Layered Concurrent Programs

Bernhard Kragl[1] and Shaz Qadeer[2]

[1] IST Austria, Klosterneuburg, Austria
[2] Microsoft Research, Redmond, USA

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.

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.

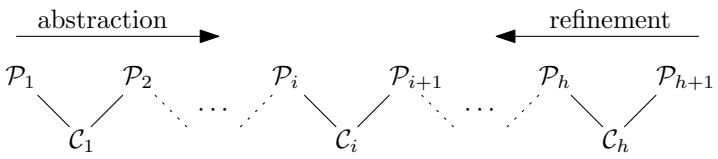


Fig. 1. Concurrent programs $\mathcal{P}_i$ and connecting checker programs $\mathcal{C}_i$ represented by a layered concurrent program $\mathcal{LP}$.

Finally, refinement naturally supports a bidirectional approach to correctness—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 [4], 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 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 [5] based on the theory of right and left movers [10] 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.

1.1 Related Work

This paper formalizes, clarifies, and extends the most important aspect of the design of Civl [6], a deductive verifier for layered concurrent programs. Hawblitzel et al. [7] 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 [4] 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 [1] 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 [2] toolset which has been used to model and verify several systems of industrial significance. TLA+ [9] 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 [11,12]. 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 Concurrent Programs

In this section we introduce a concurrent programming language. The syntax of our programming language is summarized in Figure 2.

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

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

Fig. 2. Concurrent programs

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, an *expression* e is a mapping from stores to values, and a *transition* t 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 (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, e, t). The semantics of an atomic action in an execution is to first evaluate the expression e, called the *gate*, in the current state. If the gate evaluates to *false* the execution *fails*, otherwise the program state is updated according to the transition t. *Input variables* in I can be read by e and t, and *output variables* in O can be written by t.

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	e	t
$x := x + y$	*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) `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 $\mathit{Safe}(\mathcal{P})$ if $\mathcal{P}$ is safe under preemptive semantics, and $\mathit{CSafe}(\mathcal{P})$ if $\mathcal{P}$ is safe under cooperative semantics.

2.1 Running Example

In this section, we introduce a sequence of three concurrent programs (Figure 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 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 (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 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 3(b) declares atomic actions with a

(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 ∈ 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
```

Fig. 3. Lock example

mover type [5], `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 [10]. 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 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 [14] 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.

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 3) and their connections can all be expressed in a single layered concurrent program. In Section 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 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

$$[a,b] = \{x \mid a,b,x \in \mathbb{N} \wedge a \leq x \leq b\} \qquad GS \in GVar \rightharpoonup \mathbb{I}$$
$$AS \in A \mapsto (I, O, e, t, r)$$
$$n, \alpha \in \mathbb{N} \qquad IS \in A \mapsto (I, O, e, t, n)$$
$$r \in \mathbb{I} = \{[a,b] \mid a \leq b\} \qquad PS \in P \mapsto (I, O, L, s, n, ns, A)$$
$$ns \in LVar \rightharpoonup \mathbb{N} \qquad m \in Proc$$
$$\mathcal{I} \in 2^{Store}$$
$$\mathcal{LP} \in LayeredProg ::= (GS, AS, IS, PS, m, \mathcal{I})$$
$$s \in Stmt ::= \cdots \mid \texttt{icall}\ (A, \iota, o) \mid \texttt{pcall}_\alpha\ \overline{(P_i, \iota_i, o_i)}_{i \in [1,k]} \quad (\alpha \in \{\varepsilon\} \cup [1,k])$$

Fig. 4. Layered Concurrent Programs

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 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 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.

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 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.

(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

(Program)

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

(Atomic action)

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

(Introduction action)

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

(Procedure)

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

(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)

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

(Parameter passing)

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

(Introduction call)

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

(Parallel call)

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

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

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

$ReadVars(t) = \{v \mid \exists\ \sigma, \sigma', a : (\sigma, \sigma') \in t \wedge (\sigma[v \mapsto a], \sigma') \notin t\}$

$ReadVars(e, t) = ReadVars(e) \cup ReadVars(t)$

$WriteVars(t) = \{v \mid \exists\ \sigma, \sigma' : (\sigma, \sigma') \in t \wedge \sigma(v) \neq \sigma'(v)\}$

$Nonblocking(e, t) = \forall\ \sigma \in e : \exists\ \sigma' : (\sigma, \sigma') \in t$

Fig. 5. Type checking rules for layered concurrent programs

$\mathcal{C}_n$ 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.

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

$$\begin{aligned}
gs &= \{v \mid GS(v) = [a, b] \wedge \ell \in [a+1, b]\},\\
as &= \{A \mapsto (I, O, e, t) \mid AS(A) = (I, O, e, t, r) \wedge \ell \in r\},\\
ps &= \{P \mapsto (I \cap ns|_\ell, O \cap ns|_\ell, L \cap ns|_\ell, \Gamma^P_\ell(s)) \mid PS(P) = (I, O, L, s, n, ns, _) \wedge \ell \leq n\},\\
m' &= \begin{cases} m & \text{if } \ell \in [1, h]\\ A_m & \text{if } \ell = h+1 \end{cases},
\end{aligned}$$

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

$$\begin{aligned}
\Gamma^P_\ell(\texttt{skip}) &= \texttt{skip},\\
\Gamma^P_\ell(s_1 \ ; \ s_2) &= \Gamma^P_\ell(s_1) \ ; \ \Gamma^P_\ell(s_2),\\
\Gamma^P_\ell(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2) &= \texttt{if } e \texttt{ then } \Gamma^P_\ell(s_1) \texttt{ else } \Gamma^P_\ell(s_2),\\
\Gamma^P_\ell(\texttt{icall } (A, \iota, o)) &= \texttt{skip},\\
\Gamma^P_\ell(\texttt{pcall}_\alpha\ \overline{(Q, \iota, o)}) &= \texttt{pcall } \overline{(X, \iota|_{ns_Q|_\ell}, o|_{ns_P|_\ell})},\\
&\qquad \text{for } \begin{matrix} PS(P) = (_, _, _, _, _, ns_P, _)\\ PS(Q) = (_, _, _, _, n, ns_Q, A) \end{matrix} \text{ and } X = \begin{cases} Q \text{ if } \ell \leq n\\ A \text{ if } \ell > n \end{cases}.
\end{aligned}$$

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 5) allow a pcall to invoke a procedure with the same disappearing layer as the caller.

$\mathcal{LP}^{lock}$

```
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,∞)

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 ∈ 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
```

Fig. 6. Lock example (layered concurrent program)

3.3 Running Example

We return to our lock example from Section 2.1. Figure 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 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 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 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 `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 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 6) over separate specification of each concurrent program (as in Figure 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 [7] or a data-race detector [15], the saving in code duplication is significant.

4 Refinement Checking

Section 3 described how a layered concurrent program $\mathcal{LP}$ encodes a sequence $\mathcal{P}_1, \ldots, \mathcal{P}_h, \mathcal{P}_{h+1}$ of concurrent programs. 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 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 $\mathit{CSafe}(\widetilde{\mathcal{P}}_\ell)$, then $\mathit{CSafe}(\mathcal{P}_\ell)$.*

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 $\mathit{CSafe}(\mathcal{C}_\ell)$ and $\mathit{Safe}(\mathcal{P}_{\ell+1})$, then $\mathit{CSafe}(\widetilde{\mathcal{P}}_\ell)$.*

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 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 $\mathit{MSafe}(\mathcal{P}_\ell)$, $\mathit{CSafe}(\mathcal{C}_\ell)$ and $\mathit{Safe}(\mathcal{P}_{\ell+1})$, then $\mathit{Safe}(\mathcal{P}_\ell)$.*

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 $\mathit{MSafe}(\mathcal{P}_\ell)$ and $\mathit{CSafe}(\mathcal{C}_\ell)$ for all $\ell \in [1, h]$ and $\mathit{Safe}(\mathcal{P}_{h+1})$, then $\mathit{Safe}(\mathcal{P}_1)$.*

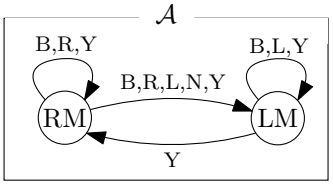
4.1 From Preemptive to Cooperative Semantics

We present the judgment $\mathit{MSafe}(\mathcal{P})$ 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 $M \in \mathit{Action} \to \{\mathrm{N}, \mathrm{R}, \mathrm{L}, \mathrm{B}\}$ 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 $\mathit{MSafe}(\mathcal{P})$ requires a mapping M that satisfies two conditions.

First, the atomic actions in $\mathcal{P}$ must satisfy the following logical commutativity conditions [7], 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* $\mathcal{A}$ on the right 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:

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

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

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.

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 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 7 extends $\mathcal{P}_1^{lock}$ (Figure 3(a)) with the variables introduced at layer 1 (globals `lock`, `pos`, `slots` and 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, e, t) in any atomic-action arm of a pcall is converted into the invocation of a fresh atomic action $(I, O, \mathit{true}, e \wedge t)$. 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

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

```
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
```

Fig. 7. Lock example (variable introduction at layer 1)

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 8. 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 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).

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

```
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
```

Fig. 8. Instrumented procedures `Enter` and `Leave` (layer 1 checker program)

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 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 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.

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 [3]. Civl has been used to verify a complex concurrent garbage collector [6] and a state-of-the-art data-race detection algorithm [15]. 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 [13] and rely-guarantee reasoning [8] 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).

References

1. Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
2. 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. *STTT*, 12(6):447–466, 2010.
3. 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.
4. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, 2009.
5. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.
6. Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV*, 2015.
7. 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, February 2015.
8. Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
9. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
10. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
11. Kenneth L. McMillan. A compositional rule for hardware design refinement. In *CAV*, 1997.
12. Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, 1998.
13. Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
14. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.
15. James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. VerifiedFT: a verified, high-performance precise dynamic race detector. In *PPoPP*, 2018.