A Rely-Guarantee-Based Simulation for Cooperative Semantics

Kevin Tran[1(✉)], Johannes Åman Pohjola[2,3], Rob Sison[1], and Gerwin Klein[1,4]

[1] UNSW Sydney, Sydney, Australia
{k.q.tran,r.sison}@unsw.edu.au
[2] Chalmers University of Technology, Gothenburg, Sweden
pohjola@chalmers.se, johannes.aman.pohjola@gu.se
[3] University of Gothenburg, Gothenburg, Sweden
[4] Proofcraft, Sydney, Australia
gerwin.klein@proofcraft.systems

Abstract. Compared to semantics with preemptively executing threads, ones with cooperative threads permit easier specification of atomicity in concurrent programs. We introduce a semantics of cooperative programs, and a simulation notion compatible with rely-guarantee proofs. We prove our simulation composes in parallel and sequentially, and that it can establish a standard trace-based notion of refinement.

Keywords: Concurrency · rely-guarantee reasoning · simulation

1 Introduction

Most semantics of concurrency suitable for reasoning about implementations use a *preemptive* semantics, where execution may alternate between threads after each step. This fine-grained concurrency is needed to model many efficiently executing programs, but makes reasoning about them more difficult. To limit concurrency, there might be constructs like atomic blocks which specify that the code inside must execute atomically. However, consider the example below on the left, a client of a lock-protected stack:

```
lock()
while ¬isEmpty(stack) do
    x ← pop(stack)
    unlock(); f(x); lock()
```

```
while ¬isEmpty(stack) do
    x ← pop(stack)
    yield
    f(x)
```

How do we specify the desired critical section, where checking the stack is empty and then popping it must be atomic, but the computation f(x) on the popped value need not be? This is awkward, because atomic blocks are

Z. Liu et al. (Eds.): ICTAC 2025, LNCS 16237, pp. 87–105, 2026.
https://doi.org/10.1007/978-3-032-11176-0_7

constrained by syntactic structure. Instead, consider a *cooperative* semantics, where a thread, once executed, keeps executing until it decides to yield. The desired granularity of atomicity is shown to the right.

The purpose of our cooperative semantics is not to be implementable for all programs, but to describe programs with arbitrary granularity of atomicity, from fine-grained efficient implementations with many yields to coarse-grained specifications with few yields. We are not advocating for (or against) cooperative multitasking as an implementation technique. Using cooperative semantics is a choice that can be made independently of whether the implementation is cooperative; a program that executes preemptively can be modelled in a cooperative semantics by yielding after every step.

Programs with coarse-grained atomicity are usually easier to reason about. For example, when verifying a program using Owicki-Gries [13] or rely-guarantee [7], one needs to consider interference at the yield points, of which there are fewer with coarse-grained atomicity. We want the properties of the specification we have proven to carry over to the fine-grained implementation, which requires a notion of *atomicity refinement* to justify. We use a standard *event trace refinement* based on set inclusion of (partial) traces of events.

We aim to prove atomicity refinement as part of the future verification of multicore configurations of the seL4 kernel [8]. There are two such configurations: one where most in-kernel execution is protected by one lock around the entire kernel, and a *multi-kernel* configuration where separate kernel instances run on separate cores and share no or almost no data structures. In both cases, the goal is to use atomicity refinement to reduce a large part of the concurrency verification to the existing sequential proofs about seL4 and only deal with concurrency in those parts where it matters. A cooperative semantics is well suited to specify such parts. However, event trace refinement is not compositional with respect to parallel composition. To scale, we therefore need a proof method for refinement that supports compositional reasoning.

To the best of our knowledge, this paper develops the first compositional technique for proving refinement between cooperatively executing concurrent programs. We prove soundness with respect to refinement (Theorem 2), transitivity (Sect. 5.1), and derive decomposition principles for parallel composition (Sect. 5.2) and sequential composition (Sect. 5.3). All these results are formalized in the proof assistant Isabelle/HOL [12].

Our technique is based on the rely-guarantee-based simulation of Liang et al. (RGSim) [11], a compositional proof technique for refinement in a preemptive semantics. Adapting it to cooperative semantics is non-trivial: the treatment of sequential composition is subtle, and requires decoupling the tracking of interference points from the tracking of the current state.

2 Syntax

Our language is based on Complx [2], a preemptively concurrent extension of Simpl [14], used in the seL4 verification to model the behavior of C programs [16].

We choose Complx because our aim in the future is to reason about the multicore configurations of seL4. The syntax is as follows:

$$\begin{aligned} op : &\quad \text{State} \to \text{State} \qquad b, g \subseteq \text{State} \\ f \in &\quad \text{Fault} \qquad\qquad\quad e : \text{State} \to \text{Event} \\ c, c' \in \text{Com} ::= &\ \textbf{skip} \mid \textbf{basic}\ op \mid c; c' \mid \textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c' \mid \textbf{while}\ b\ \textbf{do}\ c \\ &\mid \textbf{yield}\ g \mid \textbf{assert}\ f\ b \mid \textbf{print}\ e \end{aligned}$$

The first line of the definition of Com (for *command*) consists of standard imperative programming constructs [18] (**basic** used to update state), while the second line has more unusual constructs. The syntax is parametrized on a set of states State, and does not fix any particular syntax for expressions. Instead, conditions and state updates are shallowly embedded as sets and functions on states, respectively. The command **assert** f b checks if the current state satisfies b, faults with f if it does not, and resumes execution otherwise. It is used to model undefined behavior in C.

These two commands are our additions, and are not present in Complx:

- **yield** g yields control, permitting other threads to execute. A thread that has yielded becomes blocked until the state satisfies the guard g. This permits various synchronization mechanisms like blocking locks to be defined outside the core language.
- **print** e emits an event based on the current state. It has no effect on the state, but will be important for defining refinement and simulation later.

Unlike Complx, there is no syntax for parallel composition. Instead, a concurrent program is represented by a thread pool which includes one command (element of Com) per thread. Thread pools will be discussed further in Sect. 3.

3 Semantics

Our semantics is a small-step reduction semantics inspired by Abadi and Plotkin's [1] cooperative semantics. The steps are between *configurations*, which consist of a thread pool, the thread id of the active thread (if any), and a status (either a normal state or a fault). A thread pool is a partial map from thread ids to a command and a guard. The guard controls when the thread can be activated. For thread ids, $\mathbb{N}$ is merely a convenient choice of a countable set with equality.

Definition 1 (Configuration). *A configuration* $cfg = (i, T, st)$ *consists of:*

- *an optional thread id* $i \in \text{dom}(T) \sqcup \{\text{None}\}$
- *a partial map* $T : \mathbb{N} \rightharpoonup \text{Com} \times \mathcal{P}(\text{State})$
- *and a status* $st \in \text{Status}$.

$$s \in \text{State} \qquad f \in \text{Fault} \qquad st \in \text{Status} ::= \text{N } s \mid \text{F } f$$

Given $T(i) = (c, g)$, we say $\text{com}(T(i)) = c$ and $\text{guard}(T(i)) = g$. In this paper, we write thread pools as sets of pairs i.e. $T = \{(i, (c, g)), \ldots\}$.

For brevity in later definitions, we define some types of configurations. Those where the status is a fault, such as by failing an assertion, we call *faulting*. Configurations with an active thread, we call *active* configurations. Configurations without an active thread, and for convenience, not faulting, we call *inactive* configurations.

Definition 2. *We say that a configuration* (i, T, st) *is*

1. faulting *if there exists* f *such that* $st = \text{F } f$*;*
2. active *if* $i \in \text{dom}(T)$*, where* $\text{com}(T(i))$ *is called the* active command*;*
3. inactive *if* $i = \text{None}$ *and there exists* s *such that* $st = \text{N } s$*; and*
4. terminated *if it is inactive and, for all* $i \in \text{dom}(T)$*, it holds that* $\text{com}(T(i)) =$ ***skip***.

The semantics uses *evaluation contexts*, inductively defined by the grammar

$$\mathcal{C} \in \text{ECtxt} ::= [\,] \mid \mathcal{C}; c$$

As usual, $[\,]$ is a hole, and $\mathcal{C}[\cdot] : \text{Com} \to \text{Com}$ fills the hole in a context $\mathcal{C}$ with a command. In this case, $\mathcal{C}$ allows us to select the first command in a series of potentially nested (or empty) sequential compositions, which simplifies the formulation of our small-step rules.

We can now define the main step relation $\to$ of the semantics. See below for a selection of the rules. Here, we denote updating the function f at x to y by $f(x := y)(z)$, which is y if $z = x$ and $f(z)$ otherwise. The omitted rules for constructs such as if, while, etc. are standard. We focus here on the rules that are non-standard or important for cooperative semantics. If there is an active command, we perform a step by finding a redex and a corresponding evaluation context. Based on the redex, we can apply the appropriate rule. Exactly one redex exists, except in the case of **skip** where there are none. Otherwise, if there is no active command, the Activate rule lets us nondeterministically choose a thread whose guard holds, and whose command is not **skip**, to activate.

$$\text{Basic}\ \frac{i \in \mathbb{N} \qquad T(i) = (\mathcal{C}[\textbf{basic } op], g)}{(i, T, \text{N } s) \to (i, T(i := (\mathcal{C}[\textbf{skip}], g)), \text{N } op(s))}$$

$$\text{SeqSkip}\ \frac{i \in \mathbb{N} \qquad T(i) = (\mathcal{C}[\textbf{skip}; c], g)}{(i, T, \text{N } s) \to (i, T(i := (\mathcal{C}[c], g)), \text{N } s)}$$

$$\text{Yield}\ \frac{i \in \mathbb{N} \qquad T(i) = (\mathcal{C}[\textbf{yield}\ g], \_)}{(i, T, \mathrm{N}\ s) \to (\mathrm{None}, T(i := (\mathcal{C}[\textbf{skip}], g)), \mathrm{N}\ s)}$$

$$\text{AssertTrue}\ \frac{i \in \mathbb{N} \qquad T(i) = (\mathcal{C}[\textbf{assert}\ f\ b], g) \qquad s \in b}{(i, T, \mathrm{N}\ s) \to (i, T(i := (\mathcal{C}[\textbf{skip}], g)), \mathrm{N}\ s)}$$

$$\text{AssertFalse}\ \frac{i \in \mathbb{N} \qquad T(i) = (\mathcal{C}[\textbf{assert}\ f\ b], g) \qquad s \notin b}{(i, T, \mathrm{N}\ s) \to (i, T(i := (\mathcal{C}[\textbf{skip}], g)), \mathrm{F}\ f)}$$

$$\text{Print}\ \frac{i \in \mathbb{N} \qquad T(i) = (\mathcal{C}[\textbf{print}\ e], g)}{(i, T, \mathrm{N}\ s) \to (i, T(i := (\mathcal{C}[\textbf{skip}], g)), \mathrm{N}\ s)}$$

$$\text{Activate}\ \frac{i \in \mathbb{N} \qquad T(i) = (c, g) \qquad c \neq \textbf{skip} \qquad s \in g}{(\mathrm{None}, T, \mathrm{N}\ s) \to (i, T, \mathrm{N}\ s)}$$

Following Abadi and Plotkin, we call steps using the Activate rule *choice steps*, and write $\to_c$, and the other steps *active steps*, and write $\to_a$. As befits a semantics for cooperative execution, once a thread has been activated, it continues execution until it yields, faults, or reaches **skip**. Unlike Abadi and Plotkin, we do not automatically yield when the active command is **skip**. Instead, execution is suspended in what is called an *incomplete* configuration, a notion that will be important for sequential compositionality because in a cooperative semantics, not every sequential composition is a preemption point. For a thread to "properly" terminate, it must end with a **yield** rather than a **skip**.

Definition 3 (Incomplete configuration). *A configuration* (i, T, st) *is* incomplete *if* $i \in \mathbb{N}$, $\mathrm{com}(T(i)) = \textbf{\textit{skip}}$ *and there exists* s *such that* $st = \mathrm{N}\ s$. *Incomplete configurations are active: they have a thread that is still selected for execution, but that cannot make any further progress.*

Example 1. We now give a sample execution from our semantics, where State $= \mathbb{N}$. We abbreviate $\textbf{basic}(x \mapsto x + 2)$ to $x \leftarrow x + 2$. We name the rules used at each step and underline the redex when taking an active step.

$$\begin{aligned}
&(\mathrm{None},\ \{(1,\ (x \leftarrow x + 2; \textbf{yield}\ \top, \{0, 42\}))\},\ \mathrm{N}\ 0) \\
\to_c\ &(1,\ \{(1, (\underline{x \leftarrow x + 2}; \textbf{yield}\ \top, \{0, 42\}))\},\ \mathrm{N}\ 0) && \text{(Activate)} \\
\to_a\ &(1,\ \{(1, (\underline{\textbf{skip}; \textbf{yield}\ \top}, \{0, 42\}))\},\ \mathrm{N}\ 2) && \text{(Basic)} \\
\to_a\ &(1,\ \{(1, (\underline{\textbf{yield}\ \top}, \{0, 42\}))\},\ \mathrm{N}\ 2) && \text{(SeqSkip)} \\
\to_a\ &(\mathrm{None},\ \{(1, (\textbf{skip}, \top))\},\ \mathrm{N}\ 2) && \text{(Yield)}
\end{aligned}$$

We start in an inactive but not terminated configuration, and the last configuration is terminated. Since the status is N 0 in the initial configuration and the guard contains 0, the thread can be chosen for execution in the first Activate step. The next two steps, Basic and SeqSkip, are thread-internal, and the last step, Yield, returns execution to the thread pool, which contains no other active threads. Execution therefore terminates.

Example 2. Now suppose the thread did not end in a yield. We instead have:

$$\begin{aligned} & (\text{None},\ \{(1,\ (x \leftarrow x+2, \{0,42\}))\},\ \text{N } 0) \\ \rightarrow_c\ & (1,\ \{(1, (\underline{x \leftarrow x+2}, \{0,42\}))\},\ \text{N } 0) && \text{(Activate)} \\ \rightarrow_a\ & (1,\ \{(1, (\textbf{skip}, \{0,42\}))\},\ \text{N } 2) && \text{(Basic)} \end{aligned}$$

The last configuration is not terminated, but incomplete.

4 Event Trace Refinement

This section defines our notion of event trace refinement. To define our traces of events, we will not use the small-step relation directly, but introduce an analog of a preemptive step, which we call a *fragment*.

4.1 Fragments

With cooperative execution, other threads cannot run while a thread is running, so intermediate states between yield points are inaccessible from the outside. Intermediate states can be made observable via **print**, but they cannot be affected by other threads. That means execution between yield points is sequential.

We can therefore coalesce the fine-grained small-step execution of the program into a more coarse-grained sequence of *fragments* that model the state transition and events output between yield points, or an initially active configuration and a yield point.

It is also here that **print** comes into play: when executing a fragment, we track the sequence of events $e(s)$ emitted by each **print** e command at state s.

Definition 4. *If we can execute a sequence of zero or more active steps from one configuration cfg to another cfg′, emitting the event sequence es, we write*

$$cfg \xrightarrow{es}{}^{*}_{a}\ cfg'.$$

For compositionality later, it will be important to track not just the current state, but also the state at the most recent yield point. We therefore extend configurations as follows:

Definition 5 (Extended configuration). *An* extended configuration *is a pair* $xcfg = (cfg, s)$ *of a configuration* $cfg = (i, T, st)$ *and the state* $s \in \text{State}$ *at the last yield command. When* cfg *is inactive, we require the (normal) status and the last yield state to coincide:* $st = \text{N}\ s$.

Now we can define fragments.

Definition 6 (Fragments). *Given extended configurations* (cfg, s) *and* (cfg', s'), *let* $(cfg, s) \overset{es}{\Longrightarrow} (cfg', s')$ *denote a* fragment *from* (cfg, s) *to* (cfg', s') *emitting the sequence of events* es. *We define it as the conjunction of:*

1. cfg *is not faulting or incomplete*
2. cfg' *is inactive, faulting, or incomplete*
3. *if* cfg *is inactive, then there exists* cfg'' *s.t.* $cfg \rightarrow_c cfg''$ *and* $cfg'' \overset{es}{\rightarrow}_a^* cfg'$
4. *if* cfg *is active, then* $cfg \overset{es}{\rightarrow}_a^* cfg'$
5. *if* cfg' *is active (including faulting or incomplete), then* $s' = s$.

Definition 7. *If we can execute a sequence of zero or more fragments from an extended configuration* $xcfg$ *to another configuration* $xcfg'$, *emitting the sequence of events* es, *we write*

$$xcfg \overset{es}{\Longrightarrow}^* xcfg'$$

In the previous two definitions, if es is omitted, it is taken to mean the empty list, that is, no events being emitted.

We now give an example of a fragment, with State = Event = $\mathbb{N}$. We use **print** x to emit the current state as an event. Steps with something above $\rightarrow$ emit an event, the rest emit no event. Consider the following small-step sequence:

$$\begin{aligned}
&(\text{None}, \{(2, (\textbf{print}\ x; \textbf{yield}\ \top, \top))\}, \text{N}\ 0) \\
\rightarrow_c\ &(2, \{(2, (\underline{\textbf{print}\ x}; \textbf{yield}\ \top, \top))\}, \text{N}\ 0) && \text{(Activate)} \\
\overset{0}{\rightarrow}_a\ &(2, \{(2, (\underline{\textbf{skip}; \textbf{yield}\ \top}, \top))\}, \text{N}\ 0) && \text{(Print)} \\
\rightarrow_a\ &(2, \{(2, (\underline{\textbf{yield}\ \top}, \top))\}, \text{N}\ 0) && \text{(SeqSkip)} \\
\rightarrow_a\ &(\text{None}, \{(2, (\textbf{skip}, \top))\}, \text{N}\ 0) && \text{(Yield)}
\end{aligned}$$

Example 3. The sequence forms a single fragment that we can write as follows:

$$\begin{aligned}
&((\text{None}, \{(2, (\textbf{print}\ x; \textbf{yield}\ \top, \top))\}, \text{N}\ 0), 0) \\
\overset{[0]}{\Longrightarrow}\ &((\text{None}, \{(2, (\textbf{skip}, \top))\}, \text{N}\ 0), 0)
\end{aligned}$$

We can also conclude that

$$\begin{aligned} &((2, \{(2, (\textbf{print}\ x; \textbf{yield}\ \top, \top))\}, \mathrm{N}\ 0), 0) \\ \xRightarrow{[0]}\ &((\mathrm{None}, \{(2, (\textbf{skip}, \top))\}, \mathrm{N}\ 0), 0) \end{aligned}$$

Example 4. Here is an example execution of several fragments:

$$\begin{aligned} &((\mathrm{None}, \{(1, (x \leftarrow x + 1; \textbf{yield}\ \top; x \leftarrow x + 1; \textbf{yield}\ \top, \top)) \\ &\qquad , (2, (\textbf{print}\ x; \textbf{yield}\ \top, \top))\}, \mathrm{N}\ 0), 0) \\ \Rightarrow\ &((\mathrm{None}, \{(1, (\textbf{skip}; x \leftarrow x + 1; \textbf{yield}\ \top, \top)) \\ &\qquad , (2, (\textbf{print}\ x; \textbf{yield}\ \top, \top))\}, \mathrm{N}\ 1), 1) \\ \xRightarrow{[1]}\ &((\mathrm{None}, \{(1, (\textbf{skip}; x \leftarrow x + 1; \textbf{yield}\ \top, \top)) \\ &\qquad , (2, (\textbf{skip}, \top))\}, \mathrm{N}\ 1), 1) \end{aligned}$$

4.2 Refinement

Given the notion of fragments from the previous section, we can now define event trace refinement. When executing fragments, a trace of events is generated. This trace could be partial or end in a terminated, faulting or incomplete configuration. We denote these results P, T, F, I, respectively.

We define the partial traces PT from an extended configuration $xcfg$ inductively by the following rules. Here we use @ for list concatenation and [] for the empty list. Note that, as the name suggests, the trace only records events, not states.

$$\frac{xcfg = ((i, T, \mathrm{N}\ s), s')}{([], P) \in \mathrm{PT}(xcfg)} \qquad \frac{xcfg = ((i, T, \mathrm{F}\ f), s')}{([], F) \in \mathrm{PT}(xcfg)}$$

$$\frac{xcfg = (cfg, s) \qquad cfg \text{ is terminated}}{([], T) \in \mathrm{PT}(xcfg)} \qquad \frac{xcfg = (cfg, s) \qquad cfg \text{ is incomplete}}{([], I) \in \mathrm{PT}(xcfg)}$$

$$\frac{xcfg \xRightarrow{es} xcfg' \qquad (tr, r) \in \mathrm{PT}(xcfg')}{(es@tr, r) \in \mathrm{PT}(xcfg)}$$

Using the first of the above rules and the execution from Example 1:

$$\{([], P), ([], T)\} \subseteq \mathrm{PT}(((\mathrm{None}, \{(1, (x \leftarrow x + 2; \textbf{yield}\ \top, \{0, 42\}))\}, \mathrm{N}\ 0), 0)) \tag{1}$$

The program does not emit any events, so both traces record the empty list. For an example where PT is instead bound from above, we take a slightly different program that increments x twice, and yields in between. By induction on PT,

$$\mathrm{PT}(((\mathrm{None}, \{(1, (x \leftarrow x + 1; \mathbf{yield}\ \top; x \leftarrow x + 1; \mathbf{yield}\ \top, \top))\}, \mathrm{N}\ 0), 0)) \subseteq \{([], P), ([], T)\} \quad (2)$$

We can now define event trace refinement as follows:

Definition 8 (Event trace refinement). *Let $xcfg$, $xcfg'$ be extended configurations. We say that $xcfg \sqsubseteq xcfg'$ ($xcfg$* refines *$xcfg'$) if* $\mathrm{PT}(xcfg) \subseteq \mathrm{PT}(xcfg')$.

Refinement states that after executing some number of fragments from the concrete extended configuration, we can match the event trace and result by executing some number of fragments from the abstract extended configuration. The number of fragments may differ. This admits atomicity refinement where we decrease or increase the number of yield points between abstract and concrete levels, which our work aims to enable. Note that the statement of refinement here is for configurations $xcfg$ and $xcfg'$, not for programs. That is, the definition is for specific initial states of the thread pool, not over all of them.

Example 5 (Refinement). Since we showed in Eq. 1 and 2 that the traces of one program are above $\{([], P), ([], T)\}$ and the other below, we have

$$\begin{aligned} &((\mathrm{None}, \{(1, (x \leftarrow x + 1; \mathbf{yield}\ \top; x \leftarrow x + 1; \mathbf{yield}\ \top, \{0, 42\}))\}, \mathrm{N}\ 0), 0) \\ \sqsubseteq\ &((\mathrm{None}, \{(1, (x \leftarrow x + 2; \mathbf{yield}\ \top, \top))\}, \mathrm{N}\ 0), 0) \end{aligned}$$

If the programs emitted their state as events before the last yield, refinement would still hold, but only for configurations starting in the same state. Refinement would no longer hold if these threads were composed with another set of abstract and concrete threads that also modify x, even if refinement were to hold separately for these threads.

Example 6 (Non-refinement). Let

$$\begin{aligned} xcfg_c = ((\mathrm{None}, \{&(1, (x \leftarrow x + 1; \mathbf{yield}\ \top; x \leftarrow x + 1; \mathbf{yield}\ \top, \{0, 42\})) \\ &, (2, (\mathbf{print}\ x; \mathbf{yield}\ \top, \top))\}, \mathrm{N}\ 0), 0) \\ xcfg_a = ((\mathrm{None}, \{&(1, (x \leftarrow x + 2; \mathbf{yield}\ \top, \top)) \\ &, (2, (\mathbf{print}\ x; \mathbf{yield}\ \top, \top))\}, \mathrm{N}\ 0), 0) \end{aligned}$$

From Example 4, we have $([1], P) \in \mathrm{PT}(xcfg_c)$. By induction on PT, we have $\mathrm{PT}(xcfg_a) \subseteq \{([], P), ([0], P), ([0], T), ([2], P), ([2], T)\}$. Hence $xcfg_c \not\sqsubseteq xcfg_a$.

Together with Example 5 and Example 3 (noting that the fragment ends in a terminated configuration), this shows that event trace refinement is not compositional with respect to parallel composition (to be defined in Sect. 5.2).

5 Simulation

As just mentioned, although event trace refinement is an intuitive notion of behavioral preservation, it does not compose with respect to parallel composition. Since compositionality is indispensable for scalable reasoning, we follow the usual path of defining a compositional simulation instead that can be used for reasoning. To this end, we adapt the rely-guarantee-based simulation RGSim [11] to a cooperative semantics. We prove it implies our event trace refinement and is compositional with respect to parallel (Sect. 5.2) and sequential composition (Sect. 5.3).

The key difference between cooperative and preemptive semantics is that in preemptive semantics, every execution step is a yield point and therefore observable as a step. In cooperative semantics, we need to distinguish between execution states in the middle of a fragment and execution states that have reached a yield point where interference from other threads is possible. For the simulation to stay compositional, this needs additions to both the internal definition of the simulation itself, and the parameters the simulation operates on.

Our simulation is between two extended configurations, one concrete and one abstract, with additional parameters we introduce below. Let CState and AState be the concrete and abstract state set, respectively. The extra parameters are:

- $R_c, G_c \subseteq \text{CState} \times \text{CState}$, the *concrete rely and guarantee relations*
- $R_a, G_a \subseteq \text{AState} \times \text{AState}$, the *abstract rely and guarantee relations*
- $\alpha \subseteq \text{CState} \times \text{AState}$, the *state relation*
- $Q \subseteq \text{CState} \times (\mathbb{N} \rightharpoonup \mathcal{P}(\text{CState})) \times \text{AState} \times (\mathbb{N} \rightharpoonup \mathcal{P}(\text{AState}))$, the *normal postcondition*, which is a predicate on the states and the guards of all threads.
- $Q_i \subseteq \text{CState} \times \text{CState} \times \text{AState} \times \text{AState}$, the *incomplete postcondition*, which is a predicate on the current states and the states at the last yield point.

The first three items are the same as in the original RGSim, but the normal postcondition now keeps track of the guards. The incomplete postcondition is a new addition needed for sequential compositionality in Sect. 5.3. As is customary with rely-guarantee reasoning, we assume the rely relations R_c, R_a are reflexive.

As with the original RGSim, we will need the following definition:

Definition 9 (α-related transitions). *We call $\langle R_c, R_a \rangle_\alpha$ the α-related transitions in R_c and R_a. They are the set of all tuples $(s_c, s_c', s_a, s_a') \in \text{CState} \times \text{CState} \times \text{AState} \times \text{AState}$ such that $(s_c, s_a) \in \alpha$, $(s_c, s_c') \in R_c$, $(s_a, s_a') \in R_a$ and $(s_c', s_a') \in \alpha$.*

Let $xcfg_c = (cfg_c, s_c)$ be the concrete extended configuration, where $cfg_c = (i_c, T_c, st_c)$, and similarly for the abstract extended configuration $xcfg_a$. We now define the simulation between $xcfg_c$ and $xcfg_a$ coinductively.

Definition 10. *If $R_c, G_c, \alpha, R_a, G_a \vdash xcfg_c \preceq xcfg_a$ Q, Q_i then all of the following must hold:*

1. $(s_c, s_a) \in \alpha$

2. *if* $i_c = \text{None}$ *then* $i_a = \text{None}$
3. *neither* cfg_c *nor* cfg_a *are faulting or incomplete*
4. *if* cfg_c *is terminated, then there exists an extended configuration* $xcfg'_a$ *such that* $xcfg_a \Rightarrow^* xcfg'_a$, cfg'_a *is terminated,* $(s_c, s_c, s_a, s'_a) \in \langle G_c, G_a^* \rangle_\alpha$ *and* $(s_c, \text{guard} \circ T_c, s'_a, \text{guard} \circ T'_a) \in Q$, *where* $\circ$ *is function composition.*
5. *if* $xcfg_c \overset{es}{\Longrightarrow} xcfg'_c$ *and* cfg'_c *is inactive, then there exists an extended configuration* $xcfg'_a$ *such that* $xcfg_a \overset{es}{\Longrightarrow}^* xcfg'_a$, cfg'_a *is inactive,* $(s_c, s'_c, s_a, s'_a) \in \langle G_c, G_a^* \rangle_\alpha$ *and* $R_c, G_c, \alpha, R_a, G_a \vdash xcfg'_c \preceq xcfg'_a\ Q, Q_i$.
6. *if* $xcfg_c \overset{es}{\Longrightarrow} xcfg'_c$ *and* $st'_c = \text{F}\ f$, *then there exists an extended configuration* $xcfg'_a$ *such that* $xcfg_a \overset{es}{\Longrightarrow}^* xcfg'_a$ *and* $st'_a = \text{F}\ f$.
7. *if* $xcfg_c \overset{es}{\Longrightarrow} xcfg'_c$ *and* cfg'_c *is incomplete, then there exists an extended configuration* $xcfg'_a$ *such that* $xcfg_a \overset{es}{\Longrightarrow}^* xcfg'_a$, cfg'_a *is incomplete,* $(s_c, s_c, s_a, s'_a) \in \langle G_c, G_a^* \rangle_\alpha$ *and there exists* s''_c, s''_a *such that* $st'_c = \text{N}\ s''_c$, $st'_a = \text{N}\ s''_a$ *and* $((s_c, s''_c), (s'_a, s''_a)) \in Q_i$.
8. *if* $xcfg'_c$ *and* $xcfg'_a$ *are extended configurations such that* cfg'_c *and* cfg'_a *are inactive and* $(s_c, s'_c, s_a, s'_a) \in \langle R_c, R_a^* \rangle_\alpha$, *then* $R_c, G_c, \alpha, R_a, G_a \vdash xcfg'_c \preceq xcfg'_a\ Q, Q_i$.

Case 1 states that the last yield states of the concrete and abstract configurations must be related by the state relation. Case 2 states that if the concrete configuration is inactive, then the abstract one must be too. It is used to prevent situations where we are free to pick a thread on the concrete level, but on the abstract level, we are forced to execute a thread. For case 3, the simulation only includes inactive configurations and active, but not faulting or incomplete configurations. Faulting and incomplete configurations are dealt with by executing fragments from a configuration in the simulation. Case 4 is the "base case" of the simulation, when the concrete configuration is terminated. The abstract configuration is allowed to do some work before terminating. The final states and guards of all threads must satisfy the normal postcondition. Cases 5, 6, 7 require that when we execute a fragment on the concrete level, we must match it with zero or more fragments on the abstract level. The fragments must preserve the state relation and also obey the guarantee relations (if we end in an inactive configuration). Case 7 also requires that when we end up in an incomplete configuration, that the last yield states and current states obey the incomplete postcondition. Case 8 requires that the simulation be robust against interference from the environment, bounded by the rely relations and the state relation.

We can strengthen rely relations and weaken the guarantee relations and postconditions:

Theorem 1. *If* $R_c, G_c, \alpha, R_a, G_a \vdash xcfg_c \preceq xcfg_a\ Q, Q_i$ *and*

- $R'_c \subseteq R_c$ *and* $R'_a \subseteq R_a$ *(strengthening relies)*
- $G_c \subseteq G'_c$ *and* $G_a \subseteq G'_a$ *(weakening guarantees)*
- $Q \subseteq Q'$ *and* $Q_i \subseteq Q'_i$ *(weakening postconditions)*

then $R_c', G_c', \alpha, R_a', G_a' \vdash xcfg_c \preceq xcfg_a\ Q', Q_i'$.

We then prove the simulation sound with respect to the more intuitive trace refinement. First we prove a lemma, then obtain soundness as a corollary:

Lemma 1. *Let* Id *be the identity relation and* $\top$ *be the universal relation. If* $\mathrm{Id}, \top, \alpha, \mathrm{Id}, \top \vdash xcfg_c \preceq xcfg_a\ Q, Q_i$, *then* $\mathrm{PT}(xcfg_c) \subseteq \mathrm{PT}(xcfg_a)$.

Proof. By induction over PT.

Theorem 2 (Soundness). *Let* R_c, R_a *be reflexive rely relations.*
If $R_c, G_c, \alpha, R_a, G_a \vdash xcfg_c \preceq xcfg_a\ Q, Q_i$, *then* $xcfg_c \sqsubseteq xcfg_a$.

Proof. Using Theorem 1, we strengthen the relies to the Id relation and weaken the guarantees to the $\top$ relation. Unfold the definition of $\sqsubseteq$ and apply Lemma 1.

5.1 Transitivity

By focusing on the extended configurations, we can think of the simulation as a binary relation. Thus we might wonder whether or not our simulation is transitive, which would allow stepwise simulation proofs. Assuming that the rely relations and state relations are in some sense compatible, we can answer in the affirmative. The compatibility condition can informally be described as: given $\alpha \circ \beta$-related transitions, we can factor them into some α-related transitions and β-related transitions. More formally,

Definition 11. *Let* R_l, R_m, R_h *be rely relations on the low, middle and high levels. Let* α *be a state relation between the low and middle levels, and let* β *be a state relation between the middle and high levels. We say that* $\mathrm{compat}(R_l, \alpha, R_m, \beta, R_h)$, *if for all* $(s_l, s_l', s_h, s_h') \in \langle R_l, R_h \rangle_{\alpha \circ \beta}$ *and* s_m *s.t.* $(s_l, s_m) \in \alpha$ *and* $(s_m, s_h) \in \beta$, *there exists* s_m' *s.t.* $(s_l, s_l', s_m, s_m') \in \langle R_l, R_m \rangle_\alpha$ *and* $(s_m, s_m', s_h, s_h') \in \langle R_m, R_h \rangle_\beta$.

Theorem 3 (Transitivity). *Let* R_l, R_m, R_h *be rely relations on the low, middle and high levels. Let* α *be a state relation between the low and middle levels, and let* β *be a state relation between the middle and high levels. If*

- $R_l, G_l, \alpha, R_m, G_m \vdash xcfg_l \preceq xcfg_m\ Q, Q_i$
- $R_m, G_m, \beta, R_h, G_h \vdash xcfg_m \preceq xcfg_h\ Q', Q_i'$
- *and* $\mathrm{compat}(R_l, \alpha, R_m, \beta, R_h^*)$

then $R_l, G_l, \alpha \circ \beta, R_h, G_h \vdash xcfg_l \preceq xcfg_h\ Q \circ Q', Q_i \circ Q_i'$, *where* $\circ$ *means relational composition and* R_h^* *is the reflexive transitive closure.*

5.2 Parallel Composition

Although our language lacks a parallel composition operator, by taking the disjoint union of thread pools, we can obtain a somewhat restricted analog of it. This shallow embedding as a disjoint union allow us to inherit properties like associativity automatically. We can break down a simulation into simulations on each part of the disjoint union, abstracting away the behavior of the other part of the disjoint union using the rely and guarantee relations.

Given normal postconditions Q, Q' for the parts, what should the normal postcondition of the whole be? We need the states to agree, and the guards to be the disjoint union of the guards for the parts. Thus, we define $Q \sqcup Q'$, the set of tuples $(s_c, gs_c \cup gs'_c, s_a, gs_a \cup gs'_a)$ where $(s_c, gs_c, s_a, gs_a) \in Q$, $(s_c, gs'_c, s_a, gs'_a) \in Q'$, $\mathrm{dom}(gs_c) \cap \mathrm{dom}(gs'_c) = \emptyset$ and $\mathrm{dom}(gs_a) \cap \mathrm{dom}(gs'_a) = \emptyset$.

As with rely-guarantee reasoning, we need the normal postconditions to be stable under interference from the environment. Let $Q \subseteq \mathrm{CState} \times (\mathbb{N} \rightharpoonup \mathcal{P}(\mathrm{CState})) \times \mathrm{AState} \times (\mathbb{N} \rightharpoonup \mathcal{P}(\mathrm{AState}))$ be a normal postcondition and $\Lambda \subseteq \mathrm{CState} \times \mathrm{CState} \times \mathrm{AState} \times \mathrm{AState}$. We say $\mathrm{Sta}(Q, \Lambda)$, if for all $(s_c, gs_c, s_a, gs_a) \in Q$ and $(s_c, s'_c, s_a, s'_a) \in \Lambda$, we have $(s'_c, gs_c, s'_a, gs_a) \in Q$.

We now can state our parallel composition rule:

Theorem 4 (Parallel composition). *If*

1. $R_c, G_c, \alpha, R_a, G_a \vdash ((i_c, T_c, st_c), s_c) \preceq ((i_a, T_a, st_a), s_a)\ Q, Q_i$
2. $R'_c, G'_c, \alpha, R'_a, G'_a \vdash ((i'_c, T'_c, st'_c), s_c) \preceq ((i'_a, T'_a, st'_a), s_a)\ Q', Q_i$
3. $G_c \subseteq R'_c$, $G'_c \subseteq R_c$, $G_a \subseteq R'_a$ *and* $G'_a \subseteq R_a$
4. $\mathrm{dom}(T_c) \cap \mathrm{dom}(T_a) = \emptyset$ *and* $\mathrm{dom}(T'_c) \cap \mathrm{dom}(T'_a) = \emptyset$
5. $\mathrm{Sta}(Q, \langle R_c, R_a^* \rangle_\alpha)$ *and* $\mathrm{Sta}(Q', \langle R'_c, R_a'^* \rangle_\alpha)$
6. $(i''_c, st''_c, i''_a, st''_a, \mathrm{None}, \mathrm{None}) \in \{(i_c, st_c, i_a, st_a, i'_c, i'_a), (i'_c, st'_c, i'_a, st'_a, i_c, i_a)\}$

then

$$R_c \cap R'_c, G_c \cup G'_c, \alpha, R_a \cap R'_a, G_a \cup G'_a \vdash ((i''_c, T_c \cup T'_c, st''_c), s_c) \preceq ((i''_a, T_a \cup T'_a, st''_a), s_a)\ Q \sqcup Q', Q_i$$

Assumptions 1 and 2 are the simulations on the "parallel components". Assumption 3 states the rely and guarantee relations between the components are compatible. Assumption 4 states that the thread pools must have disjoint thread ids. Assumption 5 states that the normal postconditions are stable under interference. Assumption 6 is meant to formalize the idea of picking one half of the parallel composition to execute.

5.3 Sequential Composition

Even for concurrent programs, there is often a substantial amount of sequential reasoning to be done. In the preemptive case, every sequential composition is a preemption point, but with cooperative semantics this is not always the case. When executing a fragment with a sequential composition, we could either yield

or not before executing the second part. This leads us to distinguish between normal and incomplete execution of a fragment. To propagate information from the first part to the second part of a sequential composition, we use the distinction between normal and incomplete postconditions. The normal postcondition, in addition to states, tracks thread guards at the end of executing the first part so that the second part can reason about them. The incomplete postcondition of the first part allows us to take the first part of the execution into account for checking the guarantee and state relations when the second part encounters a yield instruction. We now state our sequential composition rule:

Theorem 5 (Sequential composition). *If*

1. $R_c, G_c, \alpha, R_a, G_a \vdash$
   $((i_c, \{(t_c, (c_c, g_c))\}, st_c), s_c) \preceq ((i_a, \{(t_a, (c_a, g_a))\}, st_a), s_a)\ Q, Q_i$
2. *For all* $(s_c', {gs_c}', s_a', {gs_a}') \in Q$ *such that* $(s_c', s_a') \in \alpha$,

$$R_c, G_c, \alpha, R_a, G_a \vdash ((\text{None}, \{(t_c, (c_c', {gs_c}'(t_c)))\}, \text{N}\ s_c'), s_c') \preceq ((\text{None}, \{(t_a, (c_a', {gs_a}'(t_a)))\}, \text{N}\ s_a'), s_a')\ Q', Q_i'$$

3. *For all* $(s_c', s_c'', s_a', s_a'') \in Q_i$ *and guards* g_c', g_a',

$$R_c, G_c, \alpha, R_a, G_a \vdash ((t_c, \{(t_c, (c_c', g_c'))\}, \text{N}\ s_c''), s_c') \preceq ((t_a, \{(t_a, (c_a', g_a'))\}, \text{N}\ s_a''), s_a')\ Q', Q_i'$$

4. *For all* $(s_c', {gs_c}', s_a', {gs_a}') \in Q$, *we have* $t_c \in \text{dom}({gs_c}')$ *and* $t_a \in \text{dom}({gs_a}')$, *and* $s_c' \in \text{guard}({gs_c}'(t_c))$ *and* $s_a' \in \text{guard}({gs_a}'(t_a))$
5. $c_c' \neq \textbf{skip}$ *and* $c_a' \neq \textbf{skip}$

then

$$R_c, G_c, \alpha, R_a, G_a \vdash ((i_c, \{(t_c, (c_c; c_c', g_c))\}, st_c), s_c) \preceq ((i_a, \{(t_a, (c_a; c_a', g_a))\}, st_a), s_a)\ Q', Q_i'$$

Assumption 1 is the simulation on the first part of the sequential composition. Assumption 2 is the simulation on the second part, assuming the first part terminated normally. Assumption 3 is the simulation on the second part of the sequential composition, assuming the first part terminated in an incomplete configuration. The guards are actually irrelevant since we deal with active configurations. Assumption 4 specifies that when the first part terminates normally, we are not in a state blocked by the guard.

5.4 Example

We borrow the following example from Liang et al. [11, Section 4.3]. We wish to establish a simulation between incrementing an abstract atomic counter $x \in \mathbb{N}$ by 2 and incrementing a concrete lock-protected counter by 1 twice. The concrete

state consists of a counter $x \in \mathbb{N}$, an optional thread id $i \in \mathbb{N} \sqcup \{\text{None}\}$ indicating the lock owner, and a ghost copy X of the abstract state. Then define for $i \in \mathbb{N}$:

$$\textbf{lock}\ i = (\textbf{if}\ \text{owner} = \text{None}\ \textbf{then skip else yield}\ \text{owner} = \text{None});\ \text{owner} \leftarrow i$$
$$\textbf{unlock}\ i = \textbf{assert}\ (\text{owner} = i);\ \text{owner} \leftarrow \text{None}$$

The commands on each level are:

$$\begin{aligned} c_c =& (\textbf{lock}\ w; \textbf{yield}\ \top); (x \leftarrow x + 1; \textbf{yield}\ \top); (x \leftarrow x + 1; \textbf{yield}\ \top); \\ & \textbf{unlock}\ w; \textbf{yield}\ \top \\ c_a =& x \leftarrow x + 2; \textbf{yield}\ \top \end{aligned}$$

We have rely and guarantee relations, parametrized by a thread id $i \in \mathbb{N}$.

$$\begin{aligned} R_c(i) &= \{(s_c, s_c') \mid \text{owner}(s_c) = i \implies s_c = s_c'\} \\ G_c(i) &= \{(s_c, s_c') \mid s_c' = s_c \vee ((\text{owner}(s_c) = \text{None} \implies \text{owner}(s_c') = i) \\ &\quad \wedge (\exists i' \in \mathbb{N}.\ \text{owner}(s_c) = i' \implies i' = i \wedge \text{owner}(s_c') \in \{i, \text{None}\}))\} \end{aligned}$$

The rely relation states that if a thread i holds the lock, then the environment is not allowed to change the state. The guarantee relation states that thread i can take, hold or release the lock, and cannot make any other state changes unless they have the lock.

The state relation says that X on the concrete level indeed copies the abstract x, and that when the lock is not held, the abstract and concrete x are equal:

$$\alpha = \{(s_c, s_a) \mid x(s_a) = X(s_c) \wedge (\text{owner}(s_c) = \text{None} \implies x(s_c) = X(s_c))\}$$

Our normal postcondition Q is just α on the states, and that the writer's guard is $\top$. The incomplete postcondition Q_i is not used for this example.

$$Q = \{(s_c, gs_c, s_a, gs_a) \mid (s_c, s_a) \in \alpha \wedge gs_c = \{(w, \top)\} \wedge gs_a = \{(w, \top)\}\}$$

Example 7. For any pair of states (s_c, s_a) in α, we have:

$$\begin{aligned} &R_c(w), G_c(w), \alpha, \top, \top \vdash \\ &\quad (\text{None}, \{(w, (c_c, \top))\}, \text{N}\ s_c) \preceq (\text{None}, \{(w, (c_a, \top))\}, \text{N}\ s_a)\ Q, Q_i \end{aligned}$$

Example 8. When we repeat the execution of c_c and c_a, we would also expect the simulation to hold. Indeed, using the sequential composition rule, we have:

$$\begin{aligned} &R_c(w), G_c(w), \alpha, \top, \top \vdash \\ &\quad (\text{None}, \{(w, (c_c; c_c, \top))\}, \text{N}\ s_c) \preceq (\text{None}, \{(w, (c_a; c_a, \top))\}, \text{N}\ s_a)\ Q, Q_i \end{aligned}$$

Liang et al.'s example also contains printer threads. Our Isabelle formalization similarly proves the simulation between these, and uses the parallel composition rule to prove simulation for the entire system.

6 Limitations and Future Work

The aim to use this work in the multicore seL4 verification informs some of its limitations. For instance, constructs for dynamically creating threads are not necessary for a static number of concurrent kernel instances. Parallel composition in a cooperative semantics is challenging to specify. A fork command would be a more natural extension but is unnecessary for our purposes.

The language we present is based on Complx [2]. Complx has exceptions, which are useful for modelling C constructs such as `break` and `continue`. We leave this for future work to focus on the main compositionality results first.

In our semantics there are "deadlocked" configurations: inactive configurations with threads that have not reached **skip**, but no guards are satisfied so no thread can run. Also, an active thread may run forever without yielding. Neither situation creates any fragments, thus satisfying our simulation and consequently, event trace refinement. Thus, deadlock freedom and termination are not preserved by our simulation, only safety properties on states.

We have not investigated how weak memory models would affect the semantics and have so far targeted sequential consistency only.

7 Related Work

Our mechanization of cooperative semantics is loosely based on the ideas of Abadi and Plotkin [1]. Their aim is not program verification, but instead exploring denotational semantics and connections to algebraic effects.

Liang et al. [11] introduce RGSim, a rely-guarantee-based simulation compositional with respect to constructs like parallel and sequential composition. Their work is formalized in Coq using a language with preemptive semantics. We adapt the simulation to a language with cooperative semantics while preserving parallel and sequential composition. As preemptive semantics can be expressed using cooperative semantics, our work in some ways generalizes RGSim.

For the treatment of atomicity refinement more broadly, linearizability [6] is a safety property widely used as a correctness condition for concurrent objects. It roughly states that each history of method invocations and responses is equivalent to a history where methods are executed sequentially. However, not all programs are naturally expressed as objects with methods; in particular, not those that we are interested in applying our method to.

Later work by the RGSim authors Liang and Feng [10] enables the use of liveness properties for blocking synchronization in addition to linearizability, so the same generalization may be possible in our setting. For our application, event trace refinement is sufficient, so we have not yet explored this direction further.

Elmas et al. [3,4] prove atomicity refinement and linearizability using reduction, which checks whether individual steps of a thread commute with steps of other threads, and which does not compose with respect to parallel composition. Elmas et al. use a preemptive semantics, but Civl [9] extends this line of work to use cooperative semantics. Their notion of refinement associates program steps

with assertions, and checks for preservation of end-to-end behavior and absence of assertion failures. These methods are not proven sound in a proof assistant.

Compositional notions of refinement have been used to verify concurrent compiler optimizations. Simuliris [5] uses a separation logic-based simulation to prove a fair termination preserving contextual refinement of concurrent optimizations. Contextual refinement considers the termination behavior of a program (terminating with a value, infinite execution or getting stuck) when composed with arbitrary well-formed contexts. Their language has preemptive semantics, does not consider I/O and assumes non-blocking execution.

Timany and Birkedal [15] provide a compositional separation logic-based proof method for refinement of programs with continuations, which they use as a compilation target for a cooperative concurrent language. Although they prove refinement between the target and source programs, they do not define refinement between programs in the cooperative source language. The compiler eliminates some nondeterminism on the concrete level by assuming a particular scheduling implementation using a queue, whereas we continue to allow for arbitrary interleaving of threads at yield points. Arbitrary interleaving better models the possible range of behavior of programs at the implementation level, such as when running on multiple cores and using different schedulers.

Vistrup et al. [17] use interaction trees [19] to enable reusable program logic fragments for effects on top of a pure language, including cooperative concurrency. Their program logics deal with single programs and not refinement relations between programs.

8 Conclusion

This paper has presented a concurrent imperative language with cooperative execution semantics. The language is generic over state and can be instantiated to model the behavior of a variety of more concrete imperative languages.

A cooperative semantics, unlike the usual preemptive concurrency semantics, lets us easily model different degrees of atomicity of executions within the same language without being constrained by the block structure of the language.

We have adapted the standard notion of trace refinement for cooperative semantics as a basis for the soundness of a compositional simulation that can be used for reasoning about such programs.

Our simulation for cooperative concurrent semantics is based on RGSim [11], an existing simulation formalization for the preemptive setting. The cooperative setting requires a number of subtle changes to enable compositional proof rules for reasoning about parallel and sequential composition. We have proved in Isabelle/HOL that the simulation is sound with respect to refinement, that it is compositional, and that it satisfies basic desirable properties such as transitivity.

Acknowledgement. This research was funded by the Australian Government's RTP scholarship. We thank the reviewers and Thomas Sewell for their feedback.

Disclosure of Interests. The authors have no competing interests.

References

1. Abadi, M., Plotkin, G.: A model of cooperative threads. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, pp. 29–40. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1480881.1480887
2. Amani, S., Andronick, J., Bortin, M., Lewis, C., Rizkallah, C., Tuong, J.: COMPLX: a verification framework for concurrent imperative programs. In: International Conference on Certified Programs and Proofs, pp. 138–150. SIGPLAN Notices, Paris (2017)
3. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 296–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_25
4. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, pp. 2–15. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1480881.1480885
5. Gäher, L., et al.: Simuliris: a separation logic framework for verifying concurrent program optimizations. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498689
6. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
7. Jones, C.B.: Tentative steps towards a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
8. Klein, G., et al.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2:1–2:70 (2014)
9. Kragl, B., Qadeer, S.: The civl verifier. In: 2021 Formal Methods in Computer Aided Design (FMCAD), pp. 143–152 (2021)
10. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, pp. 385–399. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2837614.2837635
11. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. SIGPLAN Not. **47**(1), 455–468 (2012). https://doi.org/10.1145/2103621.2103711
12. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer, Heidelberg (2002)
13. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976)
14. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
15. Timany, A., Birkedal, L.: Mechanized relational verification of concurrent programs with continuations. Proc. ACM Program. Lang. **3**(ICFP) (2019). https://doi.org/10.1145/3341709
16. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 97–108. ACM, Nice (2007)

17. Vistrup, M., Sammler, M., Jung, R.: Program logics à la carte. Proc. ACM Program. Lang. **9**(POPL) (2025). https://doi.org/10.1145/3704847
18. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
19. Xia, L.Y., et al.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371119