Rely-Guarantee Verification of Queue Locks with Proof Support in Isabelle/HOL

Robert J. Colvin[1,2], Scott Heiner[2], Peter Höfner[3], and Roger C. Su[3]

[1] Defence Science and Technology Group, Australia
[2] University of Queensland, Brisbane, Australia
{r.colvin, s.heiner}@uq.edu.au
[3] Australian National University, Canberra, Australia
{peter.hoefner, roger.su}@anu.edu.au

Abstract. To support rely-guarantee reasoning, we present an extension to Isabelle/HOL's built-in library, which we use to verify a hierarchy of queue locks. The framework incorporates novel features of Isabelle, and enables flexible syntax, assertion-annotations, and tactics for both automated and structured proofs. Assertion-annotations enable elegant top-down specification from an abstract queue lock to a non-trivial, practical circular-buffer queue lock.

1 Introduction

Concurrent systems are deeply integrated in the society, encompassing many safety critical systems, such as automotive, aviation, and medical devices. Reasoning about these systems is challenging, especially when considering the interference between the individual *threads* that share resources and information.

Rely-guarantee (RG) reasoning, first developed in 1983 by Jones [12], provides a formal, compositional framework to reason about such interference. Each thread *guarantees* a certain behaviour, but only when it can *rely* on its *environment* (i.e. all other threads) to maintain an acceptable level of interference. The rely- and guarantee-conditions form a contract, allowing a large, complex system to be decomposed into smaller components. One can then reason about each component separately, through the component's own rely-guarantee contract.

RG reasoning is based on Hoare triples, which feature a precondition, a command, and a postcondition. It augments the precondition with a rely condition, and the postcondition with a guarantee condition, resulting in a *quintuple*. A formulation of RG was mechanised in Isabelle/HOL by Prensa Nieto [17] in 2003. Since then, Isabelle/HOL [15] has improved considerably, with advances such as the automation tool Sledgehammer [3] being released in 2011, and the tactics language Eisbach [14] in 2016. We modernise and enhance this library.

To allow for understandable and maintainable specifications, we develop flexible syntax on top of the built-in RG library to support both the 'quintuple style' and the 'VDM keyword style' of RG specification. We also enable one to factor out a shared *data-invariant*, which would otherwise need to be repeated across the specification. Additionally we extend the syntax of commands to include

embedded *annotations* (assertions). We develop novel tactics that support both automated and structured proof styles, and take advantage of annotations to discharge many trivial side conditions. All of these aim to make the specifications and proofs easier to develop and maintain. The syntax and tactics have been applied to a variety of case studies, including the examples in the built-in Isabelle library, as well as a hierarchy of queue locks; on the former, our syntax and tactics greatly simplify their proofs and improves their readability.

Outline. Section 2 recalls the relevant background on Isabelle/HOL. Section 3 then covers the basics of RG reasoning, while presenting our syntax extensions. Section 4 introduces the enhanced RG inference rules and our tactics. Section 5 specifies and verifies the Abstract Queue Lock. Sections 6 and 7 provide more complex examples of a Ticket Lock and a Circular-Buffer Lock. Section 8 provides directions for future work and concludes.

1.1 Related Work

Mechanised Rely-Guarantee. We are not aware of any work that builds on Isabelle/HOL's built-in RG library [17], except for our earlier work [5], which motivated us to extend the library more systematically.

CSimpl [20] and CSim2 [21] are RG verification frameworks in Isabelle/HOL. They derive from the sequential-programming framework Simpl [22], which supports exception-handling and aborting behaviour, among others. These extra functionalities come at a cost of high complexity in the logic. In many verification cases (such as the ones presented in this paper), the programs of interest do not involve these extra functionalities, and are thus better suited to a more lightweight framework. Compared to those works, our framework also supports command annotations, which contribute to better usability. (Simpl has also been extended to concurrency by Complx [1], which does not use RG reasoning.)

RG has been mechanised in Rocq (formerly Coq) by Zakowski et al. [24], who based the logic on an intermediate representation language, and employed it to verify a concurrent garbage collector. We chose to use Isabelle/HOL in order to leverage its automated tools (e.g. Sledgehammer) and its structured proof language. Another formulation of RG is the algebraic style of Hayes et al. [8,6], where relies, guarantees, preconditions, postconditions, and commands are all elements of an algebra. This style of RG, however, is not yet ready for practical applications, as its foundations are still being updated recently [9].

Locks. The background and history of locks are widely discussed in the literature (e.g., [10,18]), and there have been works that use various logics and tools to verify different locks. For example, the Bakery Algorithm was verified with PVS [11], the MCS Lock was verified using the 'certified concurrent abstraction layers' methodology [13], and the CLH Lock was verified with custom-built program logics [23,19]. In comparison, we verify locks with the rely-guarantee logic in Isabelle/HOL, and thus demonstrate another technique for concurrency verification. Moreover, we present the locks hierarchically from abstract to concrete, showing how our methodology can be applied to other locks and even other concurrent data structures.

2 Isabelle/HOL

The definitions and theorems in this paper are implemented in the interactive proof assistant Isabelle/HOL [15].[4] In this section, we introduce the relevant basics, using the Abstract Queue Lock as a running example.

In Isabelle/HOL, common basic types include `nat` and `bool`. Type-variables are prefixed with apostrophes (`'a`). Functions from `'a` to `'b` have type `'a ⇒ 'b`. The symbol `∘>` denotes forward function-composition: `(f ∘> g) x = g (f x)`.

New types can be defined using the keywords **type_synonym** and **datatype**. When specifying locks, we identify individual threads using the type `thread_id`, which we define as: **type_synonym** `thread_id = nat`.

For parametric types, the type parameters are written *before* the type constructors. In particular, `'a list` denotes the type of lists, where `#` denotes the *cons* operation that adds an element to the front of a list, and `@` denotes list-concatenation. Other common parametric types include sets (`'a set`), relations (**type_synonym** `'a rel = ('a × 'a) set`), and the option-type (**datatype** `'a option = None | Some 'a`).

We further define the following two notions. The abbreviation `at_head` describes when 'an element `x` is at the head of a list `ys`'. Note that both clauses must be present in `at_head` to characterise the predicate faithfully, because the term `x = hd ys` (`x` being the head of `ys`) does not imply `x ∈ set ys` in Isabelle/HOL. The notion `pred_to_rel` lifts a predicate `p` (encoded as a set) to a relation, in which `p` being true in the pre-state implies that `p` is true in the post-state.

abbreviation `at_head x ys ≡ x ∈ set ys ∧ x = hd ys`
definition `pred_to_rel p ≡ {(s,s'). s ∈ p ⟶ s' ∈ p}`

2.1 Records

New types with multiple fields can be defined using the keyword **record**, which is used in this paper to model the states of concurrent programs. For example, the state of the Abstract Queue Lock is defined as:

record `queue_lock = queue :: thread_id list`

This definition introduces a new type `queue_lock`, with a single field `queue`, which corresponds to a function of type `queue_lock ⇒ thread_id list`.

The built-in RG library further defines the following specialised set-builder notations, which are often used on records.

(1) The acute symbol (´) abbreviates certain function applications in a set-builder expression. For example, ⦃ `distinct ´queue ∧ length ´queue < 5` ⦄ is equivalent to `{x. distinct (queue x) ∧ length (queue x) < 5}`.

This notation is not just for the fields of a record type, but also for any function on the record type. Moreover, there can be multiple different functions prefixed by the acute symbol within one pair of double braces.

(2) For set-builder expressions describing relations, the symbols $^{\underline{o}}$ and $^{\underline{a}}$ denote variables of the pre-state (the *old*-state) and the post-state (the *after*-state).

[4] Our files can be found at http://hoefner-online.de/submissions/IsabelleLocks.zip.

For example, the following two expressions are equivalent:

```
{‖ length ᵒqueue < length ᵃqueue ‖}
{ (x1, x2). length (queue x1) < length (queue x2) }
```

This notation, ^{o}`x` and ^{a}`x`, corresponds to the standard RG-notation `x` and x'.

3 Rely-Guarantee Reasoning

Developed by Jones [12], the rely-guarantee (RG) reasoning paradigm augments a Hoare triple with a *rely* and a *guarantee*, which are relations on states and describe the allowed interaction between a thread and its *environment* (i.e. all other threads that runs in parallel). A thread requires every environment-step to satisfy the rely, and if so, each step of the thread would uphold the guarantee.

A basic sentence in RG is a quintuple that consists of a precondition `P` (a set of states), a rely-relation `R` (a relation on states), a program/command `c` (of type `'a com`), a guarantee-relation `G` (a relation on states), a postcondition `Q` (a set of states). If `c` starts in a state satisfying `P` and each step of the environment satisfies `R`, then `c` will finish in a state satisfying `Q`, with each transition satisfying `G`. Collectively, `P`, `R`, `G`, and `Q` are called the *specification components*.

In the original library [17], an RG sentence is written as `⊢ c sat [P, R, G, Q]`. In our syntax extension, a basic RG sentence is written `rely: R guar: G code: {P} c {Q}`, or more concisely `{P,R} c {G,Q}`. These are closer to the common notation used in the RG literature.

The original library defines sequential commands using a simple while-language; the type `'a com` encompasses sequential commands that act on states of type `'a`. A parallel composition combines sequential commands in a list. Annotations will be defined in Section 3.2.

3.1 Data-Invariant

Often, all four specification components contain a common *data-invariant* `I` that can be factored out. Such an RG sentence with six components is written as (1) or (2) below, both of which abbreviate (3).

(1) `{P,R} c // I {G,Q}`
(2) `rely: R guar: G inv: I code: { P } c { Q }`
(3) `{P ∩ I, R ∩ pred_to_rel I} c {G ∩ pred_to_rel I, Q ∩ I}`

Here, `pred_to_rel` lifts the predicate from set to relation, as defined just before Section 2.1. This syntactic sugar makes specifications more concise by avoiding replication of what is often a long expression, thus minimising translation errors.

3.2 Annotated Commands

When reasoning about programs, it is common and useful practice to interleave assertions amid instructions, which illustrates more clearly how the individual instructions affect the state and how they interact (e.g. [1,16]). The commands (`'a com`) in the original library do not allow for such annotations of assertions, so we define the new type *annotated commands* (`'a anncom`) below.

```
datatype 'a anncom = NoAnno 'a com | BasicAnno  'a ⇒ 'a
  | SeqAnno    'a anncom 'a set    'a anncom
  | CondAnno   'a bexp   'a anncom 'a anncom
  | WhileAnno  'a bexp   'a set    'a anncom
  | ...
```

The constructor `NoAnno` directly wraps a non-annotated command. Meanwhile, `BasicAnno f` abbreviates `NoAnno (Basic f)`, where `(Basic f)` is a non-annotated command that encodes the state-transformation function `f`, which models `SKIP`, single assignments, and multiple assignments. Multiple assignments (used in Sections 6 and 7) are performed in a single step, by combining them with function-composition and wrapping the resultant function in a Basic instruction. Such a 'multi-assignment' is used to couple auxiliary instructions with concrete instructions, or to model RMW instructions such as fetch-and-increment.

In a sequential composition of two commands (`SeqAnno c1 p c2`), the intermediate assertion `p` serves as the postcondition of `c1` and the precondition of `c2`, thus transforming the overall proof goal into two subgoals. The remaining cases follow the typical patterns of assertions, with some instruction-types omitted. A typical RG sentence on annotated commands is written: `{P, R} c {G, Q}`, and the notation for data-invariants is supported similarly as in Section 3.1, except that the invariant is pushed through the syntactic structure to be included in the annotations.

A parallel composition is defined as a list of annotated commands. The annotated commands in such a list often have the same form, as in the examples of locks later in this paper. For this type of parallel composition, we express its specification using the *multi-parallel* sentence below. Let `P`, `R`, `G`, and `Q` be the precondition, rely, guarantee, and postcondition of the 'global' parallel composition. Furthermore, the parallel composition consists of `n` threads, where each thread `i` has precondition `P i`, rely `R i`, annotated command `c i`, guarantee `G i`, and postcondition `Q i`. The multi-parallel sentence is then written as follows:

```
annotated  global_init: P  global_rely: R
  ‖ i < n @
{P i, R i} c i {G i, Q i}
global_guar: G  global_post: Q
```

The global specification components allow the parallel composition to be embedded into a wider context, thus enabling compositional reasoning.

3.3 Example (Specification of Abstract Queue Lock)

We now demonstrate the use of annotated commands and their RG sentences (Section 3.2), by specifying parts of the Abstract Queue Lock. The full specification of the Abstract Queue Lock will be completed later in Section 5.

A lock is a synchronisation mechanism to ensure that only one thread at a time can enter its 'critical section' of code and access a shared resource. Before a thread enters its critical section, it must first *acquire* the lock; if the lock is

already held by another thread, the acquiring thread waits until the lock becomes available. Once the thread has finished executing its critical section, it *releases* the lock, allowing other threads to acquire it.

The Abstract Queue Lock manages access to the critical section by a shared queue to guarantee the *first-in first-served* property. The command `acquire` consists of two steps: joining the queue, and checking whether it has reached the head of the queue. As soon as the thread is at the head of the queue, it has successfully acquired the lock. This corresponds to the following two commands.[5]

```
´queue := ´queue @ [t]   ;   WHILE (hd ´queue ≠ t) DO SKIP OD
```

The loop of the second command, which contains the empty body, is referred to as a *spinloop*. Thread `t` is said to *spin* on the value `hd ´queue ≠ t`.

The `acquire` command should only be invoked by Thread `t` when it is not already holding or queueing for the lock. This means that the precondition of `acquire`, invoked by Thread `t`, is {| `t ∉ set ´queue` |}. After the first instruction, `t` becomes a part of the queue, but is not necessarily at the head; hence, the intermediate assertion is {| `t ∈ set ´queue` |}. After `t` finishes executing `acquire`, it will be at the head of the queue (and, by definition, holding the lock); hence, the postcondition is {| `at_head t ´queue` |}.

Furthermore, each thread should only acquire the lock one at a time. That means that each thread should only occur at most once in the queue, which is formalised by the data-invariant {| `distinct ´queue` |} – see Section 3.1. Suppose for now that the rely and guarantee of `acquire` are `R` and `G` respectively. Then, the RG sentence for `acquire` is written as follows, after combining the annotated command with the pre- and postconditions.[6]

```
rely: R  guar: G  inv: {| distinct ´queue |}
anno_code:
  {| t ∉ set ´queue |}
NoAnno (´queue := ´queue @ [t]) .;
  {| t ∈ set ´queue |}
NoAnno (WHILE hd ´queue ≠ t DO SKIP OD)
  {| at_head t ´queue |}
```

Our proof assistance includes further syntactic sugar to simplify the presentation of annotated commands, but we omit that in this paper for clarity.

4 Inference Rules and Tactics

We now turn to the inference rules on RG sentences involving our extended syntax. Based on these inference rules, we then define our tactics, which support both *structured* and *semi-automatic* approaches to theorem-proving.

[5] The record-field `queue` acts on an underlying state, so the acute symbol is needed to represent this hidden function application (see Section 2.1).

[6] We have omitted some brackets to improve readability.

4.1 Inference Rules

We summarise the subset of RG inference rules we apply in this paper. RG reasoning requires that every pre- and postcondition in a proof is stable under environment interference: a set (predicate) `S` is *stable* under a transition relation `R`, if the post-state `x'` satisfies `S`, whenever the pre-state satisfies `S`.

```
definition stable :: 'a set ⇒ 'a rel ⇒ bool where
  stable S R ≡ ∀ x x' . x ∈ S ∧ (x, x') ∈ R ⟶ x' ∈ S
```

Our first rule is on Basic instructions, which encompass single or multiple assignments, `SKIP`, and state-transformations in general. To guarantee non-interference from the environment, the pre- and postconditions need to be stable under the rely. If the start state is in `P`, then `f` should result in a state in `Q`. The guarantee should contain the identity-relation restricted to `P`; this accounts for the possibility of the current thread not taking any step. Meanwhile, the state transformation `f` needs to uphold the guarantee as well.

$$\frac{\texttt{stable P R} \quad \texttt{stable Q R} \quad \texttt{P} \subseteq \{\!|\, \texttt{´f} \in \texttt{Q} \,|\!\} \quad \forall \texttt{s. s} \in \texttt{P} \longrightarrow \texttt{(s, s)} \in \texttt{G} \quad \forall \texttt{s. s} \in \texttt{P} \longrightarrow \texttt{(s, f s)} \in \texttt{G}}{\texttt{\{P,R\} Basic f \{G,Q\}}}\ \text{(basic)}$$

The rule for spinloops is a specialised version of the original rule for `WHILE`. As before, the pre- and postconditions must be stable under the rely. The guarantee must contain the identity-relation restricted to `P`; however, as a spinloop changes nothing, we can assume that the guarantee contains the full identity relation. Finally, if a state satisfies the precondition but not the guard, then we exit the spinloop, and the same state must satisfy the postcondition.

$$\frac{\texttt{stable P R} \quad \texttt{stable Q R} \quad \texttt{Id} \subseteq \texttt{G} \quad \texttt{P} \cap \texttt{- b} \subseteq \texttt{Q}}{\texttt{\{P,R\} WHILE b DO SKIP OD \{G,Q\}}}\ \text{(spinlock)}$$

The two rules above involve non-annotated commands, as `Basic` commands and spinloops do not need annotations in practice. The annotated version of all the existing RG rules [4] can be implemented in our framework, including the well-known rule for sequential composition, where `M` is the 'middle' assertion.

$$\frac{\texttt{\{P,R\} ac1 \{G,M\}} \quad \texttt{\{M,R\} ac2 \{G,Q\}}}{\texttt{\{P,R\} ac1 ; \{M\} ac2 \{G,Q\}}}\ \text{(seq)}$$

This rule demonstrates the usefulness of annotated commands. In contrast to the original, non-annotated version, we can now explicitly characterise the mid-state as part of the RG sentence. All other rules—such as the two-branch if-block, one-branch if-block, and infinite loop—can be found in our source files.

4.2 Tactics

In interactice proof assistants, a *tactic* (or *method*) is a procedure that transforms proof goals in an attempt to solve them (semi-)automatically; either the goal is solved entirely, or some (simpler) subgoals remain.

Tactics in Isabelle/HOL were historically written in the ML programming language, which requires detailed understanding of ML and the internals of Isabelle/HOL. The Eisbach language [14] abstracts many of these advanced, ML-level features into high-level constructs in Isabelle/HOL's input language, thus

allowing for easier and more natural specification of tactics. Tactics in Eisbach are defined using the same syntax as proofs. For example,

```
method conj_solver = (rule conjI; assumption)
```

defines a tactic `conj_solver` that attempts to resolve a conjunction automatically. It uses the *structured concatenation* combinator (`;`), which applies the method `assumption` to all subgoals that emerge from the application of the rule `conjI`.

Our framework features around ten different tactics, all of which are defined in Eisbach. Here we only describe the ones we regularly use in our case-studies.

Our main tactic, `rg_anno_ultimate`, is based on the inference rules on annotated commands. It tries to apply all possible inference rules, and when it succeeds, it generates the appropriate proof skeleton. This tactic is vital during the development process, as the structured proof skeleton lets us easily identify any missing lemma or any error in the specification.

After applications of `rg_anno_ultimate`, we reach RG sentences on the indivisible `Basic` commands and spinloops. Here, we apply the tactics `method_basic` or `method_spinloop`. These basic tactics can generate proof skeletons according to the rules in Section 4.1; in simpler cases, they can be combined with built-in automatic tactics (such as `fastforce`) using the structured concatenation combinator (`;`) to discharge the goal in one line.

The tactics above are mainly aimed for the structured style of proofs. In the automated style, our tactic `rg_proof_expand` eagerly simplifies an RG sentence and its subgoals, using Isabelle/HOL's built-in automated reasoners. This tactic first applies the built-in `auto` tactic, while restricting its rule-set to a collection of RG-related lemmas—including the RG inference rules and their alternative formulations. This phase decomposes all the RG sentences into proof obligations, on which the tactic then invokes the built-in simplifier `simp`.

The automated tactic `rg_proof_expand` is useful when the specification has been finalised, and when the program of interest does not require many additional lemmas—such as the 'findP' program, which is a standard example in concurrency verification. The findP program involves `n` threads that concurrently search in an array for an entry satisfying some predicate P. Specifically, each thread `i` searches only the array-entries whose indices are congruent to `i` mod `n`.

This program was verified in the original RG library, with a proof that consists of more than 25 apply-commands, including many manual choices of primitive rules and manual instantiations of rules.

Our proof below is systematic and many side conditions are automatically discharged. The application of `rg_proof_expand` leaves us ten subgoals that purely involve list-indexing and could not be discharged by `simp` alone. The first seven subgoals are discharged by the standard method `force`. Isabelle/HOL automatically finds the proofs of the final three subgoals, using the Sledgehammer tool [3].

```
apply rg_proof_expand
        apply force+
  apply (metis linorder_neqE_nat mod_aux)
 apply (metis antisym_conv3 mod_aux)
by (metis leD mod_less_eq_dividend)
```

5 Abstract Queue Lock

We now pick up from Section 3.3, and complete the specification of the Abstract Queue Lock. We discuss the correctness properties, the rely and guarantee, and the main RG theorem. These discussions in this abstract setting will later guide us in formalising the specifications of the Ticket Lock and Circular Buffer Lock.

As discussed earlier, the main data structure of this lock is a queue, modelled as a list in Isabelle/HOL. Each thread that wishes to enter the critical section joins at the end of the list and leaves from its head. In a queue, each element can occur at most once. Hence, the data-invariant is expressed as the set of states where the queue contains distinct elements: {| `distinct ´queue` |}.

We view a thread as holding the lock if and only if that thread is at the head of the queue. As a queue has at most one unique head, this modelling choice directly upholds the *mutual exclusion property*—that 'the lock is held by at most one thread at a time'.

Contract (Rely and Guarantee). There are other desirable properties of locks that are described in terms of the interaction between a thread and its environment. When Thread `t` holds the lock, the lock cannot be taken away by the environment, and only `t` itself can release the lock; we call this the *self-releasing property*. In the context of queue locks, this translates to the statement *head stays at the head*.

The concurrent system of a lock is symmetric in that every thread has the same behaviour. This symmetry enables the rely and the guarantee to be described by a common relation, which we term the *contract*. Thread `i` relies on its contract being met, and guarantees to respect the contracts of all other threads. The latter can be expressed in terms of the `for_others` function below. It takes an indexed relation `r`, where `r i` in our context represents the contract of Thread `i`. Now, `for_others r i` is defined below as the intersection of all other threads' contracts `r j`, where `j` $\neq$ `i`. This relation is what Thread `i` must guarantee.

```
abbreviation for_others :: ('i ⇒ 's rel) ⇒ 'i ⇒ 's rel where
  for_others r i ≡ ⋂ j∈-{ i }. r j
```

Overall, the following is the contract of the Abstract Queue Lock. The first clause states that a thread cannot be added to or removed from the queue by its environment. The second clause states that the *head stays at the head*. Both are classical rely-conditions.

```
abbreviation queue_contract :: thread_id ⇒ queue_lock rel where
  queue_contract t ≡ {| (t ∈ set ᵒqueue ⟷ t ∈ set ᵃqueue)
                      ∧ (at_head t ᵒqueue ⟶ at_head t ᵃqueue) |}
```

Theorems. On the top-level, each thread repeatedly uses the lock in the pattern `WHILE True DO (acquire ; release) OD`. We omit the critical section between

`acquire` and `release`, as it does not access the lock. In the body of the infinite loop, `acquire` consists of two steps, enqueuing and spinning, while `release` consists only of the single dequeuing step. The local invariant, assertions, and contract are as presented earlier.

The top-level RG sentence is hence the global parallel theorem below. The queue is initially empty. Since there is no other actor outside of the collection of threads, the rely is the identity relation (no external interference) and the guarantee is the universal relation (all variables are potentially modified). Finally, because the outer loop never terminates (a continuously executing system), the global postcondition is left as the trivial empty set.

```
theorem qlock_global: assumes 0 < n shows
 annotated  global_init: {| ´queue = [] |}  global_rely: Id
    || i < n @ {| i ∉ set ´queue |}, queue_contract i
  WHILE True DO  {stable_guard: {| i ∉ set ´queue |} }
    NoAnno (´queue := ´queue @ [i]) .;
      {| i ∈ set ´queue |}
    NoAnno (WHILE hd ´queue ≠ i DO SKIP OD) .;
      {| at_head i ´queue |}
    NoAnno (´queue := tl ´queue) OD // {| distinct ´queue |}
  { for_others queue_contract i, {} }
  global_guar: UNIV  global_post: {}
```

For this relatively simple program, our automated tactic `rg_proof_expand` transforms this theorem into four subgoals. Among these, the first three subgoals are RG sentences corresponding to the three instructions inside the infinite loop. Note that `rg_proof_expand` decomposes the global parallel sentence into the infinite loop, and then decomposes the infinite loop into its three constituent instructions, while discharging most of the side-condition checks automatically. Now, the first two RG sentences can be discharged by single lines, while the last RG sentence can be resolved in a separate lemma using a structured proof (see our source files for more detail). The final small subgoal can also be discharged automatically. These result in the following five-line proof.

```
apply rg_proof_expand
   apply (method_rg_basic_named; fastforce)
  apply (method_spinloop; fastforce)
 using qlock_rel apply fastforce
using assms by fastforce
```

6 Ticket Lock

We now specify the Ticket Lock as an implementation of the Abstract Queue Lock. The state of the Ticket Lock consists of three fields: (1) `myticket :: thread_id ⇒ nat`, (2) `now_serving :: nat`, and (3) `next_ticket :: nat`.

Every thread locally stores a ticket number, and this collection of local variables is modelled globally by the `myticket` function. When Thread `t` joins the

queue, it sets `myticket t` to be the value `next_ticket`, and atomically increments `next_ticket`; this corresponds to the atomic Fetch-And-Add instruction, which is supported on most computer systems. Thread `t` then waits until the `now_serving` value becomes equal to its own ticket number `myticket t`. When Thread `t` leaves the queue, it increments `now_serving`.

This corresponds to the following code for `acquire` and `release`. Note that we use forward function composition to model the Fetch-And-Add atomic block.

```
acquire ≡ ((myticket t := next_ticket) ○>
            (next_ticket := next_ticket + 1)) ;
            WHILE now_serving ≠ myticket t DO SKIP OD)

release ≡ now_serving := now_serving + 1
```

Conceptually, Thread `t` is in the queue if and only if `now_serving` $\leq$ `myticket t`, and is at the head if and only if `now_serving = myticket t`.

6.1 Invariant

We now formalise the invariant of the Ticket Lock. The first three clauses are inequalities, while the last two clauses will be explained in more detail below.

```
abbreviation tktlock_inv ≡ {| ´now_serving ≤ ´next_ticket
  ∧ (1 ≤ ´now_serving) ∧ (∀ i. ´myticket i < ´next_ticket)
  ∧ bij_betw ´myticket ´tktlock_contending_set
                       {´now_serving ..< ´next_ticket}
  ∧ inj_img ´myticket positive_nats |}
```

The predicate `bij_betw f A B` holds if and only if `f` is bijective when its domain is restricted to `A` and its codomain restricted to `B`. The penultimate clause of the invariant stipulates that for every valid state `s`, the function `myticket s` is bijective between the set of queuing/contending threads (those threads whose tickets are not smaller than `now_serving`) and the set of tickets in use (those numbers from `now_serving` up to, but not including `next_ticket`). In the final clause of the invariant, `inj_img f B` holds if and only if `f` is injective when its codomain is restricted to `B`. Hence, the final clause of the invariant ensures that the function `myticket s` is injective when 0 is excluded from its codomain. In other words, all threads, whose tickets are non-zero, hold unique tickets.

6.2 Contract

The contract of the ticket lock, `tktlock_contract` below, describes the expected behaviour of the environment. The first clause ensures that the local variable `myticket t` does not change. Meanwhile, the global variables `next_ticket` and `now_serving` must not decrease, which is described by the second and third clause of `tktlock_contract`.

The last two clauses of `tktlock_contract` correspond to the two clauses of the contract of the Abstract Queue Lock, where `t` $\in$ `set queue` and `at_head t queue` under the Abstract Queue Lock respectively translate to `now_serving` $\leq$ `myticket t` and `now_serving = myticket t` under the Ticket Lock.

```
abbreviation tktlock_contract t ≡ {| °myticket t = ªmyticket t
  ∧ °next_ticket ≤ ªnext_ticket ∧ °now_serving ≤ ªnow_serving
  ∧ (°now_serving ≤ °myticket t ⟷ ªnow_serving ≤ ªmyticket t)
  ∧ (°now_serving = °myticket t ⟶ ªnow_serving = ªmyticket t) |}
```

6.3 RG Theorems

Similar to the Abstract Queue Lock, the specification of the Ticket Lock is stated as a global parallel theorem. An application of our tactic `method_anno_ultimate` generates six named subgoals. The latter five are side-condition checks, which are easily discharged by `fastforce`, with appropriate lemmas supplied. The main subgoal, presented as the following lemma, concerns the three key steps: *enqueue*,[7] *spin*, and *dequeue*.

```
rely: tktlock_contract i  guar: for_others tktlock_contract i
 inv:  tktlock_inv        anno_code:
     {| ´myticket i < ´now_serving |}
   BasicAnno ((´myticket i ← ´next_ticket) o>
              (´next_ticket ← ´next_ticket + 1)) .;
     {| ´now_serving ≤ ´myticket i |}
   NoAnno (WHILE ´now_serving ≠ ´myticket i DO SKIP OD) .;
     {| ´now_serving = ´myticket i |} }
   NoAnno (´now_serving := ´now_serving + 1)
     {| ´myticket i < ´now_serving |}
```

This RG sentence on an annotated command reduces to three subgoals with an application of `method_anno_ultimate`. These three subgoals correspond to the three instructions in the lemma; each is a RG sentence on a non-annotated, with pre- and postconditions taken from the appropriate annotations.

On each of these subgoals, our tactics `method_basic` and `method_spinloop` generate named cases, which we easily discharge after identifying and establishing the needed lemmas (possibly with the aid of Sledgehammer).

7 Circular Buffer Lock

Under the Ticket Lock, every queuing thread spins on the same 'now-serving' variable. On cache-coherent machines, this creates extra cache-updating traffic and hinders the hardware performance. This problem can be mitigated using a queue lock where every queuing thread spins on a different memory-location. In this section, we specify a Circular Buffer Lock, which is a more involved implementation of the Abstract Queue Lock, inspired by other array-based locks [2,7].

First, we define the types `thread_id` and `index` to be synonyms of the natural numbers. We also define a new type: **datatype** `flag_status = Pending | Granted`. The Circular Buffer Lock assumes a fixed number of threads, which we

[7] Note that in the *enqueue* step, a left-arrow denotes a function that updates the field of a record, and `o>` denotes forward function composition (see Section 2).

define as a constant `NumThreads`. This constant is assumed positive, which we enforce using a **locale** in Isabelle/HOL. Based on this constant, we allocate an array of size `ArraySize`, which is `NumThreads + 1`.

The state of our Circular Buffer Lock is modelled by the record `cblock_state`, which consists of the following fields:

- `myindex :: thread_id ⇒ index` – a function that, for each thread, stores an index into an array of flags.
- `flag_mapping :: index ⇒ flag_status:` – an array of size `ArraySize` that stores values of type `flag_status`.
- `tail :: index` – an index representing the tail of the queue, which is used when a thread joins the queue.
- `aux_head :: index` – an auxiliary variable that stores the index used by the thread at the head of the queue; the head of the queue spins on the flag `flag_mapping aux_head`.
- `aux_queue :: thread_id list` – the auxiliary queue of threads.
- `aux_mid_release :: thread_id option` – an auxiliary variable that signals if a thread has executed the first instruction of `release`, but not the second.

We initialise the array of flags with `Granted` in the zeroth entry and `Pending` in all other entries. The indices `tail` and `aux_head` are initialised to `0`. The queue is initially empty, and no thread is in the middle of `release`. (The definition of the initial state can be found in our source files.)

A typical state is illustrated by Fig. 1. There are two threads, with Thread 1 preceding Thread 0 in the auxiliary queue. The index held by Thread 1 is `myindex 1 = 0`. As the zeroth entry of the array of flags is `Granted`, Thread 1 holds the lock. The index held by Thread 0 is `myindex 0 = 1`. Thread 0 thus spins on the flag with index 1, and waits for it to become `Granted`.

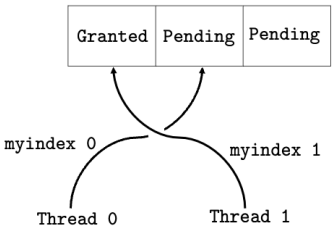

Fig. 1. Example

Similar to the previous queue locks, the `acquire` command of our Circular Buffer Lock consists of two conceptual steps. (1) To join the queue, Thread `t` stores the global index `tail` locally as `myindex t`, and atomically increments `tail` modulo the array size. (2) Thread `t` then spins on its flag, which is the entry in the array at index `myindex t`. When this flag changes from `Pending` to `Granted`, the thread has reached the head of the queue.

```
acquire ≡ Basic ((myindex[t] ← tail) ○>
                 (tail ← (tail + 1) mod ArraySize)) ;
          WHILE flag_mapping (myindex t) = Pending DO SKIP OD
```

When Thread `t` releases the lock, it sets its flag to `Pending`. Then it sets the flag of the next thread to `Granted`, which corresponds to 'next' entry in the array, modulo the array size.

```
release ≡ flag_mapping[myindex t] := Pending ;
          flag_mapping[((myindex t + 1) mod ArraySize)] := Granted
```

Auxiliary Variables. The `release` command consists of the single conceptual step of exiting the queue, but is implemented here as two separate instructions. Hence, the auxiliary variable `aux_mid_release` indicates when a thread is between the two lines of `release`, and allows us to express the assertion there.

The other two auxiliary variables, `aux_head` (the *head-index*) and `aux_queue`, store information that can in principle be inferred from the concrete variables (i.e. the non-auxiliary variables). However, explicitly recording this information as auxiliary variables greatly simplifies the verification process.

In the program, these auxiliary variables need to be updated atomically with the relevant instructions. Below is the code of `release` with the auxiliary variables included. (Auxiliary variables are added to `acquire` in a similar way.)

```
release' ≡
  Basic ((flag_mapping[myindex t] ← Pending) ○>
         (aux_mid_release ← Some t)) ;
  Basic ((flag_mapping[((myindex t + 1) mod ArraySize)] ← Granted) ○>
         (aux_queue ← tl aux_queue) ○>
         (aux_head ← (aux_head + 1) mod ArraySize) ○>
         (aux_mid_release ← None))
```

7.1 Invariant

The invariant of the Circular Buffer Lock is stated as separate parts below, all of which are of type `cblock_state set`. The first definition `invar_flag` relates `flag_mapping` with the head-index `aux_head`, and consists of two clauses. (1) At every index that is not the head-index, the flag must be `Pending`. (2) As for the head-index itself, there are two possibilities. When the thread at the head of the queue invoked `release` but has only executed its first instruction, `aux_mid_release` becomes set to `Some t`; in this case, the flag at the head-index is set to Pending, but the thread remains in the queue. In all other cases, `aux_mid_release = None`, and the flag at the head-index is always Granted

```
definition invar_flag ≡ {| (∀ i ≠ ´aux_head. ´flag_mapping i = Pending)
  ∧ (´flag_mapping ´aux_head = Pending ⟷ ´aux_mid_release ≠ None) |}
```

The next clause `invar_queue` describes the relationship between the auxiliary queue and the other variables, including the set of "used indices", whose definition can be found in our source files. The clause involving `map` relates the queuing threads to the implies a number of further properties, such as the distinctness of `aux_queue` (which mirrors the invariant of the Abstract Queue Lock), and the injectivity of `myindex` (i.e. each queuing thread has a unique index).

```
definition invar_queue ≡ {| (∀ t. t ∈ set ´aux_queue ⟶ t < NumThreads)
  ∧ map ´myindex ´aux_queue = ´used_indices |}
```

The overall invariant is the conjunction of the invariant-clauses above, with three additional simple inequalities concerning `tail`, `aux_head`, and `NumThreads`, which are omitted here and can be found in our source files.

7.2 Contract

The contract of the Circular Buffer Lock is devised along the observations: (1) local variables do not change; (2) global variables may change; and (3) auxiliary variables change similarly as in the Abstract Queue Lock.

The first two areas are covered by `contract_raw`, quoted below. The only local variable `myindex t` does not change. The global variable `tail` may change, but is not included in the contract, as changes to `tail` are not restricted. However, the other global variable `flag_mapping` is allowed to change only in specific ways. As `flag_mapping` stores information about the head of the conceptual queue, its allowed changes naturally relate to the *head stays the head* property. Under the Circular Buffer Lock, Thread `t` is at the head of the queue when `flag_mapping (myindex t) = Granted`. Meanwhile, note that `myindex t` can become outdated if Thread `t` is not in the queue. Hence, we need the premise `t ∈ set ᵒaux_queue` before the *head stays the head* statement in the final clause of `contract_raw`.

```
definition contract_raw :: thread_id ⇒ cblock_state rel where
  contract_raw t ≡ {| ᵒmyindex t = ᵃmyindex t
   ∧ (t ∈ set ᵒaux_queue ⟶ ᵒflag_mapping (ᵒmyindex t) = Granted
      ⟶ ᵃflag_mapping (ᵃmyindex t) = Granted) |}
```

For the auxiliary variable `aux_queue` we require the same two clauses as in the contract of the Abstract Queue Lock. As for `aux_mid_release`, only the head of the queue can invoke `release` and hence modify `aux_mid_release`. Therefore, the second clause of `contract_raw` has the extra equality in the consequent.

```
definition contract_aux :: thread_id ⇒ cblock_state rel where
  contract_aux t ≡ {| (t ∈ set ᵒaux_queue ⟷ t ∈ set ᵃaux_queue)
    ∧ (at_head t ᵒaux_queue ⟶
       at_head t ᵃaux_queue ∧ ᵒaux_mid_release = ᵃaux_mid_release) |}
```

7.3 RG Theorems

Similar to Section 6.3, the overall theorem is a global parallel RG sentence, where each thread runs an infinite loop of `acquire` followed by `release`. Our tactic reduces it to a number of side-condition checks and the following subgoal.

```
 rely: cblock_contract t   guar: for_others cblock_contract t
  inv:  cblock_invar t     anno_code:
      {| t ∉ set ´aux_queue |}
    BasicAnno (acq_line1 t) .;
      {| t ∈ set ´aux_queue |}
    NoAnno (WHILE ´flag_mapping (´myindex t) = Pending DO SKIP OD) .;
      {| at_head t ´aux_queue ∧ ´aux_mid_release = None |}
    BasicAnno (rel_line1 t) .;
      {| at_head t ´aux_queue ∧ ´aux_mid_release = Some t |}
    BasicAnno (rel_line2 t)
      {| t ∉ set ´aux_queue |}
```

This lemma is reduced to four RG sentences by `method_anno_ultimate`, with each subgoal corresponding to each of the four instructions. For each of these

subgoals, we then apply the suitable tactic to generate a proof skeleton, from which we identify any required lemma, patch any missing piece in the specification, and finally prove the overall theorem.

8 Conclusion

We have presented an extension of the rely-guarantee library in Isabelle/HOL. By verifying both high-level specifications (Section 5) and realistic code (Sections 6 and 7) we have illustrated its usability. This extension includes support for RG to be written in familiar syntactic forms, and specifies a global parallel composition operator with global specification components. Commands of the language can be annotated with assertions, which we use to build up a structured representation of commands with pre- and postconditions and intermediate states. Additionally, a common invariant can be separately reasoned about using this framework, instead of repeating this in the four specification components. Novel proof tactics improve readability of proofs, and better assist with specification development and automated theorem-proving.

We have applied the extended RG library to verify a collection of lock algorithms, including an Abstract Queue Lock, a Ticket Lock and a Circular Buffer Lock. In each case, we have been able to reason directly with individual lines of code, using the assertion-annotations and the improved basic syntax structure. We have shown that all three algorithms maintained their correctness properties when operated under the scenarios outlined by their specification components.

Previously, we have verified an implementation of a CLH lock [5] using some rudimentary proof assistance presented in this paper. For future work, we plan to apply our extended RG library systematically to more case-studies, such as the MCS lock and the networking buffer used in the OpenBSD operating system. We also plan to further improve the library, such as adding support for weak-memory analysis or liveness-property specification.

Acknowledgments. This work was funded by the Department of Defence, and administered through the Advanced Strategic Capabilities Accelerator.

References

1. Amani, S., Andronick, J., Bortin, M., Lewis, C., Rizkallah, C., Tuong, J.: Complx: A verification framework for concurrent imperative programs. In: Certified Programs and Proofs. p. 138–150 (2017). https://doi.org/10.1145/3018610.3018627
2. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distributed Syst. **1**(1), 6–15 (1990). https://doi.org/10.1109/71.80120
3. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: Frontiers of Combining Systems. pp. 12–27 (2011). https://doi.org/10.1007/978-3-642-24364-6_2

4. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Log. Comput. **17**(4), 807–841 (2007). https://doi.org/10.1093/logcom/exm030
5. Colvin, R.J., Hayes, I.J., Heiner, S., Höfner, P., Meinicke, L., Su, R.C.: Practical rely/guarantee verification of an efficient lock for seL4 on multicore architectures. In: The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part I. pp. 65–87. LNCS 14780 (2024). https://doi.org/10.1007/978-3-031-66676-6_4
6. Colvin, R.J., Hayes, I.J., Meinicke, L.A.: Designing a semantic model for a wide-spectrum language with concurrency. Form. Asp. Comput. **29**(5), 853–875 (2017). https://doi.org/10.1007/s00165-017-0416-4
7. Graunke, G., Thakkar, S.: Synchronization algorithms for shared-memory multiprocessors. Computer **23**(6), 60–69 (1990). https://doi.org/10.1109/2.55501
8. Hayes, I.J.: Generalised rely-guarantee concurrency: An algebraic foundation. Form. Asp. Comput. **28**, 1–22 (2016). https://doi.org/10.1007/s00165-016-0384-0
9. Hayes, I.J., Meinicke, L.A., Evangelou-Oost, N.: Restructuring a concurrent refinement algebra. In: Relational and Algebraic Methods in Computer Science. pp. 135–155 (2024). https://doi.org/10.1007/978-3-031-68279-7_9
10. Herlihy, M., Shavit, N., Luchangco, V., Spear, M.: The Art of Multiprocessor Programming. Morgan Kaufmann, 2 edn. (2020)
11. Hesselink, W.H.: Mechanical verification of Lamport's Bakery algorithm. Sci. Comput. Program. **78**(9), 1622–1638 (2013). https://doi.org/10.1016/j.scico.2013.03.003
12. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
13. Kim, J., Sjöberg, V., Gu, R., Shao, Z.: Safety and liveness of MCS lock—layer by layer. In: Programming Languages and Systems (APLAS). pp. 273–297 (2017). https://doi.org/10.1007/978-3-319-71237-6_14
14. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for Isabelle. J. Autom. Reasoning **56**, 261–282 (2016). https://doi.org/10.1007/s10817-015-9360-2
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
16. Nipkow, T., Prensa Nieto, L.: Owicki/Gries in Isabelle/HOL. In: Fundamental Approaches to Software Engineering. pp. 188–203 (1999). https://doi.org/10.1007/978-3-540-49020-3_13
17. Prensa Nieto, L.: The rely-guarantee method in Isabelle/HOL. In: Programming Languages and Systems (ESOP). pp. 348–362 (2003). https://doi.org/10.1007/3-540-36575-3_24
18. Raynal, M., Taubenfeld, G.: A visit to mutual exclusion in seven dates. Theor. Comput. Sci. **919**, 47–65 (2022). https://doi.org/10.1016/j.tcs.2022.03.030
19. Reinhard, T., Jacobs, B.: Ghost signals: Verifying termination with busy waiting. In: Computer-Aided Verification. pp. 27–50 (2021). https://doi.org/10.1007/978-3-030-81688-9_2
20. Sanán, D., Zhao, Y., Hou, Z., Zhang, F., Tiu, A., Liu, Y.: CSimpl: A rely-guarantee-based framework for verifying concurrent programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 481–498 (2017). https://doi.org/10.1007/978-3-662-54577-5_28

21. Sanan, D., Zhao, Y., Lin, S.W., Yang, L.: CSim2: Compositional top-down verification of concurrent systems using rely-guarantee. ACM Trans. Program. Lang. Syst. **43**(1) (2021). https://doi.org/10.1145/3436808
22. Schirmer, N.: A sequential imperative programming language syntax, semantics, Hoare logics and verification environment. Archive of Formal Proofs (2008), https://isa-afp.org/entries/Simpl.html, Formal proof development
23. Windsor, M., Dodds, M., Simner, B., Parkinson, M.J.: Starling: Lightweight concurrency verification with views. In: Computer-Aided Verification. pp. 544–569 (2017). https://doi.org/10.1007/978-3-319-63387-9_27
24. Zakowski, Y., Cachera, D., Demange, D., Petri, G., Pichardie, D., Jagannathan, S., Vitek, J.: Verifying a concurrent garbage collector with a rely-guarantee methodology. J. Autom. Reasoning **63**(2), 489–515 (2019). https://doi.org/10.1007/s10817-018-9489-x