Flexible Refinement Proofs in Separation Logic

Flexible Refinement Proofs in Separation Logic

Aurel Bílý, Christoph Matheja, and Peter Müller

Department of Computer Science, ETH Zurich, Switzerland
{aurel.bily, cmatheja, peter.mueller}@inf.ethz.ch

Abstract. Refinement transforms an abstract system model into a concrete, executable program, such that properties established for the abstract model carry over to the concrete implementation. Refinement has been used successfully in the development of substantial verified systems. Nevertheless, existing refinement techniques have limitations that impede their practical usefulness. Some techniques generate executable code automatically, which generally leads to implementations with sub-optimal performance. Others employ bottom-up program verification to reason about efficient implementations, but impose strict requirements on the structure of the code, the structure of the refinement proofs, as well as the employed verification logic and tools.
In this paper, we present a novel refinement technique that removes these limitations. Our technique uses separation logic to reason about efficient concurrent implementations. It prescribes only a loose coupling between an abstract model and the concrete implementation. It thereby supports a wide range of program structures, data representations, and proof structures. We make only minimal assumptions about the underlying program logic, which allows our technique to be used in combination with a wide range of logics and to be automated using off-the-shelf separation logic verifiers. We formalize the technique, prove the central trace inclusion property, and demonstrate its usefulness on several case studies.

1 Introduction

Refinement is a powerful technique for the formal development of correct systems. It is especially useful for concurrent and distributed systems, because it allows one to establish system-wide invariants on the level of abstract models and preserve them when decomposing the system into its components. It is, thus, not surprising that several recent developments of verified systems employ refinement reasoning [2,10,14,19,22].

Traditionally, refinement is applied to mathematical models of software, for instance, in formalisms such as Event-B [1], TLA$^+$ [17], or the higher-order logics supported by interactive theorem provers such as Coq [8]. The final executable program is then produced automatically by a code generator. However, this approach generally leads to sub-optimal implementations that do not fully utilize the language features needed to produce efficient code, such as mutable heap structures and concurrency. Manually optimizing the generated code would forfeit the correctness guarantees provided by the formal development.

To address these shortcomings, recent work has combined refinement with bottom-up program verification techniques that support more features of modern programming languages [10,15,33,34]. While these approaches substantially increase the range of programs that can be developed using refinement, they come with their own limitations. First, several existing techniques restrict the structure of the executable program, which reduces expressiveness and limits the efficiency of the executable code. For instance, the methodology used in IronFleet [10] supports protocol-level concurrency, but restricts the implementations of individual components to execute sequentially. Moreover, the structure of the code must closely follow the structure of the abstract TLA$^+$model. Similarly, the Igloo methodology [33] does not allow threads to perform I/O operations concurrently. Second, most refinement techniques that support bottom-up code development are closely tied to a particular program logic, which often impedes adoption and automation. For example, refinement in DeepSpec [15] is tied to the VST logic [5], DISEL [32] comes with its own dedicated logic, and Trillium [34] leverages Iris [13]. All of these logics are very expressive, but require substantial manual effort. The underlying program logic may also impose limitations. For instance, Trillium inherits Iris's restriction to finitary behaviors, which precludes operations such as non-deterministically choosing from an infinite set. However, non-deterministic choice is essential for the specification of abstract system models, for instance, when the concrete algorithm to determine a value is an implementation decision that is taken in a later refinement step.

In this paper, we present a novel methodology to prove that an implementation refines an abstract model given as a transition system. Our methodology enables *flexible* refinement proofs along four dimensions:

1. *Abstract model.* We do not prescribe a specific formalism for abstract models, but support any transition system whose transition relation can be specified in first-order logic.
2. *Program structure.* Our methodology uses separation logic [29] to support efficient implementations, in particular, mutable state and arbitrary concurrency structures.
3. *Logic and automation.* We make only minimal assumptions about the underlying program logic, which allows our methodology to be used in combination with a wide range of logics and to be automated using off-the-shelf separation logic verifiers.
4. *Proof structure.* Our methodology prescribes only a very loose coupling between the abstract model and the concrete implementation. This maximizes flexibility when choosing the program structure (for instance, control flow, concurrency, and thread synchronization), the data representation (supporting for instance, local and shared state), and proof structure (for instance, allowing coupling invariants to be expressed via a combination of local assertions and lock invariants).

Approach. Our goal is to prove that a given implementation refines an abstract model, that is, each finite trace of the abstract model corresponds to a trace in

the transition system. This trace inclusion property guarantees that any safety property proved for the abstract model also holds for the implementation.

Our abstract models are expressed as (possibly infinite) transition systems in the style of TLA$^+$, Event-B, or other refinement frameworks. We assume that some variables of the abstract transition system (*ATS*) are declared to be part of the environment in which the system executes. These are the variables that can be observed about an execution; that is, we require trace inclusion only after projecting traces to the environment variables. Consequently, a concrete implementation may perform arbitrary internal operations as long as it manipulates the environment according to *ATS*. A typical example for environment variables are the input and output streams of a program, but our methodology also supports other cases such as the variables shared between threads. We assume that a program manipulates the environment only via dedicated operations, such as the methods of an I/O library, whose specifications express how they manipulate the environment in terms of the environment variables of *ATS*.

To reason about *ATS* and the concrete implementation within one program logic, we embed *ATS* as global ghost state into the concrete implementation: each variable of *ATS* is represented as a ghost variable of the program. The concrete program must implement the transitions of *ATS* via atomic program operations and corresponding updates of the ghost state. We identify these operations via annotations in the code; they include in particular the operations used to manipulate the environment. Proof obligations enforce that each such operation performs a valid transition of *ATS* and that they execute atomically. We do not prescribe how these atomicity checks are performed: they are trivial for sequential programs, could be performed syntactically by identifying a set of atomic operations (such as compare-and-swap and native I/O operations), could be ensured via a global locking strategy, or could be discharged in a logic that can reason about atomicity, such as TaDA [30]. This approach allows us to use any separation logic that is able to reason about shared state, in particular, standard separation logic [29], concurrent separation logic [24], and the numerous separation logics for fine-grained concurrency, e.g., [9,30,31].

A key virtue of our methodology is that refinement proofs are organized around a minimal core, namely the proof obligations showing that the program manipulates the environment variables according to the abstract model. Discharging these proof obligations generally requires suitable coupling invariants between the abstract and the concrete state, as well as updates to the ghost state to maintain them. However, we do not prescribe how to express and prove those, which enables programmers to flexibly structure the code, data, and proof.

Contributions. We make the following technical contributions:

- We present a novel verification methodology for refinement proofs that provides more flexibility than prior work in terms of the supported class of programs, the choice of verification logic and tools, and the organization of the refinement proof itself (Sec. 2).

- We formalize an instance of our methodology for a concurrent language and a standard separation logic. Our soundness proof shows trace inclusion, which implies that safety properties of the abstract model also hold for the concrete implementation (Sec. 3 and Sec. 4).
- We demonstrate the expressiveness of our methodology by encoding a series of interesting examples into the Viper language [23]. Our evaluation shows in particular that our methodology can be automated with existing separation logic verifiers (Sec. 5). We will submit our examples as an artifact.

2 Overview

In this section, we explain our methodology on a simple example that prints all integers in ascending order, starting from an arbitrary, non-negative initial value. To illustrate the flexibility of our methodology, we refine an abstract model into a concurrent implementation. The concrete state is stored in local variables of the individual threads, which requires a non-trivial, decentralized coupling invariant. This section provides an informal overview; the details of the programming language and proof rules will be formalized in the next sections.

Abstract Model. The abstract model of our system is defined in Fig. 1. We represent the output stream of the system `stdOut` as a sequence of integers. This variable belongs to the environment, whereas `count` is internal to the system. The initial state has an arbitrary non-negative value for `count`, which illustrates the common case that abstract models choose values non-deterministically. A valid transition adds the current value of `count` to the output stream and then increments it.

Embedding of the Abstract Model. As explained in the introduction, we embed the abstract model as ghost state into the concrete implementation. To this end, we declare a global ghost variable:

```
ghost var count: Int
```

We assume that the environment variable `stdOut` is predeclared for each program. It is manipulated via a `print(x)` statement, which requires ownership of `stdOut` (in the sense of separation logic: $\texttt{stdOut} \mapsto \_$) and appends its argument `x` to the output stream. Other input and output channels, as well as other I/O operations are handled analogously.

To refine the abstract model, the concrete implementation may manipulate `count` and `stdOut` only according to the abstract model presented in Fig. 1. We

Vars: $\texttt{count: Int, stdOut: Seq[Int]}$
Init: $0 \leq \texttt{count} \land \texttt{stdOut} = []$
Next: $\texttt{stdOut}' = \texttt{stdOut ++ [count]} \land \texttt{count}' = \texttt{count} + 1$

Fig. 1. Abstract model of our example. Here, unprimed and primed variables refer the variables values before and after the transition, respectively.

enforce this requirement by protecting both variables with a *ghost lock*. Like a regular lock, a ghost lock is equipped with a lock invariant that must be established when the lock is initialized and must be preserved by the operations between an acquire and a release operation. In contrast to a regular lock, a ghost lock is not present at run time; that is, it does not block execution and, thus, cannot cause deadlock. Erasing ghost locks from the program is sound because the operations between an acquire and a release must execute atomically.

In our language, we initialize the ghost lock with a dedicated `Init` ghost statement, which checks the `Init` assertion from the abstract model as well as the lock invariant (in particular, it transfers ownership of the variables in the lock invariant from the executing thread to the ghost lock). A dedicated `Next` ghost block statement acquires the ghost lock, executes the block of the statement, and then releases the ghost lock. Upon release, it checks that the `Next` assertion from the abstract model holds between the state in which the ghost lock was acquired and the state where it is released. That is, the `Next` statement executes *one* transition of the abstract model (or stutters). The block of a `Next` statement must be atomic; we enforce this requirement syntactically, that is, we check that the body consists of at most one atomic statement plus an arbitrary number of ghost operations. However, our methodology also supports more sophisticated approaches, for instance, logics that can prove atomicity [30]. Note that both `Init` and `Next` are ghost statements, that is, part of the specification. The executable program contains their blocks, but not the manipulation of the ghost lock. They are supported by any separation logic that handles locks.

The invariant of the ghost lock must contain at least fractional ownership [3] to each variable of the abstract model, which ensures that once the ghost lock has been initialized, those variables can be modified *only* within a `Next` statement and, thus, the `Next` relation of the abstract model is checked on each modification. In our example, the ghost lock invariant contains fractional ownership of the `count` variable and full ownership of `stdOut`:

$$\texttt{count} \overset{\frac{2}{3}}{\mapsto} \_ \;*\; \texttt{stdOut} \mapsto \_$$

Our proof rules enforce that `print`—the only way to modify `stdOut`—can be executed only after the ghost lock has been initialized with an `Init` statement. Consequently, the above lock invariant guarantees that `print` operations can occur only within a `Next` block and, thus, all changes to the environment are checked to comply with the abstract model. We assume that `print` is atomic, which is standard for I/O operations.

Fig. 2 shows the concrete implementation. After initializing the abstract state, the `Init` operation checks that the initial state is valid and creates the ghost lock that guards further updates to the abstract state, in particular, all executions of `print`. Note that our implementation fixes a concrete initial value for `count`, whereas the abstract model permits an arbitrary non-negative number (see Fig. 1). Resolving non-determinism is very common during refinement. We explain the body of the `Init` statement below.

```
count := 0 ;
Init {
    evenTurn := true ; lastEven := −1 ; lastOdd := 0 ;
    lock ℒ { /* even-thread */ ‖ /* odd-thread */ }
}
```

Fig. 2. The concrete implementation of our example. The even-thread is presented in Fig. 3; the odd-thread is analogous.

```
var c := 0 ;
while true  invariant lastEven ↦(1/2) 2 ∗ c − 1
{
    with ℒ when evenTurn  {
        Next {  /∗ print + ghost update ∗/
            print(2 ∗ c) ; count := count + 1 }
        lastEven := lastEven + 2 ;
        evenTurn := !evenTurn ;
        c := c + 1
    }  /∗ release lock ℒ ∗/
}
```

Fig. 3. Implementation of the thread that prints the even numbers. The `Next` block marks an atomic transition; its body is atomic because the `print` statement is atomic and the subsequent assignment is a ghost statement.

Concrete Program State and Coupling. Our concrete implementation uses two concurrent threads (parallel branches) that print the even and odd numbers, respectively. To synchronize these two threads, we declare a global Boolean variable that indicates whether the next number to be printed is even:

```
var evenTurn: Bool
```

This variable is protected by a global (regular, non-ghost) lock $\mathcal{L}$.

Fig. 3 shows the implementation of the thread that prints the even numbers; the other thread is analogous. Both threads loop indefinitely. In each iteration, they acquire the lock $\mathcal{L}$ (waiting until `evenTurn` is true resp. false), execute a transition (explained later), flip `evenTurn`, and release the lock. Each thread has a local variable `c` that counts how many numbers it has printed. This design illustrates that our methodology supports flexible combinations of global concrete state (such as `evenTurn`) and local concrete state (such as `c`). While global concrete state can easily be connected to the abstract state via lock invariants, local variables require more flexible ways of expressing the coupling invariant.

For instance, at the beginning of each loop iteration of the even-thread, the abstract counter `count` is equal to $2 * \texttt{c}$ in case `evenTurn` is true. This condition allows us to prove that the `print` operation in the loop body is indeed permitted by the abstract model. However, this coupling invariant cannot be included

in a lock invariant because locks do not protect local variables. Nor can it be expressed as a loop invariant because that would require that the even-thread holds on to some ownership of `count`, which would prevent the odd-thread from ever updating it.

A relation between the local variable `c` and the shared variable `count` can be proved in many ways, for instance, by using classical rely-guarantee reasoning [12] or concurrent abstract predicates [9]. Our refinement methodology is compatible with any such logic. In our example, we use a standard encoding of the Owicki-Gries counter [26] in separation logic. For this purpose, we introduce two global ghost variables `lastEven` and `lastOdd` that keep track of the effect of each individual thread:

```
ghost var lastEven: Int, lastOdd: Int
```

We relate these ghost variables to the local variable `c` in each thread via the thread's loop invariant (see Fig. 3), and also to the global `count` via the lock invariant of lock $\mathcal{L}$:

$$\begin{aligned}&\texttt{evenTurn} \mapsto \_ * \texttt{lastEven} \overset{\frac{1}{2}}{\mapsto} \_ * \texttt{lastOdd} \overset{\frac{1}{2}}{\mapsto} \_ * \texttt{count} \overset{\frac{1}{3}}{\mapsto} \_ * \\ &(\texttt{evenTurn} \Rightarrow \texttt{count} = \texttt{lastOdd} \wedge \texttt{lastEven} = \texttt{lastOdd} - 1) * \\ &(\neg\texttt{evenTurn} \Rightarrow \texttt{count} = \texttt{lastEven} \wedge \texttt{lastOdd} = \texttt{lastEven} - 1)\end{aligned}$$

The lock and the loop invariants *together* form the coupling invariant for our example. For the even-thread, we get $\texttt{lastEven} = 2 * c - 1$ from the loop invariant, `evenTurn` from the `with`-statement, and $(\texttt{evenTurn} \Rightarrow \texttt{count} = \texttt{lastOdd} \wedge \texttt{lastEven} = \texttt{lastOdd} - 1)$ from the lock invariant. These three conditions together imply $\texttt{count} = 2 * c$, which is required to show that the printed value is permitted by the abstract model.

Discussion. Our example illustrates that our methodology enables flexible refinement proofs, which are required to support a wide range of efficient implementations. We refined an abstract model into a concurrent implementation that uses both local and mutable shared state, as well as thread synchronization via locks. The proof makes only minimal assumptions about the underlying program logic. Concretely, we use concurrent separation logic, locks, ghost variables, and fractional permissions. These features are supported and automated by many existing separation logic verifiers. For instance, an encoding of our example into Viper verifies automatically in around 3.8s. Combining our refinement methodology with other, more advanced program logics is possible.

Finally, our example demonstrates that our methodology enables flexible proof structures. Proofs are essentially derived backwards from those statements that manipulate environment variables, here, `print`. These statements require that the abstract state has been initialized and that the modification of the environment variables is permitted by the abstract model. Any proof structure that establishes these properties is compatible with our methodology. In our example, we use a combination of loop invariants and lock invariants, connected via global ghost variables, to establish the necessary coupling relation. This flexibility is essential to support a wide range of data, control, and concurrency structures.

3 Preliminaries: Concurrent Separation Logic

Our verification technique for refinement proofs does not depend on a particular program logic but can, in principle, be integrated into most separation logic-based verification techniques. To make this claim more concrete, we will formalize (in Sec. 4) our methodology on top of an elementary formalization of concurrent separation logic with fractional permissions by Vafeiadis [35]. Since Vafeiadis' soundness proof generalizes well to more advanced concurrent separation logics, such as [9], we expect the same when using our technique for refinement on top of such advanced logics.

This section briefly recapitulates the main ingredients of concurrent separation logic (CSL). More precisely, we introduce a small concurrent programming language, its underlying model of program states, and its operational semantics. Furthermore, we discuss CSL's assertion language and proof rules.

3.1 Programming Language

We consider a small programming language that supports heap-manipulating instructions, structured concurrency, and locks. More precisely, the set **Cmds** of *commands* in our programming language is given by the grammar

$$
\begin{array}{lcl}
C & ::= & C_{\text{base}} \\
& & \mid C \,;\, C \\
& & \mid \texttt{if}\ E\ \{\, C \,\}\ \texttt{else}\ \{\, C \,\} \\
& & \mid \texttt{while}\ E\ \{\, C \,\} \\
& & \mid C \parallel C \\
& & \mid \texttt{lock}\ \mathcal{L}\ \{\, C \,\} \\
& & \mid \texttt{with}\ \mathcal{L}\ \texttt{when}\ E\ \{\, C \,\} \\
& & \mid \texttt{within}\ \mathcal{L}\ \{\, C \,\} \textit{semantics}
\end{array}
\qquad
\begin{array}{lcl}
C_{\text{base}} & ::= & \texttt{skip} \\
& & \mid x\ \texttt{:=}\ E \\
& & \mid [E]\ \texttt{:=}\ E \\
& & \mid x\ \texttt{:=}\ [E] \\
& & \mid \texttt{free}(E) \\
& & \mid x\ \texttt{:=}\ \texttt{new}(E)
\end{array}
$$

where x is a *variable* in the set **Vars**, E is an *expression* over variables, and $\mathcal{L}$ is a *lock identifier* taken from the arbitrary, but finite set $\mathbf{Locks} = \{\, \mathcal{L}, \dots \}$.

We briefly go over our language: in addition to the assignment x `:=` E and the usual control-flow structures, $C_1 \parallel C_2$ is the *parallel composition* of C_1 and C_2; `lock` $\mathcal{L}\ \{\, C \,\}$ *declares a new lock* $\mathcal{L}$ that can be used in C and expires after termination of C; the *conditional critical region* (CCR) `with` $\mathcal{L}$ `when` $E\ \{\, C \,\}$ acquires lock $\mathcal{L}$ if condition E holds (and waits otherwise), executes C, and releases $\mathcal{L}$ again upon termination of C; `within` $\mathcal{L}\ \{\, C \,\}$ is an internal command that we will use to indicate that C is executed while holding the lock $\mathcal{L}$. Furthermore, $[E]$ denotes the value at the memory address given by E. x `:=` $[E]$ *reads* the value at address E and assigns it to x; $[E]$ `:=` E' *writes* the value of E' to the address E. Moreover, `free` (E) disposes of location E, and x `:=` `new`(E) allocates a free memory address, assigns it to x, and stores E at that address. We use structured concurrency and global locks to simplify the formalization, but our methodology also supports dynamic threads and locks.

3.2 Program States

A *program state* $(s, \mathfrak{h})$ consists of a *stack* s, i.e., a valuation of variables, and a *heap* $\mathfrak{h}$ modeling dynamically allocated memory. Formally, we fix a set $\mathbf{Vals} = \{ v, \dots \}$ of *values* containing, e.g., Booleans, integers, and sequences. The set **Stacks** consists of all mappings s from variables in **Vars** to values in **Vals**, i.e.,

$$\mathbf{Stacks} \triangleq \mathbf{Vars} \to \mathbf{Vals} \ .$$

Moreover, we fix a countably infinite set **Addrs** of *memory addresses*, and the set $\mathbf{Perms} \triangleq [0, 1]$ of *fractional permissions*, where permission $\rho = 1$ means write access, $\rho \in (0, 1)$ means read access, and $\rho = 0$ means no access, respectively.

The set **Heaps** of *heaps* consists of all finite partial functions $\mathfrak{h}$ that map addresses in their domain $\mathit{dom}(\mathfrak{h}) \subseteq \mathbf{Addrs}$ to permission-value pairs, i.e.,

$$\mathbf{Heaps} \triangleq \mathbf{Addrs} \rightharpoonup_{\text{fin}} (\mathbf{Perms} \times \mathbf{Vals}) \ .$$

For $\mathfrak{h}(a) = (\rho, v)$, we define the projections $\mathsf{perm}(\mathfrak{h}(a)) = \rho$ and $\mathsf{val}(\mathfrak{h}(a)) = v$. We denote by $\left\{ a_1 \overset{\rho_1}{\longmapsto} v_1, \dots, a_n \overset{\rho_n}{\longmapsto} v_n \right\}$ the heap $\mathfrak{h}$ with $\mathit{dom}(\mathfrak{h}) = \{ a_1, \dots, a_n \}$ and $\mathfrak{h}(a_i) = (\rho_i, v_i)$, where $i \in \{ 1, \dots, n \}$; the empty heap is denoted by $h_\emptyset$.

The *addition* of $\mathfrak{h}_1 \oplus \mathfrak{h}_2$ of heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ is defined as

$$(\mathfrak{h}_1 \oplus \mathfrak{h}_2)(a) \triangleq \begin{cases} (\rho_1 + \rho_2, v) & \text{if } a \in \mathit{dom}(\mathfrak{h}_1) \cap \mathit{dom}(\mathfrak{h}_2) \text{ and } \mathfrak{h}_1(a) = (\rho_1, v) \\ & \qquad \text{and } \mathfrak{h}_2(a) = (\rho_2, v) \text{ and } \rho_1 + \rho_2 \leq 1 \\ \mathfrak{h}_1(a) & \text{if } a \in \mathit{dom}(\mathfrak{h}_1) \setminus \mathit{dom}(\mathfrak{h}_2) \\ \mathfrak{h}_2(a) & \text{if } a \in \mathit{dom}(\mathfrak{h}_2) \setminus \mathit{dom}(\mathfrak{h}_1) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We write $\mathit{def}(\mathfrak{h}_1 \oplus \mathfrak{h}_2)$ if $(\mathfrak{h}_1 \oplus \mathfrak{h}_2(a)$ is well-defined for all $a \in \mathit{dom}(\mathfrak{h}_1) \cup \mathit{dom}(\mathfrak{h}_2)$. Furthermore, $\mathfrak{h}\,[a := v]$ denotes the heap $\mathfrak{h}$ except that a maps to v (with permission 1), i.e.,

$$\mathfrak{h}\,[a := v]\,(a') \triangleq \begin{cases} (1, v) & \text{if } a' = a \\ \mathfrak{h}(a') & \text{if } a' \neq a \ . \end{cases}$$

For every stack s, $s\,[x := v]$ is defined analogously. We define the heap $\mathfrak{h}\,[a := \bot] \triangleq \{ a' \mapsto \mathfrak{h}(a') \mid a' \in \mathit{dom}(\mathfrak{h}) \setminus \{a\} \}$, which is obtained from $\mathfrak{h}$ by removing a from its domain. Finally, the set **NormHeaps** of *normal heaps* consists of all heaps h that assign the full permission 1 to every address, i.e.,

$$\mathbf{NormHeaps} \triangleq \{ \mathfrak{h} \in \mathbf{Heaps} \mid \forall a \in \mathit{dom}(\mathfrak{h}) \ . \ \mathsf{perm}(\mathfrak{h}(a)) = 1 \} \ .$$

We typically write h instead of $\mathfrak{h}$ to highlight that a heap is normal; by slight abuse of notation, we use $h(a)$ as a shortcut for $\mathsf{val}(h(a))$. Clearly, for every heap $\mathfrak{h}$, there exists a heap $\mathfrak{h}'$ such that $\mathfrak{h} \oplus \mathfrak{h}'$ is a normal heap.

3.3 Operational Semantics

The semantics of commands is defined in terms of a small-step execution relation.

Toward a formal definition, we first clarify the semantics of expressions. We do not fix a specific syntax for expressions; instead, we assume that every expression E is associated with a function $E\colon \mathbf{Stacks} \to \mathbf{Vals}$ such that $E(s)$ is the value obtained from evaluating E in stack s.

The set **Confs** of *program configurations* c consists of all command-stack-normal-heap triples plus a dedicated error state `abort`, that is,

$$\mathbf{Confs} \triangleq (\mathbf{Cmds} \times \mathbf{Stacks} \times \mathbf{NormHeaps}) \cup \{\, \texttt{abort} \,\} .$$

We use normal heaps because permissions are a reasoning concept that does not exist when executing a program. The *small-step operational semantics* of commands is defined as the execution relation $\to \subseteq \mathbf{Confs} \times \mathbf{Confs}$ given by the rules in Fig. 4. Most rules in Fig. 4 are standard (cf. [35]) and reflect the behavior informally explained in Sec. 3.1; we briefly discuss particularities.

Usage and Declaration of Locks. The operational semantics uses the command structure to record which locks are currently declared and acquired, respectively. That is, $\texttt{lock}\ \mathcal{L}\ \{\, C \,\}$ indicates that $\mathcal{L}$ is declared in C; the internal command $\texttt{within}\ \mathcal{L}\ \{\, C \,\}$ indicates that C holds $\mathcal{L}$ during its execution. We denote by $\mathbf{Locks}(C)$ the set of all locks that are declared in C. Analogously, $\mathbf{locked}\ (C)$ denotes the set of all locks that are currently held by C. For example, if C' is *not* of the form $\texttt{lock}\ \mathcal{L}\ \{\ldots\}$ or $\texttt{within}\ \mathcal{L}\ \{\ldots\}$, and C is given by

$$\texttt{lock}\ \mathcal{L}_1\ \{\, \texttt{lock}\ \mathcal{L}_2\ \{\ldots \texttt{lock}\ \mathcal{L}_n\ \{\, \texttt{within}\ \mathcal{L}'_1\ \{\ldots \texttt{within}\ \mathcal{L}'_m\ \{\, C' \,\}\,\}\,\}\ldots\} ,$$

then $\mathbf{Locks}(C) = \{\, \mathcal{L}_1, \ldots, \mathcal{L}_n \,\}$ and $\mathbf{locked}(C) = \{\, \mathcal{L}'_1, \ldots, \mathcal{L}'_m \,\}$.

The semantics of $\texttt{lock}\ \mathcal{L}\ \{\, C \,\}$ consequently keeps the lock $\mathcal{L}$ declared and performs a step of C until termination; after that, the lock declaration expires.

CCRs and Parallel Composition. For CCRs $\texttt{with}\ \mathcal{L}\ \texttt{when}\ E\ \{\, C \,\}$, we first check whether condition E holds in the current state. If so, we acquire lock $\mathcal{L}$ and enter the CCR by moving to the internal command $\texttt{within}\ \mathcal{L}\ \{\, C \,\}$, which will execute C, and release the lock upon termination of C. If E does not hold, no transition is possible, i.e., we need to wait for other threads.

The rules for the parallel composition $C_1 \parallel C_2$ model all interleaved executions of C_1 and C_2. They include a sanity check stating that C_1 and C_2 do not hold the same lock at the same time, i.e., locks provide mutual exclusion. Moreover, there is a rule that prematurely aborts executions whenever a command admits a data race, that is, C_1 (resp. C_2) writes to and C_2 (resp. C_1) accesses (i.e., reads from or writes to) the same address that is *outside* of a CCR, and thus without protection by a lock.

$$
\begin{array}{lll}
(\textsc{Assign}) & x := E, s, h \to \mathtt{skip}, s\,[x := E(s)]\,, h & \\
(\textsc{Read}) & x := [E], s, h \to \mathtt{skip}, s\,[x := h(v)]\,, h & \text{if } E(s) = v \in \mathit{dom}\,(h) \\
(\textsc{ReadA}) & x := [E], s, h \to \mathtt{abort} & \text{if } E(s) \notin \mathit{dom}\,(h) \\
(\textsc{Write}) & [E] := E', s, h \to \mathtt{skip}, s, h\,[a := E'(s)] & \text{if } E(s) = a \in \mathit{dom}\,(h) \\
(\textsc{WriteA}) & [E] := E', s, h \to \mathtt{abort} & \text{if } E(s) \notin \mathit{dom}\,(h) \\
(\textsc{Alloc}) & x := \mathtt{new}(E), s, h \to \mathtt{skip}, s\,[x := a]\,, h\,[a := E(s)] & \text{if } a \in \mathbf{Addrs} \setminus \mathit{dom}\,(h) \\
(\textsc{Free}) & \mathtt{free}(E), s, h \to \mathtt{skip}, s, h\,[a := \bot] & \text{if } E(s) = a \in \mathit{dom}\,(h) \\
(\textsc{FreeA}) & \mathtt{free}(E), s, h \to \mathtt{abort} & \text{if } E(s) \notin \mathit{dom}\,(h) \\
(\textsc{With}) & \mathtt{with}\ \mathcal{L}\ \mathtt{when}\ E\ \{\, C\,\}, s, h \to \mathtt{within}\ \mathcal{L}\ \{\, C\,\}, s, h & \text{if } E(s) = \mathtt{true} \\
(\textsc{Ite1}) & \mathtt{if}\ E\ \{\, C_1\,\}\ \mathtt{else}\ \{\, C_2\,\}, s, h \to C_1, s, h & \text{if } E(s) = \mathtt{true} \\
(\textsc{Ite2}) & \mathtt{if}\ E\ \{\, C_1\,\}\ \mathtt{else}\ \{\, C_2\,\}, s, h \to C_2, s, h & \text{if } E(s) = \mathtt{false} \\
(\textsc{While}) & \mathtt{while}\ E\ \{\, C\,\}, s, h \to \mathtt{if}\ E\ \{\, C\ ;\ \mathtt{while}\ E\ \{\, C\,\}\,\}\ \mathtt{else}\ \{\, \mathtt{skip}\,\}, s, h &
\end{array}
$$

$$
\begin{array}{llll}
(\textsc{WithinL}) & \mathtt{within}\ \mathcal{L}\ \{\, C\,\}, s, h \to \mathtt{abort} & \text{if } \mathcal{L} \in \mathbf{locked}(C) & \\
(\textsc{WithinS}) & \mathtt{within}\ \mathcal{L}\ \{\, \mathtt{skip}\,\}, s, h \to \mathtt{skip}, s, h & (\textsc{SeqS}) & \mathtt{skip}\ ;\ C_2, s, h \to C_2, s, h \\
(\textsc{LockS}) & \mathtt{lock}\ \mathcal{L}\ \{\, \mathtt{skip}\,\}, s, h \to \mathtt{skip}, s, h & (\textsc{ParS}) & \mathtt{skip} \parallel \mathtt{skip}, s, h \to \mathtt{skip}, s, h
\end{array}
$$

$$
(\textsc{Seq})\ \frac{C_1, s, h \to C_1', s', h'}{C_1\ ;\ C_2, s, h \to C_1'\ ;\ C_2, s', h'} \qquad (\textsc{SeqA})\ \frac{C_1, s, h \to \mathtt{abort}}{C_1\ ;\ C_2, s, h \to \mathtt{abort}}
$$

$$
(\textsc{Par1})\ \frac{C_1, s, h \to C_1', s', h' \quad \mathbf{locked}(C_1') \cap \mathbf{locked}(C_2) \ = \ \emptyset}{C_1 \parallel C_2, s, h \to C_1' \parallel C_2, s', h'} \qquad (\textsc{Par1A})\ \frac{C_1, s, h \to \mathtt{abort}}{C_1 \parallel C_2, s, h \to \mathtt{abort}}
$$

$$
(\textsc{Par2})\ \frac{C_2, s, h \to C_2', s', h' \quad \mathbf{locked}(C_1) \cap \mathbf{locked}(C_2') \ = \ \emptyset}{C_1 \parallel C_2, s, h \to C_1' \parallel C_2, s', h'} \qquad (\textsc{Par2A})\ \frac{C_2, s, h \to \mathtt{abort}}{C_1 \parallel C_2, s, h \to \mathtt{abort}}
$$

$$
(\textsc{Race})\ \frac{(\mathbf{accesses}(C_1, s) \cap \mathbf{writes}(C_2, s)) \cup (\mathbf{writes}(C_1, s) \cap \mathbf{accesses}(C_2, s)) \neq \emptyset}{C_1 \parallel C_2, s, h \to \mathtt{abort}}
$$

$$
(\textsc{Lock})\ \frac{C, s, h \to C', s', h'}{\mathtt{lock}\ \mathcal{L}\ \{\, C\,\}, s, h \to \mathtt{lock}\ \mathcal{L}\ \{\, C'\,\}, s', h'} \qquad (\textsc{LockA})\ \frac{C, s, h \to \mathtt{abort}}{\mathtt{lock}\ \mathcal{L}\ \{\, C\,\}, s, h \to \mathtt{abort}}
$$

$$
(\textsc{Within})\ \frac{C, s, h \to C', s', h'}{\mathtt{within}\ \mathcal{L}\ \{\, C\,\}, s, h \to \mathtt{within}\ \mathcal{L}\ \{\, C'\,\}, s', h'}
$$

$$
(\textsc{WithinA})\ \frac{C, s, h \to \mathtt{abort}}{\mathtt{within}\ \mathcal{L}\ \{\, C\,\}, s, h \to \mathtt{abort}}
$$

Fig. 4. Rules of the small-step operational semantics for commands in **Cmds**. Here, $\mathbf{locked}(C)$ is the set of all locks $\mathcal{L}$ held by C, i.e., $\mathtt{within}\ \mathcal{L}\ \{\ldots\}$ is a sub-command of C. Furthermore, $\mathbf{accesses}(C, s)$ (resp. $\mathbf{writes}(C, s)$) denotes the set of those addresses a that are not exclusively owned by C, i.e., appear only in sub-commands $\mathtt{Next}\ \{\ldots\}$ or $\mathtt{with}\ \mathcal{L}\ \mathtt{when}\ E\ \{\ldots\}$), and are accessed (resp. modified) by C given stack s.

Table 1. Semantics of assertions.

P	$s, \mathfrak{h} \models P$ iff
E	$E\ (\mathrm{s}) = \mathtt{true}$
$Q \wedge R$	$s, \mathfrak{h} \models Q$ and $s, \mathfrak{h} \models R$
$\neg Q$	$s, \mathfrak{h} \not\models Q$
$\forall x\ .\ Q$	for all $v \in \mathbf{Vals}$, $s\,[x := v]\,, \mathfrak{h} \models Q$
$\exists x\ .\ Q$	exists $v \in \mathbf{Vals}$ s.t. $s\,[x := v]\,, \mathfrak{h} \models Q$
$\mathtt{emp}$	$dom\,(\mathfrak{h}) = \emptyset$
$Q * R$	exists $\mathfrak{h}_1, \mathfrak{h}_2$ s.t. $\mathfrak{h} = \mathfrak{h}_1 \oplus \mathfrak{h}_2$ and $s, \mathfrak{h}_1 \models Q$ and $s, \mathfrak{h}_2 \models R$
$Q \mathrel{-\!\!*} R$	for all $\mathfrak{h}'$, $(def(\mathfrak{h} \oplus \mathfrak{h}')$ and $s, \mathfrak{h}' \models Q)$ implies $s, \mathfrak{h} \oplus \mathfrak{h}' \models R$
$\bigstar_{i \in I} P_i$	$I = \emptyset$ or exists $j \in I$ s.t. $s, \mathfrak{h} \models P_j * \bigstar_{i \in I \setminus \{j\}} P_i$

3.4 Assertions

Syntax. The syntax of *separation logic assertions* includes, amongst others, all Boolean expressions E supported by our programming language Formally, the set **SL** of separation logic assertions is given by the grammar

$$P \ ::= \ E \mid P \wedge P \mid \neg P \mid \forall x\ .\ P \mid \exists x\ .\ P \qquad \text{(FOL)}$$

$$\mid \mathtt{emp} \mid E \overset{\rho}{\mapsto} E \mid P * P \mid P \mathrel{-\!\!*} P \ \mid \bigstar_{i \in I} P_i\ , \qquad \text{(SL)}$$

where E is a Boolean expression over variables, x is a variable in **Vars**, ρ is a permission in **Perms**, and I is a finite set such that each $i \in I$ is associated with an assertion P_i. We use syntactic sugar, such as $P \Rightarrow Q$, $P \vee P$ and, in particular, $E \overset{\rho}{\hookrightarrow} E \triangleq E \overset{\rho}{\mapsto} E * \mathtt{true}$, $E \overset{\rho}{\mapsto} \_ \triangleq \exists y\ .\ E \overset{\rho}{\hookrightarrow} y$, and $E \overset{\rho}{\hookrightarrow} \_ \triangleq \exists y\ .\ E \overset{\rho}{\hookrightarrow} y$.

We denote by **FOL** the set of all first-order logic formulas, i.e., those formulas that can be constructed from the first line of the above grammar.

Semantics. Assertions are interpreted over pairs $(s, \mathfrak{h})$ consisting of a stack s and a heap (with permissions) $\mathfrak{h}$. Tab. 1 shows the formal semantics of assertions.

Intuitively, $\mathtt{emp}$ specifies the empty heap; the points-to assertion $E \overset{\rho}{\mapsto} E'$ specifies that the heap contains *exactly one* address E that is mapped to E' with permission ρ; $E \overset{\rho}{\hookrightarrow} E'$ is an intuitionistic version stating that the heap contains *at least* permission ρ for the address E, which is mapped to E'. $E \overset{\rho}{\mapsto} \_$ and $E \overset{\rho}{\hookrightarrow} \_$ are analogous but do not require a specific value at the address given by E. The separating conjunction $P * Q$ specifies that the heap can be partitioned into two parts such that one part satisfies P and the other part satisfies Q. $\bigstar_{i \in I} P_i$ is an iterative version of the separating conjunction. Finally, the magic wand $P \mathrel{-\!\!*} Q$ specifies that Q holds in a heap $\mathfrak{h}$ after it has been extended by any heap that satisfies P (and can be added to $\mathfrak{h}$).

We call an assertion P *valid*, written $\models P$, if and only if $s, \mathfrak{h} \models P$ holds for all stacks $s \in \mathbf{Stacks}$ and heaps $\mathfrak{h} \in \mathbf{Heaps}$.

We denote by $\mathbf{FV}(P)$ the set of *free variables* (i.e., those that are not bound by quantifiers) of assertion P. Moreover, $P[x/v]$ denotes the *substitution* of every free occurrence of variable x in assertion P by v.

3.5 Proof System

The last step of our recapitulation of concurrent separation logic (CSL) presents the formal triples and proof rules for reasoning about commands. To formalize both, we first introduce lock environments and lock invariants.

Lock Environments. In contrast to the operational semantics (cf. Sec. 3.3), which has a global view on the full program state, the CSL proof system considers only the local state i.e., those parts of the current program state that are accessible to the local command. The remainder of the global state is either framed around the command or protected by locks. To this end, we associate every declared lock $\mathcal{L}$ with a *lock invariant* R that specifies the part of the global state that is protected by $\mathcal{L}$. Formally, the assignment of lock invariants to locks is captured by a *lock environment* Γ, which is given by the grammar

$$\Gamma \ ::= \ \emptyset \mid \Gamma, \mathcal{L}\colon R \ ,$$

where $\mathcal{L} \in \mathbf{Locks}$ and R is an $\mathbf{SL}$ assertion representing the lock invariant. Moreover, given a lock environment Γ, the *lock invariant of lock* $\mathcal{L}$ is

$$\Gamma(\mathcal{L}) \ \triangleq \ \begin{cases} R & \text{if } \Gamma = \Gamma', \mathcal{L}\colon R \\ \Gamma'(\mathcal{L}) & \text{if } \Gamma = \Gamma', \mathcal{L}'\colon R' \text{ and } \mathcal{L}' \neq \mathcal{L} \\ \mathsf{emp} & \text{if } \Gamma = \emptyset \ . \end{cases}$$

Intuitively, if $\Gamma(\mathcal{L}) = R$, then R specifies a portion of the global state that is not part of the local state but is shared with other threads and can be modified by them. In particular, if $\Gamma(\mathcal{L}) = \mathsf{emp}$, then $\mathcal{L}$ does not appear in Γ (or the declared invariant is emp) and nothing is shared with other threads.

Declaring, acquiring, and releasing the lock $\mathcal{L}$ can be understood as a transfer of the portion of the state specified by R between the local state and Γ, i.e., the part of the global state that is shared with other threads: whenever we declare $\mathcal{L}$, R is shared with other threads and thus moved from the local state into Γ; once the declaration expires, R is transferred back into the local state. Whenever we acquire the lock $\mathcal{L}$, R is moved from Γ into the local state; whenever we release the lock $\mathcal{L}$, R is moved back from the local state into Γ.

CSL Judgments. Judgments that can be derived in CSL are of the form

$$\Gamma \ \vdash \ \{P\} \ C \ \{Q\} \ ,$$

where Γ is a lock environment such that $\bigstar_{\mathcal{L} \in \mathbf{Locks}} \Gamma(\mathcal{L})$ describes the shared state, $P \in \mathbf{SL}$ is the precondition evaluated in the local state, C is a command in $\mathbf{Cmds}$, and $Q \in \mathbf{SL}$ is the postcondition evaluated in the local state. Fig. 5 shows the standard proof rules of CSL [35]. In particular, the aforementioned transfer between local and shared state can be observed in the rules (Lock) and (With). A detailed discussion of the CSL proof rules is found in [24,4].

$$(\textsc{Skip})\frac{}{\Gamma \vdash \{P\}\ \texttt{skip}\ \{P\}} \qquad (\textsc{Assign})\frac{x \notin \mathbf{FV}(\Gamma)}{\Gamma \vdash \{P[x/E]\}\ x\ \texttt{:=}\ E\ \{P\}}$$

$$(\textsc{Write})\frac{}{\Gamma \vdash \left\{E \overset{1}{\mapsto} \_\right\}\ [E]\ \texttt{:=}\ E'\ \left\{E \overset{1}{\mapsto} E'\right\}}$$

$$(\textsc{Read})\frac{x \notin \mathbf{FV}(E, E', \Gamma)}{\Gamma \vdash \left\{E \overset{\rho}{\mapsto} E'\right\}\ x\ \texttt{:=}\ [E]\ \left\{E \overset{\rho}{\mapsto} E' \wedge x = E'\right\}}$$

$$(\textsc{Alloc})\frac{x \notin \mathbf{FV}(\Gamma, E)}{\Gamma \vdash \{\texttt{emp}\}\ x\ \texttt{:=}\ \texttt{new}(E)\ \left\{x \overset{1}{\mapsto} E\right\}}$$

$$(\textsc{Free})\frac{E \notin \mathbf{GhostAddrs}}{\Gamma \vdash \left\{E \overset{1}{\mapsto} \_\right\}\ \texttt{free}(E)\ \{\texttt{emp}\}}$$

$$(\textsc{Seq})\frac{\Gamma \vdash \{P\}\ C_1\ \{R\} \quad \Gamma \vdash \{R\}\ C_2\ \{Q\}}{\Gamma \vdash \{P\}\ C_1\ ;\ C_2\ \{Q\}}$$

$$(\textsc{Cond})\frac{\Gamma \vdash \{P \wedge E\}\ C_1\ \{Q\} \quad \Gamma \vdash \{P \wedge \neg E\}\ C_2\ \{Q\}}{\Gamma \vdash \{P\}\ \texttt{if}\ E\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}\ \{Q\}}$$

$$(\textsc{While})\frac{\Gamma \vdash \{I \wedge E\}\ C\ \{I\}}{\Gamma \vdash \{I\}\ \texttt{while}\ E\ \{\,C\,\}\ \{I \wedge \neg E\}}$$

$$(\textsc{Par})\frac{\begin{array}{c}\Gamma \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \mathbf{FV}(P_1, C_1, Q_1) \cap \mathbf{Mod}(C_2) = \emptyset \\ \Gamma \vdash \{P_1\}\ C_2\ \{Q_2\} \quad \mathbf{FV}(P_2, C_2, Q_2) \cap \mathbf{Mod}(C_1) = \emptyset\end{array}}{\Gamma \vdash \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

$$(\textsc{Lock})\frac{\Gamma, \mathcal{L}\colon R \vdash \{P\}\ C\ \{Q\}}{\Gamma \vdash \{R * P\}\ \texttt{lock}\ \mathcal{L}\ \{\,C\,\}\ \{R * Q\}}$$

$$(\textsc{With})\frac{\Gamma \vdash \{(P * R) \wedge E\}\ C\ \{Q * R\}}{\Gamma, \mathcal{L}\colon R \vdash \{P\}\ \texttt{with}\ \mathcal{L}\ \texttt{when}\ E\ \{\,C\,\}\ \{Q\}}$$

$$(\textsc{Frame})\frac{\Gamma \vdash \{P\}\ C\ \{Q\} \quad \mathbf{FV}(R) \cap \mathbf{Mod}(C) = \emptyset}{\Gamma \vdash \{P * R\}\ C\ \{Q * R\}}$$

$$(\textsc{Cons})\frac{\begin{array}{c}\Gamma \vdash \{P\}\ C\ \{Q\} \\ \models P' \Rightarrow P \qquad \models Q \Rightarrow Q'\end{array}}{\Gamma \vdash \{P'\}\ C\ \{Q'\}} \qquad (\textsc{Ex})\frac{\Gamma \vdash \{P\}\ C\ \{Q\} \quad x \notin \mathbf{FV}(C)}{\Gamma \vdash \{\exists x\ .\ P\}\ C\ \{Q\}}$$

$$(\textsc{Conj})\frac{\begin{array}{c}\forall \mathcal{L}\ .\ \Gamma(\mathcal{L})\ \text{precise} \\ \Gamma \vdash \{P_1\}\ C\ \{Q_1\} \\ \Gamma \vdash \{P_2\}\ C\ \{Q_2\}\end{array}}{\Gamma \vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}} \qquad (\textsc{Disj})\frac{\begin{array}{c}\Gamma \vdash \{P_1\}\ C\ \{Q_1\} \\ \Gamma \vdash \{P_2\}\ C\ \{Q_2\}\end{array}}{\Gamma \vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

Fig. 5. Inference rules inherited from concurrent separation logic. Here, ρ is a permission; $\mathbf{Mod}(C)$ denotes all variables modified by command C, i.e., those that appear on the left-hand side of assignments. Moreover, $\mathbf{FV}(E)$, $\mathbf{FV}(C)$, and $\mathbf{FV}(\Gamma)$ denote the free variables of in expression E, command C (i.e., all variables accessed by C), and lock environment Γ (i.e., the union of $\mathbf{FV}(\Gamma(\mathcal{L}))$ for all $\mathcal{L} \in \mathbf{Locks}$), respectively. Furthermore, we define $\mathbf{FV}(A, B, C) \triangleq \mathbf{FV}(A) \cup \mathbf{FV}(B) \cup \mathbf{FV}(C)$. Finally, an assertion P is *precise* iff for all $s, \mathfrak{h}_1, \mathfrak{h}_2, \mathfrak{h}_1', \mathfrak{h}_2'$, if $def(\mathfrak{h}_1 \oplus \mathfrak{h}_2)$ and $\mathfrak{h}_1 \oplus \mathfrak{h}_2 = \mathfrak{h}_1' \oplus \mathfrak{h}_2'$ and $s, \mathfrak{h}_1 \models P$ and and $s, \mathfrak{h}_2 \models P$, then $\mathfrak{h}_1 = \mathfrak{h}_2$.

4 Methodology and Formalization

We now present the details of our methodology and formalize it on top of the concurrent separation logic (CSL) introduced in Sec. 3. We first discuss how abstract models are encoded into program states. After that, we extend the programming language, its operational semantics, and the CSL proof system with refinement-specific commands and rules. Finally, we show that our methodology is sound, i.e., a proof in our program logic guarantees that the (finite) traces of the given implementation are included in the traces of the abstract model. Further details and proofs are provided in the appendix.

4.1 Abstract Models

Our methodology takes as input an abstract model that should be refined by a concrete implementation. More precisely, we assume that an abstract model is provided as a (potentially infinite-state) abstract transition system.

Toward a formal definition, recall from Sec. 3.4 the definition of heap-independent assertions in first-order logic (**FOL**). Moreover, given a sequence of variables $\vec{x} = (x_1, \ldots, x_k)$, we denote by $\vec{x}'$ the same sequence in which each x_i is replaced by a primed version, i.e., $\vec{x}' = (x'_1, \ldots, x'_k)$.

Definition 1 (Abstract Transition System (ATS)). *An abstract transition system is a quadruple* $ATS \triangleq (\mathbb{k}, \vec{x}, Init, Next)$*, where*

- $\vec{x} = (x_1, \ldots, x_{\mathbb{k}})$ *is a repetition-free sequence of* $\mathbb{k} \geq 1$ *variables,*
- *the* initial state formula $Init(\vec{x})$ *is an* **FOL** *assertion over* $\vec{x}$*, and*
- *the* next state formula $Next(\vec{x}, \vec{x}')$ *is an* **FOL** *assertion over* $\vec{x}$ *and* $\vec{x}'$*.* △

We consider only ATSs that are *stutter-invariant*, that is, for all sequences of values $\vec{v} \in \mathbf{Vals}^{\mathbb{k}}$, the formula $Next(\vec{v}, \vec{v})$ is valid. Stutter invariance is desirable in the context of refinement proofs as concrete implementations should always be allowed to perform more fine-grained computation steps. Furthermore, every ATS can be turned into a stutter-invariant one by considering the modified next-state formula $Next(\vec{x}, \vec{x}') \ \vee \ \bigwedge_{1 \leq i \leq \mathbb{k}} x'_i = x_i$.

Throughout the rest of this paper, we fix a stutter-invariant abstract transition system $ATS \triangleq (\mathbb{k}, \vec{x}, Init, Next)$ representing the abstract model we would like to prove refinement of.

To reason about refinement, we need to formalize the observable traces of ATS. Every evaluation of an ATS's variables $\vec{x}$ constitutes one of its *states*. Formally, the *state space* of ATS is $\Sigma_{ATS} \triangleq \mathbf{Vals}^{\mathbb{k}}$. A (finite) *path* of ATS is a sequence of states $\sigma_1 \ldots \sigma_n$, where $n \geq 1$, such that

1. $\models Init(\sigma_1)$, i.e., σ_1 is an initial state of ATS, and
2. for all $i \in [1, n)$, we have $\models Next(\sigma_i, \sigma_{i+1})$, i.e., for every state but the last one, ATS admits a transition to the next state on the path.

We collect in the set $\mathbf{Paths}(ATS)$ all finite paths of the transition system ATS.

A *trace* then projects every state on a path to the values corresponding to those variables that are observable, e.g., variables modeling I/O channels as outlined in Sec. 2. For simplicity, we assume that exactly one variable, x_1, is observable; generalizing to arbitrary sets of observable variables is straightforward. We denote by $obs(\sigma)$ the projection of state σ to the value assigned to the observable variable x_1, that is, if $\sigma = (v_1, \ldots, v_{\mathbb{k}})$, then $obs(\sigma) = v_1$.

Definition 2. *The set* $\mathbf{Traces}(ATS)$ *of* observable traces *of* ATS *is given by*

$$\mathbf{Traces}(ATS) \triangleq \{obs(\sigma_1) \ldots obs(\sigma_n) \mid n \in \mathbb{N} \text{ and } \sigma_1 \ldots \sigma_n \in \mathbf{Paths}(ATS)\}. \qquad \triangle$$

State Encoding. To reason about the behavior of ATSs with the machinery offered by concurrent separation logic (cf. Sec. 3), we encode states of ATSs within program states—more precisely: as part of the heap.

We fix a set $\mathbf{GhostAddrs} \subseteq \mathbf{Addrs}$ of dedicated *ghost addresses* that cannot be allocated or disposed of—they thus need to be already allocated before program execution. Every variable x_i of ATS is then encoded as a ghost address $\lfloor x_i \rfloor \in \mathbf{GhostAddrs}$ at which we store the current evaluation of x_i. We use `stdOut` as a synonym for $\lfloor x_1 \rfloor$—the address of the only observable variable x_1.

Given a heap $\mathfrak{h}$, the contents of the addresses $\lfloor x_1 \rfloor, \ldots, \lfloor x_{\mathbb{k}} \rfloor$ in $\mathfrak{h}$ encode the current state of ATS. We introduce a function get_state: $\mathbf{Heaps} \to \Sigma_{ATS}$ which extracts the transition system's state from the heap:

Definition 3. *The* state extraction function get_state: $\mathbf{Heaps} \to \Sigma_{ATS}$ *is*

$$get\_state(\mathfrak{h}) \triangleq \begin{cases} (v_1, \ldots, v_{\mathbb{k}}) & \text{if } \lfloor x_1 \rfloor, \ldots, \lfloor x_{\mathbb{k}} \rfloor \in dom(\mathfrak{h}) \text{ and} \\ & \quad \textit{for all } i \in [1, \mathbb{k}],\ \mathfrak{h}(\lfloor x_i \rfloor) = (\rho_i, v_i) \\ & \qquad \text{and } \rho_i > 0 \\ \text{undefined} & \textit{otherwise}\ . \end{cases} \qquad \triangle$$

While ghost addresses cannot be allocated or disposed, (ghost) commands can read and modify their contents just like with ordinary addresses. Hence, we can enrich a concrete implementation with ghost commands to model updates to the state of the abstract model ATS.

Ghost Locks. We employ a dedicated lock $\mathcal{G}$ to protect the ghost locations $\lfloor x_1 \rfloor, \ldots, \lfloor x_{\mathbb{k}} \rfloor$ representing the current state of the abstract model ATS from undesirable modifications. To this end, we require that any lock invariant G chosen for $\mathcal{G}$ contains *some* permission to each of the locations $\lfloor x_1 \rfloor, \ldots, \lfloor x_{\mathbb{k}} \rfloor$—thus preventing modification of their content without acquiring $\mathcal{G}$ first. Formally:

Assumption 1. For any lock invariant G associated with the ghost lock $\mathcal{G}$, there exists a permission $\rho > 0$ such that the following entailment is valid:

$$\models \quad \big(G \quad \Rightarrow \quad \lfloor x_1 \rfloor \overset{\rho}{\hookrightarrow} \_ * \ldots * \lfloor x_{\mathbb{k}} \rfloor \overset{\rho}{\hookrightarrow} \_\big) \qquad \triangle$$

$$\text{(Print)}\quad \texttt{print}(E), s, h \to \texttt{skip}, s, h\,[\texttt{stdOut} := h(\texttt{stdOut}) \mathbin{+\!\!+} E(s)] \text{ if } \texttt{stdOut} \in \mathit{dom}(h)$$
$$\text{(PrintA)}\quad \texttt{print}(E), s, h \to \texttt{abort} \text{ if } \texttt{stdOut} \notin \mathit{dom}(h)$$
$$\text{(Init)}\quad \texttt{Init}\ \{\, C \,\}, s, h \to \texttt{lock}\ \mathcal{G}\ \{\, C \,\}, s, h$$
$$\text{(Next)}\ \frac{\mathit{atomic}(C) \qquad C, s, h \to^* \texttt{skip}, s', h'}{\texttt{Next}\ \{\, C \,\}, s, h \to \texttt{skip}, s', h'}$$

Fig. 6. Extension of the rules in Fig. 4 that determine the operational semantics. Here, $\to^*$ denotes the reflexive and transitive closure of execution relation $\to$.

Our methodology prescribes that $\mathcal{G}$ is a *ghost lock* such that it can be safely erased at runtime and thus cannot block execution. To guarantee the above property, our formalization treats $\mathcal{G}$ differently from other locks: it is a dedicated lock that is invisible to programmers and thus can neither be declared nor locked by them. Instead, it will be governed by the refinement-specific ghost commands `Init` and `Next` in a way such that $\mathcal{G}$ is indeed a ghost lock.

4.2 Programming Language and Operational Semantics

We now extend the programming language from Sec. 3.1 with an output operation $\texttt{print}(E)$ and the refinement-specific commands $\texttt{Init}\ \{\, C \,\}$ and $\texttt{Next}\ \{\, C \,\}$. Formally, we expand the grammar in Sec. 3.1 as follows:

$$C \ ::= \ \texttt{print}(E) \mid \texttt{Init}\ \{\, C \,\} \mid \texttt{Next}\ \{\, C \,\} \mid \ \ldots$$

We use ghost addresses taken from **GhostAddrs** to formalize the semantics of each of the above commands; Fig. 6 summarizes how we formally extend the execution relation $\to$ from Sec. 3.3. We briefly go over the new rules:

$\texttt{print}(E)$ appends the value of E to the standard output stream, which we represent by a mathematical sequence stored at the ghost address $\texttt{stdOut} \in$ **GhostAddrs**. Print thus modifies the only observable variable of the abstract mode *ATS*. If `stdOut` has not been allocated, we abort execution.

The command $\texttt{Init}\ \{\, C \,\}$ is, operationally speaking, a dedicated command for declaring the ghost lock $\mathcal{G}$. In fact, the rule (Init) desugars it to an explicit lock declaration $\texttt{lock}\ \mathcal{G}\ \{\, C \,\}$.[1] Conceptually, `Init` takes the important role of marking the system as initialized, i.e., after entering `Init`, the concrete model must be in an observable state that matches the (observable part of) the abstract model's initial state. The command $\texttt{Next}\ \{\ C\ \}$ is, operationally speaking again, a dedicated command for the conditional critical region that acquires the ghost lock $\mathcal{G}$, executes C, and releases $\mathcal{G}$, that is, it can be viewed as syntactic sugar for $\texttt{with}\ \mathcal{G}\ \texttt{when true}\ \{\, C \,\}$. However, to ensure that $\mathcal{G}$ is indeed a *ghost* lock and can thus be safely erased at runtime, we additionally require that every command C put into a $\texttt{Next}\ \{\, C \,\}$ block is *atomic*. That is, C can be executed in a single step without interference from other threads. Consequently, $\texttt{Next}\ \{\, C \,\}$ is atomic and

[1] While we do not allow programmers to explicitly use the ghost lock $\mathcal{G}$, the rules of our operational semantics can use it like any other lock.

thus executed in a single step, as reflected by the (NEXT) rule: if the execution of C terminates in a configuration c, then `Next` $\{\, C \,\}$ transitions to configuration c in a single step.

We assume that I/O operations, such as `print`(E), are atomic and that adding ghost code to an atomic command yields an atomic command. Apart from these assumptions, our methodology does not prescribe how to ensure atomicity—be it through known atomic statements, a global locking strategy, or formal proofs of atomicity in a program logic (cf. Sec. 2).

Observable Traces. To formalize program refinement, we need to define the observable traces induced by a program configuration. Similar to the definition of traces of the abstract model ATS, we project every program configuration in an execution $c_1 \dots c_n$ to its observable state once the system has been initialized, which is recorded by the operational semantics in its command structure:

Definition 4. *A command C is* initialized*, written $init(C)$, if and only if the ghost lock $\mathcal{G}$ is declared in C, i.e., $\mathcal{G} \in \mathbf{Locks}(C)$.* △

In our formalization, only the content of the standard output stream, which is modified via `print`(E) calls, is observable. The *observable state* $obs(c)$ of a configuration c is defined as the empty or singleton sequence

$$obs(c) \;\triangleq\; \begin{cases} [h(\texttt{stdOut})] & \text{if } c = (C, s, h) \text{ and } init(C) \\ [\,] & \text{otherwise.} \end{cases}$$

A *trace* is then obtained from a (finite) execution by mapping configurations to their observable state and concatenating the resulting sequences:

Definition 5. *The set* **Traces(I)** *of finite* traces *induced by a set* $\mathbf{I} \subseteq \mathbf{Confs}$ *is*

$$\mathbf{Traces(I)} \;\triangleq\; \{\, obs(c_1) \dots obs(c_n) \mid n \in \mathbb{N} \text{ and } c_1 \in \mathbf{I}, c_2, \dots, c_n \in \mathbf{Confs} \text{ and } c_1 \to \dots \to c_n \,\}\,. \qquad \triangle$$

The definition of $obs()$ ignores the observable state as long as the system is not initialized, i.e., we have not reached `Init` $\{\dots\}$ yet. This is justified as long as observable state is never modified before initialization and, thus, the implementation performs no observable action that needs to be matched by the abstract model. Our proof system will guarantee this property.

4.3 Proof System

Recall from Sec. 3.5 the proof system of concurrent separation logic (CSL). We now extend CSL to a sound program logic for proving refinement in the extended programming language introduced earlier in this section. Our assertion language is the same as for CSL, i.e., it consists of all **SL** assertions (see Sec. 3.4).

$$(\textsc{Init})\ \frac{\Gamma, \mathcal{G}\colon G, \mathcal{I}\colon \texttt{emp} \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}}{\Gamma \vdash_{\mathcal{R}} \{(\exists \vec{y}\ .\ \lfloor \vec{x} \rfloor(\vec{y}) \wedge \mathit{Init}(\vec{y}) \wedge G) * P\}\ \texttt{Init}\ \{C\}\ \{G * Q\}}$$

$$(\textsc{Next})\ \frac{\mathit{atomic}(C) \quad \Gamma \vdash_{\mathcal{R}} \{(\lfloor \vec{x} \rfloor(\vec{o}) \wedge G) * P\}\ C\ \{(\exists \vec{y}\ .\ \lfloor \vec{x} \rfloor(\vec{y}) \wedge \mathit{Next}(\vec{o}, \vec{y}) \wedge G) * Q)\}}{\Gamma, \mathcal{G}\colon G \vdash_{\mathcal{R}} \{P\}\ \texttt{Next}\ \{C\}\ \{Q\}}$$

$$(\textsc{Print})\ \frac{}{\Gamma, \mathcal{I}\colon \texttt{emp} \vdash_{\mathcal{R}} \left\{\texttt{stdOut} \overset{1}{\mapsto} E'\right\}\ \texttt{print}(E)\ \left\{\texttt{stdOut} \overset{1}{\mapsto} (E' \texttt{++} E)\right\}}$$

Fig. 7. Proof rules for the new commands. Here, $\vec{o} = (o_1, \ldots, o_{\mathbb{k}})$ are fresh variables. Moreover, for some $\rho > 0$, $\lfloor \vec{x} \rfloor(\vec{y}) \triangleq \lfloor x_1 \rfloor \overset{\rho}{\hookrightarrow} y_1 * \ldots * \lfloor x_{\mathbb{k}} \rfloor \overset{\rho}{\hookrightarrow} y_{\mathbb{k}}$.

Judgments. Judgments that can be derived in our logic (notice the $\vdash_{\mathcal{R}}$ to distinguish our judgments from CSL judgments) are of the form

$$\Gamma \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}\ ,$$

where Γ is a lock environment such that $\bigstar_{\mathcal{L} \in \mathbf{Locks}} \Gamma(\mathcal{L})$ describes the shared state, $P \in \mathbf{SL}$ is the precondition evaluated in the local state, C is a command and $Q \in \mathbf{SL}$ is the postcondition evaluated in the local state.

Fig. 7 shows the proof rules for the new commands `Init`, `Next`, and `print`(E). Furthermore, our proof system inherits all proof rules from CSL (cf. Fig. 5 where we tacitly replace $\vdash$ by $\vdash_{\mathcal{R}}$ and exclude dedicated ghost locks in the rules (Lock) and (With)). We briefly go over the rules for the new commands:

Initialization Blocks. The rule (Init) can be viewed as a stronger version of the rule (Lock), which declares *two* locks at once—the ghost lock $\mathcal{G}$ with lock invariant G and a second dedicated ghost lock $\mathcal{I}$ with an empty lock invariant.

The ghost lock $\mathcal{G}$ has already been used in our operational semantics; it governs access to the state of the abstract model ATS. The lock $\mathcal{I}$ is never acquired but serves as a marker in our proof system to record that an initialization block has been reached; we will explain why we need it in more detail further below.

(Init) is a stronger version of (Lock); in addition to declaring a lock, it verifies that the abstract model's state, i.e., the contents $y_1, \ldots, y_{\mathbb{k}}$ of the ghost addresses $\lfloor x_1 \rfloor, \ldots, \lfloor x_{\mathbb{k}} \rfloor$, is a valid initial state in ATS, i.e., $\mathit{Init}(y_1, \ldots, y_{\mathbb{k}})$ holds. By Assumption 1, we know that the ghost lock's invariant G holds the necessary permissions to access the contents of these ghost addresses.

Regarding the second ghost lock $\mathcal{I}$, notice that the output stream, that is, the observable content of `stdOut`, needs to be part of the program state before initialization, since the ghost address `stdOut` cannot be allocated by the implementation itself. In Def. 5, we formalized the traces of an implementation in a way such that the content of `stdOut` is ignored before initialization. The underlying rationale is that the implementation performs no observable action before it is initialized and we know the state of the abstract model ATS. To guarantee that there are indeed no modifications of observable state—in our case: there are no `print`(E) calls—before initialization, we use the ghost lock $\mathcal{I}$. It acts as a

token that is created upon initialization, stays in the lock environment, since it is never acquired (in contrast to the ghost lock $\mathcal{G}$), and needs to be part of the lock environment Γ for all rules that modify observable state, such as (PRINT).

Next Blocks. The rule (NEXT) can be viewed as a specialized version of the rule (WITH), applied to ghost lock $\mathcal{G}$ and wait condition `true`. However, it contains an additional premise to ensure that executing `Next { C }` can be simulated by taking a *single* transition of ATS. Since C is atomic, the postcondition $Next(\vec{o}, \vec{y})$ in the premise achieves this effect (where $\vec{o}$ and $\vec{y}$ capture the state of ATS before and after execution of `Next { C }`). For initialized commands, Assumption 1 guarantees that changes to the transition system's state are possible only when holding the lock $\mathcal{G}$, i.e., inside of `Next { C }` blocks; outside of such blocks, ATS will perform a stutter step for every step of the concrete implementation.

Print Statements. Finally, to understand the (PRINT) rule, we notice that `print`(E) is—if we ignore atomicity—syntactic sugar for the command

$$C \triangleq x \texttt{ := } [\texttt{stdOut}] \texttt{ ; } [\texttt{stdOut}] \texttt{ := } x \texttt{ ++ } E,$$

where x is a fresh variable that is not used anywhere else. Assuming that `print`(E) and C are identical, the rule (PRINT) can then be derived using the standard rules in Fig. 5 (in sequential separation logic, i.e., for $\Gamma = \emptyset$).

4.4 Soundness

Intuitively, soundness means that deriving $\Gamma \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}$ implies that, for every execution of C that starts in a configuration with a local state given by P and a shared state given by Γ, the trace of the execution is also a trace of the abstract model ATS. We now formalize and prove the above soundness claim for our refinement logic. We build upon Vafeiadis' [35] existing soundness proof for CSL (cf. Sec. 3). Hence, we first restate the main ingredients for proving CSL sound, where we slightly generalize to simplify their re-use.

Let $\mathsf{succ}(c, c')$ be a predicate over *two* program configurations; we will use $\mathsf{succ}(c, c')$ only if there is a transition $c \to c'$. Soundness of CSL as well as our refinement logic is based on the following notion of configuration safety:

Definition 6 (Generalized Configuration Safety). *The n-step safety predicate $\mathsf{safe}_n[\![\,\mathsf{succ}\,]\!]\,(C, s, \mathfrak{h}, \Gamma, Q)$ is recursively defined as follows:*

- $\mathsf{safe}_0[\![\,\mathsf{succ}\,]\!]\,(C, s, \mathfrak{h}, \Gamma, Q)$ *holds always.*
- $\mathsf{safe}_{n+1}[\![\,\mathsf{succ}\,]\!]\,(C, s, \mathfrak{h}, \Gamma, Q)$ *holds if and only if*
  1. $\mathbf{accesses}(C, s) \subseteq dom(\mathfrak{h})$*;*
  2. *for all* $\mathfrak{h}_\mathrm{F}$*, if* $def(\mathfrak{h} \oplus \mathfrak{h}_\mathrm{F})$*, then* $(C, s, \mathfrak{h} \oplus h) \not\to \texttt{abort}$*;*
  3. *if* $C = \texttt{skip}$*, then* $s, h \models Q$*;*
  4. *for all* C'*,* s'*,* $\mathfrak{h}_\mathrm{F}$*,* $\mathfrak{h}_\mathrm{S}$*,* $\mathfrak{h}'$*, if* $s, \mathfrak{h}_\mathrm{S} \models \bigstar_{\mathcal{L} \in \mathbf{locked}(C') \setminus \mathbf{locked}(C)} \Gamma(\mathcal{L})$ *and* $(C, s, \mathfrak{h} \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \to (C', s', \mathfrak{h}')$*, then there exist* $\mathfrak{h}''$ *and* $\mathfrak{h}'_\mathrm{S}$ *such that (a)* $\mathfrak{h}' = \mathfrak{h}'' \oplus \mathfrak{h}'_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}$*;*

(b) $s', \mathfrak{h}'_{\mathrm{S}} \models \bigstar_{\mathcal{L} \in \mathbf{locked}(C) \setminus \mathbf{locked}(C')} \Gamma(\mathcal{L})$*;*
(c) $\mathsf{succ}(C, s, \mathfrak{h} \oplus \mathfrak{h}_{\mathrm{S}} \oplus \mathfrak{h}_{\mathrm{F}}, C', s', \mathfrak{h}')$ *holds; and*
(d) $\mathsf{safe}_n[\![\, \mathsf{succ} \,]\!] (C', s', \mathfrak{h}'', \Gamma, Q)$ *holds.* △

Intuitively, $\mathsf{safe}_n[\![\, \mathsf{succ} \,]\!] (C, s, \mathfrak{h}, \Gamma, Q)$ means that every configuration c with command C, local state $\mathfrak{h}$ and shared state $\mathfrak{h}_{\mathrm{S}}$ given by Γ, and, optionally, additional global state $\mathfrak{h}_{\mathrm{F}}$, is *safe* w.r.t. lock environment Γ and postcondition Q for $n \geq 0$ steps; that is, every configuration c' reached from c via one transition (1) has no data race, (2) does not abort execution, (3) satisfies postcondition Q if execution terminated, (4 a,b) respects the lock invariants in Γ, (4 c) if we reached c' via a transition $c'' \to c'$, then $\mathsf{succ}(c'', c')$ holds, and (4 d) is itself safe for $n-1$ steps.

Definition 7 (Valid CSL Triples). *We write* $\Gamma \models \{P\}\, C\, \{Q\}$ *(read: the triple* $\{P\}\, C\, \{Q\}$ *is* valid *given lock environment* Γ*) if and only if*

$$\forall n, s, \mathfrak{h}\colon \quad s, \mathfrak{h} \models P \quad \text{implies} \quad \mathsf{safe}_n[\![\, \mathtt{true} \,]\!] (C, s, h, \Gamma, Q) \ .$$

Vafeiadis [35] used the above notions of configuration safety and validity to prove that every CSL judgment that can be derived using the rules in Fig. 5 yields a valid triple; the original soundness proof for CSL is due to Brookes [4]. Formally:

Theorem 1 (Soundness of CSL). *If* $\Gamma \vdash \{P\}\ C\ \{Q\}$*, then* $\Gamma \models \{P\}\, C\, \{Q\}$*.*

Compared to CSL, our refinement logic introduces one rule for each of the new commands $\mathtt{Init}\ \{\, C \,\}$, $\mathtt{Next}\ \{\, C \,\}$, and $\mathtt{print}(E)$ (see Fig. 7). If we only consider configuration safety as in CSL, then, as discussed in Sec. 4.3, each of these rules is either a special case of the existing rules ((Print)) or a variant of an existing rule with an even stronger precondition ((Init), (Next)). Hence, the soundness of CSL carries over to our logic (recall that $\vdash_{\mathcal{R}}$ indicates refinement judgments):

Lemma 1. *If* $\Gamma \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}$*, then* $\Gamma \models \{P\}\, C\, \{Q\}$*.*

App. C provides further details. To formalize that the implementation refines the abstract model *ATS*, we extend the above definition of configuration safety such that, if the system is initialized, every concrete transition $c \to c'$ taken is simulated by a transition of *ATS* (or stutters). Moreover, if the system becomes initialized after a step, we require that the encoded state of *ATS* is a valid initial state.

Our formalization uses the state extraction function *get_state* from Def. 3 to refer to the abstract model's state encoded in the heap. We then consider the configuration safety predicate $\mathsf{safe}_n[\![\, \mathsf{refsucc} \,]\!] (C, s, \mathfrak{h}, \Gamma, Q)$, where the predicate $\mathsf{refsucc}((C, s, \mathfrak{h}), (C', s', \mathfrak{h}'))$ holds iff

(i) if $\mathit{init}(C)$ holds, then $\models \mathit{Next}(\mathit{get\_state}(\mathfrak{h}), \mathit{get\_state}(\mathfrak{h}'))$;
(ii) if $\mathit{init}(C)$ does not hold and $\mathit{init}(C')$ holds, then $\models \mathit{Init}(\mathit{get\_state}(\mathfrak{h}'))$.

Definition 8 (Valid Refinement Triples). *We write* $\Gamma \models_{\mathcal{R}} \{P\}\, C\, \{Q\}$ *(read: the triple* $\{P\}\, C\, \{Q\}$ *is* valid for refinement of *ATS given* Γ*) if and only if*

$$\forall n, s, \mathfrak{h}\colon \quad s, \mathfrak{h} \models P \quad \text{implies} \quad \mathit{safe}_n[\![\, \mathit{refsucc} \,]\!]\,(C, s, h, \Gamma, Q)\ .$$

As for CSL, our refinement logic is sound in the sense that formally derived judgments yield valid triples:

Theorem 2. *If* $\Gamma \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}$*, then* $\Gamma \models_{\mathcal{R}} \{P\}\, C\, \{Q\}$*.*

Proof (Sketch). The proof is by structural induction over the rules of our refinement logic (found in Fig. 5 and Fig. 7). In each case, we show that the predicate $\mathsf{safe}_{n+1}[\![\, \mathsf{refsucc} \,]\!]\,(C, s, h, \Gamma, Q)$ holds; we first invoke Lemma 1 such that only two proof obligations remain for every possible step

$$c = (C, s, \mathfrak{h} \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \to (C', s', \mathfrak{h}'' \oplus \mathfrak{h}'_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) = c' \qquad (\dagger)$$

considered in Def. 6.(4): (a) $\mathsf{refsucc}(c, c')$ and (b) $\mathsf{safe}_n[\![\, \mathsf{refsucc} \,]\!]\,(C', s', h'', \Gamma, Q)$. There are three main cases: First, for axioms and (Next), we have $C' = \mathtt{skip}$; then (a) is immediate, since $\neg \mathit{init}(C)$ and $\neg \mathit{init}(C')$ hold; (b) follows from Lemma 2 below. Second, for the (Init) rule, (a) is immediate by the rule's premise; (b) follows from Lemma 3 below. Third, and finally, for all other rules, (a) and (b) follow directly from the induction hypothesis. Further details are found in App. D. □

Lemma 2. *For all* n, s, h, Γ, Q*, if* $s, h \models Q$*, then* $\mathit{safe}_n[\![\, \mathit{refsucc} \,]\!]\,(\mathtt{skip}, s, \mathfrak{h}, \Gamma, Q)$*.*

Lemma 3. *If* C *contains no two different sub-commands* C_1 *and* C_2 *such that* $\mathbf{locked}(C_1) \cap \mathbf{locked}(C_2) \neq \emptyset$ *and* $\mathbf{FV}(G) \cap \mathbf{Mod}(C) = \emptyset$*, then*

$$\Gamma, \mathcal{G}\colon G \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\} \quad \text{implies} \quad \Gamma \models_{\mathcal{R}} \{P * G\}\, \mathtt{lock}\ \mathcal{G}\ \{\, C \,\}\, \{Q * G\}\,.$$

We will use validity in the proof of trace inclusion. Furthermore, we need to define the initial configurations that determine the traces of the concrete implementation. We consider those traces of the concrete implementation that start in a (global) state that at least covers the local state specified by the precondition P and the shared state described by the lock invariants in Γ.

Definition 9 (Initial Configurations). *For a command* C*, an assertion* P*, and a lock environment* Γ*, the set* $\mathbf{I}(C, P, \Gamma)$ *of* initial configurations *is*

$$\mathbf{I}(C, P, \Gamma) \ \triangleq\ \{(C, s, \mathfrak{h} \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \mid \mathit{def}(\mathfrak{h} \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \text{ and } s, \mathfrak{h} \models P \text{ and } s, \mathfrak{h}_\mathrm{S} \models \bigstar_{\mathcal{L} \in \mathbf{Locks} \setminus \mathbf{locked}(C)} \Gamma(\mathcal{L})\}\ .$$

To guarantee trace inclusion, we exclude commands that suddenly become "uninitialized" because the $\mathtt{Init}\ \{\, C \,\}$ block expires but execution continues, e.g., $\mathtt{Init}\ \{\, C' \,\}\ ;\ \lfloor x_2 \rfloor\ \mathtt{:=}\ 17\ ;\ \mathtt{Init}\{\, C'' \,\}$. Such ill-formed commands can easily be detected by a syntactic check. More precisely, we call a command C *continuously initialized* if there exist commands C_1, C_2 such that C_1 contains no sub-command of the form $\mathtt{Init}\ \{\ldots\}$ and $C = (C_1\ ;\ \mathtt{Init}\ \{\, C_2 \,\})$.

Table 2. Case studies used for evaluation. **Data ref.** indicates whether there is interesting data refinement between the abstract model and the implementation. **Threads** indicates the number of nodes or worker threads in the implementation, where a $*$ means the threads are spawned dynamically after the model was already initialized. **Sync.** indicates the kind of synchronization primitive used, if any. **Idiom** indicates the reasoning used on top of our methodology. **SLOC** indicates standard lines of code including annotations. **Time** indicates verification time in seconds, measured as an average of the wall-clock runtime over 10 runs using Viper's symbolic execution verification backend on an Intel Core i9-10885H 2.40GHz CPU with 16 GiB of RAM.

Example	Data ref.	Threads	Sync.	Idiom	SLOC	Time
`alternating`	—	2	Lock	Owicki-Gries	118	3.78
`barrier`	—	N	Barrier	Guards	383	6.88
`cons_producer`	—	2	Lock	Guards	213	4.15
`echo_server`	—	1	—	—	69	3.39
`ring_leader`	—	N	—	—	279	7.74
`trees_product`	✓	$N*$	—	—	192	3.48
`trees_record`	✓	$N*$	—	Rely-guarantee	268	4.45

Theorem 3 (Soundness). *For every continuously-initialized command C,*

$$\Gamma \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\} \quad \text{implies} \quad \mathbf{Traces}(\mathbf{I}(C, P, \Gamma)) \subseteq \mathbf{Traces}(ATS)\ .$$

A detailed proof is found in App. E.

5 Evaluation

To evaluate our verification technique, we verified seven case studies using the separation logic-based automated verifier Viper [23]. All of our examples verify using Viper's existing verification backends, demonstrating that our methodology is supported by readily available verification tools. We will make our case studies available as an artifact. The examples show the flexibility of our proofs in the four dimensions mentioned in Sec. 1 and are summarized in Tab. 2. We briefly describe each of the examples in the following.

`alternating` demonstrates how multiple threads can collaborate to achieve the overall behavior of an abstract system. This example is described in Sec. 2.

`barrier` is an adaptation of the barrier example from Armada [22], originally from Cohen and Schirmer [7]. In this example, multiple threads are started and must pass a barrier before exiting. A guarded transition system is used to ensure that each thread can only perform transitions related to its own state.

`cons_producer` is an example with a consumer-producer setup, in which one thread adds values to a shared buffer and the other consumes them. We show that at all points an invariant is maintained, namely the relationship between the number of values produced, the number of values consumed, and the number of values left in the shared buffer. `echo_server` demonstrates the use of I/O methods to model both standard input and standard output.

`ring_leader` implements leader election in a ring, with a TLA$^+$specification adapted from the TLA$^+$examples repository, based on Chang and Roberts [6].

`trees_product` and `trees_record` are both examples demonstrating data refinement: the abstract model uses a mathematical datatype to represent a tree of values, whereas the implementation uses a heap-allocated array representation of the tree. The `trees_record` example further demonstrates dynamic threading to process the input tree in parallel, and a two-state monotonicity invariant.

Discussion. Our case studies demonstrate that our methodology is highly flexible w.r.t. to the structure of both programs and proofs. In particular, they use local and shared mutable state (including dynamic heap data structures), concurrency with dynamic thread creation and synchronization via locks and barriers, as well as three different reasoning idioms.

The evaluation also shows that our methodology enables automating refinement proofs using an off-the-shelf verification tool. While we used Viper as a concrete tool, no example relies on features that are genuinely Viper-specific, which supports the claim that our approach is flexible w.r.t. to the underlying logic. The verification time for each example is below 8s, which demonstrates that our methodology is well-suited for SMT-based automation.

6 Related Work

In this section, we survey refinement techniques that combine abstract models and executable code.

Various approaches [20,28,32,36] develop implementations that are correct by construction by refining abstract models within Coq and then extracting executable OCaml programs. Similarly, Liu et al. [21] model distributed systems in Maude's rewriting logic and compile them into implementations running in distributed Maude sessions. The code extracted by these approaches is typically sub-optimal (for instance, does not use mutable data structures) and cannot interface with existing libraries, which is often necessary in practice. In contrast, our methodology uses bottom-up verification and can handle efficient implementations using concurrency and mutable state.

Trillium [34] is a refinement technique based on separation logic. Like our methodology, it supports a wide range of program and data structures. Trillium is based on Iris [13] and formalized in Coq, which enables foundational proofs at the expense of substantial manual effort. Trillium inherits some of Iris's limitations. In particular, it is limited to finitary behaviors and, thus, does not support the common case that abstract models choose a value non-deterministically from an infinite set. Moreover, Trillium expresses coupling invariants via Iris's invariants, which complicates reasoning about system initialization, especially allocation. Our methodology does not have these limitations. Trillium supports liveness properties, which we do not handle yet.

Armada [22] supports the verification of concurrent, high-performance code written in a C-like language. To achieve refinement against an abstract model,

the user specifies a sequence of steps to gradually transform the implementation into the specification. Non-trivial refinement steps require complex Dafny [18] proofs showing a connection between two state machines. Unlike Armada, our methodology does not convert programs to state machines and the coupling between the abstract model and the implementation can be much looser.

The CIVL verifier [11,16] also organizes the refinement proof into multiple layers. Each layer is a *structured concurrent program*, where the concurrent behavior is reflected in the program structure. This structure simplifies the proof obligations and allows automation, but also reduces program flexibility. Refinement steps are based on a set of trusted tactics. By contrast, our methodology imposes no restrictions on the program or proof structure.

Igloo [33] connects abstract models to concrete implementations via dedicated I/O specifications [27]. Similarly to our work, they support a variety of separation logics to reason about concrete implementations. However, their technique does not allow threads to perform I/O operations concurrently, whereas our methodology has no such limitation.

Similar to our methodology, IronFleet [10] embeds abstract models as ghost state into executable programs and automates verification using an SMT-based verifier, in their case Dafny. However, their refinement technique imposes severe restrictions on the executable program. It must be sequential and its structure must mirror the structure of the abstract model. IronFleet supports both safety and liveness properties, whereas our approach focuses on safety properties and leaves liveness as future work.

The refinement technique [15] used in DeepSpec [2] is based on the Verified Software Toolchain (VST) [5], a framework for verifying C programs via a separation logic embedded in Coq. Instead of transition systems, they specify the intended system behavior using interaction trees [37], which are embedded into VST's separation logic. In contrast, our methodology allows us to apply standard separation logics and existing program verifiers.

Oortwijn and Huisman [25] embed process calculus models into a concurrent SL, which is automated using Viper. Their refinement approach preserves state assertions, but it is unclear whether arbitrary trace properties are preserved.

7 Conclusion

We have introduced a methodology for refinement proofs in separation logic that is flexible in terms of the type of abstract model used, the structure of the concrete implementation, the underlying logic and tool chain, as well as the structure of the proofs themselves. We have formalized the methodology on top of concurrent separation logic and demonstrated its applicability on several case studies using the automated verifier Viper. As future work, we plan to extend our methodology to liveness properties.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Appel, A.W., Beringer, L., Chlipala, A., Pierce, B.C., Shao, Z., Weirich, S., Zdancewic, S.: Position paper: the science of deep specification. Philosophical Transactions of the Royal Society A **375** (Oct 2017)
3. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) Static Analysis (SAS). pp. 55–72 (2003)
4. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1-3), 227–270 (2007)
5. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reasoning **61**(1-4), 367–422 (2018)
6. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Communications of the ACM **22**(5), 281–283 (1979)
7. Cohen, E., Schirmer, B.: From total store order to sequential consistency: A practical reduction theorem. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 403–418. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Coq Development Team, T.: The Coq Reference Manual, version 8.10 (2019), available electronically at `http://coq.inria.fr/documentation`
9. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 6183, pp. 504–528. Springer (2010)
10. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Symposium on Operating Systems Principles (SOSP). pp. 1–17. ACM (2015)
11. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: International Conference on Computer Aided Verification. pp. 449–465. Springer (2015)
12. Jones, C.B.: Developing methods for computer programs including a notion of interference. Ph.D. thesis, University of Oxford, UK (1981), `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064`
13. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming (2018)
14. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Symposium on Operating Systems Principles (SOSP). pp. 207–220. ACM (2009)
15. Koh, N., Li, Y., Li, Y., Xia, L., Beringer, L., Honoré, W., Mansky, W., Pierce, B.C., Zdancewic, S.: From C to interaction trees: specifying, verifying, and testing a networked server. In: Mahboubi, A., Myreen, M.O. (eds.) Certified Programs and Proofs (CPP). pp. 234–248. ACM (2019)
16. Kragl, B., Qadeer, S., Henzinger, T.A.: Refinement for structured concurrent programs. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 12224, pp. 275–298. Springer (2020)

17. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
18. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
19. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Principles of Programming Languages (POPL). pp. 42–54. ACM (2006)
20. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: Bodík, R., Majumdar, R. (eds.) Principles of Programming Languages (POPL). pp. 357–370. ACM (2016)
21. Liu, S., Sandur, A., Meseguer, J., Ölveczky, P.C., Wang, Q.: Generating correct-by-construction distributed implementations from formal maude designs. In: Lee, R., Jha, S., Mavridou, A. (eds.) NASA Formal Methods. Lecture Notes in Computer Science, vol. 12229, pp. 22–40. Springer (2020)
22. Lorch, J.R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J.R., Zhao, X.: Armada: low-effort verification of high-performance concurrent programs. In: Donaldson, A.F., Torlak, E. (eds.) Programming Language Design and Implementation (PLDI). pp. 197–210. ACM (2020)
23. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer (2016)
24. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) Concurrency Theory (CONCUR). Lecture Notes in Computer Science, vol. 3170, pp. 49–67. Springer (2004)
25. Oortwijn, W., Huisman, M.: Practical abstractions for automated verification of message passing concurrency. In: Ahrendt, W., Tarifa, S.L.T. (eds.) Integrated Formal Methods (iFM). Lecture Notes in Computer Science, vol. 11918, pp. 399–417. Springer (2019)
26. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Commun. ACM **19**(5), 279–285 (1976). https://doi.org/10.1145/360051.360224, https://doi.org/10.1145/360051.360224
27. Penninckx, W., Jacobs, B., Piessens, F.: Sound, modular and compositional verification of the input/output behavior of programs. In: Vitek, J. (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 9032, pp. 158–182. Springer (2015)
28. Rahli, V., Vukotic, I., Völp, M., Veríssimo, P.J.E.: Velisarios: Byzantine fault-tolerant protocols powered by coq. In: Ahmed, A. (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 10801, pp. 619–650. Springer (2018)
29. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. pp. 55–74 (2002)
30. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 8586, pp. 207–231. Springer (2014)
31. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Grove, D., Blackburn, S.M. (eds.) Programming Language Design and Implementation (PLDI). pp. 77–87. ACM (2015)

32. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. PACMPL **2**(POPL), 28:1–28:30 (2018)
33. Sprenger, C., Klenze, T., Eilers, M., Wolf, F.A., Müller, P., Clochard, M., Basin, D.: Igloo: Soundly linking compositional refinement and separation logic for distributed system verification. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). vol. 4. ACM (2020)
34. Timany, A., Gregersen, S.O., Stefanesco, L., Gondelman, L., Nieto, A., Birkedal, L.: Trillium: Unifying refinement and higher-order distributed separation logic. CoRR **abs/2109.07863** (2021)
35. Vafeiadis, V.: Concurrent separation logic and operational semantics. In: MFPS. Electronic Notes in Theoretical Computer Science, vol. 276, pp. 335–351. Elsevier (2011)
36. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the Raft consensus protocol. In: Avigad, J., Chlipala, A. (eds.) Certified Programs and Proofs (CPP). pp. 154–165 (2016)
37. Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. **4**(POPL), 51:1–51:32 (2020)

Table 3. Notational conventions and metavariables used throughout the paper.

Entities	Metavariables	Domain	Defined
Commands	C	**Cmds**	Sec. 3.1
Variables	x, y, z, ...	**Vars**	Sec. 3.1
Expressions	E	**Stacks** $\to$ **Vals**	Sec. 3.3
Locks	$\mathcal{L}$, $\mathcal{R}$	**Locks**	Sec. 3.1
Values	v	**Vals**	Sec. 3.2
Stacks	s	**Stacks**	Sec. 3.2
Heaps	$\mathfrak{h}$	**Heaps**	Sec. 3.4
Normal heaps	h	**NormHeaps**	Sec. 3.2
Addresses	a	**Addrs**	Sec. 3.2
Configurations	c	**Confs**	Sec. 3.3
Assertions	$P, Q, R, \ldots$	**SL**	Sec. 3.4
Lock environments	Γ		Sec. 3.5
Ghost addresses	$\mathtt{stdOut}$, $\lfloor x_1 \rfloor, \ldots, \lfloor x_\mathbb{k} \rfloor$	**GhostAddrs**	Sec. 4.1
Ghost lock with invariant	$\mathcal{G} : G$		Sec. 4.1

A Quick Reference

As a quick reference, Tab. 3 summarizes the notational conventions that have been introduced in Sec. 3 (above the line) and Sec. 4 (below the line).

B Lemmas from CSL

We will use the following lemmas taken from [35]:

Lemma 4. *For all n, s, h, Γ, Q, if $s, h \models Q$, then $\mathit{safe}_n[\![\, \mathtt{true} \,]\!] (\mathtt{skip}, s, \mathfrak{h}, \Gamma, Q)$.*

Lemma 5. *If $\forall n\colon \mathit{safe}_n[\![\, \mathtt{true} \,]\!] (C, s, \mathfrak{h}, \Gamma, Q)$ and $\mathit{def}(\mathfrak{h} \oplus \mathfrak{h}_\mathrm{F})$ and*

$$C, s, \mathfrak{h} \oplus \mathfrak{h}_\mathrm{F} \to^* \mathtt{skip}, s', h',$$

then there exists $\mathfrak{h}''$ such that $h' = \mathfrak{h}'' \oplus \mathfrak{h}_\mathrm{F}$ and $s', \mathfrak{h}'' \models Q$.

C Proof of Lemma 1

Claim. If $\Gamma \vdash_\mathcal{R} \{P\}\ C\ \{Q\}$, then $\Gamma \models \{P\}\, C\, \{Q\}$.

Proof. By Def. 7, it suffices to show that

$$\begin{aligned} &\Gamma \vdash_\mathcal{R} \{P\}\ C\ \{Q\} \quad \text{implies} \\ &\quad \forall n, s, \mathfrak{h}\colon \qquad s, \mathfrak{h} \models P \quad \text{implies} \quad \mathsf{safe}_n[\![\, \mathtt{true} \,]\!] (C, s, h, \Gamma, Q)\ . \end{aligned} \tag{1}$$

Since $\mathsf{safe}_0[\![\,\mathtt{true}\,]\!]\,(C, s, \mathfrak{h}, \Gamma, Q)$ holds always by Def. 6, it suffices to prove that $\mathsf{safe}_{n+1}[\![\,\mathtt{true}\,]\!]\,(C, s, \mathfrak{h}, \Gamma, Q)$ holds for an arbitrary, but fixed $n \in \mathbb{N}$.

By structural induction over the rules for of our refinement logic for deriving judgments $\Gamma \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}$ (found in Fig. 5 and Fig. 7).

We present the case (Print) in detail further below. As discussed in Sec. 4.3, the rule (Init) is a variant of the CSL rule (Lock) with a stronger precondition, and the rule (Next) is a special case of the CSL rule (With); we thus omit detailed proofs. since all other rules are identical to the CSL proof rules; their proof is thus completely analogous to the proof of Thm. 1.

*The case (*Print*).* Recall the rule

$$(\textsc{Print})\ \frac{}{\Gamma, \mathcal{I}\colon \mathtt{emp} \vdash_{\mathcal{R}} \left\{\mathtt{stdOut} \overset{1}{\mapsto} E'\right\}\ \mathtt{print}(E)\ \left\{\mathtt{stdOut} \overset{1}{\mapsto} (E' \mathbin{+\!\!+} E)\right\}}$$

and assume

$$s, \mathfrak{h} \models \mathtt{stdOut} \overset{1}{\mapsto} E'. \tag{2}$$

Consequently,

$$\mathfrak{h} \;=\; \left\{\, \mathtt{stdOut} \overset{1}{\mapsto} E'(s) \,\right\}. \tag{3}$$

We discharge all items in Def. 6 to show that

$$\mathsf{safe}_{n+1}[\![\,\mathtt{true}\,]\!]\left(\mathtt{print}(E), s, \mathfrak{h}, \Gamma, \mathcal{I}\colon \mathtt{emp}, \mathtt{stdOut} \overset{1}{\mapsto} (E' \mathbin{+\!\!+} E)\right). \tag{4}$$

For Def. 6.1, consider the following:

$$\begin{aligned} & \mathbf{accesses}(\mathtt{print}(E), s) = \{\,\mathtt{stdOut}\,\} && (5)\\ =\ & \mathit{dom}(h) && (\text{by } (3)) \end{aligned}$$

Def. 6.2 is immediate by (3) and the rules of our operational semantics. Def. 6.3 is trivial. For Def. 6.4, assume C', s', $\mathfrak{h}_\mathrm{F}$, $\mathfrak{h}_\mathrm{S}$, $\mathfrak{h}'$ such that

$$s, \mathfrak{h}_\mathrm{S} \models \bigstar_{\mathcal{L} \in \mathbf{locked}(C') \setminus \mathbf{locked}(C)} \Gamma(\mathcal{L}) \tag{6}$$

and

$$(\mathtt{print}(E), s, \mathfrak{h} \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \to (C', s', \mathfrak{h}') \tag{7}$$

By our operational semantics and (3), there is only one transition as above. For this transition, we have

$$C' = \mathtt{skip}, \quad \text{and} \quad \mathfrak{h}_\mathrm{S} = \mathtt{emp}, \quad \text{and} \tag{8}$$

$$h' = \left\{\, \mathtt{stdOut} \overset{1}{\mapsto} E'(s) \mathbin{+\!\!+} E(s) \,\right\} \oplus \mathfrak{h}_\mathrm{F} \tag{9}$$

Hence, for $\mathfrak{h}'' = h'$ and $\mathfrak{h}'_\mathrm{S} = \mathtt{emp}$, Def. 6.(a)-(c) hold immediately. Def. 6.(d) holds by Lemma 4. □

D Missing lemmas for the proof of Thm. 2

D.1 Proof of Lemma 2

Claim. For all n, s, h, Γ, Q, if $s, h \models Q$, then $\mathsf{safe}_n[\![\,\mathsf{refsucc}\,]\!]\,(\mathtt{skip}, s, \mathfrak{h}, \Gamma, Q)$.

Proof. By definition, $\mathsf{safe}_0[\![\,\mathsf{refsucc}\,]\!]\,(\mathtt{skip}, s, \mathfrak{h}, \Gamma, Q)$ holds always. Assume

$$s, h \models Q. \tag{10}$$

By Lemma 4, we know that

$$\mathsf{safe}_{n+1}[\![\,\mathsf{true}\,]\!]\,(\mathtt{skip}, s, \mathfrak{h}, \Gamma, Q)\,. \tag{11}$$

Then $\mathsf{safe}_{n+1}[\![\,\mathsf{refsucc}\,]\!]\,(\mathtt{skip}, s, \mathfrak{h}, \Gamma, Q)$ holds as well, because Def. 6.(1-3) holds by (11). Furthermore, since our operational semantics does not admit any transition starting with command `skip`, Def. 6.4 holds vacuously. □

D.2 Proof of Lemma 3

Claim. If C contains no two different sub-commands C_1 and C_2 such that $\mathbf{locked}(C_1) \cap \mathbf{locked}(C_2) \neq \emptyset$ and $\mathbf{FV}(G) \cap \mathbf{Mod}(C) = \emptyset$,

$$\Gamma, \mathcal{G} \colon G \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\} \quad \text{implies} \quad \Gamma \models_{\mathcal{R}} \{P * G\}\,\mathtt{lock}\ \mathcal{G}\ \{\,C\,\}\,\{Q * G\}\,.$$

Proof (Sketch). By structural induction on the rules of our refinement logic we show that $\Gamma, \mathcal{G} \colon G \vdash_{\mathcal{R}} \{P\}\ C\ \{Q\}$ and $s, \mathfrak{h} * G \models P$ implies

$$\forall n \in \mathbb{N}\colon \quad \mathsf{safe}_n[\![\,\mathsf{refsucc}\,]\!]\,(\mathtt{lock}\ G\ \{\,C\,\}, s, h, \Gamma, Q * G)\ . \tag{12}$$

We present the case for the (Next) in detail as it is the only rule, where the predicate refsucc is not immediately discharged by either applying the induction hypothesis or performing a stutter step.

The remaining cases are very similar to Vafeiadis [35] soundness proof for CSL (and in particular require an additional complete induction on n for some cases, such as parallel composition).

The case (Next). Recall from Fig. 7 the rule

$$\text{(Next)}\ \frac{\Gamma \vdash_{\mathcal{R}} \{(\lfloor \vec{x} \rfloor(\vec{o}) \wedge G) * P\}\ C\ \left\{ \underbrace{\exists \vec{y}\ .\ \lfloor \vec{x} \rfloor(\vec{y}) \wedge \mathit{Next}(\vec{o}, \vec{y}) \wedge G) * Q)}_{R} \right\}^{\mathit{atomic}(C)}}{\Gamma, \mathcal{G} \colon G \vdash_{\mathcal{R}} \{P\}\ \mathtt{Next}\ \{\,C\,\}\ \{Q\}}$$

and assume that

$$s, \mathfrak{h} \models P * G. \tag{13}$$

We then need to show that, for all $n \in \mathbb{N}$,

$$\mathsf{safe}_n[\![\,\mathsf{refsucc}\,]\!]\,(\mathtt{lock}\ \mathcal{G}\ \{\,\mathtt{Next}\ \{\,C\,\}\,\}, s, h, \Gamma, Q * G)\,. \tag{14}$$

The case $n = 0$ always holds by Def. 6. For $n > 0$, we discharge Def. 6.(1)-(4):

– Def. 6.(1) is immediate, since $\mathbf{accesses}(\texttt{lock}\ \mathcal{G}\ \{\,\texttt{Next}\ \{\,C\,\}\,\}, s) = \emptyset$;
– Def. 6.(2) is immediate, by Lemma 1 and the rules of our operational semantics;
– Def. 6.(3) is trivial; and
– for Def. 6.(4) assume C', s', $\mathfrak{h}_\mathrm{F}$, $\mathfrak{h}_\mathrm{S}$, $\mathfrak{h}'$ such that

$$s, \mathfrak{h}_\mathrm{S} \models \bigstar_{\mathcal{L} \in \mathbf{locked}(C') \setminus \mathbf{locked}(C)} \Gamma(\mathcal{L}) \tag{15}$$

and

$$(C, s, \mathfrak{h} \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \to (C', s', \mathfrak{h}'). \tag{16}$$

Our operational semantics has exactly one rule that admits such a transition, namely the one with premise $C, s, h \oplus \mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F} \to^* \texttt{skip}, s', h'$. Now, by the (Next) rule's premise and Thm. 1, we have

$$\forall m \colon \mathsf{safe}_m[\![\,\texttt{true}\,]\!]\,(C, s, \mathfrak{h}, \Gamma, R)\,. \tag{17}$$

Hence, by Lemma 5, there exist $\mathfrak{h}''$ such that

$$h' \;=\; \mathfrak{h}'' \oplus (\mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F}) \text{ and} \tag{18}$$

$$s', \mathfrak{h}'' \models R. \tag{19}$$

Then Def. 6.(4a) holds for $\mathfrak{h}''$ as above and $\mathfrak{h}'_\mathrm{S} = h_\emptyset$; Def. 6.(4b) holds, since $\mathbf{locked}(\texttt{lock}\ \mathcal{G}\ \{\,\texttt{Next}\ \{\,C\,\}\,\}\texttt{lock}\ \mathcal{G}\ \{\,\texttt{Next}\ \{\,C\,\}\,\}) = \mathbf{locked}(\texttt{lock}\ \mathcal{G}\ \{\,\texttt{skip}\,\})$; Def. 6.(4c) follows from (13) and $s', (h'' \oplus (\mathfrak{h}_\mathrm{S} \oplus \mathfrak{h}_\mathrm{F})) \models R$, where R is the (Next) rule's postcondition; and Def. 6.(dc) holds by Lemma 2.

□

E Proof of Thm. 3

Claim. For every continuously initialized command C,

$$\Gamma \vdash_\mathcal{R} \{P\}\ C\ \{Q\} \quad \text{implies} \quad \mathbf{Traces}(\mathbf{I}(C, P, \Gamma)) \subseteq \mathbf{Traces}(ATS)\ .$$

Proof. By Thm. 2, we have

$$\Gamma \models_\mathcal{R} \{P\}\, C\, \{Q\} \tag{20}$$

Moreover, since $\mathit{init}(C)$ holds iff $\mathcal{G} \in \mathbf{Locks}C$ and there is no proof rule for (sub-)commands of the form $\texttt{lock}\ \mathcal{G}\ \ldots$, we have

$$\neg \mathit{init}(C) \tag{21}$$

Now, assume that

$$c_1 = (C_1, s_1, h_1) \in \mathbf{I}(C_1, P, \Gamma) \tag{22}$$

and, for some arbitrary, but fixed, $n \in \mathbb{N}$,

$$c_1 \to c_2 \to \ldots \to c_n\ . \tag{23}$$

By (20), (22), Def. 9, and Def. 8 we have

$$\mathsf{safe}_{n+2}[\![\,\mathsf{refsucc}\,]\!]\,(C, s, h, \Gamma, Q)\ . \tag{24}$$

By Def. 6.(2) and (22), we can safely assume that

$$\forall i \in \{\,1, \ldots, n\,\} : \quad c_i = (C_i, s_i, h_i)\ . \tag{25}$$

We distinguish two cases: First, assume there exists an $m' \in \{\,1, \ldots, n\,\}$ such that $\mathit{init}(C_{m'})$; let m be the minimal such index m'. By (21), we know that $m \geq 2$. Hence, by Lemma 6 below,

$$\mathit{get\_state}(h_m) \ldots \mathit{get\_state}(h_n) \in \mathbf{Paths}(\mathit{ATS}) \tag{26}$$

Now, if $C_n = \mathtt{skip}$, then we have:

$$\begin{aligned}
& \mathbf{Traces(I}(C_1, P, \Gamma)\mathbf{)} \\
\ni\ & \mathit{obs}(c_1) \ldots \mathit{obs}(c_n) \\
=\ & \underbrace{\mathit{obs}(c_1) \ldots \mathit{obs}(c_{m-1})}_{=\,[\,]}\, \mathit{obs}(c_m) \ldots \mathit{obs}(c_{n-1})\, \underbrace{\mathit{obs}(c_n)}_{=\,[\,]} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Def. of } \mathit{obs}(.),\ \neg \mathit{init}(\mathtt{skip})) \\
=\ & \mathit{obs}(c_m) \ldots \mathit{obs}(c_{n-1}) \qquad \text{(Def. of } \mathit{obs}(.);\ m \text{ is minimal by assumption)} \\
=\ & \mathit{get_state}(h_m) \ldots \mathit{get_state}(h_{n-1}) \qquad\qquad\qquad\qquad \text{(Def. of } \mathit{obs}(.)) \\
\in\ & \mathbf{Paths}(\mathit{ATS}). \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by (26))}
\end{aligned}$$

Conversely, if $C_n \neq \mathtt{skip}$, then we have:

$$\begin{aligned}
& \mathbf{Traces(I}(C_1, P, \Gamma)\mathbf{)} \\
\ni\ & \mathit{obs}(c_1) \ldots \mathit{obs}(c_n) \\
=\ & \underbrace{\mathit{obs}(c_1) \ldots \mathit{obs}(c_{m-1})}_{=\,[\,]}\, \mathit{obs}(c_m) \ldots \mathit{obs}(c_{n-1}) \mathit{obs}(c_n) \\
=\ & \mathit{obs}(c_m) \ldots \mathit{obs}(c_{n-1}) \mathit{obs}(c_n) \quad \text{(Def. of } \mathit{obs}(.);\ m \text{ is minimal by assumption)} \\
=\ & \mathit{get_state}(h_m) \ldots \mathit{get_state}(h_n) \qquad\qquad\qquad\qquad\qquad\qquad \text{(Def. of } \mathit{obs}(.)) \\
\in\ & \mathbf{Paths}(\mathit{ATS}). \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by (26))}
\end{aligned}$$

Second, assume there for all $m' \in \{\,1, \ldots, n\,\}$, we have $\neg \mathit{init}(C_{m'})$. Then, by Def. of $\mathit{obs}(.)$, we have

$$\mathbf{Traces(I}(C_1, P, \Gamma)\mathbf{)}\ \ni\ \mathit{obs}(c_1) \ldots \mathit{obs}(c_n)\ =\ [\,]\ \in\ \mathbf{Paths}(\mathit{ATS}). \tag{27}$$

Lemma 6. *For all $2 \leq m \leq n$ and all $(C_1, s_1, h_1), \ldots, (C_n, s_n, h_n)$, if*

(i) C_1 *is continuously initialized and*
(ii) $\models_{\mathcal{R}} \{P\}\, C_1\, \{Q\}$ *and*
(iii) $(C_1, s_1, h_1) \in \mathbf{I}(C, P, \Gamma)$ *and*
(iv) $C_1, s_1, h_1 \rightarrow \ \ldots\ \rightarrow C_n, s_n, h_n$ *and*
(v) $\mathit{init}(C_m)$ *and, for all $j < m$, not* $\mathit{init}(C_j)$,

then $\mathit{get\_state}(h_m) \ldots \mathit{get\_state}(h_n) \in \mathbf{Paths}(\mathit{ATS})$.

Proof. By induction on n.

Induction base. Assume *(i)-(v)* hold for $n = m = 2$. By *(iii)*, there exist heaps $\mathfrak{h}$, $\mathfrak{h}_S$ such that

$$h_1 \;=\; \mathfrak{h} \oplus \mathfrak{h}_S \tag{28}$$
$$s_1, \mathfrak{h} \;=\; P \tag{29}$$
$$s_1, \mathfrak{h}_S \models \bigstar_{\mathcal{L} \in \mathbf{Locks} \setminus \mathbf{locked}(C_1)} \Gamma(\mathcal{L}) \tag{30}$$
$$\tag{31}$$

Hence, by *(ii)*,

$$\mathsf{safe}_3[\![\, \mathsf{refsucc} \,]\!]\, (C_1, s_1, \mathfrak{h}, \Gamma, Q) \;\; . \tag{32}$$

Now, consider the transition

$$C_1, s_1, \mathfrak{h} \oplus \mathfrak{h}_S \oplus h_\emptyset \rightarrow \; \ldots \; \rightarrow C_2, s_2, h_2 \tag{33}$$

which exists by *(iv)*. By *(v)*, we have $\mathit{init}(C_1)$ does not hold and $\mathit{init}(C_2)$ holds. Hence, by Def. 6.(4c) and definition of $\mathsf{refsucc}$,

$$\models \mathit{Init}(\mathit{get\_state}(h_2)) \tag{34}$$

and thus also $\mathit{get\_state}(h_2) \in \mathbf{Paths}(\mathit{ATS})$.

Induction hypothesis. For an arbitrary, but fixed, $n \geq 2$, assume that *(i)-(v)* imply $\mathit{get\_state}(h_m) \ldots \mathit{get\_state}(h_n) \in \mathbf{Paths}(\mathit{ATS})$.

Induction step. Assume that *(i)-(v)* hold for some $2 \leq m \leq n+1$, where $n \geq 2$. The case $m = n+1$ is analogous to the induction base. Hence, assume $m < n+1$. By *(iii)*, there exist heaps $\mathfrak{h}$, $\mathfrak{h}_S$ such that

$$h_1 \;=\; \mathfrak{h} \oplus \mathfrak{h}_S \tag{35}$$
$$s_1, \mathfrak{h} \;=\; P \tag{36}$$
$$s_1, \mathfrak{h}_S \models \bigstar_{\mathcal{L} \in \mathbf{Locks} \setminus \mathbf{locked}(C_1)} \Gamma(\mathcal{L}) \tag{37}$$
$$\tag{38}$$

Hence, by *(ii)*,

$$\mathsf{safe}_{n+3}[\![\, \mathsf{refsucc} \,]\!]\, (C_1, s_1, \mathfrak{h}, \Gamma, Q) \;\; . \tag{39}$$

By repeated unfolding the above predicate, we know that there exist $\mathfrak{h}'$ and $\mathfrak{h}'_S$ such that

$$h_n \;=\; \mathfrak{h}' \oplus \mathfrak{h}'_S \tag{40}$$
$$\mathsf{safe}_2[\![\, \mathsf{refsucc} \,]\!]\, (C_n, s_n, \mathfrak{h}', \Gamma, Q) \tag{41}$$
$$s_1, \mathfrak{h}_S \models \bigstar_{\mathcal{L} \in \mathbf{Locks} \setminus \mathbf{locked}(C_n)} \Gamma(\mathcal{L}) \tag{42}$$
$$\tag{43}$$

Now, consider the transition below, which exists by *(iv)*.

$$C_n, s_n, \mathfrak{h}' \oplus \mathfrak{h}'_S \oplus h_\emptyset \to \ldots \to C_{n+1}, s_{n+1}, h_{n+1} \tag{44}$$

Since $m < n+1$, *(i)* and *(v)* yield that $init(C_n)$ holds. By Def. 6.(4c) and definition of refsucc, this means

$$\models Next(get\_state(\mathfrak{h}_n), get\_state(\mathfrak{h}_{n+1})) \tag{45}$$

By I.H., we also know that

$$get\_state(h_m) \ldots get\_state(h_n) \in \mathbf{Paths}(ATS) \tag{46}$$

Hence,

$$get\_state(h_m) \ldots get\_state(h_n) get\_state(h_{n+1}) \in \mathbf{Paths}(ATS), \tag{47}$$

which finishes the proof. □