School of Computer Science & Engineering

Trustworthy Systems Group

A rely-guarantee-based simulation for cooperative semantics

Kevin Tran Johannes Åman Pohjola
Rob Sison Gerwin Klein

k.q.tran@unsw.edu.au

Background

- Shared-memory concurrency enables efficient execution
- But interference complicates reasoning
- Executable code has pervasive potential interference
- Some programs behave “almost sequentially”, with only a few interference points e.g. those using locks
- Such programs are nicer to reason about

© K. Tran et al. 2026, CC-BY-SA 4.0

Aim

How to formally prove an executable program behaves “almost sequentially”?

How?

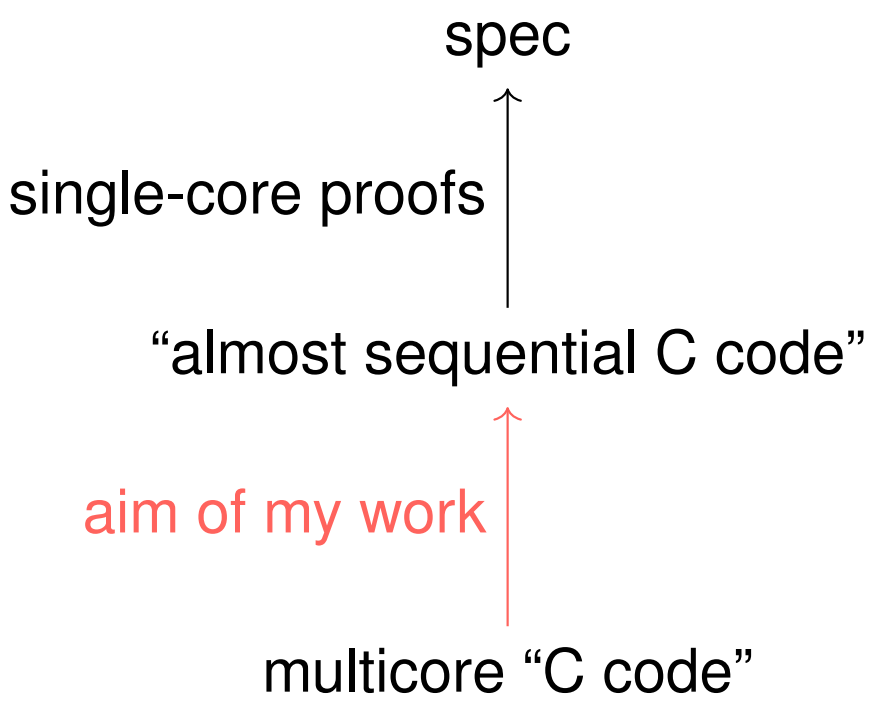
A rely-guarantee-based simulation for cooperative semantics:

- A proof method for event trace refinement
- Composes with respect to parallel&sequential composition
- Formalized in proof assistant Isabelle/HOL

© K. Tran et al. 2026, CC-BY-SA 4.0

Motivation

- seL4, an operating system kernel
- Single core seL4 is verified
- Want to verify big lock seL4



© K. Tran et al. 2026, CC-BY-SA 4.0

Structure

1. What “almost sequential” means
2. What behaving almost sequentially means
3. How to prove something behaves almost sequentially

© K. Tran et al. 2026, CC-BY-SA 4.0

“Almost sequential”?

- Client processing items of a lock-protected stack
- Want to allow local computations `f` to be done concurrently
- Don’t want to deal with concurrency otherwise

```
lock();
while !is_empty(stack) {
    x = pop(stack);
    unlock();
    f(x);
    lock();
}
unlock();
```


© K. Tran et al. 2026, CC-BY-SA 4.0

How to specify this program?

```
lock();
while !is_empty(stk) {
    x = pop(stk);
    unlock();
    f(x);
    lock();
}
unlock();
```

- Executable code typically has preemptive semantics
- Threads can be interleaved at each step
- Interference is pervasive and implicit
- Locks don't *specify* the absence of interference

© K. Tran et al. 2026, CC-BY-SA 4.0

Cooperative semantics

- Thread keeps executing until it hits a `yield` point
- Interference only at explicit `yield` points

preemptive implementation

```
lock();
while !isEmpty(stk) {
    x = pop(stk);
    unlock();
    f(x);
    lock();
}
unlock();
```


© K. Tran et al. 2026, CC-BY-SA 4.0

Cooperative semantics

- Thread keeps executing until it hits a `yield` point
- Interference only at explicit `yield` points

preemptive implementation

```
lock();
while !isEmpty(stk) {
    x = pop(stk);
    unlock();
    f(x);
    lock();
}
unlock();
```

cooperative specification

```
while !isEmpty_s(stk) {
    x = pop_s(stk);
    yield;
    f(x);
    yield;
}
yield;
```

- $\texttt{pop}_s$ is sequential version of `pop`

© K. Tran et al. 2026, CC-BY-SA 4.0

“Almost sequential”?

What is an “almost sequential” program?

One with few `yield`s in a cooperative semantics.

© K. Tran et al. 2026, CC-BY-SA 4.0

Cooperative semantics

- **Does not imply cooperative multithreading!**
- For specification, not execution
- Model preemptive code with pervasive yielding

© K. Tran et al. 2026, CC-BY-SA 4.0

Cooperative semantics details

- Small-step operational semantics on *configurations* with thread pool, active thread id, status (normal state or fault)
- Thread pool maps thread id to command and guard
- `yield guard` blocks thread until `guard` satisfied
- define synchronization mechanisms on top
- e.g. given bool `u`, `lock()` can be `yield u; u = false`

$$
\begin{aligned}
&(0, \{0 : (\texttt{lock()}, \texttt{true})\}, \text{Normal}\{\texttt{u} : \texttt{true}\}) \\
&\rightarrow (\text{None}, \{0 : (\texttt{skip;u=false}, \texttt{u})\}, \text{Normal}\{\texttt{u} : \texttt{true}\}) \\
&\rightarrow (0, \{0 : (\texttt{skip;u=false}, \texttt{u})\}, \text{Normal}\{\texttt{u} : \texttt{true}\}) \\
&\rightarrow (0, \{0 : (\texttt{u=false}, \texttt{u})\}, \text{Normal}\{\texttt{u} : \texttt{true}\}) \\
&\rightarrow (0, \{0 : (\texttt{skip}, \texttt{u})\}, \text{Normal}\{\texttt{u} : \texttt{false}\})
\end{aligned}
$$

© K. Tran et al. 2026, CC-BY-SA 4.0

Behaving almost sequentially?

- Executable code has pervasive potential interference
- Locks give atomicity, enabling sequential reasoning
- e.g. lock owned counter `x`

preemptive

```
lock();
x = x + 1;
x = x + 1;
unlock();
```


© K. Tran et al. 2026, CC-BY-SA 4.0

Behaving almost sequentially?

- Executable code has pervasive potential interference
- Locks give atomicity, enabling sequential reasoning
- e.g. lock owned counter `x`

preemptive

```
lock();
x = x + 1;
x = x + 1;
unlock();
```

cooperative translation

```
lock(); yield;
r = x; yield;
r = r + 1; yield;
x = r; yield;
r = x; yield;
r = r + 1; yield;
x = r; yield;
unlock(); yield;
```


© K. Tran et al. 2026, CC-BY-SA 4.0

Refinement ($\sqsubseteq$)

- Formalizes and generalizes atomicity
- Transitive relation between *closed* programs
- Concrete program has no more behavior than abstract

concrete

```
lock(); yield;
r = x; yield
r = r + 1; yield;
x = r; yield;
r = x; yield;
r = r + 1; yield;
x = r; yield;
unlock(); yield;
```

$\sqsubseteq$

```
lock();
r = x;
r = r + 1;
x = r;
r = x;
r = r + 1;
x = r;
unlock();
yield;
```


© K. Tran et al. 2026, CC-BY-SA 4.0

Refinement ($\sqsubseteq$)

- Formalizes and generalizes atomicity
- Transitive relation between *closed* programs
- Concrete program has no more behavior than abstract

concrete

```
lock(); yield;
r = x; yield
r = r + 1; yield;
x = r; yield;
r = x; yield;
r = r + 1; yield;
x = r; yield;
unlock(); yield;
```

$\sqsubseteq$

```
lock();
r = x;
r = r + 1;
x = r;
r = x;
r = r + 1;
x = r;
unlock();
yield;
```

$\sqsubseteq$

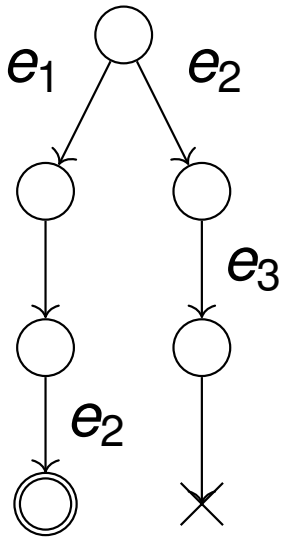
abstract

```
x = x + 2;
yield;
```


© K. Tran et al. 2026, CC-BY-SA 4.0

Event trace refinement ($\sqsubseteq$)

- Programs may emit events during execution
- Execution has result e.g. partial, terminated, faulted
- Associate program with set of traces of events and result
- Refinement is set inclusion



$(e_1, \text{Partial})$
$(e_1\, e_2, \text{Terminated})$
$(e_2, \text{Partial})$
$(e_2\, e_3, \text{Partial})$
$(e_2\, e_3, \text{Faulted})$

© K. Tran et al. 2026, CC-BY-SA 4.0

Behaving almost sequentially

What behaving almost sequentially means

Refinement between executable code and an abstract program with few yields.

© K. Tran et al. 2026, CC-BY-SA 4.0

Proving event trace refinement

- Definition vs. proof method
- Using definition directly is inconvenient because of non-compositionality w.r.t. parallel composition

concrete

```
x = x + 1; yield;
x = x + 1; yield;
```

$\sqsubseteq$

abstract

```
x = x + 2; yield;
```

```
x = x + 1;
yield;
x = x + 1;
yield;
```
$\parallel$
```
print(x);
yield;
```
$\not\sqsubseteq$
```
x = x + 2;
yield;
```
$\parallel$
```
print(x);
yield;
```

What we want when proving refinement

Composes with respect to parallel composition

© K. Tran et al. 2026, CC-BY-SA 4.0

Challenge

No prior work that is:

1. Proof method for refinement
2. For cooperative semantics
3. Composes with respect to parallel composition

Approach

Use a rely-guarantee-based simulation which satisfies 1 and 3 and adapt to 2

© K. Tran et al. 2026, CC-BY-SA 4.0

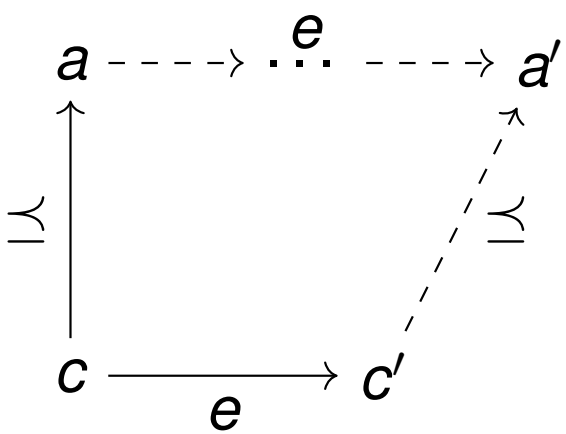
Rely-guarantee reasoning

- Enables compositional reasoning w.r.t. parallel composition
- Each component has rely and guarantee relation
- Must prove guarantee e.g. when program doesn't hold lock, it doesn't change lock owned state
- Can assume rely e.g. when program holds lock, environment doesn't change lock owned state
- Relies must be compatible with guarantees of other threads

© K. Tran et al. 2026, CC-BY-SA 4.0

Simulation ($\preceq$)

- Relation $c \preceq a$ between concrete and abstract program
- Proof method for refinement ($c \preceq a \implies c \sqsubseteq a$)
- Each concrete step is matched by zero or more abstract steps



if $c \xrightarrow{e} c'$ then there exists a' s.t. $a \xrightarrow{e}{}^{*} a'$ and $c' \preceq a'$

© K. Tran et al. 2026, CC-BY-SA 4.0

Rely-guarantee simulation (RGSim)

- By Liang et al. (POPL '12)
- Rely and guarantee relations on concrete (R, G) and abstract ($\mathbb{R}, \mathbb{G}$) level
- Step invariant (α) between concrete and abstract state
- Steps taken by concrete and abstract program in simulation must satisfy respective guarantees
- Stable under rely relations
- Postcondition (Q) enables sequential compositionality

$$R, G, \alpha, \mathbb{R}, \mathbb{G} \vdash \mathit{cfg}_c \preceq \mathit{cfg}_a\ Q$$

© K. Tran et al. 2026, CC-BY-SA 4.0

Adapting RGSim to cooperative semantics

- Refinement and simulation typically work on each step
- In our setting, intermediate steps shouldn't matter
- Define analog of preemptive step: fragment
- Fragment: Sequence of steps from thread activation until yield or fault or incomplete termination

$$
\begin{aligned}
& (\texttt{x = x + 1; yield;},\ldots) \\
\rightarrow\ & (\texttt{skip; yield;},\ldots) \\
\rightarrow\ & (\texttt{yield;},\ldots) \\
\rightarrow\ & (\texttt{skip;},\ldots)
\end{aligned}
\Bigg\} \textit{fragment}
$$

© K. Tran et al. 2026, CC-BY-SA 4.0

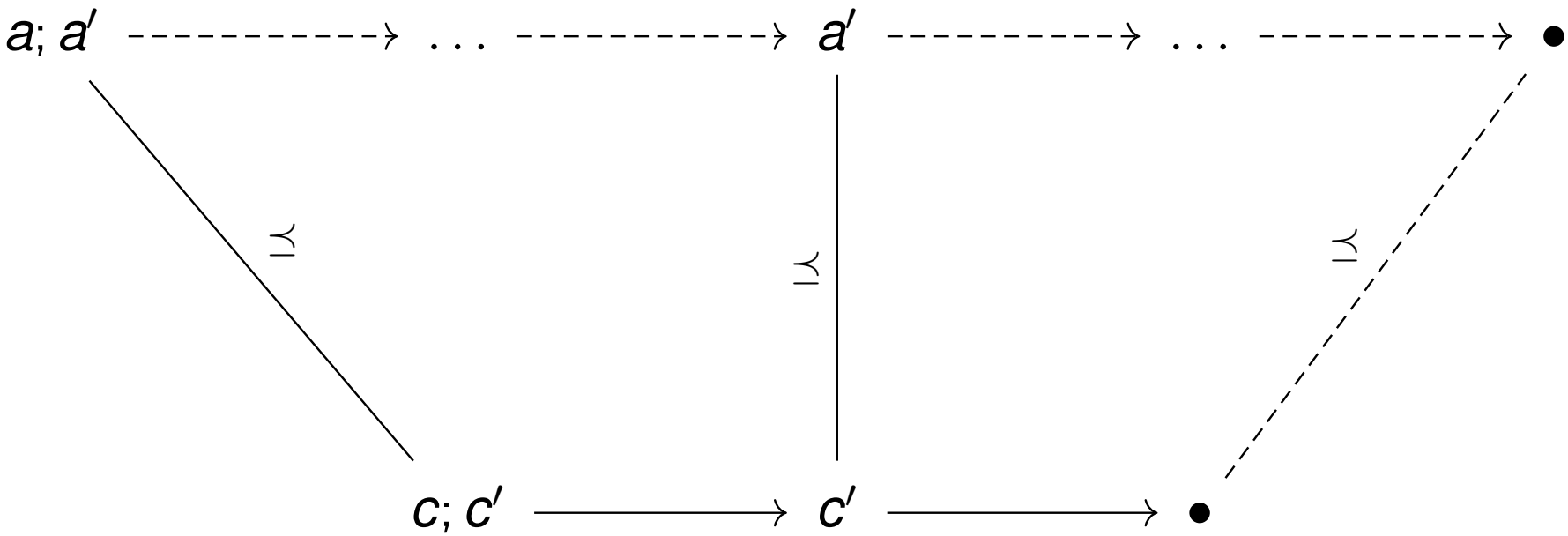
Sequential composition, cooperatively

- We want to prove $c; c' \preceq a; a'$ using $c \preceq a$ and $c' \preceq a'$
- To establish simulation, we need to consider fragments starting from $c; c'$
- c might temporarily violate guarantee or invariant
- e.g. guarantee is x maintains parity and $c = c' = \texttt{x=x+1}$
- What happens when the fragment straddles c and c'?

© K. Tran et al. 2026, CC-BY-SA 4.0

Simulation proof idea

- Try break fragment in two, and match each fragment using $c \preceq a$ and $c' \preceq a'$, respectively



© K. Tran et al. 2026, CC-BY-SA 4.0

Sequential composition, cooperatively

- Fragment starting from *c* must terminate
- What should a fragment do when execution terminates?

© K. Tran et al. 2026, CC-BY-SA 4.0

Sequential composition, cooperatively

- Fragment starting from *c* must terminate
- What should a fragment do when execution terminates?
- Implicitly yielding on termination of *c* forces us to reestablish guarantee/invariant
- Instead, not yielding on termination of *c* is considered *incomplete*, deferring the checks until the next yield
- Store last yield state to check at next yield

© K. Tran et al. 2026, CC-BY-SA 4.0

Sequential composition

- Explicitly yield for normal termination
- Store guards on termination to filter to configurations where the second fragment is not blocked
- Since we can terminate normally or incompletely, we have two kinds of postconditions:
  - The normal postcondition stores the final states as usual, as well as guards on termination
  - The incomplete postcondition stores the final states and last yield states

© K. Tran et al. 2026, CC-BY-SA 4.0

My adaptation of RGSim

$$R, G, \alpha, \mathbb{R}, \mathbb{G} \vdash (cfg_c, s_c) \preceq (cfg_a, s_a)\ Q, Q_i$$

- Fragments instead of steps
- Configurations cfg_c, cfg_a
- Last yield states s_c, s_a
- Rely (R, $\mathbb{R}$), guarantee (G, $\mathbb{G}$)
- Step invariant α
- Normal postcondition Q with guards
- Incomplete postcondition Q_i

© K. Tran et al. 2026, CC-BY-SA 4.0

Summary

- A proof method for event trace refinement
- Composes with respect to parallel&sequential composition
- Formalized in proof assistant Isabelle/HOL

How to formally prove an executable program behaves “almost sequentially”?

Using a rely-guarantee-based simulation for cooperative semantics!

© K. Tran et al. 2026, CC-BY-SA 4.0

The end

© K. Tran et al. 2026, CC-BY-SA 4.0