad g{\cdot}\ell \notin \rho}{(g,\, TC[\ell][\texttt{call}\ A], \Omega) \xrightarrow{\Sigma} \lightning}\ \textsc{ActionFail}$$

$$\frac{s' = \text{if } (\ell \in le) \text{ then } s_1 \text{ else } s_2}{(g,\, TC[\ell][\texttt{if}\ le\ \texttt{then}\ s_1\ \texttt{else}\ s_2], \Omega) \xrightarrow{\Sigma} (g,\, TC[\ell][s'], \Omega)}\ \textsc{If}$$

$$(g,\, TC[\ell][\texttt{call}\ P], \Omega) \xrightarrow{\Sigma} (g,\, (\ell, \mathcal{P}.P){\cdot}\, TC[\ell][\texttt{skip}], \Omega)\ \textsc{Call}$$

$$(g,\, (\ell_1, \texttt{skip}){\cdot}\, TC[\ell_2][s], \Omega) \xrightarrow{\Sigma} (g,\, TC[\ell_1 \triangleright \ell_2][s], \Omega)\ \textsc{Return}$$

$$(g,\, TC[\ell][\texttt{async}\ P], \Omega) \xrightarrow{\Sigma} \begin{cases} (g,\, TC[\ell][\texttt{skip}], (\ell, P) \uplus \Omega) & \text{if } P \notin \Sigma \\ (g,\, (\ell, \texttt{call}\ P)^\sharp{\cdot}\, TC[\ell][\texttt{skip}], \Omega) & \text{if } P \in \Sigma \end{cases}\ \textsc{Async}$$

$$(g,\, (\ell, \texttt{skip})^\sharp{\cdot}\, \vec{f}, \Omega) \xrightarrow{\Sigma} (g, \vec{f}, \Omega)\ \textsc{AsyncReturn}$$

Figure 4.7: Concurrent tracking semantics $\xrightarrow{M,Q,\Sigma}$ and sequential synchronized semantics $\xrightarrow{\Sigma}$

Sequential synchronized semantics. In ③ we are concerned with the sequential execution of Q, with asynchronous calls to procedures in Σ being synchronized. We formally define the *sequential synchronized semantics* $\xrightarrow{\Sigma}$ (Figure 4.7) that executes a single thread and stores a multiset of pending asyncs. In ActionStep, the pending asyncs of an atomic action are added to the already existing pending asyncs. For an asynchronous call to P, Async records a pending thread creation if $P \notin \Sigma$, and synchronizes the call if $P \in \Sigma$. The synchronized stack frame is marked with $\sharp$ such that it is popped in AsyncReturn without writing return variables to the caller. This technicality is necessary in our formalization. In practice, asynchronously called procedures simply cannot have return parameters.

With the concurrent tracking semantics and the sequential synchronized semantics we can now formally define *Sync* and *Refinement*.

Sync. $\mathit{Sync}(\mathcal{P}, M, Q, \Sigma)$ comprises the following two conditions:

S1 $(g, (\ell, \texttt{call}\ \mathit{Main}, \top)) \xRightarrow{M,Q,\Sigma}{}^{*} (g', \mathit{TC}[\ell'][s][m] \uplus \mathcal{T})$ implies $m \neq \bot$;

S2 $(g, (\ell, \texttt{call}\ \mathit{Main}, \top)) \xRightarrow{M,Q,\Sigma}{}^{*} (g', \mathit{TC}[\ell'][\texttt{call}\ P][\mathrm{LM}] \uplus \mathcal{T})$ implies $(g', (\ell', \texttt{call}\ P), \varnothing) \xrightarrow{\Sigma}{}^{*} (g'', (\ell'', \texttt{skip}), \Omega'')$.

S1 states that executions respect the required mover sequences, i.e., no violation is reachable in the tracking semantics. S2 (the cooperation condition) states that every procedure call in the left-mover phase can be completed. The repeated application of S1 allows us to complete partial executions of Q. Note that S2 also captures asynchronous calls to procedures P with $P \in \Sigma$, since the operational semantics rewrites `async` P into `call` P.

Refinement. $\mathit{Refinement}(\mathcal{P}, Q, \Sigma, \rho, \tau)$ comprises the following two conditions:

R1 $\rho \cap \{g{\cdot}\ell \mid (g, (\ell, \mathcal{P}.Q), \varnothing) \xrightarrow{\Sigma}{}^{*} \lightning\} = \varnothing$;

R2 $\rho \circ \{(g{\cdot}\ell, g'{\cdot}\ell', \Omega) \mid (g, (\ell, \mathcal{P}.Q), \varnothing) \xrightarrow{\Sigma}{}^{*} (g', (\ell', \texttt{skip}), \Omega)\} \subseteq \tau$.

R1 states that the gate of A is strong enough to filter out all failures of Q, and R2 states that the transition relation of A captures all non-failing executions of Q.

4.5 Verifying Synchronization

In this section we show how the premises of the Synchronize rule can be efficiently checked in practice. The *MoverValid* and *Refinement* premises both lead to standard verification conditions. In particular, the constraints of *MoverValid* state the commutativity of individual atomic actions, and the constraints of *Refinement* state that a sequential procedure is summarized by a transition relation, which can be readily handed off to logical reasoning engines. Thus we focus on *Sync* which we decompose as follows:

$$\frac{\mathit{StaticSync}(\mathcal{P}, M, Q, \Sigma, \mathit{Pre}) \quad \mathit{Safe}(\mathcal{P}, \mathit{Pre}) \quad \mathit{Terminates}(\mathcal{P}, \Sigma, \mathit{Pre}, \mathit{Red})}{\mathit{Sync}(\mathcal{P}, M, Q, \Sigma)}$$

We establish *Sync* in three steps. First, *StaticSync* is a static control-flow analysis that over-approximates the tracking semantics. It uses the domain of a *precondition mapping*

Pre, a partial mapping from procedure names to sets of stores. If *StaticSync* succeeds, it guarantees S1 (i.e., that $\bot$ cannot be reached) and that all procedures P called with mover phase LM in S2 are in $\mathrm{dom}(Pre)$. Second, we over-approximate the possible stores $g'{\cdot}\ell'$ at these calls. For that, *Safe* requires that *Pre* denotes valid preconditions, i.e., if `call` P is reachable with store $g'{\cdot}\ell'$, then $g'{\cdot}\ell' \in Pre(P)$ for all $P \in \mathrm{dom}(Pre)$. Then finally, to establish S2, it remains to show that there is some terminating sequential execution from $(g', (\ell', \texttt{call}\ P), \varnothing)$ for every $P \in \mathrm{dom}(Pre)$ and $g'{\cdot}\ell' \in Pre(P)$. *Terminates* reduces these cooperation checks to standard termination checks on a restricted program. In particular, the *restriction function Red* limits the nondeterministic behavior of some atomic actions. Then showing that *all* executions in the restricted program terminate implies that there is *some* terminating execution in the original program (given that *Red* is not allowed to make atomic actions blocking).

StaticSync. Let $\mathcal{E}$ be the function that maps a mover type to the corresponding set of edges in $\mathcal{A}$, e.g., $\mathcal{E}(\mathrm{R}) = \{\mathrm{RM} \to \mathrm{RM}, \mathrm{RM} \to \mathrm{LM}\}$. We define an interprocedural control flow analysis that lifts $\mathcal{E}$ to a mapping $\widehat{\mathcal{E}}$ on statements, corresponding to the paths a statement may take in the tracking semantics. We write $\mathit{StaticSync}(\mathcal{P}, M, Q, \Sigma, Pre)$ if there is a solution $\widehat{\mathcal{E}}(\mathcal{P}.Q) \neq \varnothing$ to the following equations w.r.t. M, Σ and *Pre*:

$$
\begin{aligned}
\widehat{\mathcal{E}}(\texttt{skip}) &= \mathcal{E}(\mathrm{B}) \\
\widehat{\mathcal{E}}(\texttt{call}\ A) &= \mathcal{E}(M(A)) \\
\widehat{\mathcal{E}}(s_1; s_2) &= \widehat{\mathcal{E}}(s_1) \circ \widehat{\mathcal{E}}(s_2) \\
\widehat{\mathcal{E}}(\texttt{if}\ le\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &= \widehat{\mathcal{E}}(s_1) \cap \widehat{\mathcal{E}}(s_2) \\
\widehat{\mathcal{E}}(\texttt{call}\ P) &= \begin{cases} \widehat{\mathcal{E}}(\mathcal{P}.P) & \text{if } P \in \mathrm{dom}(Pre) \\ \widehat{\mathcal{E}}(\mathcal{P}.P) \cap \{\mathrm{RM}\}^2 & \text{if } P \notin \mathrm{dom}(Pre) \end{cases} \\
\widehat{\mathcal{E}}(\texttt{async}\ P) &= \begin{cases} \{\mathrm{LM}\}^2 & \text{if } P \notin \Sigma \\ \{\mathrm{LM}\}^2 \cap \widehat{\mathcal{E}}(\mathcal{P}.P) & \text{if } P \in \Sigma \cap \mathrm{dom}(Pre) \\ \varnothing & \text{if } P \in \Sigma \setminus \mathrm{dom}(Pre) \end{cases}
\end{aligned}
$$

The equations on the left capture regular control-flow propagation. The equation for `call` P has two cases. If $P \in \mathrm{dom}(Pre)$, we do not restrict the call since P is cooperative. However, if $P \notin \mathrm{dom}(Pre)$ we must restrict the call to stay in the right-mover phase, because we cannot rely on the cooperation condition to complete partial executions of Q. The equation for `async` P has three cases. If $P \notin \Sigma$, we do not synchronize P and thus only require the caller to be followed by only left movers. If $P \in \Sigma \cap \mathrm{dom}(Pre)$, we additionally require the invoked procedure P to be only left movers. For synchronized procedures we always have to establish cooperation, thus the case $P \in \Sigma \setminus \mathrm{dom}(Pre)$ is not allowed.

If $\mathit{StaticSync}(\mathcal{P}, M, Q, \Sigma, Pre)$, then S1 holds and for every `call` P reachable with LM in S2 we have $P \in \mathrm{dom}(Pre)$. Hence, we must check cooperation for all procedures in $\mathrm{dom}(Pre)$.

Safe. Now that we know the procedures that need to be checked for cooperation, we want to know the stores from which to check cooperation. For that, *Pre* must denote valid

preconditions. We write $Safe(\mathcal{P}, Pre)$, if $(g, (\ell, \texttt{call}\ Main)) \Rightarrow^* (g', TC[\ell'][\texttt{call}\ P] \uplus \mathcal{T})$ implies $g' \cdot \ell' \in Pre(P)$ for all $P \in \mathrm{dom}(Pre)$.

Terminates. Finally, we establish S2 by showing that all procedures P in $\mathrm{dom}(Pre)$ cooperate from states in $Pre(P)$. Suppose that cooperation holds, but termination (which is stronger) does not. Such a difference between termination and cooperation must be due to nondeterminism. Thus, if we suitably restrict the nondeterminism to eliminate nonterminating behaviors, proving termination for the restricted program implies cooperation for the original program. Formally, a *restriction function* Red is a partial mapping from atomic action names to atomic actions, such that for all $A \in \mathrm{dom}(Red)$ with $\mathcal{P}.A = (\rho, \tau)$ it holds that $Red(A) = (\rho, \tau')$ with $\tau' \subseteq \tau$ and $Red(A)$ is nonblocking. Let $\mathcal{P}^{Red}$ be the program equal to $\mathcal{P}$, except that $\mathcal{P}^{Red}.A = Red(A)$ for $A \in \mathrm{dom}(Red)$.

We write $Terminates(\mathcal{P}, \Sigma, Pre, Red)$, if for all $P \in \mathrm{dom}(Pre)$ and $g \cdot \ell \in Pre(P)$, there is no infinite sequential synchronized $\mathcal{P}^{Red}$-execution $(g, (\ell, \texttt{call}\ P), \varnothing) \xrightarrow{\Sigma} \cdots$. Notice that these termination checks can now be solved by a standard termination checker for sequential programs. While it is possible for the programmer to explicitly provide restricted atomic actions, in practice we did not found this necessary for any of our examples. Instead, a fixed policy to resolve nondeterministic branching (e.g., always take the *then* branch) was enough. For example, recall the program in Figure 4.1 (c). Always taking the *then* branch (i.e., resolving the nondeterministic choice to *true*) allows us to prove termination and thus implies cooperation of the original program.

Theorem 6. *If we have* $StaticSync(\mathcal{P}, M, Q, \Sigma, Pre)$, $Safe(\mathcal{P}, Pre)$, *and* $Terminates(\mathcal{P}, \Sigma, Pre, Red)$, *then* $Sync(\mathcal{P}, M, Q, \Sigma)$ *holds.*

4.6 Eliminating Pending Asynchrony

In the previous two sections we showed how the SYNCHRONIZE rule allows to summarize procedures to atomic actions, by either directly synchronizing asynchronous calls or keeping them as pending asyncs. In this section we present the complementary PENDINGASYNCELIM rule to eliminate pending asyncs from atomic actions.

Let A be an atomic action with pending asyncs to a procedure P. Eliminating those pending asyncs requires that (1) P is summarized to an atomic action, say B, and (2) B must be a left mover, since we will directly compose its effect with A. Now we show the construction of the new gate and update for A. The new gate is obtained by filtering out all states from the gate of A that can cause B to fail. In other words, we strengthen A's gate such that it cannot make a transition to a state where B fails:

$$Gt(\rho_A, \tau_A, \rho_B, P) = \left\{ \sigma \in \rho_A \mid \forall \begin{array}{c} g', \ell', \\ \ell_P, \Omega \end{array} : \begin{array}{c} (\sigma, g' \cdot \ell', (\ell_P, P) \uplus \Omega) \in \tau_A \\ \implies g' \cdot \ell_P \in \rho_B \end{array} \right\}$$

The new update consists of two parts. First, we take all transitions without pending asyncs to P:

$$Upd_1(\tau_A, P) = \{(\sigma, \sigma', \Omega) \in \tau_A \mid \neg \exists \ell_P : (\ell_P, P) \in \Omega\}$$

Second, we compose all transitions with a pending async to P with the transitions of B:

$$Upd_2(\tau_A, \tau_B, P) = \left\{ (\sigma, g'' \cdot \ell', \Omega \uplus \Omega') \mid \exists \begin{array}{c} g', \ell_P, \\ \Omega, \ell'' \end{array} : \wedge \begin{array}{c} (\sigma, g' \cdot \ell', (\ell_P, P) \uplus \Omega) \in \tau_A \\ (g' \cdot \ell_P, g'' \cdot \ell'', \Omega') \in \tau_B \end{array} \right\}$$

Notice that the transitions of B can have pending asyncs that are absorbed into the resulting transition. Combining all pieces, we obtain the following rule for eliminating pending asyncs:

$$\boxed{\textsc{PendingAsyncElim}}$$
$$\frac{\begin{array}{c}\mathcal{P}.P = \texttt{call}\ B \qquad \mathcal{P}.B = (\rho_B, \tau_B) \qquad M(B) \in \{\mathrm{L}, \mathrm{B}\} \\ \rho'_A = Gt(\rho_A, \tau_A, \rho_B, P) \qquad \tau'_A = Upd_1(\tau_A, P) \cup Upd_2(\tau_A, \tau_B, P)\end{array}}{\mathcal{P} \uplus [A \mapsto (\rho_A, \tau_A)] \rightsquigarrow \mathcal{P} \uplus [A \mapsto (\rho'_A, \tau'_A)]}$$

Example 1. Recall our motivating lock service example from Section 4.2.2. Eliminating the pending async in `ACQUIRE` is a formal application of PendingAsyncElim with $P =$ `Callback`, $A =$ `ACQUIRE`, and $B =$ `CALLBACK`. The resulting action (the new A) is `ACQUIRE'`.

Theorem 7. *If $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ using the* PendingAsyncElim *rule, then $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.*

PendingAsyncElim eliminates a single pending async to P in A. Iterative application of the rule allows us to eliminate finitely many pending asyncs. In theory, PendingAsyncElim can be generalized with an induction schema to eliminate unboundedly many pending asyncs, but we did not find this necessary in practice.

4.7 Evaluation

We implemented our verification method in Civl [56], a verification system for concurrent programs based on automated and modular refinement reasoning. In Civl, a program is specified and verified across multiple layers of refinement. At each layer, procedures can be declared to refine atomic actions and henceforth appear atomic to higher layers. This means that an input program with layer annotations implicitly describes the program at multiple levels of abstraction, and Civl automatically checks refinement between programs on adjacent layers.

We implemented and verified a collection of nine benchmarks, of which five expand on our motivating example from Section 4.2.1, one is a ping-pong agreement protocol that exercises the notion of cooperation, and the remaining three examples are discussed in the remainder of this section to illustrate (1) the interaction with Civl and modular verification via pending asyncs, (2) the applicability to challenging concurrency, and (3) one-shot synchronization of nested asynchronous calls. Overall, our benchmarks capture realistic patterns of asynchronous computation. All benchmarks are verified by our tool in less than three seconds. The implementation and benchmarks are available at `https://github.com/boogie-org/boogie`.

The proof rules introduced in this paper are crucial to preserving the layered verification approach in Civl and exploiting it to construct compact and highly-automated proofs with simple invariants [70]. Without our new rules, Civl proofs of our benchmarks would amount to single-layer proofs with monolithic invariants in a style similar to classical proofs of distributed systems in modeling frameworks such as TLA+ [76].

4.7.1 Lock Service

In this section we illustrate how synchronization and pending async elimination are offered to a programmer in Civl by revisiting the lock service example from Section 4.2.2.

```
action {:atomic}{:layer 1,1}
CAS_l (oldval:Tid, newval:Tid) returns (b:bool)
{
  if (l == oldval) {
    l := newval; b := true;
  } else {
    b := false;
  }
}

procedure {:layer 1}{:refines ACQUIRE}
Acquire (tid:Tid)
{
  var b:bool;
  b := false;
  while (!b) call b := CAS_l(nil, tid);
  async call Callback(tid);
}
```

```
action {:atomic}{:layer 2,3} ACQUIRE (tid:Tid)
{
  assert tid != nil;
  assume l == nil;
  l := tid;
  async call Callback(tid);
}

procedure {:layer 2}{:refines CALLBACK}
Callback (tid:Tid)
{
  /* not shown */
}

action {:left}{:layer 3} CALLBACK (tid:Tid)
{
  assert tid != nil && l == tid;
  x := x + 1;
  l := nil;
}
```

Figure 4.8: Lock service in Civl (excerpt)

Figure 4.8 shows a fragment of our Civl implementation. First, let us understand the layer annotations in more detail. A procedure has a single layer number x that denotes the layer at which the procedure is shown to refine an atomic action. At all layers up to x calls to the procedure behave according to its implementation, and at layers higher than x calls to the procedure behave according to its refined atomic action. Atomic actions have an associated layer range $[x, y]$, which denotes at which layers the action is "available". For each layer, the set of available atomic actions is subject to pairwise commutativity checks. In Figure 4.8, the procedure `Acquire` is declared to refine the atomic action `ACQUIRE` at layer 1, which causes Civl to apply synchronization. The implementation makes two calls, a synchronous call to a compare-and-swap operation which is already atomic at layer 1, and an asynchronous call to `Callback`. Since `Callback` is refined at the higher layer 2, the asynchronous call results in a pending async in the atomic action `ACQUIRE`. Thus, at layer 2, `ACQUIRE` is exactly the client-independent specification of `Acquire` we presented in Figure 4.2 (b).

Now `Callback` (whose implementation is not shown) is declared to refine `CALLBACK` at layer 2. This causes Civl to apply pending async elimination in `ACQUIRE` at layer 3; the pending async to `Callback` is replaced with the effect of `CALLBACK`. Thus, at layer 3, `ACQUIRE` corresponds to `ACQUIRE'` in Figure 4.2 (e).

This example illustrates two important aspects of our proof method and its integration into Civl. First, on the conceptual side, our method enables independent and modular reasoning about the lock service implementation and its client. The atomic action `ACQUIRE` can be (1) proved for a different implementation of the lock without the need to re-verify the client, and (2) used to reason about a different client by letting Civl apply pending async elimination for a different client (i.e., `Callback` implementation). Second, on the practical side, the application of synchronization and pending async elimination in Civl is driven by layer annotations. The programmer does not have to explicitly write the program under consideration at every layer of abstraction and specify the transformation that connects them. Instead, Civl automatically constructs per-layer versions of procedures and atomic actions.

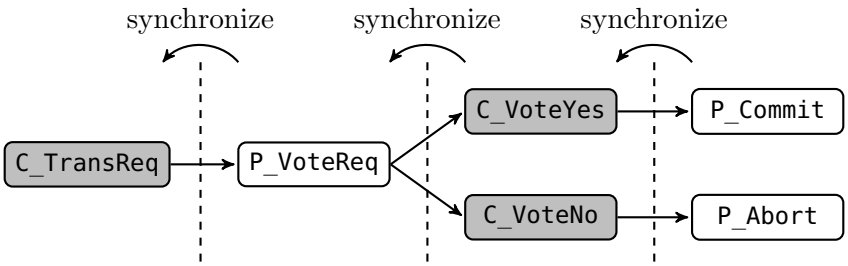


Figure 4.9: 2PC call hierarchy (from left to right) and proof outline (right to left)

4.7.2 Two-phase Commit

In this section we show that our method applies to realistic programs with intricate concurrency by verifying full functional correctness of the two-phase commit (2PC) protocol. The protocol employs a *coordinator* process to consistently replicate transactions among a set of *participant* processes. In the first phase, the coordinator broadcasts incoming request to all participants, which respond either with a "yes" vote to commit, or a "no" vote to abort. In the second phase, the coordinator processes incoming votes as follows: (1) If *all* participants voted "yes" it broadcasts a "commit" message, or (2) as soon as *a single* participant votes "no" it broadcasts an "abort" message. Due to asynchrony and message reordering, the protocol implementation must be robust against unexpected situations. For example, a participant can receive an abort message before it receives the corresponding vote request.

Figure 4.9 shows the message handlers of the protocol we implemented in Civl, together with the asynchronous communication structure. For example, `P_VoteReq` is a participants handler for vote requests, which asynchronously invokes either the coordinators `C_VoteYes` or `C_VoteNo` handler. To reason about the protocol, we use a variable `state` such that for every transaction `xid` and process `pid`, `state[xid][pid]` is one of `INIT`, `COMMIT`, or `ABORT`. We prove a top-level atomic action specification for `C_TransReq` that states that for a fresh `xid`, `state[xid]` is consistently updated, i.e., there are no two processes such that one is `COMMIT` and the other one `ABORT`. Figure 4.9 also shows the proof outline, making repeated use of synchronization. Here we focus on the first synchronization of `P_Commit` and `P_Abort`, which requires them to be left movers. A priori these operations do not commute, because they write the conflicting values `COMMIT` and `ABORT` to `state[xid][pid]`, respectively. However, by making it explicit that the coordinator has to decide on a transaction first, the following abstractions are commutative:

```
action P_Commit (pid,xid):
  assert state[xid][C] == COMMIT
  state[xid][pid] := COMMIT
```

```
action P_Abort (pid,xid):
  assert state[xid][C] == ABORT
  state[xid][pid] := ABORT
```

Our proof of 2PC confirms that the benefit of reduced invariant complexity in structured multi-layer refinement proofs [56] carries over to the asynchronous setting. In particular, we could state the central correctness invariant in terms of the protocol mechanism (i.e., voting and phases) after hiding low-level implementation details (i.e., counting).

4.7.3 Task Distribution Service

Finally, we verified a task distribution service inspired by a set of benchmarks from [12]. This example captures a whole class of similar benchmarks, where a set of *independent* tasks is processed by passing through a sequence of stages. The result of every stage is asynchronously communicated to the next stage, and different tasks can run through different stages. However, concurrent tasks do not interfere with each other. With this key difference to examples like 2PC, we can avoid the overhead of stepwise synchronization over several layers. Instead, synchronization can be applied to eliminate long (and even unbounded) chains of asynchronous calls in a single layer.

To summarize, synchronization is applicable to tightly interfering programs using program layers, and less interference leads to even simpler proofs.

4.8 Related Work

The idea of taming concurrency through synchrony is also at the heart of other works. Brisk [12] computes canonical sequentializations of message-passing programs by matching sends with corresponding receives. Our work differs in the programming model (dynamic thread creation vs. parametric processes with blocking receives) and the verification goal (deductive functional correctness vs. automatic deadlock-freedom). The work in [20] proposes the notion of robustness against concurrency as correctness condition for a class of event-driven programs. That is, the sequential behavior of a program is the underlying specification, and asynchronous executions are checked to conform to sequential executions. In contrast, we use synchronization to simplify the verification of safety properties.

There are several recent papers on mechanized verification of distributed systems. IronFleet [55] embeds TLA-style state-machine modeling [76] into the Dafny verifier [77] to refine high-level distributed systems specifications into low-level executable implementations. They use a fixed 3-layer design and one-shot reductions to atomic actions, while our program layers are more flexible. Ivy [98] organizes the search for an inductive invariant as a collaborative process between automatic verification attempts and user guided generalizations of counterexamples to induction in a graphical model. They use a restricted modeling and specification language that makes their verification conditions decidable. We rely on small partitioned verification conditions that can be discharged by an SMT solver [35]. PSync [40] uses a synchronous round-based model of communication for the purpose of program design and verification, shifting the complexity of efficient asynchronous execution to a runtime system. We allow explicit control over low-level details at the potential cost of increased verification effort. Verdi [116] lets the programmer provide a specification, implementation, and proof of a distributed system under an idealized network model. Then the application is automatically transformed into one that handles faults via verified system transformers. The rely-guarantee rule of [49] and the ALS types of [67] target a weaker form of asynchrony, where a single task queue atomically executes one task at a time.

Concurrent separation logic (CSL) [92] was devised for modular reasoning about multi-threaded shared-memory programs, focusing on the verification of fine-grained concurrent data structures. CSL adequately addresses the problem of reasoning about low-level concurrency related to dynamic memory allocation, but still suffers from the complications of a monolithic approach to invariant discovery for protocol-level concurrency. Recently,

CSL has been applied to message-passing programs. The approach in [93] uses CSL to link implementation steps to atomic actions, and then relies on a model checker to explore the interleavings of those atomic actions. The work in [104] addresses the composition of verified protocols using ideas from separation logic. The actor services of [107] focus on compositional verification of response properties of message-passing programs.

4.9 Conclusion

The contribution of this paper are proof rules to simplify the reasoning about asynchronous concurrent programs. The impact of our work must be understood in the context of our two-pronged strategy for aiding interactive and automated verification of asynchronous programs. First, our proof rules enable asynchronous computation to be summarized analogous to the summarization of synchronous computation by pre- and post-conditions. This capability enables the construction of syntax-driven and structured proofs of asynchronous programs. Second, the program simplification enabled by our proof rules attacks the nemesis of complex invariants induced by a large number of interleaved executions. Instead of writing a large and complex invariant justifying the overall correctness of the program, the programmer may now write a sequence of simpler invariants, each justifying a program simplification.

Our proof method decomposes the task of proving the correctness of a large asynchronous program into formulating and automatically discharging smaller independent proof obligations. These proof obligations show that an atomic action commutes with other atomic actions; that an atomic action summarizes the effect of a statement in a given context; and that an assertion is an inductive invariant for a simpler program, where asynchronous procedure calls are replaced by synchronous (immediate) atomic actions. Using our method, the automatable part of a concurrent verification problem—i.e., the safety proof given an inductive invariant—remains automatable, and the creative part—i.e., the discovery of an appropriate invariant—is greatly simplified by structuring it into smaller proof obligations, each of which can still be discharged automatically.

Acknowledgments

This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

5 Inductive Sequentialization of Asynchronous Programs

Abstract. Asynchronous programs are notoriously difficult to reason about because they spawn computation tasks which take effect asynchronously in a nondeterministic way. Devising inductive invariants for such programs requires understanding and stating complex relationships between an unbounded number of computation tasks in arbitrarily long executions. In this paper, we introduce *inductive sequentialization*, a new proof rule that sidesteps this complexity via a *sequential reduction*, a sequential program that captures every behavior of the original program up to reordering of coarse-grained commutative actions. A sequential reduction of a concurrent program is easy to reason about since it corresponds to a simple execution of the program in an idealized synchronous environment, where processes act in a fixed order and at the same speed. We have implemented and integrated our proof rule in the Civl verifier, allowing us to provably derive fine-grained implementations of asynchronous programs. We have successfully applied our proof rule to a diverse set of message-passing protocols, including leader election protocols, two-phase commit, and Paxos.

5.1 Introduction

Asynchronous programming is widely adopted for building responsive and efficient software. Unlike synchronous procedure calls that block the caller and hence must be executed immediately, asynchronous procedure calls do not block the caller and can be executed in parallel. Depending on the nature of the application, an asynchronous call could either execute in the same process (on another thread), a different process on the same node, or a different node entirely. Asynchronous programming is essential for distributed fault-tolerant software and client-server applications.

Asynchronous programs are notoriously hard to get right. There is inherent nondeterminism in their semantics due to the different orders in which asynchronous calls can execute. This complexity is exacerbated by further nondeterminism due to the execution platform, e.g., network delays and partitions in distributed applications. A promising approach to proving the correctness of realistic implementations is to go through a sequence of *abstraction* steps. Each abstraction step leads to a successively simpler program such that the correctness of the most abstract program implies the correctness of the most concrete program. Alternatively, a realistic implementation could be derived from an abstract (and obviously correct) program through a sequence of *refinement* steps. Devising an

automated program verifier that enables refinement proofs is non-trivial and has received a great deal of attention recently [26, 56, 55, 52, 112, 71, 34, 53].

Each link in the chain of programs connected together by refinement steps must be justified by a proof rule. A useful proof rule must be sound, broadly applicable, and able to simplify reasoning about programs. Many such proof rules already exist including (1) variable introduction and elimination useful for changing the state representation, (2) reduction [80] for eliminating preemptions and creating atomic blocks, and (3) summarization for creating summaries of a block of code that executes atomically. These proof rules work together symbiotically and have been implemented to great effect in several refinement-oriented program verifiers [41, 26, 56].

In this paper, we introduce *Inductive Sequentialization* (IS), a new proof rule that works harmoniously with the other aforementioned proof rules, thereby extending the overall applicability of refinement-oriented program verifiers. IS simplifies reasoning about unbounded concurrent executions of an asynchronous program by reducing its correctness to that of a *single* interleaving of the concurrently-executing actions of the program. Our experience shows that the sequential reduction established by IS for a distributed protocol (like Chang-Roberts [27] and Paxos [75]) corresponds to an execution of the protocol in an idealized environment where processes execute in a fixed order, at the same speed, and messages are delivered immediately unless they are lost. This is the simplest execution of the protocol to reason about.

The goal of IS is to show that an asynchronous program $\mathcal{P}$ is a refinement of a sequential program $\mathcal{S}$. Here refinement means that the summary, the relation between initial and terminating states, of $\mathcal{P}$ is included in that of $\mathcal{S}$. Since any non-terminating program can be abstracted by one that terminates after a nondeterministically chosen number of steps, IS is capable of reasoning about all reachable states and arbitrary safety properties of asynchronous programs. IS combines inductive reasoning, showing that $\mathcal{S}$ summarizes a single fixed interleaving π of the asynchronous calls in $\mathcal{P}$, with commutativity reasoning, showing that focusing only on this single interleaving is sound.

The induction argument in IS is based on a user-provided (nondeterministic) procedure $\mathcal{I}$ that represents *all* prefixes of π. The asynchronous procedure calls whose effect is not included in a particular execution of $\mathcal{I}$ remain asynchronous and executable in an arbitrary order. This proof artifact is the analog of an inductive invariant in a proof of safety. However, unlike classical inductive invariants, inductiveness in our proof rule is shown only w.r.t. a *single* operation at a time, determined by π. The verification conditions prescribed by IS check that $\mathcal{S}$ is the "maximal" prefix of $\mathcal{I}$, which represents the complete interleaving π where no asynchronous calls remain to execute.

It remains to be shown that the sequential order determined by π and summarized by $\mathcal{I}$ captures all terminating executions of $\mathcal{P}$. To achieve this goal, we exploit the concept of a *left mover* [80], an atomic operation that may execute earlier than other concurrently-executing operations without changing the final state. If all operations in $\mathcal{P}$ are left movers, then they can be reordered arbitrarily, in particular following the fixed interleaving π, thus allowing us to conclude that every terminating state of $\mathcal{P}$ can be reached by π. This approach does not work on practical protocols because the concrete operations in $\mathcal{P}$ are not left movers. However, we observed that if π is suitably chosen, then for each operation A executed in π, there is an abstraction A' of A such that A' is a left mover and A' behaves identically to A in the context of π. This observation allows us to replace A with A' when performing the inductiveness check in the IS proof rule. This

tight combination of induction, reduction, and abstraction is one of the main technical contributions of our work (described in detail in Section 5.4).

The applicability of IS is governed by two hypotheses which hold in well-designed asynchronous systems: (1) to ensure responsiveness, these systems extensively use short-living asynchronous tasks, e.g., message handlers, that can execute in parallel, and (2) asynchrony is meant to improve performance but not modify the logic of the application, i.e., the asynchronous behaviors should be equivalent to idealized *synchronous* behaviors where processes act at the same speed and the infrastructure, e.g., the network, is synchronous. The second hypothesis in particular has been addressed and justified in the context of a wide class of applications [20, 22, 29, 42, 34]. The first hypothesis offers more opportunities for a proof tactic based on reordering actions in an execution, while the second enables the reduction to reasoning about a single interleaving.

We have implemented IS as an extension of Civl [56], a verification system for concurrent programs based on automated refinement reasoning. This extension allows IS to be interleaved with the other proof tactics implemented by Civl. We have evaluated the usability of IS on a diverse set of message-passing protocols, including leader election protocols like Chang-Roberts, a non-trivial version of two-phase commit where nodes can abort early, and Paxos. We demonstrate that our proof rule enables sequentializations of all these protocols with a high degree of automation. Our evaluation shows that IS supports simple sequential reductions of complex protocols. Furthermore, the proof artifacts needed to establish the soundness of these reductions are also devised thinking only about a single fixed interleaving. Exploiting IS and other proof rules already implemented in Civl, we are able to derive protocol implementations comprising fine-grained, verified event handlers which are similar to the unverified implementations written by programmers today.

In summary, this paper contributes: (1) a new proof rule called Inductive Sequentialization for eliminating concurrency from asynchronous programs, (2) an implementation of this rule in the Civl verifier, and (3) a demonstration of its usefulness on a variety of challenging examples.

5.2 Overview

In this section we provide an overview of inductive sequentialization (IS). We motivate the challenges of deductive verification of asynchronous programs on a running example and then illustrate the concepts of IS on this example.

5.2.1 Motivation

Verification of concurrent programs. We present our verification technique in a general framework based on (gated) atomic actions over shared state and asynchronous thread creation, which abstracts away the details of any particular programming system irrelevant to our development. We will illustrate these concepts and our contribution on an example.

Figure 5.1-① shows a simple *broadcast consensus* protocol for n nodes (numbered from 1 to n) to agree on a common value. The local states of the nodes are represented using arrays, i.e., `value`$[i]$ holds the input value of node i and `decision`$[i]$ stores the final decision of node i. For each node i there are two concurrent threads created by

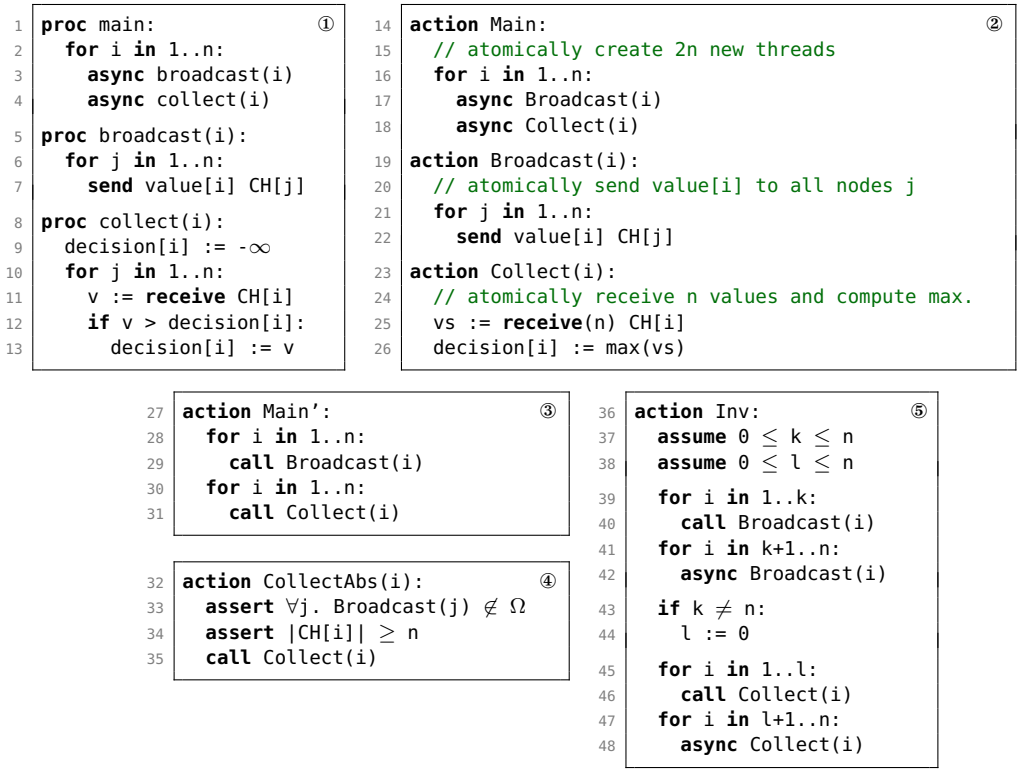


Figure 5.1: Broadcast consensus protocol. ① Original program. ② Program after reduction to atomic actions. ③ Sequentialization. ④ Abstraction of `Collect` action. ⑤ Partial sequentialization.

the asynchronous calls in procedure `main`: one thread executes procedure `broadcast`(i) which sends the value of node i to every other node j, and the other thread executes procedure `collect`(i) which receives n values and stores the maximum as its decision. We consider the channels `CH`$[i]$ for exchanging messages to be multisets (or bags) which models a network where messages can be delayed and delivered out-of-order, and the `receive` statement is blocking. Since every node receives the values of all other nodes, it is the case that, after the protocol finishes, all nodes must have decided on the same value, i.e.,

$$\forall i, j \in [1, n].\ \texttt{decision}[i] = \texttt{decision}[j], \tag{5.1}$$

where $[1, n]$ denotes the set of integers from 1 to n. However, proving this property directly on the code in Figure 5.1-① is notoriously complicated, i.e., requires an inductive invariant that is disproportionally complicated given the simplicity of the protocol. The challenge is that the send and receive operations across all nodes can execute in many different orders. An inductive invariant has to capture all of these orders, and represent every possible intermediate state that can occur. In (5.2) below we show that, even after reduction, the required inductive invariant remains complicated. This is in contrast to the following intuitive reasoning a programmer would employ to understand the correctness of the protocol:

> "First, all nodes send their values to each other (the order does not matter), and then, consequently, every node receives the same set of n values to compute the maximum (the order does not matter)."

Our proof rule is designed to facilitate this kind of reasoning about only a representative set of execution orders. In particular, we enable the programmer to think and reason about the program *sequentially*. To justify that we can focus the reasoning task on certain sequential execution orders and ignore all other concurrent execution orders, we build on the theory of mover types and reduction [80, 41, 56].

Atomic actions, mover types, and reduction. An execution of the program in Figure 5.1-① is naturally understood as an interleaving of small atomic (i.e., uninterruptible) actions of different threads. For instance, reading or writing a variable, sending a message, and spawning a new thread are all examples of fine-grained atomic actions. However, atomic actions are equally well suited to specify coarser-grained operations, and then the verification task can be understood as the sound summarization of fine-grained concurrent executions by large atomic actions. Concretely, we consider atomic actions of the form (ρ, τ), where ρ is a set of states (or one-state predicate), called *gate*, that specifies the states from which the action does not fail (like an assertion), and τ is a transition relation (or two-state predicate) that specifies the possible state transitions when the action executes (possibly including newly created threads). Note that the separation of ρ and τ is important to distinguish failure from blocking.

To formalize the idea that the execution order of atomic actions sometimes does not matter, we assign a *mover type* [47] to every atomic action in a program. An atomic action is a *left (right) mover* if it can be commuted to the left (right) of every other atomic action executed by a different thread, without altering the outcome of the execution. For example, over bag channels as in Figure 5.1-①, where messages can be received in an arbitrary order, `receive` is a right mover and `send` is a left mover. Furthermore, asynchronous calls (i.e., just the action of creating a new thread) are left movers, and local variable accesses like reading `value`$[i]$ and writing `decision`$[i]$ are both left and right movers (because no two concurrent threads access them at the same index i). Note that commutativity is checked pairwise among the pool of actions in a given program, only using the action definitions without considering reachable program executions. Thus an action can be a mover in one program, but not in another.

Given the mover types of the atomic actions in a program, consider a thread that, according to the static program order, executes a sequence of atomic actions with the following mover types: first a sequence of zero or more right movers, then at most one non-mover, and finally a sequence of zero or more left movers. We call such a sequence *atomic*, because any execution where these actions are interleaved with actions from other threads can be permuted into an equivalent execution where the atomic sequence is uninterrupted by other threads. Following this argument, the *reduction* method lets us summarize atomic sequences into bigger atomic actions. Figure 5.1-② shows the result of applying reduction to Figure 5.1-①, where all three procedures are atomic; `main` is a sequence of left-moving asynchronous calls, `broadcast` is a sequence of left-moving sends and both-moving reads of `value`$[i]$, and `collect` is a sequence of right-moving receives and both-moving reads and writes of `decision`$[i]$.

Here we want to stress two important points. First, we conveniently represent atomic actions as code blocks. While this makes, e.g., the action `Broadcast`(i) (Figure 5.1-②) visually appear the same as the procedure `broadcast`(i) (Figure 5.1-①), it represents an atomic broadcast of `value`$[i]$ to all other nodes in one single step. Second, atomic actions can create new concurrent threads, represented as asynchronous calls. For example,

executing action `Main` (Figure 5.1-②) has the effect of atomically creating $2n$ new threads (n `Broadcast`'s and n `Collect`'s), without yet executing any of their steps. We call these new threads *pending asyncs* (PAs), since their future effect is not summarized into the parent action. Formally, a PA is an action name together with parameter values, and we denote a set of pending asyncs with the variable Ω.

For the presentation in this paper we assume that programs are given as a set of atomic actions with PAs. In practice, this means that reduction is typically applied before our new technique, e.g., using the framework of layered concurrent programs [70]. In theory, this assumption is without loss of generality, since a non-atomic sequence of actions `A;B` can be represented with `A` having a PA to its continuation `B`.

Atomic actions are no silver bullet. Reducing a program to atomic actions with PAs is no panacea for the deductive verification of concurrent programs. In general, PAs still cause many different concurrent execution orders, and an inductive invariant has to capture all of them. For example, consider the inductive invariant for Figure 5.1-②:

$$\begin{aligned}
&\big(\Omega = \{\texttt{Main}\} \wedge (\forall i \in [1,n].\ \texttt{CH}[i] = \varnothing)\big)\\
\vee\ &\big(\exists D \subseteq [1,n].\ (\forall i \in [1,n].\ \texttt{CH}[i] = \{\texttt{value}[j] \mid j \in D\}) \wedge\\
&\quad \Omega = \{\texttt{Broadcast}(i) \mid i \in [1,n] \setminus D\} \uplus \{\texttt{Collect}(i) \mid i \in [1,n]\}\big)\\
\vee\ &\big(\exists D \subseteq [1,n].\ (\forall i \in [1,n] \setminus D.\ \texttt{CH}[i] = \{\texttt{value}[j] \mid j \in [1,n]\} \wedge\\
&\quad (\forall i \in D.\ \texttt{decision}[i] = \max\{\texttt{value}[j] \mid j \in [1,n]\}) \wedge\\
&\quad \Omega = \{\texttt{Collect}(i) \mid i \in [1,n] \setminus D\}\big)
\end{aligned} \tag{5.2}$$

The first disjunct captures the initial state with a single PA to `Main` and all channels empty, the second disjunct captures the intermediate states where any subset of nodes D performed their `Broadcast` and the remaining `Broadcast`'s and all `Collect`'s are still pending, and the third disjunct captures the intermediate states where any subset of nodes D performed their `Collect`. Setting $D = [1,n]$ in the third disjunct implies the correctness property (5.1) and that no PAs are left (i.e., $\Omega = \varnothing$). Note that in this example the `Collect`'s happen after the `Broadcast`'s, because the `Collect`'s block until there are n messages in their corresponding channel. However, there are still two sources of complexity in reasoning with invariant (5.2) that our new method addresses. First, the ordering of `Broadcast`'s before `Collect`'s is not made explicit in the invariant; to show the inductiveness of (5.2) we have to prove that in a state with remaining `Broadcast`'s (i.e., satisfying the second disjunct) the `Collect`'s are blocked. Second, the execution order among the `Broadcast`'s and among the `Collect`'s does not matter, and thus we only want to reason about the "sequential" execution of `Broadcast`'s happening in order from 1 to n, and similarly for `Collect`'s.

5.2.2 Inductive Sequentialization

In this paper we provide an approach to enable sequential reasoning about asynchronous concurrent programs in the form of a program-transforming (refinement) proof rule called *inductive sequentialization (IS)*. A first idea of IS is to exploit mover types to *eliminate* PAs from atomic actions. By that we mean instead of an action creating a PA that takes effect asynchronously at a later time, we establish conditions that let us reason about the PA taking effect immediately, and thus combine it with the calling action. In particular,

this is the case if the PA is a left mover, because then it can be moved earlier in an execution, to immediately follow its caller. However, atomic actions can generally create unboundedly many PAs, and the elimination of one PA can also introduce new ones if the eliminated PA has PAs itself. Our solution with IS to eliminate unboundedly many PAs at once is an *induction* scheme that has to address the following challenges:

C1 How to express intermediate results during the elimination of unboundedly many PAs?

C2 How to control the order of eliminating PAs to enable sequential reasoning?

C3 How to eliminate PAs that are not left movers?

We illustrate these challenges and how they are solved by IS on the consensus protocol in Figure 5.1-②. In particular, we show how an application of IS derives that the consensus protocol is a sound refinement of the sequential program `Main'` in Figure 5.1-③. `Main'` represents a very simple schedule of the consensus protocol where all `Broadcast`'s are executed before all `Collect`'s, and in a round-robin fashion.

Challenge C1 is addressed by an *invariant action*, namely `Inv` in Figure 5.1-⑤. It represents (summarizes) the intermediate results during the induction, i.e., all prefixes of the schedule defining `Main'`. Therefore, either only some pending `Broadcast`'s are already eliminated and the rest of the PAs are still pending (when $k < n$), or all `Broadcast`'s and some number of `Collect`'s are already eliminated (when $k = n$). Note that the number of `Broadcast`'s or `Collect`'s that are summarized by `Inv` is chosen nondeterministically. This allows `Inv` to summarize *all* prefixes of the schedule defining `Main'`, one prefix for every choice of k and l. While we believe that the code of `Inv` is quite simple to understand, this is of course not the only way to represent prefixes of `Main'`. In general, IS is not sensitive to a particular representation.

Customary for an induction, IS has a base case and an induction step. The *base case* of IS, i.e., that the effect of `Main` is captured by `Inv`, is satisfied with $k = 0$. For the *induction step*, i.e., that the elimination of a `Broadcast` or `Collect` PA from `Inv` is still captured by `Inv`, we want to proceed with our sequential intuition and thus have to address challenge C2.

Every `Broadcast` PA created by a transition of `Inv` is a left mover, and thus any one of them could be eliminated next. However, the natural choice is to eliminate `Broadcast`$(k + 1)$. For `Inv` this is also the only way to satisfy the induction step, by advancing from k to $k+1$. To communicate this choice to IS, the proof rule is parameterized with a *choice function* that selects the next PA to eliminate from any state with PAs left to eliminate. The choice function for our example always selects the `Broadcast`(i) PA with the smallest parameter i, as long as there exists one, and otherwise it selects the `Collect`(i) PA with the smallest parameter i.

The `Collect` actions are, however, not left movers, manifesting challenge C3. First, receives do not commute to the left of sends, and second, left movers also have to satisfy a *non-blocking* condition, namely that it is always possible to execute the action (from any state that satisfies its gate). A `Collect` action blocks in every state that has less than n messages to receive. The solution provided by IS is that *abstractions* for the atomic actions to be eliminated can be provided, which are used both for establishing left-moverness and to eliminate the PA selected by the choice function in the induction step. Note that there

always exists a trivial abstraction that satisfies the mover conditions. Given an inductive safety invariant I, e.g., the one in Equation 5.2, every action can be abstracted to an arbitrary step between two states satisfying the invariant (i.e., an action defined by $\rho = I$ and $\tau = I \wedge I'$, where I' uses primed variables to represent the end state of a transition), which commutes with itself. This abstraction is of course not useful, since our goal is to avoid reasoning about this invariant in the first place.

We abstract `Collect` to `CollectAbs` given in Figure 5.1-④, which strengthens the gate (represented as assertion) from $\rho_{\texttt{Collect}} = \mathit{true}$ to $\rho_{\texttt{CollectAbs}} = \forall j.\ \texttt{Broadcast}(j) \notin \Omega \wedge |\texttt{CH}[i]| \geq n$. This assertion represents a condition which holds in the schedule defining `Main'`, i.e., there are no concurrent `Broadcast`'s when a `Collect` action is spawned and the channel accessed by the `Collect` already contains n messages. This makes `CollectAbs` non-blocking and a left mover. Thus IS is applicable and we show how `CollectAbs` is used in the induction step.

Assume that some prefix of `Collect`'s from 1 to l are already eliminated, and `Collect`$(l+1)$ should be eliminated next (as indicated by the choice function). This is where the supplied abstraction comes in; instead of `Collect`$(l+1)$ we perform the induction step with `CollectAbs`$(l+1)$. In particular, this means that we need to show that after the transition of `Inv` it holds that Ω contains no `Broadcast`'s and $|\texttt{CH}[l+1]| \geq n$. Observe that this is an entirely sequential verification condition, which holds because all `Broadcast`'s happen before the `Collect`'s in `Inv`. There are two important points to note about abstractions supplied to IS. First, these abstractions are merely proof artifacts used during IS. They are neither introduced into the program before nor left in the program after IS. Second, an abstraction is always only used for the single PA selected by the choice function. In particular, in `Inv` (Figure 5.1-⑤), `CollectAbs` is neither used for the already sequentialized `Collect`'s (line 46) nor for the remaining PAs after $l+1$ (line 48). While in this example the gate of `CollectAbs` also holds there, this is not the case in general (see Section 5.4). Thus abstraction during IS is more powerful than abstraction before applying IS.

Finally, similar to a sequential loop invariant, which allows us to fast-forward through all iterations of a loop, the invariant action in IS allows us to fast-forward through all eliminations of PAs. For `Inv` (Figure 5.1-⑤) this means that we want to fast-forward to the point where all `Broadcast`'s and `Collect`'s have been eliminated. This is the case when $k = l = n$, and thus the result obtained by IS is the atomic action `Main'` in Figure 5.1-③. The formal guarantee of IS is that `Main` (Figure 5.1-②) *refines* `Main'` (Figure 5.1-③). Hence, we can replace reasoning about `Main` with reasoning about `Main'`. This action captures exactly how we imagined the broadcast consensus protocol to execute sequentially, and IS guarantees that this is a sound summary of all concurrent executions. Now we can prove property (5.1) using simple sequential reasoning, i.e., using sequential loop invariants for a particular execution order, as opposed to the complicated flat inductive invariant (5.2). Note also that the proof artifacts required to apply IS, i.e., the invariant action `Inv` and the abstraction `CollectAbs`, were themselves devised from this particular execution order only.

5.3 Preliminaries

In this section we provide the necessary definitions to formalize IS in the next section.

Variables and stores. Let *Var* be a set of *variables* partitioned into *global variables* *GVar* and *local variables* *LVar*. A *store* is a mapping $\sigma : Var \to Val$ that assigns a *value* from a domain *Val* to every variable. Similarly, $g : GVar \to Val$ is a *global store* and $\ell : LVar \to Val$ is a *local store*. Let $g{\cdot}\ell$ denote the combination of g and ℓ into a store.

Actions and programs. Let $\mathcal{A}$ be a set of *action names* (usually denoted by uppercase letters like A in this paper). A *pending async (PA)* is a pair (ℓ, A) of a local store ℓ and an action name A (ℓ holds parameter values for A). A *gated atomic action*, or *action* for short, is a pair (ρ, τ), where the *gate* ρ is a set of stores and the *transition relation* τ is a set of transitions (σ, g, Ω) where σ is a (combined global and local) store, g is a global store, and Ω is a finite multiset of pending asyncs. A *program* is a finite mapping from action names to actions. Every program $\mathcal{P}$ must contain the dedicated action name `Main`, i.e., $\texttt{Main} \in \mathrm{dom}(\mathcal{P})$, and every action name that appears in $\mathcal{P}$ must be mapped to an action. For a set of action names $\mathcal{E}$ and a transition $t = (\sigma, g, \Omega)$ we define $PA_{\mathcal{E}}(t)$ to be the set of PAs to $\mathcal{E}$ in Ω, i.e., $PA_{\mathcal{E}}(t) = \{(\ell, A) \in \Omega \mid A \in \mathcal{E}\}$. To simplify the notation we will identify a PA (ℓ, A) with the singleton multiset $\{(\ell, A)\}$, and thus write $(\ell, A) \uplus \Omega$ for adding (ℓ, A) to Ω. We write $\mathcal{P}[A \mapsto a]$ to denote the program $\mathcal{P}'$ that is equal to $\mathcal{P}$ except that $\mathcal{P}'(A) = a$.

Executions. A *configuration* is a pair (g, Ω) of a global store g and a finite multiset of pending asyncs Ω,[1] or a unique *failure configuration* $\lightning$. We define the transition relation $\xrightarrow{\mathcal{P}}$ (omitting $\mathcal{P}$ when it is understood from the context) as

$$\frac{g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g', \Omega') \in \tau}{(g, \underline{(\ell, A)} \uplus \Omega) \to (g', \Omega \uplus \Omega')} \qquad \frac{g{\cdot}\ell \notin \rho}{(g, \underline{(\ell, A)} \uplus \Omega) \to \lightning}$$

where $\mathcal{P}(A) = (\rho, \tau)$. In a configuration (g, Ω), any PA $(\ell, A) \in \Omega$ can be scheduled to execute next; if the gate of A does not hold (i.e., $g{\cdot}\ell \notin \rho$) then the program "fails", otherwise a transition $(g{\cdot}\ell, g', \Omega') \in \tau$ atomically updates the global store to g' and creates new PAs Ω' (that are added to Ω). Underlining optionally denotes the PA that is executed in a transition. An *execution* π is a sequence of configurations $c_0 \to c_1 \to \cdots$. We call an execution *initialized* if it starts in a configuration $(g, (\ell, \texttt{Main}))$ with a single PA to `Main`, *terminating* if it ends in a configuration $(g, \varnothing)$ with no PAs, and *failing* if it ends in the failure configuration $\lightning$.

Refinement. We define the notion of refinement between both actions and programs [56]. Let $\circ$ denote the relation composition operator (sets are unary relations). In particular, $\rho \circ \tau = \{(\sigma, g, \Omega) \in \tau \mid \sigma \in \rho\}$ denotes the subset of transitions in τ that start from a store $\sigma \in \rho$.

Definition 1. An action $a_1 = (\rho_1, \tau_1)$ *refines* an action $a_2 = (\rho_2, \tau_2)$, denoted $a_1 \preccurlyeq a_2$, if (1) $\rho_2 \subseteq \rho_1$ and (2) $\rho_2 \circ \tau_1 \subseteq \tau_2$. We also say that a_2 *abstracts* a_1.

The first condition states that a_2 has to preserve the failures of a_1. The second condition states that a_2 has to preserve the transitions of a_1 for initial stores from which

[1] In our formalization we use multisets of PAs both "statically" in the definition of actions, and "dynamically" in configurations.

a_2 cannot fail. Thus, a_2 can fail more often than a_1. For programs we are interested in the preservation of failing and terminating behaviors of initialized executions. Let $\mathit{Good}(\mathcal{P})$ be the set of initial stores from which $\mathcal{P}$ cannot fail, and $\mathit{Trans}(\mathcal{P})$ the relation between initial and final stores of terminating executions:

$$\mathit{Good}(\mathcal{P}) = \big\{ g{\cdot}\ell \mid \neg \big(g, (\ell, \texttt{Main})\big) \xrightarrow{\mathcal{P}}{}^{*} \lightning \big\};$$
$$\mathit{Trans}(\mathcal{P}) = \big\{ (g{\cdot}\ell, g') \mid \big(g, (\ell, \texttt{Main})\big) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \big\}.$$

Definition 2. A program $\mathcal{P}_1$ *refines* a program $\mathcal{P}_2$, denoted $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, if (1) $\mathit{Good}(\mathcal{P}_2) \subseteq \mathit{Good}(\mathcal{P}_1)$ and (2) $\mathit{Good}(\mathcal{P}_2) \circ \mathit{Trans}(\mathcal{P}_1) \subseteq \mathit{Trans}(\mathcal{P}_2)$. We also say that $\mathcal{P}_2$ *abstracts* $\mathcal{P}_1$.

Intuitively, this notion of refinement establishes a relationship between the summaries (input-output relations) of $\mathcal{P}_1$ and $\mathcal{P}_2$. If the programs contain no assertions (i.e., $\mathit{Good}(\mathcal{P}_1)$ and $\mathit{Good}(\mathcal{P}_2)$ contain all possible stores), it requires that the summary of the "concrete" program $\mathcal{P}_1$ is included in the summary of the "abstract" program $\mathcal{P}_2$. When assertions are present, it requires that $\mathcal{P}_2$ fails more often (condition 1) and that the summary of $\mathcal{P}_1$, restricted to initial states where $\mathcal{P}_2$ does not fail, is included in the summary of $\mathcal{P}_2$ (condition 2). This is sound in the sense that if $\mathcal{P}_2$ does not fail, then (1) $\mathcal{P}_1$ does not fail as well, and (2) any property of terminating states of $\mathcal{P}_2$ is also valid for the terminating states of $\mathcal{P}_1$.

Proposition 1. *If $a \preccurlyeq a'$, then $\mathcal{P}[A \mapsto a] \preccurlyeq \mathcal{P}[A \mapsto a']$.*

Left movers. An action $l = (\rho_l, \tau_l)$ is a *left mover w.r.t. an action* $x = (\rho_x, \tau_x)$ if

1. the gate of l is *forward-preserved* by x, i.e., if ρ_l remains true after executing x whenever it was true before,

$$g{\cdot}\ell_l \in \rho_l \wedge (g{\cdot}\ell_x, g', \Omega) \in \rho_x \circ \tau_x \implies g'{\cdot}\ell_l \in \rho_l;$$

2. the gate of x is *backward-preserved* by l, i.e., if ρ_x is true before executing l whenever it is true afterwards,

$$(g{\cdot}\ell_l, g', \Omega) \in \rho_l \circ \tau_l \wedge g'{\cdot}\ell_x \in \rho_x \implies g{\cdot}\ell_x \in \rho_x;$$

3. l *commutes to the left of* x, i.e., if executing x before l leads to a global store that is also possible when executing l before x,

$$g{\cdot}\ell_l \in \rho_l \wedge (g{\cdot}\ell_x, \bar{g}, \Omega_x) \in \tau_x \circ \rho_x \wedge (\bar{g}{\cdot}\ell_l, g', \Omega_l) \in \tau_l$$
$$\implies \exists \hat{g}.\ (g{\cdot}\ell_l, \hat{g}, \Omega_l) \in \tau_l \wedge (\hat{g}{\cdot}\ell_x, g', \Omega_x) \in \tau_x;$$

4. l is *non-blocking*, i.e., if it contains a transition $(\sigma, g, \Omega) \in \tau_l$ from any store σ satisfying the gate ρ_l.

Furthermore, l is a *left mover w.r.t. a program* $\mathcal{P}$, denoted by $\mathit{LeftMover}(l, \mathcal{P})$, if it is a left mover w.r.t. every action in $\mathcal{P}$.

5.4 Inductive Sequentialization

In this section we present the *inductive sequentialization (IS)* proof rule. The context of IS is a program $\mathcal{P}$, an action name M, and a set of action names $\mathcal{E}$. The goal of IS is to eliminate all PAs to $\mathcal{E}$ from M, i.e., to summarize M together with the future effects of the PAs to $\mathcal{E}$ it creates. In particular, the IS proof rule replaces M with a new action that contains no PAs to $\mathcal{E}$. Formally, $\mathcal{P}$ is transformed into a program $\mathcal{P}'$ that is equal to $\mathcal{P}$, except that the action name M is re-mapped to a new action $(\rho_{M'}, \tau_{M'})$, i.e., $\mathcal{P}' = \mathcal{P}[M \mapsto (\rho_{M'}, \tau_{M'})]$. Notice that in general, M does not have to be the `Main` action of $\mathcal{P}$.

The correctness requirement for IS is that $\mathcal{P}$ refines $\mathcal{P}'$, which means that $\mathcal{P}'$ has to preserve both failing and terminating behaviors of $\mathcal{P}$ (see Definition 2). In particular, every terminating state of $\mathcal{P}$ must also be reachable by $\mathcal{P}'$:

$$(g, (\ell, \texttt{Main})) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \implies (g, (\ell, \texttt{Main})) \xrightarrow{\mathcal{P}'}{}^{*} (g', \varnothing).$$

The natural strategy to prove this property is to show that every terminating $\mathcal{P}$-execution π can be rewritten into a terminating $\mathcal{P}'$-execution π' with the same final state, by turning every transition of M in π into a transition of M' in π'. We illustrate this process in Figure 5.2, where ① shows the final part of a $\mathcal{P}$-execution. First M executes from a configuration with two other PAs to X and Y, which creates two new PAs to A and B, and then X, B, Y, A execute to reach a terminating configuration. Suppose $\mathcal{E} = \{A, B\}$, and thus our goal is to obtain the execution in ⑥ which executes M' instead of M, which does not create PAs to A and B. We do so by setting up an induction that stepwise eliminates A and B from the execution in ①. The central artifact for this induction is an *invariant action* I that has to be provided as input to IS. Then the first step in ②, constituting the base case of the induction, is to execute I instead of M, which requires that every transition of M is also a transition of I (or more precisely, that M refines I). At this point the transition of I denotes an "empty sequentialization" which we are going to extend in the next steps to "partial sequentializations", until we obtain the "complete sequentialization" M'. In doing so we control the constructed sequentialization through a *choice function* that determines for every partial sequentialization a *single* PA to sequentialize next. Concretely, in ② we first want to sequentialize A and then B, and thus the choice function selects A in the transition of I (marked with a box around A). We commute A to the left of Y, B, and X to obtain ③, which requires that A is a *left mover* w.r.t. the actions in $\mathcal{P}$. Then the *induction condition* of IS guarantees that the composition of I and A is possible as a single transition of I (corresponding to an extended partial sequentialization), and thus we obtain ④. Crucially, the transition of I in ③ only has to be inductive w.r.t A. Now we proceed similarly with the PA to B—commute B to the left of X and absorb it into I—to obtain ⑤. However, it might be the case that B is not an unconditional left mover. Therefore it is possible to supply abstractions for actions in $\mathcal{E}$ to IS, like B^* for B in ④. These abstractions can take into account the context in which they are sequentialized, e.g., B^* can rely on the fact that A already executed. Finally, in ⑥ we replace I with M', which is constructed from I by removing every transition that has PAs to $\mathcal{E}$, and thus obtain the desired $\mathcal{P}'$-execution.

We remark that in general, IS sequentializes not only the PAs created directly by M, but also transitively created PAs. This capability is essential to sequentialize unbounded sequences of PAs. Furthermore, M' can still have PAs to actions disjoint from $\mathcal{E}$. Then IS

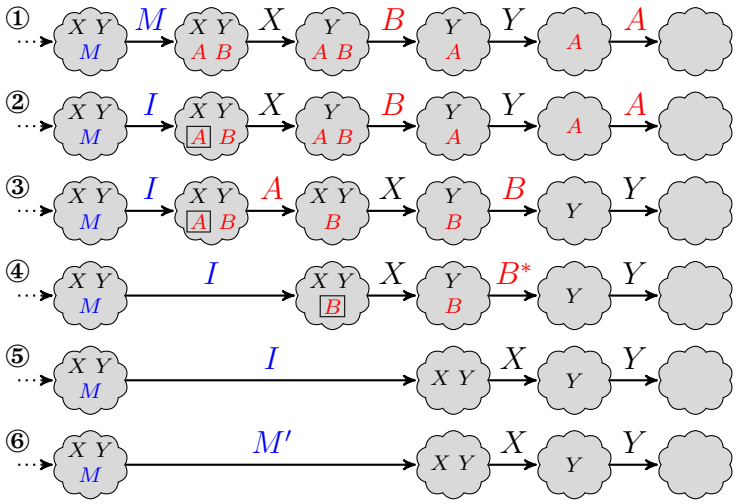


Figure 5.2: Illustration of the induction argument. Clouds represent the set of PAs in a configuration (stores are not shown) and the arrow labels indicate the actions that execute in the transitions from one configuration to the next.

can be applied to M' again, and in Section 5.5.3 we show that iterated application of IS can be beneficial in practice.

Inductive sequentialization. The formal definition of the IS proof rule is given in Figure 5.3. Besides the program $\mathcal{P}$, action name M, and set of action names $\mathcal{E}$ which frame the rule, an *invariant action* (ρ_I, τ_I), a *choice function* f, an *abstraction function* α, and a *well-founded order* over configurations $\gg$ have to be provided. The choice function f selects from every transition t of the invariant action that creates PAs to $\mathcal{E}$ (i.e., $PA_{\mathcal{E}}(t) \neq \varnothing$) one of these PAs. The *abstraction function* α is such that $\alpha(A)$ is an abstraction of $\mathcal{P}(A)$ for every $A \in \mathcal{E}$. Note that we can set $\alpha(A) = \mathcal{P}(A)$ for every A that should not be abstracted. The induction argument outlined above is enabled by three *induction conditions*. To start the induction (cf. Figure 5.2, ①-②), condition **(I1)** requires the invariant action to be an abstraction of the action we rewrite. To end the induction (cf. Figure 5.2, ⑤-⑥), condition **(I2)** requires M to be re-mapped to an action that abstracts the invariant action with all transitions that contain PAs to $\mathcal{E}$ removed. To absorb a PA into the invariant action (cf. Figure 5.2, ③-④), condition **(I3)** is two-fold, corresponding to the failure and behavior preservation requirement. First, after every transition t of the invariant action, if A is the PA selected by the choice function and A^* the abstraction of A, then the gate of A^* has to be satisfied. In other words, any potential failure of A^* has to be propagated into the gate of I. Second, when t is composed with a transition of A^*, then the resulting composite transition must be contained in I and thus possible in a single step. To commute the actions in $\mathcal{E}$ to the appropriate position in the sequentialization (cf. Figure 5.2, ②-③), the *left-mover condition* **(LM)** requires that every abstracted action $\alpha(A)$ is a left mover w.r.t. the original actions in the program $\mathcal{P}$. Thus, abstracted actions do not have to be left movers w.r.t. each other, which is evident in Figure 5.2 where at most one abstraction at a time is part of the execution. Finally, the *cooperation condition* **(CO)** strengthens the standard non-blocking conditions of left movers. It states that is must be possible to execute every abstracted action such that the configuration decreases in some well-founded order $\gg$. (Recall that $\gg$ is well-founded, if there is no infinite sequence $c_0, c_1, c_2, \ldots$ of configurations such that $c_i \gg c_{i+1}$ for every $n \in \mathbb{N}$.) This ensures that it is always possible

$$\boxed{\textbf{Given: } \mathcal{P}, M, \mathcal{E} \qquad \textbf{To invent: } \rho_I, \tau_I, f, \alpha, \gg}$$

$$\mathcal{P}(M) = (\rho_M, \tau_M) \qquad \mathcal{E} \subseteq \mathrm{dom}(\mathcal{P}) \qquad \mathit{WellFounded}(\gg)$$

$$\mathrm{dom}(f) = \{t \in \tau_I \mid \mathit{PA}_{\mathcal{E}}(t) \neq \varnothing\} \qquad t \in \mathrm{dom}(f) \implies f(t) \in \mathit{PA}_{\mathcal{E}}(t)$$

$$\mathrm{dom}(\alpha) = \mathcal{E} \qquad A \in \mathcal{E} \implies \mathcal{P}(A) \preccurlyeq \alpha(A)$$

$$\textbf{(I1)}\ (\rho_M, \tau_M) \preccurlyeq (\rho_I, \tau_I) \qquad \textbf{(I2)}\ (\rho_I, \{t \in \tau_I \mid \mathit{PA}_{\mathcal{E}}(t) = \varnothing\}) \preccurlyeq (\rho_{M'}, \tau_{M'})$$

$$\textbf{(I3)}\ t = (\sigma, g, (\ell, A) \uplus \Omega) \in \rho_I \circ \tau_I \wedge f(t) = (\ell, A) \wedge \alpha(A) = (\rho_{A^*}, \tau_{A^*}) \implies$$
$$g{\cdot}\ell \in \rho_{A^*} \wedge \big((g{\cdot}\ell, g', \Omega') \in \tau_{A^*} \implies (\sigma, g', \Omega \uplus \Omega') \in \tau_I\big)$$

$$\textbf{(LM)}\ A \in \mathcal{E} \implies \mathit{LeftMover}(\alpha(A), \mathcal{P})$$

$$\textbf{(CO)}\ A \in \mathcal{E} \wedge \alpha(A) = (\rho_{A^*}, \tau_{A^*}) \wedge g{\cdot}\ell \in \rho_{A^*} \implies$$
$$\exists g', \Omega'.\ (g{\cdot}\ell, g', \Omega') \in \tau_{A^*} \wedge (g, (\ell, A) \uplus \Omega) \gg (g', \Omega \uplus \Omega')$$

$$\mathcal{P} \preccurlyeq \mathcal{P}[M \mapsto (\rho_{M'}, \tau_{M'})]$$

Figure 5.3: Inductive sequentialization (IS) proof rule.

for the PA elimination process to eventually finish, instead of indefinitely introducing new PAs to be eliminated. While the cooperation condition might seem exotic, its sole purpose is the prevention of unsound IS on pathological corner cases, and the condition is easy to satisfy in practice (see below).

Example 2. Recall the broadcast consensus protocol from Figure 5.1. We formally apply IS to transform `Main` to `Main'` by eliminating all PAs to `Broadcast` and `Collect`. Thus we set $M =$ `Main`, $\mathcal{E} = \{$`Broadcast`, `Collect`$\}$, $M' =$ `Main'`, and $I =$ `Inv`. Recall that `Inv` represents all partial sequentializations where `Broadcast`'s execute in the fixed order from 1 to n, followed by the `Collect`'s executing in the fixed order from 1 to n. Thus **(I1)** `Main` is summarized by `Inv` (for $k = l = 0$) and **(I2)** `Main'` summarizes the only execution of `Inv` without remaining PAs (for $k = l = n$). We define the choice function f such that it either selects the `Broadcast` PA with the smallest parameter if one exists, or otherwise the `Collect` PA with the smallest parameter. `Broadcast` is a left mover w.r.t. $\{$`Main`, `Broadcast`, `Collect`$\}$ and does not need to be abstracted. However, `Collect` is not a left mover because it does not commute with `Broadcast` and it also does not satisfy the non-blocking condition. Thus we supply the abstraction $\alpha($`Collect`$) =$ `CollectAbs` that strengthens the gate to assert that there are no `Broadcast`'s left and at least n messages to receive, which makes `CollectAbs` a left mover. Now the induction condition **(I3)** requires us to discharge the gate of `CollectAbs` in a sequential context when we compose it with `Inv`. This is possible since `Inv` executes all `Broadcast`'s before any `Collect`. We set $\gg$ such that $c \gg c'$ if and only if c has more PAs than c'. Then $\gg$ is clearly well-founded because the number of PAs cannot be negative, and the cooperation condition **(CO)** is satisfied because the execution of `Broadcast`/`Collect` decreases the number of PAs (since they do not create new PAs).

Cooperation is necessary. We illustrate the need for the cooperation condition on the following program.

```
1 action Main:     action Rec:      action Fail:
2   async Rec        async Rec        assert false
3   async Fail
```

This program can fail in two steps by executing `Main` followed by `Fail`. Suppose we want to apply IS with $M = \texttt{Main}$, $\mathcal{E} = \{\texttt{Rec}\}$, and $I = \texttt{Main}$, which satisfies all conditions except cooperation. In particular, notice that `Fail` is not in $\mathcal{E}$ and thus the induction condition does not apply to it. But then M' is constructed from I by removing all transitions from `Main` that have PAs to `Rec`, which means all transitions. Thus, the transition relation of M' is empty, i.e., $\tau_{M'} = \varnothing$ (which we can also represent as `assume false`). Then replacing M with M' would result in a program that cannot fail, which is unsound according to the definition of refinement. The cooperation condition prevents the application of IS because the execution of `Rec` in any configuration results in exactly the same configuration, and thus it is impossible to decrease in any well-founded order.

Checking cooperation is easy. The cooperation condition in Figure 5.3 is *global* in the sense that it requires a decrease $(g, (\ell, A) \uplus \Omega) \gg (g', \Omega \uplus \Omega')$ for any possible Ω. This is the most general condition needed in our soundness proof, but in practice we did not find it necessary for $\gg$ to depend on Ω. Instead it is natural for $\gg$ to be *monotonic*, i.e., that $(g, \Omega) \gg (g', \Omega')$ implies $(g, \Omega \uplus \Omega'') \gg (g', \Omega' \uplus \Omega'')$. Then cooperation can be checked *locally* by showing $(g, (\ell, A)) \gg (g', \Omega')$. Furthermore, we found the following simple generic pattern to apply to all of our examples in Section 5.5. Devise a map g from configurations c to tuples $(x_1, \ldots, x_n)$, such that every x_i is either the number of messages in a certain channel, or the number of PAs of a certain action. Then define $c \gg c'$ if and only if $g(c) > g(c')$, where $>$ denotes the lexicographic order of tuples of natural numbers. This construction guarantees that $\gg$ is well-founded and monotonic, and the cooperation condition is easy to check syntactically.

5.4.1 Soundness Proof

In this section, let $\mathcal{P} \preccurlyeq \mathcal{P}'$ be derived by an application of IS.

Lemma 3. *For every failing $\mathcal{P}$-execution π starting with a transition of M there exists a failing $\mathcal{P}'$-execution π' starting from the same configuration with a transition of M, i.e.,*

$$(g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}}{}^{*} \lightning \implies (g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}'}{}^{*} \lightning .$$

Moreover, π' does not execute more PAs to M than π.

Proof. Let π be a failing $\mathcal{P}$-execution that starts with a transition of M:

$$\pi = (g, \underline{(\ell, M)} \uplus \Omega) \to \cdots \to \lightning .$$

We show how to rewrite π into a failing $\mathcal{P}'$-execution

$$\pi' = (g, \underline{(\ell, M)} \uplus \Omega) \to \cdots \to \lightning .$$

We (conceptually) replace the first transition of M in π with the invariant action I.

Case 1. $g{\cdot}\ell \notin \rho_I$. Then because $\rho_{M'} \subseteq \rho_I$ we obtain

$$\pi' = (g, \underline{(\ell, M)} \uplus \Omega) \to \lightning .$$

Case 2. $g{\cdot}\ell \in \rho_I$. Then because of $(\rho_M, \tau_M) \preccurlyeq (\rho_I, \tau_I)$ we must have

$$\pi = (g, \underline{(\ell, M)} \uplus \Omega) \to (g', \Omega \uplus \Omega') \to \cdots \to \lightning .$$

Furthermore, some transition $t \in \tau_I$ can simulate the first transition in π, and we denote π'' the remainder of π after the first transition:

$$t = (g{\cdot}\ell, g', \Omega') \in \tau_I; \qquad \pi'' = (g', \Omega \uplus \Omega') \to \cdots \to \lightning .$$

We consider the lexicographically ordered measure on π'' comprising (1) the length of π'', ordered by $\geq$, and (2) the final configuration in π'' before the failure, ordered by $\gg$.

Case 2.1. $PA_{\mathcal{E}}(t) = \varnothing$. Then $t \in \tau_{M'}$ and we obtain $\pi' = \pi$.

Case 2.2. $PA_{\mathcal{E}}(t) \neq \varnothing$. Let $(\ell', A) \in \Omega'$ be the PA selected by the choice function, i.e., $f(t) = (\ell', A)$. Let A^* be the abstraction of A. By the induction condition the gate of A^*, which is stronger than the gate of A, holds at the beginning of π'' (i.e., $g{\cdot}\ell' \in \rho_{A^*}$) and because of forward preservation it also holds in every later configuration of π''. In particular, the execution of (ℓ', A) cannot be the failing transition.

Case 2.2.1. (ℓ', A) executes in π''. We turn this transition of A into on of A^*, which can simulate the transition of A. Because A^* is a left mover, we stepwise commute it to the left in π'' such that it becomes the first transition. Let t' be the corresponding transition in A^*. (Note that some action X that we move A^* to the left of could now fail and thus π'' could be shortened.) By the induction condition t and t' can be composed into a single transition $t'' \in \tau_I$. We are in Case 2 with decreased measure.

Case 2.2.2. (ℓ', A) does not execute in π''. We insert a transition of (ℓ', A^*) into π'' right before the failure. Recall that the gate of A^* is satisfied, and by the cooperation condition we can execute A^* such that the final configuration decreases according to $\gg$. Because of backward preservation, the original failure is preserved after A^*. We proceed as in Case 2.2.1 to move A^* to the left and absorb it into I. Then we are again in Case 2 with decreased measure.

The above rewriting process obviously does not introduce new transitions of M into π'. □

Lemma 4. *For every terminating $\mathcal{P}$-execution π starting with a transition of M there exists a $\mathcal{P}'$-execution π' that starts from the same configuration as π with a transition of M and either fails or ends in the same configuration as π, i.e.,*

$$(g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \implies (g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}'}{}^{*} c$$

where $c \in \{\lightning, (g', \varnothing)\}$. Moreover, π' does not execute more PAs to M than π.

Proof. We rewrite π into π' exactly the same way as in the proof of Lemma 3. If no failure is introduced (which is possible in Case 1 and Case 2.2.1) we are guaranteed to preserve the final configuration of π (and never reach Case 2.2.2). Otherwise we obtain a failing π' from Lemma 3. □

Theorem 8. *The IS proof rule in Figure 5.3 is sound.*

Proof. By repeated application of Lemma 3 and Lemma 4 every $\mathcal{P}$-execution π can be rewritten into a refinement-preserving $\mathcal{P}'$-execution π'. □

5.5 Evaluation

In this section, we report on our experience of using IS for the verification of functional correctness of a diverse set of example programs (see Table 5.1). We argue that IS is *applicable* (to realistic programs), *automatable* (using sequential verifiers), and *user-friendly* (by appealing to sequential intuition). Our tool and all examples are publicly available as part of the Boogie project [2] and long-term archived [69].

5.5.1 Implementation

We implemented IS as an extension of Civl [56], a verification system for concurrent programs based on automated and modular refinement reasoning. Civl implements the framework of layered concurrent programs [70] where the input consists of a description of a sequence of (related) programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ and the verification goal is to establish the chain of refinement $\mathcal{P}_1 \preccurlyeq \cdots \preccurlyeq \mathcal{P}_n$. The justification for every refinement step between two subsequent programs $\mathcal{P}_i \preccurlyeq \mathcal{P}_{i+1}$ is compiled into sequential verification conditions of the Boogie verifier [14], which are then discharged automatically by an SMT solver. We seamlessly integrated IS into Civl, such that every refinement step can now either be an IS transformation or an existing Civl transformation.

Our integration of IS into Civl comprises roughly 2500 lines of C# code, addressing the following challenges. First, we adapted the type checker to integrate IS into the language of layered concurrent programs, which avoids extensive repetition of program parts that do not change in most refinement steps (observed as hindering, e.g., in [26]). Second, we designed a representation of PAs as multisets in the generalized array theory [36] and extended the existing refinement checker with the capability to summarize unboundedly many asynchronous calls as PAs. Third, the actual conditions of IS (Figure 5.3) are compiled to sequential verification conditions in Boogie. In particular, one sequential Boogie procedure encodes each of the following checks: (1) M refines I, (2) I is inductive w.r.t. the abstraction of a chosen PA, (3) I without transitions that create PAs to $\mathcal{E}$ refines M', and (4) A refines its abstraction $\alpha(A)$ for $A \in \mathcal{E}$. The left-mover conditions are automatically discharged by the existing mover engine. Due to this fine-grained decomposition, we can generate targeted error messages for failed checks that are local to at most two actions.

5.5.2 Verification Methodology

In this section we report on the verification methodology we followed to verify our examples and specifically illustrate it on our most significant example, Paxos.

Paxos. The Paxos [75] protocol establishes consensus among a set of unreliable nodes in an asynchronous network without a central coordinator. We consider a single-decree Paxos variant that establishes consensus on a single value. Conceptually, Paxos operates in a sequence of (increasingly numbered) rounds, where each round is associated to a *proposer* node. The proposers communicate with a set of *acceptor* nodes to try to either decide on a newly proposed value or to learn about a previously decided value. Since the proposers operate concurrently on different rounds, Paxos resolves conflicts using a mechanism that requires proposers to collect in two subsequent phases "enough" responses

```
1  var acceptorState: Node -> AcceptorState
2  var decision: Round -> Option<Value>
3  var joinChannel: Round -> Bag<JoinResponse>
4  var voteChannel: Round -> Bag<VoteResponse>
5  proc Paxos()                    proc Conclude(r: Round, v: Value)
6  proc StartRound(r: Round)       proc Join(r: Round, n: Node)
7  proc Propose(r: Round)          proc Vote(r: Round, n: Node, v: Value)
```

(a) Concrete implementation

```
8  datatype VoteInfo(value: Value, nodes: Set<Node>)
9  var joinedNodes: Round -> Set<Node>
10 var voteInfo: Round -> Option<VoteInfo>
11 var pendingAsyncs: Bag<PA>
12 action Propose(r: Round) returns (pending_async PAs: Bag<PA>):
13   var ns: Set<Node>, v: Value
14   assert Propose(r) ∈ pendingAsyncs
15   assert voteInfo[r] = None()
16   if (*):
17     assume ns ⊆ joinedNodes[r] ∧ IsQuorum(ns)
18     ... // compute v from r, ns, and voteInfo
19     voteInfo[r] := Some(VoteInfo(v, ∅))
20     PAs := {Vote(r, n, v) | n: Node} ⊎ {Conclude(r, v)}
21   ...
```

(b) Atomic actions for applying inductive sequentialization

```
22 action ProposeAbs(r: Round) returns (pending_async PAs: Bag<PA>):
23   assert {StartRound(r') ∈ pendingAsyncs | r' ≤ r} = ∅
24   assert {Join(r', n')   ∈ pendingAsyncs | r' ≤ r} = ∅
25   ... // same as Propose
26 action Paxos() seq Paxos' with PaxosInv
27 elim StartRound, StartRoundAbs elim Propose, ProposeAbs elim ...
28   ...
29 action Paxos'():
30   assert ∀ r. decision[r] = None()
31   havoc decision with ∀ r1, r2, v1, v2.
32     decision[r1] = Some(v1) ∧ decision[r2] = Some(v2) ⟹ v1 = v2
33 action PaxosInv() returns (pending_async PAs: Bag<PA>, choice c: PA)
34   ...
```

(c) Inductive sequentialization

Figure 5.4: Excerpts from our Paxos proof.

(called quorum) from acceptors, while acceptors stop working on a round when they hear about a higher round. Thus every round can remain undecided (in general, consensus cannot be guaranteed in an asynchronous network), but we want to prove that two rounds never decide on conflicting values.

Implementation. Our examples are implemented as low-level concurrent programs $\mathcal{P}_1$ that only use primitive atomic actions, like reading or writing a single memory address, and sending or receiving a single message. Figure 5.4(a) shows the variable and procedure declarations of our Paxos implementation. The procedures `Propose` and `Conclude` are associated to the proposer role in the Paxos protocol, while `Join` and `Vote` are associated to the acceptor role. A client calls `Paxos`, which creates an arbitrary number of asynchronous `StartRound` tasks. For each round, the corresponding `StartRound` task creates one `Join` task per acceptor and one `Propose` task. According to each acceptor's current state (in `acceptorState`), `Join` sends a `JoinResponse` message to a channel in `joinChannel`.

`Propose` executes a loop that receives from this channel and aggregates the response messages. If a quorum is reached, it proposes a value by creating one `Vote` task per acceptor and one `Conclude` task. Then, the `Vote` tasks send `VoteResponse` messages that are aggregated in a loop by `Conclude`. If a quorum is reached, `Conclude` updates `decision` for the corresponding round from `None()` to `Some(v)` where `v` is the decided value.

Atomic actions. IS operates on atomic actions, and thus we first apply an existing Civl transformation on $\mathcal{P}_1$ to obtain suitable atomic actions that summarize the low-level procedures, forming $\mathcal{P}_2$. Crucially, this step does not require any concurrent invariants related to the correctness of the protocol. However, the subsequently enabled application of IS significantly simplifies the construction of the actual proof of functional correctness. Furthermore, we note that the structured proofs obtained by our methodology are not only—in our experience—simpler to construct, but also more modular and thus better suited for change than flat inductive invariants. For example, changing low-level details in the implementation only requires a revision of $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, but does not affect the rest of the proof.

For Paxos we also make use of Civl's capability to change the state representation of the program. Concretely, we hide the implementation variables `acceptorState`, `joinChannel`, and `voteChannel`, and instead introduce the abstract variables `joinedNodes` and `voteInfo` shown in Figure 5.4(b). Also, we introduce `pendingAsyncs` to hold the current set of pending asyncs. As an example, Figure 5.4(b) shows the action summary of `Propose`. Instead of a loop that aggregates messages from a channel, it atomically initializes `voteInfo[r]` to `VoteInfo(v,∅)` if there is a quorum in `joinedNodes[r]`, where `v` is the proposed value and ∅ the initially empty set of acceptors that voted on it. Notice that the action `Propose` has an additional specially-declared output variable `PAs` that represents the PAs created by the action.

Inductive sequentialization. In our experience, the key to apply IS is the intuition of idealized, sequential executions of the program. The main creative task is the invention of this sequentialization, while all required proof artifacts are derived from it. In particular, the invariant action I and the choice function f are determined from partial sequential executions, M' summarizes completed sequential executions, and left-moving abstractions α can assert to only execute in the sequential context.

The sequentialization idea for Paxos is to execute one round at a time (in increasing order), and within each round execute actions in a fixed order. In particular, abbreviating action names with their first letter and denoting round boundaries by a vertical bar, the sequentialization looks as follows:

$$\mathtt{S}(1)\ \mathtt{J}(1,1)\ \mathtt{J}(1,2)\ \mathtt{P}(1)\ \mathtt{V}(1,1,\_)\ \mathtt{V}(1,2,\_)\ \mathtt{C}(1,\_) \mid \mathtt{S}(2)\ \mathtt{J}(2,1)\ \ldots$$

To preserve all original behaviors of the protocol, we observed that the effect of rounds being blocked from reaching a decision due to overlapping proposals or out-of-order message delivery is equivalent to both acceptors and proposers nondeterministically dropping incoming messages. For example, notice that the state update in `Propose` is guarded by a nondeterministic conditional on line 16 (which is not present in the low-level implementation $\mathcal{P}_1$ but introduced in $\mathcal{P}_2$).

Our goal is to apply IS to transform `Paxos` to `Paxos'` in Figure 5.4(c). `Paxos'` is a straight-forward high-level specification of Paxos, stating that the protocol consistently updates `decision`, i.e., no two rounds decide on conflicting values. A client could be provided with an API to query `decision` and would then use `Paxos'` to reason about its own consistency. The application of IS is declared on action `Paxos` (line 26), which prescribes the use of invariant action `PaxosInv` to simultaneously eliminate all other actions using the left-mover abstractions given in the `elim` clauses (line 27). For example, `ProposeAbs` in Figure 5.4(c) strengthens the gate of `Propose` with the information that, in the sequentialization, only `Join` and `StartRound` actions with higher round numbers can still be pending. All our abstractions are of this simple kind. The choice function is specified by the programmer using a special output variable `c` of `PaxosInv`, see line 33.

Our invariant action `PaxosInv` consists of four parts:

1. *Sequentialization*: Rounds execute one after another, and within rounds there is a fixed order of phases.
2. *Quorum before decision*: If there was a decision for value v, then there must have been a proposal and a quorum of nodes that voted for v (in the same round).
3. *Voting after decision*: If there was a decision in round r_1 for value v_1 and some higher round r_2 votes on value v_2, then $v_2 = v_1$.
4. *Safety*: If two rounds reach a decision, then it is on the same value.

Property 1 encodes the sequentialization order and lets us discharge the gates of our left-mover abstractions. Properties 2/3/4 capture the core mechanism of the protocol and are quite easy to state.

Invariant complexity. We demonstrate the significant simplifications afforded by IS in terms of invariant complexity by comparing against the baseline of standard "asynchrony-aware" inductive invariants (over the original asynchronous program). In particular, we compare to the well-documented Ivy invariants given in [97], but stress that these invariants are representative for other systems like [76, 55, 116, 26]. While these works have excellent contributions elsewhere, the methodology to deal with the protocol complexity boils down to the above baseline. The only other approaches we know of that focus on improving this particular aspect are [12, 112, 71, 34], but they do not apply to our example programs (see Section 5.6).

Properties 2/3/4 above correspond roughly to formulas (4)-(7) in [97]. However, the Ivy invariant requires five additional conjuncts (8)-(12), which capture more complicated dependencies of overlapping rounds and are much harder to invent. Due to sequentialization, we do not need any analogue of these in our invariant.

5.5.3 Other Case Studies

We demonstrate the broad applicability of IS by applying it on the examples listed in Table 5.1. These examples cover a wide variety of characteristics of concurrent programs, including modes of concurrency (tightly synchronized, mostly independent, coordination-focused, phase-oriented, long-running), communication topologies (complete, star, ring,

Table 5.1: Examples verified with IS.

Example	**#IS**	**#LOC** Total	**#LOC** IS	**#LOC** Impl	**Time** sec
Broadcast consensus	2	396	108	121	1.0
Ping-Pong	1	281	91	106	0.9
Producer-Consumer	1	225	65	93	0.9
N-Buyer	4	681	251	256	2.6
Chang-Roberts	2	377	117	135	1.1
Two-phase commit	4	553	181	222	1.4
Paxos	1	1168	534	302	4.2

pipeline), channel types (bags, queues), and specifications (consensus, unique leader, assertions). We avoided any hidden simplifications in the communication between processes (e.g., arranging broadcast-receive communication with a set of nodes sequentially instead of concurrently), and included realistic performance optimizations which generally complicate verification.

Column #IS reports the number of IS applications. For some programs we preferred the repeated application of IS, although the proof could be accomplished by a single application. This is because an action that is eliminated in one IS application disappears from the pool of actions w.r.t. which left-moverness has to be established in a subsequent IS application. For example, as an alternative to the one-shot proof of the broadcast consensus protocol in Figure 5.1 we performed a proof that first eliminates `Broadcast` in one IS application, and then `Collect` in a second IS application. Then the abstraction `CollectAbs` in Figure 5.1-④ does not need the gate on line 33, because `CollectAbs` does not have to commute to the left of `Broadcast`.

The #LOC columns report numbers of Civl lines of code. Civl, as Boogie, is an intermediate verification language not optimized for conciseness. Our files contain a lot of boilerplate code that would be part of a library for any frontend language. Concretely, this includes declarations of builtin SMT types, type declarations for pending asyncs, theory axioms (e.g., for sets), primitive atomic actions (e.g., send/receive), etc. Thus, besides (1) the total lines we also report (2) the lines related to IS steps, and (3) the lines related to the implementation $\mathcal{P}_1$ and existing Civl step $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.

The last column reports the total verification time. Our tool is fast and thus suitable for interactive development. However, we acknowledge observing run-time fluctuations caused by small (semantically irrelevant) modifications, likely due to heuristics for quantifier reasoning. Improving the robustness of checkers for complex verification conditions is an important avenue for future work.

We finally provide a brief description of the remaining examples besides broadcast consensus and Paxos.

Ping-Pong. In this example a Ping process sends increasing numbers to a Pong process, expecting the number to be acknowledged back. Our sequentialization makes the alternation of the Ping and Pong process explicit. We verify assertions in the program, which state that the Pong processes receives increasing numbers, and the Ping process receives correct acknowledgments.

Producer-Consumer. This is a variation of the Ping-Pong example, where a producer enqueues increasing numbers into a shared queue, and a consumer dequeues numbers from the queue and verifies that they are indeed increasing. The Producer-Consumer example has more concurrent executions than the Ping-Pong example, because the producer can be arbitrarily faster than the consumer, and thus the queue can grow arbitrarily big. However, IS reduces the program to a sequentialization where the producer and consumer alternate, and thus the queue contains at most one element.

N-Buyer. In this example n buyer processes coordinate the purchase of an item from a seller. That is, one buyer requests a quote for the item from the seller, then the buyers coordinate their individual contribution, and finally if the contributions are enough to buy the item, and order is placed. This example was adapted from [25] and is representative for the coordination protocols targeted by session types. We added and verified a functional correctness specification that states that if a final order is placed then the sum of contributions promised by the buyers actually adds up to the price of the ordered item.

Chang-Roberts. This is a leader election protocol in a ring topology [27]. Each node starts by sending its own (unique) ID to its neighbor in the ring, and then forwards incoming messages with IDs greater than its own. When a node receives its own ID, it declares itself as leader. We prove that there can be at most one leader. Our sequentialization follows from the intuition that only the node with the highest ID, say m, can become a leader, and for that its ID has to traverse the ring once. We sequentialize the program such that each node runs to completion, starting with the neighbor of m, then the neighbor of the neighbor of m, and so on, and finally m.

Two-phase commit (2PC). 2PC is a protocol for collectively deciding on *committing* or *aborting* a transaction. The protocol consists of a coordinator and n participants, and proceeds in two phases. During the first phase, the coordinator sends vote requests to all the participants and collects their votes, which can indicate to either commit or abort the transaction. If all of the participants have voted for committing the transaction, the coordinator initiates the second phase by sending *commit* messages to all participants. Otherwise, it sends an *abort* message. When the participants receive the decision message from the coordinator, they finalize the transaction.

We consider an optimized and realistic implementation of the 2PC protocol. First, in both phases, the coordinator broadcasts a message and then waits to receive responses. Second, the coordinator can send "*early abort*" messages. While receiving votes, it can terminate the first phase and abort the transaction as soon as it receives a negative vote, without waiting for the remaining votes. Thus some of the participants might receive a decision message even before seeing a request. Therefore, the last optimization is that the participants can process request and decision messages concurrently, in contrast to processing the decision message sequentially after the request message.

We verified that all participants consistently commit or abort a transaction, and that commit only happens if all participants voted for commit. We established a sequential reduction of 2PC using 4 applications of IS (each IS application enlarging the sequentialized prefix until removing asynchrony altogether). The sequential reduction follows the natural flow of the protocol: broadcasting vote request messages, followed by vote responses from a

nondeterministic number of participants, followed by the broadcast of the decision message, and the finalization of the transaction.

5.6 Related Work

We review works concerning the design of proof systems for reasoning about concurrent or distributed systems. We focus first on proof systems that include some form of reduction, i.e., behavior-preserving transformations that reduce the number of interleavings, which are closer to our work, and subsequently discuss other related works.

Reduction. Lipton's reduction theory [80] introduced the concept of *movers* to define a program transformation that creates atomic blocks of code. QED [41] expanded the scope of Lipton's theory by introducing iterated application of reduction and abstraction over gated atomic actions. CIVL [56] builds upon the foundation of QED, adding invariants [95, 63], refinement layers [70], and pending asyncs [71]. Inductive sequentialization builds upon this prior work, introduces a new scheme for reasoning inductively over unbounded concurrent executions, and thus provides an alternative to the classic approach of inductive invariants.

The work described above takes the general approach of reasoning about concurrent programs via simplifying program transformations. Recent research projects have advocated the need to incorporate an increasing set of sound program transformations. CSPEC [26] takes an approach similar to CIVL but mechanizes all metatheory within the Coq theorem prover [108] for flexibility and sound extensibility. Armada [81] also has flexible and mechanized metatheory whose usefulness is demonstrated by implementing a variety of program transformations, including those catering to fine-grained concurrency and weak memory models.

Movers have also been used to define an equivalence-preserving transformation that eliminates buffers in message-passing programs [12, 112]. These works define a restricted class of programs and prove that reasoning about the set of *rendezvous* executions of these programs, where messages are delivered instantaneously, is complete, i.e., any other execution is equivalent to a rendezvous execution, up to reordering of mover actions. Our example programs in Section 5.5 fall outside this class, e.g., because of ring topology (Chang-Roberts), optimizations (2PC, Paxos), or loop-carried state (Ping-Pong). In general, removing message buffers does *not* necessarily lead to a sequential program. Concurrency can still be present due to the different orders in which messages can be sent or received by different processes. For instance, von Gleissenthall et al. [112] consider a simpler variation of Paxos where the communication between a proposer p and an acceptor a_1 does not interleave with the communication between p and another acceptor a_2. The reduction to rendezvous communication, which remains a *concurrent* program, still contains all the complexity due to acceptors receiving messages from different rounds in an arbitrary order (which is not present in our sequentialization).

In the context of asynchronous programs, Kragl et al. [71] use left movers to derive atomic action summaries for procedures with asynchronous calls, i.e., they define a behavior-preserving transformation where asynchronous calls can be assumed to be synchronous provided that their body is a left mover. Inductive sequentialization solves the orthogonal problem of eliminating an unbounded number of PAs from atomic actions using induction. In particular, the work in [71] does not apply to the examples presented in Section 5.5

(their versions of Ping-Pong and two-phase commit do not model explicit communication through message-passing).

In the context of message-passing programs, Elrad and Francez's reduction theory [42] introduced the concept of *communication-closed layer*, which is a sequence of actions where every send action is paired with a corresponding receive action. They propose a program transformation that reduces a given program to a sequence of communication-closed layers. This simplifies reasoning since the lifetime of a message is limited to a single layer. Damian et al. [34] provides a concrete instantiation of this theory in the context of fault-tolerant distributed protocols that relies on common implementation idioms. While the result of this transformation is not a purely sequential program as in our case, it does provide a significant reduction in the number of schedules to reason about. Conceptually, our work is phrased in a more generic setting that does not rely on the specifics of the input program. The approach of Damian et al. [34] requires low-level annotations about local variables and messages, and various syntactic constraints on executions. For instance, Damian et al. [34] cannot deal with Chang-Roberts or our 2PC with "early-abort" (independently of the programming model) because of syntactical constraints on the executions (see Condition V of Definition 2 in that paper). Chang-Roberts is not admitted because the messages do not encode a notion of time and 2PC is not admitted because the coordinator interleaves computation steps (taking a decision) with receiving votes. They can also not deal with the other examples (including our version of Paxos) because they are written using asynchronous procedure calls (Damian et al. [34] deals with protocols written as the composition of a number of long-running processes executing sequential code). Concerning Paxos, Damian et al. [34] considered several optimized variations which we believe are in the reach of IS as well. Given the limited time, we chose to evaluate IS over a diverse set of communication patterns and specifications instead of additional Paxos features.

Verification of distributed systems. There are several recent papers on mechanized verification of distributed systems. IronFleet [55] embeds TLA-style state-machine modeling [76] into the Dafny verifier [77] to refine high-level distributed systems specifications into low-level executable implementations. Ivy [98] organizes the search for an inductive invariant as a collaborative process between automatic verification attempts and user guided generalizations of counterexamples to induction in a graphical model. They use a restricted modeling and specification language that makes their verification conditions decidable. Padon et al. [97] presents a methodology for (manually) instrumenting program code which ensures that the verification conditions generated by Ivy fall into the decidable effectively-propositional fragment of first-order logic. Verdi [116] lets the programmer provide a specification, implementation, and proof of a distributed system under an idealized network model. Then the application is automatically transformed into one that handles faults via verified system transformers. The rely-guarantee rule of Gavran et al. [49] and the ALS types of Kloos et al. [67] target a weaker form of asynchrony, where a single task queue atomically executes one task at a time. Unlike our approach, all the above perform asynchronous reasoning which significantly complicates the invariants. PSync [40] uses a synchronous model of communication for the purpose of program design and verification, shifting the complexity of efficient asynchronous execution to a runtime system.

Concurrent separation logic (CSL) [92] was devised for modular reasoning about multi-threaded shared-memory programs, focusing on the verification of fine-grained concurrent data structures. CSL adequately addresses the problem of reasoning about low-level concurrency related to dynamic memory allocation, but still suffers from the complications

of a monolithic approach to invariant discovery for protocol-level concurrency. Recently, CSL has been applied to message-passing programs. The approach in [93] uses CSL to link implementation steps to atomic actions, and then relies on a model checker to explore the interleavings of those atomic actions. The work in [104] addresses the composition of verified protocols using ideas from separation logic. The actor services of [107] focus on compositional verification of response properties of message-passing programs.

Sequentialization in bounded model checking. Reducing concurrent program verification to sequential program verification has also been used in the context of bounded model checking, e.g., [100, 43, 21, 18, 109, 74]. In this case, the reduction encodes the control nondeterminism due to the interleaving semantics into data nondeterminism, and assumes a certain bound on interleavings, e.g., a bounded number of context switches [99]. The resulting sequential program still exhibits all the complexity due to interleavings, but is more amenable to symbolic reasoning using SMT solvers.

5.7 Conclusion

We presented inductive sequentialization, a new induction-based methodology for proving the correctness of an asynchronous program. This methodology establishes sequential reductions, which capture all the behaviors of the original program, up to reordering of commutative actions. The proofs using inductive sequentialization are much simpler than those relying on standard inductive invariants since they sidestep the problem of reasoning about arbitrarily many and arbitrarily long interleavings.

IS is a blend of induction, reduction and abstraction, which derives its power from the tight combination of the three. Its applicability is particularly enhanced in well-designed asynchronous systems which favor short-living asynchronous tasks in place of long-living tasks that reduce responsiveness, and where asynchrony is transparent in the sense that it does not affect the logic of the application. This has been demonstrated through the verification of a number of implementations of paradigmatic distributed protocols.

In the future we plan to further investigate the potential of IS to simplify the construction of formal proofs of distributed systems in other application areas, e.g., Byzantine fault tolerance, and blockchain protocols.

Acknowledgments

This research was supported in part by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).

6 Conclusions

This dissertation presented a new framework for the deductive verification of concurrent programs. We introduced a refinement methodology over structured concurrent programs, where fine-grained procedures are gradually abstracted to coarser-grained atomic actions. Proof construction by a human is decomposed into small manageable pieces, and proof checking by a machine is decomposed into modular verification conditions. We believe that formally verified implementations can only become practical when programming and verification are combined into a single activity. Our work facilitates this unification by representing all layers of abstraction in a multi-layer refinement proof (from low-level implementations to high-level specifications) in the same uniform formalism, and compactly expressing all layers and their connection as a single layered concurrent program.

We integrated novel reduction-based program simplifications into our methodology, which *synchronize* and *sequentialize* asynchronous computations, appealing to the intuition programmers have about simple interleavings of their programs. We applied our reductions to a number of challenging examples, and demonstrated that even complicated distributed protocols like Paxos admit inductive sequentialization proofs that are much simpler (to construct) than existing proofs. A common question asks when exactly our reduction arguments do or do not apply. In theory, an inductive invariant proof can be turned into an inductive sequentialization proof. This is not very satisfactory though, since the invention of complicated inductive invariants is what we wanted to avoid in the first place. The question should really be, when exactly is inductive sequentialization useful. This question is very much open, and will only be possible to be answered with further experience. Clearly, contrived examples where every interleaving produces a different result (e.g., a set of n threads, all of which concurrently append their unique identifier to a shared list) are not amenable to a reduction argument. However, the results in this dissertation on a broad variety of realistic examples and concurrency patterns is encouraging. We hope to see this trend continue on further examples, and firmly believe that reductions will play a key role in making formal proofs of concurrent programs mainstream.

While the verification of sophisticated concurrent algorithms and realistic implementations will certainly remain a challenge, our decomposition and structuring mechanism enables programmers to face this challenge in a principled way.

We conclude with possible directions for future work.

Verification-condition generation. Civl programs are compiled to sequential verification conditions in Boogie. There are several choices for doing this compilation, and

several choices for splitting or combining different checks. We would like to better understand the tradeoffs between these choices, and if any of them should be exposed to the programmer. For example, for every Civl procedure, sequential verification (checking preconditions, postconditions, and loop invariants), noninterference checking, and refinement checking are combined into a single Boogie procedure. Splitting these checks decreases the size of individual verification conditions, but results in (linear) code duplication. Also, noninterference checking is performed w.r.t. all invariants across all procedures at once. Again, splitting these checks would decrease the size of individual verification conditions, but also result in code duplication (e.g., quadratic duplication to check noninterference of a procedure w.r.t. only one other procedure at a time).

Automation. Since we rely on automated theorem provers to discharge verification conditions, it is important to provide the programmer with mechanisms to make progress in case automated verification fails. In particular, quantified formulas can make proofs very brittle (i.e., small changes in the input have a large impact on prover performance). For certain checks, Civl supports the targeted injection of lemmas (to avoid prolific global quantified axioms) and witnesses (for quantifier instantiation). We would like to have systematic support for such mechanisms.

Integrated development environment. The language of layered concurrent programs provides new challenges and opportunities for programming support in an integrated development environment. For example, displaying a particular program layer, refactoring support for layers, etc.

Code generation. Although we can verify fine-grained realistic implementations by starting with a program that only uses primitive atomic operations, we would like to be able to generate executable binaries from verified Civl programs.

Prophecy variables. We are interested in adding support for prophecy variables to Civl. While intuitively and theoretically [4] useful, there is almost no support for prophecy variables in verification tools. Prophecy variables were added to QED to enable backward reasoning [105]. However, we are not aware of any stable implementation of prophecy variables in a verifier based on logical verification conditions. Only recently prophecy variables were added to the separation-logic framework Iris [65].

Bibliography

[1] Boogie (release). https://www.nuget.org/packages/Boogie.

[2] Boogie (source code). https://github.com/boogie-org/boogie.

[3] Jepsen. https://jepsen.io/.

[4] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2), 1991. doi:10.1016/0304-3975(91)90224-P.

[5] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. 1996. doi:10.1017/CBO9780511624162.

[6] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6), 2010. doi:10.1007/s10009-010-0145-y.

[7] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *LICS*, 1996. doi:10.1109/LICS.1996.561320.

[8] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: modularity in model checking. In *CAV*, 1998. doi:10.1007/BFb0028774.

[9] Krzysztof R. Apt. Formal justification of a proof system for communicating sequential processes. *J. ACM*, 30(1), 1983. doi:10.1145/322358.322372.

[10] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 2(3), 1980. doi:10.1145/357103.357110.

[11] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. 1998. doi:10.1007/978-1-4612-1674-2.

[12] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. In *OOPSLA*, 2017. doi:10.1145/3133934.

[13] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002. doi:10.1145/503272.503274.

[14] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005. doi:10.1007/11804192_17.

[15] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. In *OOPSLA*, 2018. doi:10.1145/3276514.

[16] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. In *IFM*, 2017. doi:10.1007/978-3-319-66845-1_7.

[17] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Robustness against relaxed memory models. In *Software Engineering*, 2014. URL: http://dl.gi.de/handle/20.500.12116/30973.

[18] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. In *TACAS*, 2012. doi:10.1007/978-3-642-28756-5_31.

[19] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV*, 2017. doi:10.1007/978-3-319-63390-9_28.

[20] Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. Verifying robustness of event-driven asynchronous programs against concurrency. In *ESOP*, 2017. doi:10.1007/978-3-662-54434-1_7.

[21] Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. On sequentializing concurrent programs. In *SAS*, 2011. doi:10.1007/978-3-642-23702-7_13.

[22] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the completeness of verifying message passing programs under bounded asynchrony. In *CAV*, 2018. doi:10.1007/978-3-319-96142-2_23.

[23] Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. Reasoning about TSO programs using reduction and abstraction. In *CAV*, 2018. doi:10.1007/978-3-319-96142-2_21.

[24] Tracy Camp, Phil Kearns, and Mohan Ahuja. Proof rules for flush channels. *IEEE Trans. Software Eng.*, 19(4), 1993. doi:10.1109/32.223804.

[25] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in Go: statically-typed endpoint APIs for dynamically-instantiated communication structures. In *POPL*, 2019. doi:10.1145/3290342.

[26] Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *OSDI*, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/chajed.

[27] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5), 1979. doi:10.1145/359104.359108.

[28] Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. Hitting families of schedules for asynchronous programs. In *CAV*, 2016. doi:10.1007/978-3-319-41540-6_9.

[29] Ching-Tsun Chou and Eli Gafni. Understanding and verifying distributed algorithms using stratified decomposition. In *PODC*, 1988. doi:10.1145/62546.62556.

[30] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, 1981. doi:10.1007/BFb0025774.

[31] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *LICS*, 1989. doi:10.1109/LICS.1989.39190.

[32] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, 2009. doi:10.1007/978-3-642-03359-9_2.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977. doi:10.1145/512950.512973.

[34] Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In *CAV*, 2019. doi:10.1007/978-3-030-25543-5_20.

[35] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008. doi:10.1007/978-3-540-78800-3_24.

[36] Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, 2009. doi:10.1109/FMCAD.2009.5351142.

[37] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. 2001.

[38] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, 2013. doi:10.1145/2491956.2462184.

[39] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In *ESEC/FSE*, 2015. doi:10.1145/2786805.2786861.

[40] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, 2016. doi:10.1145/2837614.2837650.

[41] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, 2009. doi:10.1145/1480881.1480885.

[42] Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3), 1982. doi:10.1016/0167-6423(83)90013-8.

[43] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In *POPL*, 2011. doi:10.1145/1926385.1926432.

[44] Azadeh Farzan and Anthony Vandikas. Reductions for safety proofs. In *POPL*, 2020. doi:10.1145/3371081.

[45] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2), 2001. doi:10.1016/S0304-3975(00)00102-X.

[46] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009. doi:10.1145/1542476.1542490.

[47] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003. doi:10.1145/781131.781169.

[48] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[49] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *CONCUR*, 2015. doi:10.4230/LIPIcs.CONCUR.2015.483.

[50] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. 1996. doi:10.1007/3-540-60761-7.

[51] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997. doi:10.1007/3-540-63166-6_10.

[52] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building certified concurrent OS kernels. *Commun. ACM*, 62(10), 2019. doi:10.1145/3356903.

[53] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *PLDI*, 2018. doi:10.1145/3192366.3192381.

[54] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011. doi:10.1145/1926385.1926424.

[55] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, 2015. doi:10.1145/2815400.2815428.

[56] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV*, 2015. doi:10.1007/978-3-319-21668-3_26.

[57] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. Technical Report MSR-TR-2015-8, Microsoft Research, 2015. URL:

https://www.microsoft.com/en-us/research/publication/automated-and-modular-refinement-reasoning-for-concurrent-programs/.

[58] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, 2002. doi:10.1145/503272.503279.

[59] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990. doi:10.1145/78969.78972.

[60] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969. doi:10.1145/363235.363259.

[61] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5), 1997. doi:10.1109/32.588521.

[62] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM*, 2011. doi:10.1007/978-3-642-20398-5_4.

[63] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

[64] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28, 2018. doi:10.1017/S0956796818000151.

[65] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. In *POPL*, 2020. doi:10.1145/3371113.

[66] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. Proving linearizability using partial orders. In *ESOP*, 2017. doi:10.1007/978-3-662-54434-1_24.

[67] Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. Asynchronous liquid separation types. In *ECOOP*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.396.

[68] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *PLDI*, 2020. doi:10.1145/3385412.3385980.

[69] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs (evaluated artifact), 2020. doi:10.5281/zenodo.3754772.

[70] Bernhard Kragl and Shaz Qadeer. Layered concurrent programs. In *CAV*, 2018. doi:10.1007/978-3-319-96145-3_5.

[71] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the asynchronous. In *CONCUR*, 2018. doi:10.4230/LIPIcs.CONCUR.2018.21.

[72] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for structured concurrent programs. In *CAV*, 2020. doi:10.1007/978-3-030-53288-8_14.

[73] Siddharth Krishna, Michael Emmi, Constantin Enea, and Dejan Jovanovic. Verifying visibility-based weak consistency. In *ESOP*, 2020. doi:10.1007/978-3-030-44914-8_11.

[74] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, 2008. doi:10.1007/978-3-540-70545-1_7.

[75] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998. doi:10.1145/279227.279229.

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

[77] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010. doi:10.1007/978-3-642-17511-4_20.

[78] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *FOSAD*, 2009. doi:10.1007/978-3-642-03829-7_7.

[79] Gary Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15, 1981. doi:10.1007/BF00289266.

[80] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12), 1975. doi:10.1145/361227.361234.

[81] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*, 2020. doi:10.1145/3385412.3385971.

[82] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2), 1995. doi:10.1006/inco.1995.1134.

[83] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? In *POPL*, 2018. doi:10.1145/3158134.

[84] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC*, 1990. doi:10.1145/93385.93442.

[85] Kenneth L. McMillan. *Symbolic model checking*. 1993. doi:10.1007/978-1-4615-3190-6.

[86] Kenneth L. McMillan. A compositional rule for hardware design refinement. In *CAV*, 1997. doi:10.1007/3-540-63166-6_6.

[87] Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, 1998. doi:10.1007/BFb0028738.

[88] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, 2016. doi:10.1007/978-3-662-49122-5_2.

[89] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007. doi:10.1145/1250734.1250785.

[90] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008. URL: https://www.usenix.org/conference/osdi-08/finding-and-reproducing-heisenbugs-concurrent-programs.

[91] Suha Orhun Mutluergil and Serdar Tasiran. A mechanized refinement proof of the Chase-Lev deque using a proof system. *Computing*, 101(1), 2019. doi:10.1007/s00607-018-0635-4.

[92] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3), 2007. doi:10.1016/j.tcs.2006.12.035.

[93] Wytse Oortwijn, Stefan Blom, and Marieke Huisman. Future-based static analysis of message passing programs. In *PLACES*, 2016. doi:10.4204/EPTCS.211.7.

[94] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, 1975. URL: https://hdl.handle.net/1813/6393.

[95] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5), 1976. doi:10.1145/360051.360224.

[96] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. In *OOPSLA*, 2018. doi:10.1145/3276530.

[97] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. In *OOPSLA*, 2017. doi:10.1145/3140568.

[98] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, 2016. doi:10.1145/2908080.2908118.

[99] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005. doi:10.1007/978-3-540-31980-1_7.

[100] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In *PLDI*, 2004. doi:10.1145/996841.996845.

[101] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, 1982. doi:10.1007/3-540-11494-7_22.

[102] Richard D. Schlichting and Fred B. Schneider. Using message passing for distributed programming: Proof rules, disciplines. *ACM Trans. Program. Lang. Syst.*, 6(3), 1984. doi:10.1145/579.583.

[103] Fred B. Schneider. *On Concurrent Programming*. Graduate Texts in Computer Science. 1997. doi:10.1007/978-1-4612-1830-2.

[104] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *POPL*, 2018. doi:10.1145/3158116.

[105] Ali Sezgin, Serdar Tasiran, and Shaz Qadeer. Tressa: Claiming the future. In *VSTTE*, 2010. doi:10.1007/978-3-642-15057-9_2.

[106] Scott D. Stoller and Fred B. Schneider. Verifying programs that use causally-ordered message-passing. *Sci. Comput. Program.*, 24(2), 1995. doi:10.1016/0167-6423(95)00002-A.

[107] Alexander J. Summers and Peter Müller. Actor services - modular verification of message passing programs. In *ESOP*, 2016. doi:10.1007/978-3-662-49498-1_27.

[108] The Coq Development Team. The Coq proof assistant, version 8.11.0, 2020. doi:10.5281/zenodo.3744225.

[109] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, 2009. doi:10.1007/978-3-642-02658-4_36.

[110] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, 2010. doi:10.1007/978-3-642-14295-6_40.

[111] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, 2016. doi:10.1145/2908080.2908129.

[112] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. In *POPL*, 2019. doi:10.1145/3290372.

[113] Philip Wadler. Linear types can change the world! In *IFIP Congress*, 1990.

[114] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–44. The MIT Press, 2004. doi:10.7551/mitpress/1104.003.0003.

[115] James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. VerifiedFT: a verified, high-performance precise dynamic race detector. In *PPoPP*, 2018. doi:10.1145/3178487.3178514.

[116] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015. doi:10.1145/2737924.2737958.

[117] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), 1971. doi:10.1145/362575.362577.