Mechanising Local Rely-Guarantee Reasoning over the seL4 Trace Monad

Junming Zhao
junming.zhao@unsw.edu.au
UNSW Sydney

Abstract

We present an Isabelle/HOL mechanisation of rely-guarantee reasoning combined with separation logic, developed on the seL4 trace monad: an Aczel-style trace model that makes environment interference first-class. We follow Feng's local rely-guarantee (LRG) because its frame rule fits the seL4 Microkit framework we target: a static-architecture setting in which components, their memory-region mappings, and the communication channels between them are fixed at build time.

The development so far has: (i) LRG's frame and hide rules, mechanised over the trace monad's primitives, sequential composition, and while loops; (ii) an n-ary parallel composition rule; and (iii) a three-component Microkit case study (a producer/filter/consumer pipeline over shared memory regions) composed by the n-ary parallel rule.

1 Introduction

The aim of this Isabelle/HOL development is to support verification of concurrent user-level systems built on the seL4 Microkit [24]. seL4 is a microkernel formally verified in Isabelle/HOL [15], but its guarantee stops at the kernel boundary: cross-component invariants, inter-component protocols, and other concurrent correctness properties of the composed user-level system have not been established. The setting we work with is Microkit, a static-architecture toolkit on top of the kernel: components are fixed at build time, each owns a determined set of private memory regions, and inter-component sharing happens through declared shared regions and communication channels. The recent LionsOS [10] is one such example, designed for security- and safety-critical embedded systems, and a long-term verification target of this work. For individual components, toolchains such as Pancake-to-Viper [20, 17, 28] verify each component in isolation, but their guarantees are sequential and stop at the inter-component boundary. Microkit's components run concurrently and interact through the shared state, so verifying the composed system requires concurrency reasoning.

Concurrent programs must handle interference, where one component's actions affect another's view of shared state. Reasoning about them is a long-standing research area, with rely-guarantee and separation logic as two well-known approaches. Rely-guarantee (RG) [12] handles interference cleanly: a rely R bounds the environment's steps, a guarantee G bounds the program's own, and parallel composition checks that each thread's guarantee covers the others' relies. The downside, which we revisit in § 2.2, is that relies and guarantees range over the whole state space, so threads on disjoint regions still have to mention each other. Separation logic (SL) [19]'s separating conjunction $P * Q$ describes disjoint states, and the frame rule lets a thread reason locally about its own footprint. Several frameworks combine the two: RGSep [26], SAGL [9],

and Feng's local rely-guarantee (LRG) [8]. Modern concurrent separation logics such as Iris [14], FCSL [18], and TaDA [5] can do the same through more general mechanisms (higher-order ghost state, step-indexed frameworks, user-defined invariants), at the cost of significantly more proof infrastructure. For the static-architecture setting we target, LRG fits: each action is paired with a precise invariant on the region it touches, matching Microkit's declared memory regions whose ownership is explicit and fixed.

We mechanise LRG over the seL4 trace monad [23]. The trace monad is a language-agnostic operational model in which every step of a program is recorded with a tag distinguishing self-action from environment interference; this matches RG semantics directly. Our three-component case study composes the components (each a trace-monad program) via an n-ary parallel rule. The components form a producer/filter/consumer pipeline communicating through shared memory regions, the shape of a typical Microkit IO subsystem.

Outline. Section 2 reviews the seL4 trace monad, rely-guarantee, and separation logic. Section 3 sets up the separation algebra and the trace decomposition used in this mechanisation. Section 4 develops the necessary premises of the LRG frame rule. Section 5 presents the n-ary parallel composition rule. Section 6 reports the three-component Microkit case study. Section 7 compares with related mechanisations, and Section 8 concludes.

2 Background

2.1 The seL4 trace monad

Our development is built on the trace monad of the seL4 verification framework [23], which we treat as a concrete realisation of the Aczel trace model [1, 6]. In Aczel's formulation, a program denotes a set of finite sequences of states (or, equivalently, state-tagged actions) with two kinds of step: those taken by the program itself and those taken by its environment. Rely-guarantee maps onto this model directly: a rely constrains Env steps, a guarantee constrains Me steps. The seL4 trace monad implements precisely this view in Isabelle/HOL, with an explicit data type for traces and an explicit Me/Env tag on every step.

Types. A computation $f : \Sigma \to \mathcal{P}((\{\mathsf{Me}, \mathsf{Env}\} \times \Sigma)^* \times \mathit{Out}_\alpha)$, with Σ the state type and α the type of return values, maps an input state to a set of (trace, outcome) pairs, where a trace is a finite sequence of state-tagged steps, each tag distinguishing a self-step (Me) from environment interference (Env). We write nil for the empty trace and :: for cons. The outcome ranges over

$$\mathit{Out}_\alpha \;=\; \{\mathsf{Failed}\} \,\cup\, \{\mathsf{Incomplete}\} \,\cup\, \{\mathsf{Result}(v, \sigma) \mid v \in \alpha,\, \sigma \in \Sigma\}.$$

The $\mathsf{Incomplete}$ outcome carries the strict prefixes of unfinished runs (which makes the sequential composition `bind` well-behaved under interference). Traces are stored *most-recent-first*: the head is the newest step, as `bind`'s prepending in Figure 1 makes explicit. The trace-monad primitives including `bind` are summarised in Figure 1.

Primitives. The pure/state primitives in Figure 1 are *trace-empty*: they emit the empty trace list and introduce no interference points. `commit_step`, `env_steps`, and the atomic-commit family record at least one trace step per call. Control flow uses `condition` $P\,L\,R \equiv \lambda s.\,\mathsf{if}\ P\,s\ \mathsf{then}\ L\,s\ \mathsf{else}\ R\,s$ for branching and `whileLoop` $C\,B$ for iteration.

Interference points. Aczel-style interleaving requires explicit places at which the environment is allowed to act. The trace monad has three: `env_steps` uses `select UNIV` to pick an arbitrary sequence of states and appends them Env-tagged to the trace (the rely constraint later

Pure and state primitives

$$\texttt{return}\, v \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Result}(v, s))\} \qquad \texttt{put}\, s' \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Result}((), s'))\}$$
$$\texttt{get} \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Result}(s, s))\} \qquad \texttt{modify}\, f \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Result}((), f\, s))\}$$
$$\texttt{select}\, A \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Result}(v, s)) \mid v \in A\} \qquad \texttt{fail} \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Failed})\}$$

Sequential composition

$$\texttt{bind}\, f\, g \ \equiv\ \lambda s.\ \bigcup_{(xs,\, r)\, \in\, f\, s} \Phi_g(xs,\, r)$$
$$\Phi_g(xs, r) \ =\ \begin{cases} \{(xs, r)\} & r \in \{\mathsf{Failed},\ \mathsf{Incomplete}\} \\ \{(ys +\!\!+\, xs,\ r') \mid (ys, r') \in g\, v\, \sigma\} & r = \mathsf{Result}(v, \sigma) \end{cases}$$

Interference operators

$$\texttt{commit_step} \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Incomplete}),\ ([(\mathsf{Me},\ s)],\ \mathsf{Result}((), s))\}$$
$$\texttt{env_steps} \equiv \text{non-deterministic append of } \mathsf{Env}\text{-tagged steps}$$
$$\texttt{interference} \equiv \texttt{commit_step}\ ;\ \texttt{env_steps}$$

Atomic-commit family

$$\texttt{modify_commit}\, f \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Incomplete}),\ ([(\mathsf{Me},\ f\, s)],\ \mathsf{Result}((), f\, s))\}$$
$$\texttt{read_commit}\, h \equiv \lambda s.\,\{(\mathsf{nil},\ \mathsf{Incomplete}),\ ([(\mathsf{Me},\ s)],\ \mathsf{Result}(h\, s, s))\}$$
$$\texttt{rmw_commit}\, f \equiv \lambda s.\, \mathsf{let}\, (\sigma', v) = f\, s\, \mathsf{in}\, \{(\mathsf{nil},\ \mathsf{Incomplete}),\ ([(\mathsf{Me},\ \sigma')],\ \mathsf{Result}(v, \sigma'))\}$$

Figure 1. Trace-monad primitives. The first three groups are from the upstream seL4 infrastructure [23, 3]; the atomic-commit family (with `rmw_commit` the most general form) is introduced in this paper and analysed in § 4. The `Incomplete` pair in each trace-emitting operator is what makes the operator prefix-closed.

filters them); `commit_step` records the current state as a single Me step; and `interference` ≡ `commit_step`; `env_steps` packages the two together. A program intended for parallel composition is well-formed when it begins with `env_steps` and ends with `interference`, ensuring its trace has enough Env slack to align with another thread's.

Parallel composition. Parallel composition zips two traces position-wise. A trace tr_f of $f\,s$ and a trace tr_g of $g\,s$ are *compatible* when they have the same length and agree on states at every position, with at most one Me tag per position (the other is Env, or both tags are Env). Their position-wise merge $tr_f \bowtie tr_g$ keeps the shared states and takes the non-Env tag at each position (Env if both are). With this,

$$\texttt{parallel}\, f\, g \ \equiv\ \lambda s.\, \big\{(xs, rv) \ \big|\ \exists\, tr_f\ tr_g.\ (tr_f, rv) \in f\, s\ \wedge\ (tr_g, rv) \in g\, s\ \wedge\ xs = tr_f \bowtie tr_g\big\}.$$

We use $\|$ as infix syntax for `parallel`.

2.2 Rely-guarantee reasoning

Rely-guarantee [12, 27] extends Hoare logic to shared-variable concurrency by augmenting a triple $\{P\}\, f\, \{Q\}$ with two extra binary predicates on states: a *rely* R that bounds the steps the environment may take between actions of f, and a *guarantee* G that bounds the steps f itself may take. The resulting quintuple $\{P\}, \{R\}\, f\, \{G\}, \{Q\}$ reads informally: from any state satisfying P, against an R-respecting environment, f's steps respect G and any terminating run ends in a state satisfying Q.

The RG definition over the trace monad. We require f to be prefix-closed: every prefix of a trace in $f\,s$ is itself in $f\,s$, paired with $\mathsf{Incomplete}$. The quintuple $\{P\}, \{R\}\, f\, \{G\}, \{Q\}$ then unfolds as:

$$\begin{aligned} \forall s, tr, res.\quad & P\,s \;\wedge\; (tr, res) \in f\,s \;\wedge\; \mathsf{relyCond}\, R\, s\, tr \\ \implies\quad & \mathsf{guarCond}\, G\, s\, tr \;\wedge\; (\forall rv, s'.\; res = \mathsf{Result}\,(rv, s') \implies Q\, rv\, s') \end{aligned}$$

where $\mathsf{relyCond}\, R\, s_0\, tr$ requires every Env-tagged step (s, s') in tr to satisfy $R\, s\, s'$. Similarly, $\mathsf{guarCond}\, G\, s_0\, tr$ requires every Me-tagged step to satisfy $G\, s\, s'$. Prefix-closure ensures G is checked at every Me-step of every partial run, not only at termination. Q is checked only at Result outcomes (partial correctness); $\mathsf{Incomplete}$ traces are unconstrained.

RG parallel composition rule. The defining rule of rely-guarantee [12, 27] composes two threads when each thread's guarantee is implied by the other thread's rely. Writing $G \leq R$ for pointwise implication ($\forall s\, s'.\, G\, s\, s' \implies R\, s\, s'$):

$$\frac{\{P_1\}, \{R_1\}\; f_1\; \{G_1\}, \{Q_1\} \qquad \{P_2\}, \{R_2\}\; f_2\; \{G_2\}, \{Q_2\} \qquad G_1 \leq R_2 \qquad G_2 \leq R_1}{\{P_1 \wedge P_2\}, \{R_1 \cap R_2\}\; f_1 \parallel f_2\; \{G_1 \cup G_2\}, \{Q_1 \wedge Q_2\}}\;\textsc{Par}$$

We mechanise this rule for the simplified RG definition above; the proof is a routine adaptation. Despite its compositionality, RG reasoning has a well-known inconvenience: relies and guarantees range over the whole state space. Even threads working on disjoint variables must mention every part of the state in their specifications, and adding a new thread to the system forces every existing specification to be revised. We integrate separation logic in §§ 2.3 and 3 to address this.

What we inherit and what we add. The upstream seL4 infrastructure gives us the trace monad types, primitives, `parallel`, `interference`, the RG definition, and the classic Par rule. The new contribution begins in § 3, where we layer a state-based separation algebra and a rely-parameterised trace decomposition predicate over this Aczel-style trace model, and proceeds through §§ 4 and 5 to a frame rule, a hiding rule, and an n-ary parallel composition rule.

2.3 Separation logic

LRG [8] addresses the modularity gap of RG by partitioning state into private and shared regions, so a thread's RG specification mentions only its own private state and the shared regions it touches. The frame rule then lifts this local specification to the full system state, so the local spec doesn't mention other threads' private state. This subsection introduces the separation logic machinery that our mechanisation uses.

Separation algebra. The standard semantic foundation of a separation logic is a *separation algebra* [2]: a partial commutative monoid of states with disjointness $\perp$, composition $\oplus$, and a unit ε. The separating conjunction $P * Q$ holds at a state s when s splits into disjoint sub-states satisfying P and Q:

$$(P * Q)\, s \;\equiv\; \exists\, s_1\, s_2.\; s_1 \perp s_2 \;\wedge\; s = s_1 \oplus s_2 \;\wedge\; P\, s_1 \;\wedge\; Q\, s_2.$$

LRG additionally requires invariants to be *precise*: at most one sub-state of any given state satisfies them, so the shared region's boundary is unambiguous; § 3.2 gives the formal definition.

Mechanisation on the trace monad's state. We mechanise the separation algebra as an Isabelle *locale* [16] parametrised by $(\perp, \oplus, \varepsilon)$, satisfying the standard partial-commutative-monoid axioms together with cancellation and positivity. From here on, the trace monad's state type Σ is taken to satisfy this locale: any concrete state type used with the monad provides $(\perp, \oplus, \varepsilon)$

together with the axioms above. The locale is deliberately small: it fixes only what LRG's frame and hiding rules need (separating conjunction, action separating conjunction $*_A$, precision, fencing), without committing to the richer hierarchy of separation algebra variants in the AFP entry of [16]. Concrete instances (partial map, option tile) need only discharge the locale axioms, after which all rules and proofs of §§ 3 to 5 apply.

3 Separation, fencing, and trace decomposition

This section gives the three ingredients needed to lift the separation-logic frame rule from a thread's local state to the full system state.

The first two ingredients are adopted from LRG [8] and RGSep [26], formalised inside the separation-algebra locale of § 2.3: (i) *action separating conjunction* $R_1 *_A R_2$ (§ 3.1) under which steps on disjoint sub-states compose, and (ii) *fenced actions* $I \triangleright R$ (§ 3.2) that pair an action with a precise invariant pinning the region it touches. The third is the *rely-parameterised trace decomposition* predicate `trace_sep` (§ 3.3), which we introduce to adapt LRG's one-step decomposition to the trace-monad setting.

3.1 Action separating conjunction

An *action* is a binary relation on states; the relies and guarantees of § 2.2 are actions. There are two frequently used actions: the identity $id \equiv (=)$, which leaves the sub-state fixed, and the universal action $\top \equiv \lambda_\,_.\,\mathsf{True}$, which permits any step.

Action separating conjunction $*_A$ [26, 8] lifts the standard separating conjunction $*$ from state predicates to actions:

$$(R_1 *_A R_2)\, s\, s' \;\equiv\; \exists s_1\, s_2\, s_1'\, s_2'.\; s_1 \perp s_2 \wedge s = s_1 \oplus s_2 \;\wedge\; s_1' \perp s_2' \wedge s' = s_1' \oplus s_2' \;\wedge\; R_1\, s_1\, s_1' \wedge R_2\, s_2\, s_2'.$$

A step satisfying $R_1 *_A R_2$ on a composed state factors into two independent steps on the components. $*_A$ is commutative, associative, and monotone in each argument.

The action separating conjunction is the operator that lifts a thread's local RG specification to a composed RG specification. A thread proved against (R_1, G) on its local view extends to a thread proved against $(R_1 *_A R_2, G *_A id)$ on the composed state, under suitable fencing of R_1 and R_2 (§ 3.2). Here R_2 is the rely of the rest of the system.

3.2 Fenced actions and invariant preservation

Precision. A predicate I is *precise* when, for any composite state s, at most one sub-state of s satisfies I:

$$\begin{aligned} \mathtt{precise}(I) \;\equiv\; \forall s\, s_1\, s_2\, f_1\, f_2.\;\; & s_1 \perp f_1 \;\wedge\; s = s_1 \oplus f_1 \;\wedge\; I\, s_1 \;\wedge \\ & s_2 \perp f_2 \;\wedge\; s = s_2 \oplus f_2 \;\wedge\; I\, s_2 \\ & \Longrightarrow\; s_1 = s_2. \end{aligned}$$

Precision is a classical separation-logic notion: it ensures that an invariant pins down a unique sub-state of any composite state. This uniqueness lets us speak of the specific sub-state an action operates on, not just some arbitrary sub-state.

Fence. An action R is *fenced* by an invariant I (written $I \triangleright R$) [8] when R acts only on the sub-state I pins down. In detail,

$$I \triangleright R \;\equiv\; \underbrace{(\forall s.\, I\, s \implies R\, s\, s)}_{\text{stuttering}} \;\wedge\; \underbrace{(\forall s\, s'.\, R\, s\, s' \implies I\, s \wedge I\, s')}_{\text{endpoints}} \;\wedge\; \mathtt{precise}(I).$$

Stuttering lets an environment that does nothing satisfy R if the current local state satisfies I. The endpoints clause gives invariant preservation, and precision pins down the region R acts on. Stability follows for free: $I \triangleright R \implies sta\, I\, R$, where $sta\, P\, A \equiv \forall s\, s'.\, P\, s \wedge A\, s\, s' \implies P\, s'$.

Composing under $*_A$. The below three lemmas are used to help prove the frame and hiding rules of § 5:

Lemma 1 (fence composition)**.** $I_1 \triangleright R_1 \;\wedge\; I_2 \triangleright R_2 \implies (I_1 * I_2) \triangleright (R_1 *_A R_2)$.

Fenced actions compose under separating conjunction.

Lemma 2 (frame property)**.** $I \triangleright R \;\wedge\; s = s_1 \oplus s_2 \;\wedge\; I\, s_1 \;\wedge\; (R *_A id)\, s\, s' \implies \exists s_1'.\, s' = s_1' \oplus s_2 \;\wedge\; R\, s_1\, s_1'$.

$*_A$ paired with id factors a step into a local R-step and an unchanged frame; we use this to prove the frame rule of § 5.

Lemma 3 (hiding step)**.** $I_1 \triangleright R_1 \;\wedge\; I_2 \triangleright R_2 \;\wedge\; (I_1 * I_2)\, s \;\wedge\; (R_1 *_A id)\, s\, s' \implies (R_1 *_A R_2)\, s\, s'$.

$R_1 *_A id$ implies $R_1 *_A R_2$: the frame is unchanged, and $R_2\, s_2\, s_2$ follows from the stuttering clause of $I_2 \triangleright R_2$. We use this to prove `hiding_rule` in § 5.3, where the rule lets a spec proved with a coarser hidden-side action be re-stated against a finer one.

Lemmas 1 to 3 together form the algebra of single-step actions that we inherit from LRG.

3.3 Rely-parameterised trace decomposition

In LRG's separation algebra, actions are single-step: each fenced rely R is a relation between pre- and post-states, and $*_A$ splits one such relation into two. The trace monad of § 2.1 works at a different granularity: a global execution is a finite sequence of Me/Env steps. Lifting LRG's one-step decomposition to this setting requires a list-shaped predicate.

We introduce the inductive predicate $\texttt{trace_sep}\, R\, (s_l, s_f)\; tr_l\; tr_f\; tr_g\; \bar{s}_f$, where:

- $R : \Sigma \to \Sigma \to \texttt{bool}$ is a rely;
- $(s_l, s_f) \in \Sigma \times \Sigma$ is a state split into a local sub-state s_l and a frame sub-state s_f;
- $tr_l, tr_f, tr_g \in (\{\mathsf{Me}, \mathsf{Env}\} \times \Sigma)^*$ are local, frame, and global traces, all are in chronological order (earliest step at the head);
- $\bar{s}_f \in \Sigma$ is the final frame state.

`trace_sep` decomposes a chronological global trace into a local-trace projection and a frame-trace projection, and requires every Env step on the local part to respect R:

$$\frac{s_l \perp s_f}{\texttt{trace_sep}\, R\, (s_l, s_f)\; \mathsf{nil}\; \mathsf{nil}\; \mathsf{nil}\; s_f}\ \textsc{Nil}$$

$$\frac{s_l \perp s_f \qquad s_l' \perp s_f \qquad s = s_l' \oplus s_f \qquad \texttt{trace_sep}\, R\, (s_l', s_f)\; tr_l\; tr_f\; tr_g\; \bar{s}_f}{\texttt{trace_sep}\, R\, (s_l, s_f)\; ((\mathsf{Me}, s_l') :: tr_l)\; ((\mathsf{Me}, s_f) :: tr_f)\; ((\mathsf{Me}, s) :: tr_g)\; \bar{s}_f}\ \textsc{Cons-Me}$$

$$\frac{s_l \perp s_f \qquad s_l' \perp s_f' \qquad R\, s_l\, s_l' \qquad s = s_l' \oplus s_f' \qquad \texttt{trace_sep}\, R\, (s_l', s_f')\; tr_l\; tr_f\; tr_g\; \bar{s}_f}{\texttt{trace_sep}\, R\, (s_l, s_f)\; ((\mathsf{Env}, s_l') :: tr_l)\; ((\mathsf{Env}, s_f') :: tr_f)\; ((\mathsf{Env}, s) :: tr_g)\; \bar{s}_f}\ \textsc{Cons-Env}$$

Nil is the base case: an empty trace decomposes trivially when s_l and s_f are disjoint, and the final frame state equals the starting frame. Cons-Me extends the decomposition by a thread

step: the local sub-state moves from s_l to s_l', while the frame stays at s_f, meaning the thread won't touch the environment frame. Cons-Env extends it by an environment step: both the local and frame may move (to s_l' and s_f'), but the local move must be permitted by the rely $(R\, s_l\, s_l')$.

`trace_sep` applies R to Env steps only; thread (Me) steps are checked against the guarantee G in the RG quintuple of § 2.2 instead.

Rely monotonicity. `trace_sep` is monotone in its rely:

Lemma 4 (rely monotonicity)**.** $\texttt{trace_sep}\, R\, (s_l, s_f)\, tr_l\, tr_f\, tr_g\, \bar{s}_f \wedge (\forall a\, b.\, R\, a\, b \implies R'\, a\, b) \implies \texttt{trace_sep}\, R'\, (s_l, s_f)\, tr_l\, tr_f\, tr_g\, \bar{s}_f$.

A tighter R admits fewer decompositions, so locality proved against R extends to any weaker R'.

Together with the single-step algebra of §§ 3.1 and 3.2, `trace_sep` is the basis for §§ 4 and 5. Locality and decomposition existence (§ 4) lift the single-step ingredients to whole runs of monadic programs by induction over `trace_sep`. Section 5 then uses these to prove the frame and parallel rules.

4 Locality and decomposition existence

Section 3 gave the single-step algebra (fenced actions, $*_A$) and a list-shaped decomposition predicate `trace_sep` for the trace monad. To use them in a frame rule, we need two further predicates over whole runs of monadic programs: a *locality* predicate (§ 4.1) asserting that every decomposition has a matching local run, and a *decomposition existence* predicate (§ 4.2) asserting that a decomposition exists in the first place. Together with a coverage pass over the trace monad's primitives and combinators (§ 4.3), they discharge the premises of the frame rule of § 5.

4.1 The locality predicate `is_local`

$$
\begin{aligned}
\texttt{is_local}\, I\, R\, f \;\equiv\; & \forall\, s_l\, s_f\, tr_g\, r_g\, tr_l^0\, tr_f^0\, \bar{s}_f. \\
& [\![\, s_l \perp s_f;\; I\, s_l;\; (tr_g, r_g) \in f\, (s_l \oplus s_f); \\
& \;\; \texttt{trace_sep}\, R\, (s_l, s_f)\, tr_l^0\, tr_f^0\, (\texttt{rev}\, tr_g)\, \bar{s}_f \,]\!] \\
& \implies \exists\, tr_l\, r_l.\, (tr_l, r_l) \in f\, s_l \wedge \texttt{rev}\, tr_l = tr_l^0 \wedge \\
& \quad r_g = \texttt{lift_res}\, \bar{s}_f\, r_l \wedge (\forall\, rv\, s'.\, r_l = \mathsf{Result}(rv, s') \implies I\, s').
\end{aligned}
$$

The definition has four nested levels:

- *Setup*: a starting state split as (s_l, s_f) with $I\, s_l$.
- *Global execution*: a global (trace, result) pair (tr_g, r_g) produced by f on the composed starting state $s_l \oplus s_f$.
- *Decomposition*: a `trace_sep` decomposition of tr_g into local and frame projections.
- *Local witness*: a matching run (tr_l, r_l) of f on s_l (without the frame), with $\texttt{rev}\, tr_l$ equal to the decomposition's local projection, $r_g = \texttt{lift_res}\, \bar{s}_f\, r_l$, and I preserved on success.

Here $\texttt{lift_res}\, s_f$ maps $\mathsf{Result}(v, s')$ to $\mathsf{Result}(v, s' \oplus s_f)$ and leaves Failed and Incomplete unchanged. The locality predicate asserts that for every choice of the first three, the fourth (local witness) exists.

Rely monotonicity. Strengthening R admits fewer decompositions, so locality is easier to satisfy. The direct corollary at the locality layer follows from Lemma 4.

4.2 The decomposition existence predicate

`is_local` consumes a decomposition. Where no decomposition exists, the `trace_sep` premise is unsatisfiable, so the implication holds vacuously and locality is trivially true, giving the frame rule no useful local witness. The frame rule needs a separate guarantee that decompositions exist.

$$\begin{aligned}\texttt{decomp_exists}\,R\,I\,R_g\,F\,f \;\equiv\; & \forall\, s_l\, s_f\, tr_g\, r_g.\\ & [\![\, s_l \perp s_f;\; I\, s_l;\; F\, s_f;\; (tr_g, r_g) \in f\,(s_l \oplus s_f);\\ & \;\texttt{rely_cond}\, R_g\,(s_l \oplus s_f)\; tr_g \,]\!]\\ & \;\Longrightarrow\; \exists\, tr_l^0\; tr_f^0\; \bar{s}_f.\; \texttt{trace_sep}\, R\,(s_l, s_f)\; tr_l^0\; tr_f^0\;(\texttt{rev}\; tr_g)\;\bar{s}_f\\ & \quad \wedge\; F\,\bar{s}_f.\end{aligned}$$

Here $\texttt{rely_cond}\, R\, s\, tr$ is the predicate that every `Env` step in tr respects R. Compared to `is_local`, `decomp_exists` takes two extra parameters: a *frame predicate* F on the frame, and a *global rely* R_g that the global trace respects. Both are needed when decomposing an environment step on the composed state into local and frame parts.

Why two predicates instead of one. The two predicates have different `bind` cases that don't compose into a single inductive argument. They're also useful separately: hiding (§ 5) needs only locality, not decomposition existence.

4.3 Coverage

Table 1 lists the trace monad's primitives and combinators with their locality and decomposition existence premises, falling into three categories: (i) Trace-empty primitives (`return`, `fail`, `select`, `assert`, etc.) have no premises since their traces have no steps to decompose. (ii) `Me`-emitting primitives (`commit_step` and the atomic-commit family) require the read or write function to be local on the composed state and preserve I. (iii) `Env`-emitting primitives (`env_step`, `env_steps`, `interference`) have separate premises for locality and decomposition. Locality requires $I \triangleright R$, propagating I through `Env` steps. Decomposition requires a frame-rely compatibility condition: every global R_g-step must split into a local R-step and a re-formed frame still satisfying F.

The `whileLoop` case. The l4v `whileLoop`'s inductive definition doesn't fit the layered structure of `is_local` and `decomp_exists` directly. We reduce it to an iteration-count form by binding over a non-deterministic count. The table's `whileLoop` entry follows.

5 The n-ary composition pipeline

There are two ways to compose n threads under LRG. The first, *pairwise*, treats parallel composition as a binary operator: the n-thread spec is built by repeatedly pairing two specs via Par (§ 2.2) and Hide, with Frame lifting each input to the composed state at every pairing. Mechanising it requires two substantial proof obligations, `is_local parallel` and `decomp_exists parallel`. The second, *n-ary* (which we adopt), frames each thread before any parallel composition, then combines all framed specs in one shot via a single top-level `parallel_n`. No `parallel` term is ever framed in the n-ary way, so neither lemma is needed.

5.1 The pipeline

The composition pipeline is summarised in Figure 2. Each per-thread local RG spec is first lifted by Frame; Par-N then composes the lifted specs in one shot; an optional consequence step (Csq,

Primitive / combinator	*Locality premises*	*Decomposition premises*
`return`, `fail`, `select`	—	—
`assert` P, `when` $c\,m$, `unless` $c\,m$, `maybeM` $v\,m$	m local	m decomp
`commit_step`	—	—
`modify_commit` f	f local, disj, I-pres	f local, disj
`read_commit` h	h local	—
`rmw_commit` f	write local, disj; read local; I-pres	write local, disj
`cas` $c\,f$	c local; conditional f local, disj, I-pres	c local; conditional f local, disj
`env_step`	$I \triangleright R$	frame-rely compat
`env_steps`, `interference`	$I \triangleright R$	$I \triangleright R$, frame-rely compat
`bind` $f\,g$	f, g local	f decomp; g local, decomp
`condition` $P\,L\,M$	P local; L, M local	P local; L, M decomp
`liftM` $f\,m$, `sequence` ms, `mapM` $f\,xs$, `foldM` $f\,a\,xs$	m/f local	m/f local, decomp
`whileLoop` $C\,B$	C local; B local	C local; B local, decomp

Table 1. Locality and decomposition coverage. "disj" is that the write keeps the local part disjoint from the frame; "I-pres" is invariant preservation by the write.

which strengthens pre/rely or weakens guar/post) followed by Hide gives the externally-visible RG spec.

5.2 The frame rule

The framing step lifts a per-thread spec on its local view to a spec on the composed state. It attaches a frame predicate F to the pre/post via $*$. The per-thread rely R_1 and guarantee G_1 are paired with frame-side actions R_2 and G_2 via $*_A$. The rule combines a thread-side fence pair $(I_1 \triangleright R_1, I_1 \triangleright G_1)$ with a frame-side fence pair $(I_2 \triangleright R_2, I_2 \triangleright G_2)$, the locality and decomposition predicates of § 4, and a stability obligation on F.

$$\frac{\begin{array}{c}\{P\},\{R_1\}\ f\ \{G_1\},\{Q\} \qquad \texttt{is_local}\,I_1\,R_1\,f \qquad \texttt{decomp_exists}\,R_1\,I_1\,(R_1 *_A R_2)\,F\,f \\ I_1 \triangleright R_1 \qquad I_1 \triangleright G_1 \qquad I_2 \triangleright R_2 \qquad I_2 \triangleright G_2 \qquad \mathit{sta}\,F\,R_2 \qquad P \le I_1 \qquad F \le I_2\end{array}}{\{P * F\},\{R_1 *_A R_2\}\,f\,\{G_1 *_A G_2\},\{Q *_Q F\}}\ \textsc{Frame}$$

The premises split cleanly. The first three carry the per-thread content: a local-view spec, locality, and decomposition existence against the lifted global rely. The fence pairs commit each side to a precise invariant (I_1 for the thread, I_2 for the frame), without which $*_A$-splitting the rely and guarantee would be ambiguous (§ 3.2); they also give G_1-preservation of I_1 via the endpoints clause. The stability premise $\mathit{sta}\,F\,R_2$ is the separation-logic obligation that the frame predicate is preserved by the frame-side rely. The remaining two premises bridge invariants ($P \le I_1$, $F \le I_2$). The frame property of $*_A$ (Lemma 2) discharges the trace-level obligations.

5.3 The hiding rule

The hiding rule is LRG's information-hiding move at the action-relation level. It applies when the rely splits as $R_1 *_A R_2$, where R_1 is the rely on the globally-shared state and R_2 is the rely on a locally-shared region (shared between a sub-group of threads but private from the rest of the system). Hide replaces R_2 with id, internalising the I_2 region so that the externally-visible

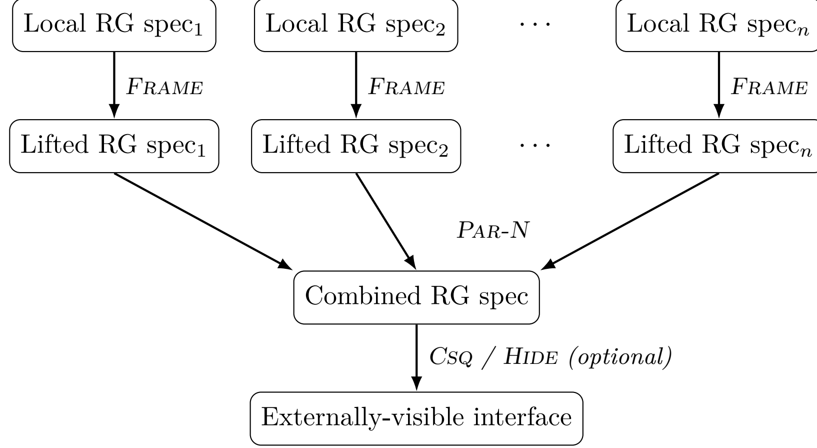

Figure 2. The n-ary RG composition pipeline. Each box is an RG quintuple $\{P\}, \{R\}\, f\, \{G\}, \{Q\}$ over the corresponding state view. Framing happens once per thread, before any parallel composition; Par-N composes all threads in a single step.

spec no longer mentions R_2.

$$\frac{\begin{array}{c}\{P\}, \{R_1 *_A R_2\}\ f\ \{G_1 *_A G_2\}, \{Q\} \\ I_1 \triangleright R_1 \qquad I_2 \triangleright R_2 \qquad I_1 \triangleright G_1 \qquad I_2 \triangleright G_2 \\ P \leq I_1 * I_2\end{array}}{\{P\}, \{R_1 *_A id\}\ f\ \{G_1 *_A G_2\}, \{Q\}}\ \textsc{Hide}$$

The rule tightens the rely from $R_1 *_A R_2$ to $R_1 *_A id$: if the thread is valid against environments that may take R_2-transitions on the I_2 region, then it is also valid against environments that leave the I_2 region unchanged. The conclusion's rely no longer mentions R_2. Soundness on traces follows from the fence and invariant premises (Lemma 3).

In the pairwise pipeline, Hide works naturally with the binary Par rule: each binary composition can be followed by a Hide step that internalises the locally-shared region. Our n-ary Par-N rule (§ 5.5) takes a different shape, producing a spec under an n-ary conjunction of $*_A$-shaped relies (one per thread's view), which does not fit Hide's binary form. An n-ary version of Hide is future work; § 6 shows how the case study works around the gap.

5.4 Defining `parallel_n`

`parallel_n` is defined as a right-associated fold over the thread list:

$$\begin{aligned} \texttt{parallel_n}\,[\,] &= \lambda_.\,\emptyset \\ \texttt{parallel_n}\,[t] &= t \\ \texttt{parallel_n}\,(t_1 :: t_2 :: ts) &= t_1 \parallel \texttt{parallel_n}\,(t_2 :: ts). \end{aligned}$$

The right-fold choice is conventional: $\parallel$ is commutative and associative on trace sets (via the zip characterisation), so `parallel_n` is well-defined under thread reordering.

5.5 The n-ary RG rule

Given a non-empty list of threads $ts = [f_1, \ldots, f_n]$ and equal-length lists of preconditions Ps, relies Rs, guarantees Gs, and post-conditions Qs, the n-ary rule reads

$$\frac{\begin{array}{c}\forall i < n.\ \{Ps_i\}, \{Rs_i\}\ f_i\ \{Gs_i\}, \{Qs_i\} \\ \forall i \neq j.\ Gs_i \leq Rs_j\end{array}}{\{\bigwedge Ps\},\ \{\bigwedge Rs\}\ \texttt{parallel_n}\,ts\ \{\bigvee Gs\},\ \{\bigwedge Qs\}}\ \textsc{Par-N}$$

PD1	PD2	PD3
r_{12}		r_{23}
r_1	r_2	r_3

Figure 3. The topology of the 3-PD model. r_{12} and r_{23} are shared memory regions; r_1, r_2 and r_3 are PD-private memory regions.

where $\bigwedge$ and $\bigvee$ denote pointwise conjunction and disjunction across the list. FRAME is applied once per individual (non-parallel) thread; the per-thread coverage is in Table 1.

6 Case study: a 3-PD Microkit example

6.1 The system model

Microkit [24] is a framework for building static system architectures on top of the formally verified seL4 microkernel [15]. The unit of structure is the *protection domain* (PD): a single thread of control running in a fixed virtual address space. PDs are event-driven and communicate through two mechanisms: (i) *channels* carry asynchronous notifications between exactly two PDs, and (ii) *shared memory regions* are statically mapped into one or more PDs with per-mapping read/write permissions. A PD can only access memory it has explicitly mapped.

We model the concurrency-visible behaviour of a 3-PD Microkit system (Figure 3): each PD is a trace-monad program; the three PDs are interleaved by a single `parallel_n` call (§ 5); the global state is a partial map from region labels to values, partitioned into three private regions (one per PD) and two shared regions; and channel notifications are represented by a boolean flag carried in each shared region. No single PD sees the whole system, and PD2 spans both shared regions, so its local view holds two protocol invariants in a separating conjunction.

The regions hold the following data:

r_1	PD1's source queue (`int list`)
r_2	PD2's empty private slot (`unit`)
r_3	PD3's log (`int list`)
r_{12}, r_{23}	shared mailbox: data (`int`) + flag (`bool`)

Each PD body is an infinite loop whose iteration is a single `modify_commit` of a pure conditional step function; the three PDs have different updating conditions and functions:

```
pdN_step s = if pdN_test s then pdN_upd s else s
pdN_body  = whileLoop (λ_ _. True) (λ_. modify_commit pdN_step) ()

-- PD1: producer
pd1_test s = r1 s ≠ [] ∧ ¬r12_flag s             -- r1 non-empty, r12 empty
pd1_upd s  = s[r1        := tl (r1 s),           -- pop head off r1
               r12_data := hd (r1 s),            -- publish to r12
               r12_flag := True]                 -- raise r12's flag

-- PD2: filter
v_ok v     = v ≥ 0                               -- example filter predicate
pd2_test s = r12_flag s ∧ ¬r23_flag s            -- r12 full, r23 empty
pd2_upd s  = let s' = s[r12_flag := False] in    -- clear r12's flag
             if v_ok (r12_data s)
             then s'[r23_data := r12_data s,     -- forward to r23
                     r23_flag := True]           -- raise r23's flag
             else s'
```

```
-- PD3: consumer
pd3_test s = r23_flag s                          -- r23 full
pd3_upd s  = s[r3       := r3 s ++ [r23_data s], -- append to r3
               r23_flag := False]                -- clear r23's flag
```

Here we simplify the model to take one atomic step for each iteration (i.e., each `pdN_test` + `pdN_upd` is atomic). Interference happens between iterations, where the trace monad inserts `Env` steps at which PDs see each other's writes through their relies.

Before invoking the n-ary pipeline, we discharge two obligations. First, the global state record instantiates the separation algebra of § 2.3, with disjointness and merging defined field-wise on its option-typed fields. Second, each `pdN_body` satisfies `is_local` and `decomp_exists`, directly from Table 1's `whileLoop` and `modify_commit` coverage.

6.2 The n-ary pipeline applied

The end-to-end proof composes the three PDs in four steps, following the pipeline of Figure 2.

Local RG (×3). The shared regions each carry a two-party protocol expressed as an invariant. r_{23} requires non-negative data when its flag is up:

$$I_{23}(s) \;\equiv\; r_{23} \in \mathrm{dom}(s) \;\wedge\; \big(\texttt{r23_flag}(s) \implies \texttt{r23_data}(s) \geq 0\big).$$

r_{12} has only the structural part, $I_{12}(s) \equiv r_{12} \in \mathrm{dom}(s)$: it carries arbitrary integers from PD1, and PD2's `v_ok` filter is what ensures only non-negative values reach r_{23}. PD1's guarantee on r_{12} leaves it untouched or, when the flag is down, atomically writes a value and raises the flag; PD2's either leaves it untouched or clears the flag after reading. PD1's rely is exactly PD2's guarantee, and vice versa; both actions are fenced by I_{12}. The protocol on r_{23} is symmetric, with PD2 in the writer role and PD3 in the reader role.

Each PD's local spec is on its footprint. PD1's spec, on footprint $r_1 \oplus r_{12}$, is

$$\{I_{12}\},\, \{R_1\}\; \texttt{pd1_body}\; \{G_1\},\, \{Q_1\}$$

where R_1 is "r_1 unchanged, and PD2 acts on r_{12} satisfying I_{12}"; G_1 is "r_1 shrinks by at most one head, and PD1 acts on r_{12} satisfying I_{12}"; and Q_1 is the body's (vacuous) post for the non-terminating loop. PD3 is symmetric on $r_3 \oplus r_{23}$. PD2 owns both shared regions: footprint $r_2 \oplus r_{12} \oplus r_{23}$ and precondition $I_{12} * I_{23}$ (fenced by Lemma 1). Each PD's local RG spec is verified against its trace-monad body via the `whileLoop` rule over `modify_commit`. This gives a local RG judgment per PD.

Frame (×3). Each local spec is lifted to the composed state by Frame (§ 5.2) against a per-PD frame predicate F_N describing the other two PDs' private footprint. For instance, F_1 asserts that $r_2, r_3 \in \mathrm{dom}(s)$, $r_{12} \notin \mathrm{dom}(s)$, and I_{23} holds on the frame. For each PD, the frame-side guarantee is id (the thread leaves the frame alone, since its `Me` steps only act on its own footprint), and the frame-side rely lets the environment modify the frame's private fields while preserving F_N. The frame's precision, stability, and fence obligations follow from Frame's premises (§ 5.2). The output is three lifted RG specs.

Pairwise compatibility (×6). Par-N (§ 5.5) requires $G_i \leq R_j$ for every $i \neq j$. We prove each of the six obligations by constructing an $*_A$ witness that aligns the two sides' state splits, then checks each field's behaviour against the per-region invariants.

N-ary composition. A single Par-N application over [`pd1_body`, `pd2_body`, `pd3_body`], using the three lifted specs and six compatibility lemmas, gives the combined spec: pre, rely, and post are pointwise conjunctions over the three threads, and the guarantee is their pointwise disjunction. This list-shaped spec is projected to a single global form in § 6.3.

6.3 CSQ-cleaning to a global invariant

To obtain a global specification of the whole system, define

$$\begin{aligned} I(s) \;\equiv\; & r_1, r_2, r_3 \in \mathrm{dom}(s) \\ & \wedge\ (\forall v \in r_3(s).\, 0 \le v) \\ & \wedge\ I_{12}(s) \wedge I_{23}(s), \end{aligned}$$

R as "I at both endpoints with all private fields unchanged", and G as "I at both endpoints". Four bridge lemmas discharge the consequence step, giving the main functional property:

Theorem 1 (Main functional property)**.**

$$\{I\}, \{R\}\ \texttt{parallel_n}\,[\texttt{pd1_body}, \texttt{pd2_body}, \texttt{pd3_body}]\ \{G\}, \{\lambda_.\, I\}.$$

The post-condition is vacuous, since the bodies are infinite loops; the substantive content is that I holds at every commit point along any trace satisfying R. Concretely:

- **Mailbox safety on** r_{23}: $\texttt{r23_flag}(s) \implies \texttt{r23_data}(s) \ge 0$;
- **PD3 log non-negativity**: $\forall v \in r_3(s).\, 0 \le v$.

Both clauses are enforced by PD2's `v_ok` filter: when PD2 forwards data from r_{12} to r_{23}, it writes only non-negative values, preserving I_{23}. PD3's appends to r_3 then draw from r_{23}'s non-negative payload, keeping the log non-negative. r_{12} itself carries arbitrary integers from PD1's queue.

We use Csq here rather than Hide because Par-N's output is a conjunction of $*_A$-shaped relies (one per PD), and Hide expects a single $R_1 *_A R_2$ shape. Extending Hide to the conjunctive shape is left as future work (§ 8).

7 Related work

RG and RGSep. Rely-guarantee originates with Jones [12]. Xu, de Roever, and He [27] gave its standard semantics, and Coleman and Jones [4] a structural soundness account. RGSep [26, 25] and SAGL [9] combined RG with separation logic. We mechanise Feng's local rely-guarantee (LRG) [8], which adds fenced actions and a hiding rule that absorbs inter-module protocols into the rely.

Jackson, Murray, and Rizkallah [11] mechanise a generalised RG-and-separation union over permission algebras in Isabelle/HOL. They model program state as a (*local*, *shared*) tuple, which does not directly fit the Microkit memory-region model: in Microkit a single region can be shared by a sub-group of PDs and hidden from others. We follow Feng's LRG, fix the underlying model to the interference-friendly trace monad [23, 3], and rely on fenced actions and an explicit hiding rule. Earlier mechanised RG work includes Prensa Nieto's rely-guarantee mechanisation in Isabelle/HOL [21] and Sanan et al.'s CSim2 [22] (top-down refinement on a custom language, without separation logic).

Concurrent separation logics. A different line of work uses higher-order ghost state and user-defined invariants instead of LRG-style fencing. Caper [7] automates verification on an RGSep-flavoured core. FCSL [18] combines fine-grained concurrency with state-transition-system reasoning in Coq. TaDA [5] introduces atomic abstract triples for reasoning about atomicity and data. Iris [14, 13] generalises this line of work in a higher-order, step-indexed framework. We are not targeting Iris-style generality; instead we pick LRG as it suits the seL4 Microkit setting, where each shared region's protocol is a small, named, precise invariant, and the externally-visible interface comes from a single hiding step.

Differences from prior work. The framework is built on the existing seL4 trace monad rather than a fresh operational model. Its language-agnostic shape and built-in self-versus-environment tagging give us an intuitive RG semantics. The composition pipeline is flat: each thread is framed independently, and a single n-ary parallel rule combines them in one step. The alternative pipeline of nested binary FRAME/PAR/HIDE is left for our next steps.

8 Conclusion and next steps

We have presented an Isabelle/HOL mechanisation of rely-guarantee plus separation logic over the seL4 trace monad, with rules following Feng's LRG. Applying LRG in this setting needs locality and decomposition-existence, which we develop on top of a rely-parameterised trace decomposition (§ 3) and prove for the primitives, sequential composition, and while loops (§ 4). The 3-PD Microkit case study (§ 6) composes the three threads via the n-ary parallel rule and proves PD3's log non-negativity from the conjunction of per-thread invariants.

Mechanisation scale. Table 2 summarises the size of the development. The framework consists of roughly 4,940 lines of Isabelle/HOL; the 3-PD case study adds 1,360 lines. All entries are sorry-free.

	Lines
Separation algebra (`Sep_Algebra`)	740
Trace decomposition (`Trace_Sep`)	310
RG primitives, atomic commit, `whileLoop` reduction	1130
Locality and decomposition coverage	1380
Frame, hiding, binary and n-ary parallel	1380
Framework total	**4940**
3-PD Microkit case study	1360

Table 2. Mechanisation scale, in lines of Isabelle/HOL.

Several directions remain open. This is a first step: the framework is sound and mechanised, and the 3-PD case study demonstrates end-to-end use, but the modelling and case-study coverage are still modest. We see two main kinds of follow-ups: (i) extending the framework's locality and hiding to handle more sophisticated patterns; and (ii) scaling the modelling to realistic Microkit deployments.

Parallel locality and decomposition. Our `trace_sep` predicate uses a two-way state split (s_l, s_f) into a local sub-state and a frame sub-state, with the shared region implicit in what the rely R permits. A three-way variant naming the shared region explicitly (private + shared + frame, as in Feng's original LRG) might let binary parallel locality follow compositionally, at the cost of refactoring every primitive's coverage proofs. This would be useful for allowing the pairwise nested PAR/HIDE pipeline.

Hiding for n-ary compositions. HIDE (§ 5.3) absorbs the rely on a locally-shared region into id via a single $R_1 *_A R_2$ split, in Feng's sense. After PAR-N, the combined rely is a conjunction of $*_A$-shaped relies (one per thread), with no single locally-shared slice exposed for HIDE to absorb. The case study works around this with a CSQ projection (§ 6.3); extending HIDE to the conjunctive shape produced by PAR-N would let the pipeline close formally.

Towards realistic Microkit verification. The current model uses coarse-grained atomic steps (each iteration is a single fused test-and-update) and minimal memory-region structure.

The next steps will involve two directions of model refinement: finer-grained atomic steps, with explicit interference points between read and write and modelling of memory-ordering effects; and larger, more structured memory regions, such as queues, ring buffers, and multi-field protocols.

For realistic Microkit deployments, we would like to have: real channel primitives (notify and protected procedure call) with their own locality coverage; scaling to LionsOS [10], targeting the network subsystem; and integration with the Pancake-to-Viper component pipeline [20, 17, 28], so that thread-local proofs can be discharged at the component level and composed in Isabelle at the system level.

Acknowledgements

I thank my supervisors, Dr. Rob Sison, Dr. Thomas Sewell, and Dr. Miki Tanaka, for their guidance, and the Trustworthy Systems team for many helpful discussions. I gratefully acknowledge my scholarship support from Cyberagentur and the seL4 Software Systems Scholarship.

References

[1] Peter Aczel. On an inference rule for parallel composition. Manuscript, Manchester University. Widely cited as the origin of the program/environment trace model used in rely-guarantee semantics., 1983.

[2] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 366–378. IEEE Computer Society, 2007. `doi:10.1109/LICS.2007.30`.

[3] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *Proceedings of TPHOLs 2008*, volume 5170 of *LNCS*, pages 167–182. Springer, 2008. `doi:10.1007/978-3-540-71067-7_16`.

[4] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007. `doi:10.1093/logcom/exm030`.

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, volume 8586 of *LNCS*, pages 207–231. Springer, 2014. `doi:10.1007/978-3-662-44202-9_9`.

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

[7] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic verification for fine-grained concurrency. In *Proceedings of the 26th European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*, pages 420–447. Springer, 2017. `doi:10.1007/978-3-662-54434-1_16`.

[8] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 315–327. ACM, 2009. `doi:10.1145/1480881.1480922`.

[9] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 173–188. Springer, 2007. doi:10.1007/978-3-540-71316-6_13.

[10] Gernot Heiser et al. Fast, secure, adaptable: LionsOS design, implementation and performance. arXiv:2501.06234 [cs.OS], 2025. URL: https://arxiv.org/abs/2501.06234.

[11] Vincent Jackson, Toby Murray, and Christine Rizkallah. A generalised union of rely-guarantee and separation logic using permission algebras. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International Conference on Interactive Theorem Proving (ITP 2024)*, volume 309 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2024.23.

[12] Cliff B. Jones. Specification and design of (parallel) programs. *IFIP Congress*, pages 321–332, 1983.

[13] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi:10.1017/S0956796818000151.

[14] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of POPL 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

[15] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of SOSP 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

[16] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *Proceedings of the 3rd International Conference on Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 332–337. Springer, 2012. doi:10.1007/978-3-642-32347-8_22.

[17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.

[18] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the 23rd European Symposium on Programming (ESOP)*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014. doi:10.1007/978-3-642-54833-8_16.

[19] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *Proceedings of CONCUR 2004*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004. doi:10.1007/978-3-540-28644-8_4.

[20] Johannes Åman Pohjola et al. Pancake: Verified systems programming made sweeter. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems (PLOS)*, pages 1–9. ACM, 2023. doi:10.1145/3623759.3624544.

[21] Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *Proceedings of the 12th European Symposium on Programming (ESOP)*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003. doi:10.1007/3-540-36575-3_24.

[22] David Sanan, Yongwang Zhao, Shang-Wei Lin, and Liu Yang. CSim2: Compositional top-down verification of concurrent systems using rely-guarantee. *ACM Transactions on Programming Languages and Systems*, 43(1), 2021. doi:10.1145/3436808.

[23] Trustworthy Systems Group. l4v: seL4 verification project. https://github.com/seL4/l4v. Accessed: 2026-05-06.

[24] Trustworthy Systems Group. The seL4 Microkit manual. https://github.com/seL4/microkit. Accessed: 2026-05-06.

[25] Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, 2008.

[26] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR)*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007. doi:10.1007/978-3-540-74407-8_18.

[27] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997. doi:10.1007/BF01211617.

[28] Junming Zhao et al. Verifying device drivers with Pancake. arXiv:2501.08249v2 [cs.PL], 2025. URL: https://arxiv.org/abs/2501.08249.