TEL AVIV UNIVERSITY אוניברסיטת תל-אביב
Tel Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
The Blavatnik School of Computer Science

MODULAR VERIFICATION OF CONCURRENCY-AWARE LINEARIZABILTY

by

Nir Hemed

under the supervision of Dr. Noam Rinetzky

A thesis submitted
for the degree of Master of Science

Submitted to the Senate of Tel Aviv University
October 2016

Abstract

Modular Verification of Concurrency-Aware Linearizabilty

Nir Hemed
School of Computer Science
Tel Aviv University

Linearizability is the de facto correctness condition for *concurrent objects* (concurrent data structures). Informally, it provides the illusion that each object operation "seems to" take effect *instantaneously at a unique point in time* between its invocation and response. Hence, *by design*, it cannot describe behaviors of *concurrency-aware concurrent objects* (CA-objects) in which several overlapping operations "seem to take effect *simultaneously*".

We introduce *concurrency-aware linearizability*, a generalized notion of linearizability which allows to formally describe the behavior of CA-objects. Based on this, we develop a thread- and procedure-modular verification technique for reasoning about CA-objects and their clients. Using our new technique, we present the first proof of linearizability of the elimination stack of Hendler *et al.* in which the stack's elimination subcomponent, which is a general-purpose CA-object, is specified and verified independently of its particular usage by the stack.

Acknowledgements

This research began 3 years ago when Noam instructed me to read an article about an algorithm that used non-fixed linearization points, and contained a proof of linearizability that seemed “not-elegant”. After reading it, I asked Noam naively what is the specification of the elimination module used in the algorithm. In the following meeting, Noam declared we have found a case study.

So, first and foremost - I would like to express my sincere gratitude to my advisor Dr. Noam Rinetzky for the continuous support of my research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my MS.C. study.

Besides my advisor, I would like to thank Prof. Viktor Vafaeidis, for his insightful comments and encouragement. Viktor joined to “clear the fog”, and provided an objective support. I am grateful for his will to harness his immense knowledge in order to help build a solution to a challenging problem.

I would also like to thank my family and friends for the support they provided me through my entire life. My dedication to this thesis was a zero-sum game, and they were on the loosing party. In particular, I must acknowledge my wife Moriah, without whose love, encouragement and support, I would not have finished this thesis.

Contents

1 Introduction **1**

2 Verification Challenge **5**

2.1 Exchanger 5

2.2 Elimination Stack 7

2.3 Synchronous Queue 9

3 Technical Background **11**

3.1 Programming Language 11

3.2 Rely/guarantee reasoning 12

4 Concurrency-Aware Linearizability (CAL) **17**

5 Specifying Concurrency-Aware Concurrent Objects **21**

5.1 Logging the object interaction using an auxiliary history variable 23

5.2 Encoding interference and cooperation using rely-guarantee conditions 24

5.3 Stack specification 25

6 Overcoming the Verification Challenges **27**

6.1 Formal Verification of the Elimination Stack 27

6.2 Formal Verification of the Exchanger 29

6.3 Formal Verification of the Synchronous Queue 31

7 Related Work **33**

7.1 Concurrency-Aware linearizabilty 33

7.2 Verification of linearizable objects 34

8 Conclusions and Future Work **35**

Bibliography **37**

Chapter 1

Introduction

Linearizability [10] is a property of the externally-observable behavior of *concurrent data structures* and is considered the de facto standard for specifying concurrent objects. Intuitively, a concurrent object is linearizable if in every execution each operation seems to take effect instantaneously between its invocation and response, and the resulting sequence of (seemingly instantaneous) operations respects a given sequential specification. However, for certain concurrent objects it is *impossible* to give sequential specification: their behavior in the presence of concurrent (overlapping) operations is, and should be, *observably different* from their behavior in the sequential setting.

We refer to such objects as *Concurrency-Aware Concurrent Objects (CA-objects)*. We show that the traditional notion of linearizability is simply not expressive enough to allow for describing all the desired behaviors of CA-objects without introducing unacceptable ones, i.e., ones which their clients would find to be too lax.

An example of a CA-object is the *exchanger* object (e.g., as found in `java.util.concurrent.Exchanger`). An exchanger allows threads to pair up and *atomically* swap elements, so that either both threads manage to swap elements or none of them does. Although exchangers are widely used in practice, for example, in applications such as genetic algorithms, pipeline designs, and implementations of thread pools and highly concurrent data structures such as channels, queues, and stacks [9, 20, 21, 24], they do not have a formal specification, which precludes modular proofs of the clients. The lack of formal specification is perhaps not so surprising: Exchangers are CA-objects, and as we show, they cannot be given a *useful* sequential specification.

In this thesis, we extend the notion of linearizability to relax the requirement that specifications be sequential, and allow them to be "*concurrency-aware*" as in the following informal specification of the exchanger:

$$\begin{array}{lcl} \{\mathsf{true}\} & t_1 : x = \texttt{exchange}(v_1) \;||\; t_2 : y = \texttt{exchange}(v_2) & \{x = (\mathsf{true}, v_2) \wedge y = (\mathsf{true}, v_1)\} \\ \{\mathsf{true}\} & t : x = \texttt{exchange}(v) & \{x = (\mathsf{false}, v)\} \end{array}$$

which says that two concurrent threads t_1 and t_2 can succeed in exchanging their values but that an exchanger can also fail to find a partner thread and return back its argument. This thesis formalizes these specifications by introducing the notion of a concurrency-aware history, in which there is a total "real-time" order between *equivalence classes* of operations. Concurrency-aware linearizability is a natural generalization of the traditional notion of linearizability, where concurrency-aware specifications are used instead of sequential ones.

Providing clear and precise specifications for concurrent objects is an important goal and is a necessary step towards *thread-modular compositional* verification techniques, i.e., ones which allow to reason about each thread separately (thread-modular verification) and to *compose* the proofs of concurrent objects from the proofs of their subcomponents (compositional verification). Designing such techniques is challenging because they have to take into account the possible *interference* by other threads on the shared subcomponents without exposing the internal structure of the latter. Modular verification of CA-objects is even more challenging as it also has to consider the cases where different threads *cooperate* to perform seemingly simultaneous operations.

Consider, for example, the *elimination stack* of Hendler, Shavit, and Yerushalmi [9], which is comprised of a lock-free stack and an *elimination module*, essentially an array of exchangers. It achieves high performance under high workloads by allowing concurrent pairs of `push` and `pop` operations to *eliminate* each other and thus reducing the contention from the main stack. Essentially, enabling the pushing thread to directly exchange the pushed value with the knowledge that the popping thread will receive it. To verify the correctness of the elimination stack, one needs to ensure that every `push` operation can be eliminated by an exactly one `pop` operation, and vice versa, and that the paired operations agree on the effect of the successful exchange on the observable behavior of the elimination stack as a whole. We present a reasoning technique which allows to provide natural specifications for such intricate interactions, and modularly verify their correct implementation.

The contributions of this thesis can be summarized as follows:

- We identify the class of concurrency-aware objects in which certain operations should "seem to take effect *simultaneously*" and provide formal means to specify them using *concurrency-aware linearizability* (CAL), a generalized notion of linearizability built on top of it as a restricted form of general concurrent specifications.

- We present a simple and effective method for verifying CAL by instrumenting the programs with an auxiliary variable that logs the CA-history of operations on concurrent objects. The unique aspects of our approach are:

    1. The ability to treat a *single* atomic action as an *sequence of operations* by *different* threads which must execute completely and without interruptions, thus providing the illusion of simultaneity, and,

2. Allowing CA-objects built over other CA-objects to define their CA-history as a function over the history of their encapsulated objects, which makes reasoning about clients straightforward.

- We present proofs for several CA-object-based algorithms, most notably we present the first *modular* proof of linearizability for the elimination stack of Handler, Shavit, and Yerushalmi [9] in which (i) the elimination subcomponent is verified independently of its particular usage by the stack, and (ii) the stack is verified using an implementation-independent *concurrency-aware specification* of the elimination module.

Outline The rest of the thesis is organized as follows. We first present a chosen set of verification challenges which will serve as running examples throughout this thesis (Chapter 2). Chapter 3 lays down the required technical background in order to provide a basis for our verification technique. In Chapter 4 we formalize *Concurency-aware linearizability*, and then, based on this definition, we show how to specify *concurency-aware objects* (Chapter 5). Chapter 6 contains the formal proofs of the verification challenges. Chapter 7 summarizes related work, and Chapter 8 presents our final conclusions and related work.

Note The definition of CAL was published in [7]. The verification technique we present was published in [8]. This thesis extends beyond the publications by adding another example of a proof of a CA-object and providing a deeper background that allows understanding the formalization of our verification technique.

Chapter 2

Verification Challenge

In this chapter, we describe an implementation of an exchanger object (Section 2.1), which we use as our running example, and of three of its clients, an elimination stack (Section 2.2), and a synchronous queue (Section 2.3).

Programming Model. We assume an imperative programming language which allows to implement *concurrent objects* using object-local variables, dynamically (heap) allocated memory and a static (i.e., apriori fixed) number of concurrent subobjects. A program is comprised of a parallel composition of sequential commands (threads), where each thread has its own local variables. Threads share access to the dynamically allocated memory and to a fixed but arbitrary number of concurrent objects. We assume that concurrent objects follow a strict ownership (encapsulation) discipline:

1. concurrent objects can manipulated only by invoking their methods.
2. subobjects contained in an object o can be used only by o, the (unique) concurrent object that contains them. and,
3. there is a strict separation between the parts of the memory used for the implementation of different objects.

The operational semantics we use is standard, the only minor exception being that the semantics maintains a map from thread identifiers to the concurrent object they currently manipulate. We denote the object-local variables of an object o by $\mathsf{Vars}(o)$. For a more formal definition, see Appendix 3.1. For readability, we write our examples in a Java-like language.

2.1 Exchanger

Figure 2.2 shows a simplified implementation of the (wait-free) exchanger object in the `java.util.concurrent` library. A client thread uses the exchanger by invoking the `exchange()` method with a value that

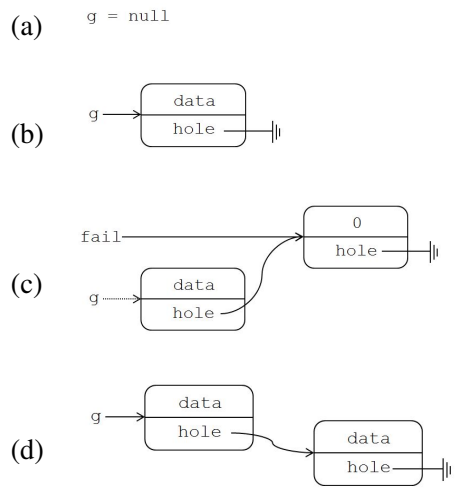


Figure 2.1: Schematic depiction of states of the exchanger.

```
1  class Offer {
2    thread_id tid;
3    int data;
4    Offer hole;
5    Offer(thread_id t, int d) {
6      tid = t;
7      data = d;
8      hole = null;
9    }
10 }
11 class Exchanger {
12   private Offer g = null;
13   private final Offer fail = new Offer(0,0);
14   (bool,int) exchange(int v) {
15     Offer n = new Offer(tid,v);
16     if (CAS(g, null, n)){ // INIT
17       sleep(50);
18       if (CAS(n.hole, null, fail)) // PASS
19         return (false,v); // FAIL
20       else
21         return (true,n.hole.data);
22     }
23     Offer cur = g;
24     if (cur != null) {
25       bool s = CAS(cur.hole, null, n); // XCHG
26       CAS(g, cur, null); // CLEAN
27       if (s)
28         return (true,cur.data);
29     }
30     return (false,v); // FAIL
31   }
32 }
```

Figure 2.2: Implementation of the exchanger.

it offers to swap. The `exchange()` method attempts to find a partner thread and, if successful, instantaneously exchanges the offered value with the one offered by the partner. It then returns a pair `(true,data)`, where `data` is the partner's value of type `int`[1]. If a partner thread is not found, `exchange()` returns `(false,v)`, indicating the client that the operation has failed. In more detail, the exchange is performed using `Offer` objects, consisting of the `data` offered for exchange and a `hole` pointer. A successful swap occurs when the `hole` pointer in the `Offer` of one thread points to the `Offer` of another thread, as depicted in Figure 2.1(d). A thread can participate in a swap in two ways, as we explain below.

The first way happens when the thread finds that the value of `g` is null (Figure 2.1(a)), it can set `g` to its `Offer` (Figure 2.2, line 16) resulting in a state like the one shown in Figure 2.1(b), and wait for a partner thread to match with (line 17). Upon awakening, it checks whether it was paired with another thread by executing a `CAS` on its own `hole` (line 18). If the `CAS` succeeds, then a match did not occur, and setting the `hole` pointer to point to the `fail` sentinel signals that the thread is no longer interested in the exchange (see Figure 2.1(c)). A failed `CAS` means that another thread has already matched the `Offer` and the exchange can complete successfully.

The second way happens when the thread finds in line 16 that the value of `g` is not null. In this case, the thread attempts to update the `hole` field of the `Offer` pointed to by `g` from its initial null value to its own `Offer` (`CAS` in line 25). An additional `CAS` (line 26) sets `g` back to null. By doing so, the thread helps to remove an already-matched offer from the global pointer; hence, the CAS in line 26 is unconditional. Moreover, this cleanup prevents having to wait for the thread that set `g` to its offer; such a wait would compromise the wait-free property of the exchanger.

2.2 Elimination Stack

The *Elimination Stack* [9] is a high-performance stack implemented using two subobjects. The first is a concurrent stack, `S`, used to implement the internal stack data structure. The internal stack is based on Treiber stack [27]. It exposes `push()` and `pop()` methods that perform CAS operations to modify the `top` of the stack. These operations fail if there is any contention on the head of the stack. The second subobject is an elimination layer `AR`, implemented as an array of exchanger objects.

Figure 2.3 shows an adapted version of the elimination stack. A `push` (respectively, `pop`) operation first tries to perform a `push` (`pop`) (lines 38, 47) operation on the main stack. In case of failure due to contention, the elimination layer is used to try and exchange a value with a thread of the opposite type. A `push` call invokes `AR.exchange` (line 40) with its input value as argument, and a `pop` call always offers a special `POP_SENTINAL` value (line 49). When `push` calls `AR.exchange`, it selects a random array entry within the elimination array's range and attempts to exchange a value with another thread.

[1] For simplicity, we consider only exchangers that swap integer values. Our results are independent of this choice.

```
1 class ElimArray {
2   Exchanger[] E = new Exchanger[K];
3   (bool, int) exchange(int data) {
4     int slot = random(0,K-1);
5     return E[slot].exchange(data);
6     }
7   }
8 class Stack {
9   class Cell {
10    Cell next;
11    int data;
12  }
13  Cell top = null;
14  bool push(int data){
15    Cell h = top;
16    Cell n = new Cell(data, h);
17    return CAS(&top, h, n);
18  }
19  (bool, int) pop(){
20    Cell h = top;
21    if (h == null)
22      return (false, 0); // EMPTY
23    Cell n = h.next;
24    if (CAS(&top, h, n))
25      return (true, h.data);
26    else
27      return (false, 0);
28  }
29 }
```

```
30 class EliminationStack {
31   final int POP_SENTINAL = INFINITY;
32   Stack S = new Stack();
33   ElimArray AR = new ElimArray();

35   bool push(int v) {
36     int d;
37     while(true){
38       bool b = S.push(v);
39       if (b) return true;
40       (b,d) = AR.exchange(v);
41       if (d == POP_SENTINAL) return true;
42     }
43   }
44   (bool,int) pop() {
45     (bool, int) (b,v);
46     while(true){
47       (b,v) = S.pop();
48       if (b) return (true,v);
49       (b,v) = AR.exchange(POP_SENTINAL);
50       if (v != POP_SENTINAL) return (true,v);
51     }
52   }
53 }
```

Figure 2.3: Elimination stack implementation (K is the number of exchanges in the elimination array.

```
1 public enum MessageType {
2     ACK, INFO
3 }
4 class Message {
5   public String body;
6   public MessageType type;
7 }
8 class MessagingQueue {
9   private Exchanger E = new Exchanger();

11  public bool send(Message m){
12    (bool, Message) (r,d);
13    m.type = MessageType.INFO;
14    do {
15      (r,d) = E.exchange(m);
16    } while (!r || d.type != MessageType.ACK)
17    return true;
18  }
```

```
19  public Message receive(){
20   (bool, Message) (r,d);
21    Message m = new Message();
22    m.type = MessageType.ACK;
23    do {
24      (r,d) = E.exchange(m);
25    } while (!r || d.type != MessageType.INFO)
26    return d;
27  }
28 }
```

Figure 2.4: Synchronous Messaging Queue

The pushing thread checks if the return value matches the `POP_SENTINAL`. Symmetrically, a popping thread that calls `AR.exchange` checks if the return value is not `POP_SENTINAL`. It is possible that the exchange will be unsuccessful, either because no exchange took place (the call to `exchange()` returned (`false`, 0)) or because the exchange was done with the same kind of operation (such as a `pop` with a `pop`). For brevity, we choose a simple approach to deal with such cases: we retry the `push` or `pop` routines.

2.3 Synchronous Queue

Synchronous queues, sometimes referred to as "zero-length queues" [21], are data structures used to transfer items between *producer* and *consumer* threads, via a standard queue API. Synchronous queues are especially efficient in high-contention systems in which data is constantly transferred and there is no need to maintain a buffer.

Figure 2.4 shows an implementation of a synchronous messaging queue that utilizes the exchanger as a sub-component. In this queue any message that is sent is acknowledged:

A *producer* thread invokes the `send(m)` operation with a message `m`. The messaging queue sets `m.type` to inform that the message is designated to be received by another thread (line 13). Then,

repeatedly, it attempts to deliver `m` to a partner thread using the exchanger (line 15)[2]. The loop terminates when the exchanger returns a value `(true,d)`, where `d` is a message with `type` "ACK", signaling that a partner thread has acknowledged the message. `send()` always returns *true* (line 17).

A *consumer* thread invokes the `receive()` operation. The messaging queue allocates a new message `m` and sets its `type` to "ACK". Message `m` is designated to serve as an acknowledgement to a producer thread (lines 21-22) that its message has been received. Then, repeatedly, it attempts to deliver `m` to a partner thread using the exchanger (line 24). The loop terminates when the exchanger returns a value `(true,d)`, where `d` is a message with `type` "INFO". This signals that message `d` has been obtained from a producer thread and thus can be returned to the consumer (line 26).

[2]Type-safety is ignored here for brevity. It is immediate to rewrite the exchanger so that it allows to exchange messages instead of integers.

Chapter 3

Technical Background

This chapter summarizes the terminology and notation used throughout the thesis and provides a short introduction to rely/guarantee reasoning, the program logic that our verification technique is based on. Our description of R/G is based on [28].

3.1 Programming Language

In this section, we present a simple concurrent imperative programming language which will be used in order to provide a uniform presentation to our reasoning technique.

We note that even though we present our logic for programs in this language, for readability we use a Java-like notation in our examples, which can be easily desugared.

We consider the following minimal imperative programming language, GPPL (standing for Generic Parallel Programming Language). Without loss of generality, we assume that the variables and methods have unique names. We do not allow to dynamically allocate concurrent objects. Thus, every program contains a fixed number of concurrent object, and every object is identified by a variable name.

Let C stand for commands, c for basic commands (e.g., assignments), B for boolean expressions, and

E for normal integer expressions. Commands, C, are given by the following grammar:

$C ::=$	$\texttt{skip}$	Empty command
	c	Basic command
	$C_1;C_2$	Sequential composition
	$C_1 + C_2$	Non-deterministic choice
	C^*	Loops
	$\langle C \rangle$	Atomic command
	$C_1 \| \| ... \| \| C_N$	Parallel composition
$c ::=$	$\texttt{assume}(B)$	Assume condition
	$x := E$	Variable assignment
	...	

A program $\mathscr{P}$ is a parallel composition of N threads, which can contain primitive commands $c \in \text{PComm}$, sequential composition $C;C$, nondeterministic choice $C + C$, iteration C^* and atomic execution $\langle C \rangle$ of C. We forbid nested atomic blocks.

GPPL is parametric with respect to the set of basic commands and expressions. The basic command $\texttt{assume}(B)$ checks whether B holds: if B is true, it reduces to $\texttt{skip}$, otherwise it diverges (loops forever). Since this dissertation discusses only partial correctness, $\texttt{assume}(B)$ is a convenient way to encode conditionals and while loops:

$$\begin{aligned} \texttt{if}(B)\, C_1\, \texttt{else}\, C_2 &\stackrel{\text{def}}{=} (\texttt{assume}(B);C_1) + (\texttt{assume}(\neg B);C_2) \\ \texttt{while}(B)\, C &\stackrel{\text{def}}{=} (\texttt{assume}(B);C)^*;\texttt{assume}(\neg B) \end{aligned}$$

Similarly, we can encode (conditional) critical regions as follows:

$$\begin{aligned} \texttt{atomic}\, C &\stackrel{\text{def}}{=} \langle C \rangle \\ \texttt{atomic}(B)\, C &\stackrel{\text{def}}{=} \langle \texttt{atomic}(B);C \rangle \end{aligned}$$

The semantic is the standard interleaving semantics, and thus omitted.

3.2 Rely/guarantee reasoning

Rely/guarantee is a compositional verification method for shared memory concurrency introduced by Jones [12]. Jones's insight was to describe interference between threads using binary relations. With single-state postconditions, we can still specify such programs, but we need to introduce a (ghost) logical variable that ties together the precondition and the postcondition. Usually, the proof rules with single-state postconditions are simpler, but the assertions become complex because of the need to introduce (ghost) logical variables [28].

Owicki-Gries

The Rely/Guarantee method can be seen as a compositional version of the Owicki-Gries method [18]. In her PhD, Owicki [17] came up with the first tractable proof method for concurrent programs. A standard sequential proof is performed for each thread; the parallel rule requires that each thread does not 'interfere' with the proofs of the other threads.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\} \quad (\dagger)}{\{P_1 \wedge P_2\}\ C_1\ ||\ C_2\ \{Q_1 \wedge Q_2\}}\ \text{OWICKI-GRIES}$$

where $(\dagger)$ is the side-condition requiring that C_1 does not interfere with the proof of C_2 and vice versa. This means that every intermediate assertion between atomic actions in the proof outline of C_2 must be preserved by all atomic actions of C_1 and vice versa. Clearly, this is a heavy requirement and the method is not compositional.

Specifications

Rely/guarantee reasoning [12] is a compositional method based on the Owicki-Gries method. The specifications consist of four components (P, R, G, Q).

- The predicates P and Q are the *pre-condition* and *post-condition*. They describe the behaviour of the thread as a whole, from the time it starts to the time it terminates (if it does). The pre-condition P, a single-state predicate, describes an assumption about the initial state that must hold for the program to make sense. The postcondition Q is a two-state predicate relating the initial state (just before the program starts execution) to the final state (immediately after the program terminates). The post-condition describes the overall effect of the program to the state.

- R and G summarise the properties of the individual atomic actions invoked by the environment (in the case of R) and the thread itself (in the case of G). They are two-state predicates, relating the state $\overleftarrow{\sigma}$ before each individual atomic action to σ, the one immediately after that action. The *rely condition* R models all atomic actions of the environment, describing the interference the program can tolerate from its environment. Conversely, the *guarantee condition* G models the atomic actions of the program, and hence it describes the interference that it imposes on the other threads of the system.

There is a well-formedness condition on rely/guarantee specifications: the precondition and the postcondition must be stable under the rely condition, which means that they are resilient to interference from the environment. Coleman and Jones [3] have stability as an implicit side-condition at every proof rule. This is, however, unnecessary: in this thesis all of the assertions are stable.

Definition 1 (Stability) *A binary relation Q is **stable under** a binary relation R if and only if $(R;Q) \Longrightarrow Q$ and $(Q;R) \Longrightarrow Q$, where ; stands for the composition relation.*

The definition says that doing an environment step before or after Q should not make Q invalid. Hence, by induction, if Q is stable, then doing any number of environment transitions before and after Q should not invalidate Q. For single state predicates, these checks can be simplified, and we get the following lemma.

Lemma 3.2.1 *A single state predicate P is **stable under** a binary relation R if and only if* $(P(\overleftarrow{\sigma}) \wedge R(\overleftarrow{\sigma}, \sigma)) \Longrightarrow P(\sigma)$.

When two threads are composed in parallel, the proof rules require that the guarantee condition of the one thread implies the rely condition of the other thread and vice versa. This ensures that the component proofs do not interfere with each other.

Proof rules

We turn to the rely/guarantee proof rules for the simple programming language introduced in 3.1. Let $C\ \text{sat}_{\text{RG}}\ (P,R,G,Q)$ stand for the judgment that the command C meets the specification (P,R,G,Q). The first rule allows us to weaken a specification. A specification is weakened by weakening its obligations (the postcondition and the guarantee condition) and strengthened by weakening its assumptions (the precondition and the rely condition). When developing a program from its specification, it is always valid to replace the specification by a stronger one.

$$\frac{C\ \text{sat}_{\text{RG}}\ (P,R,G,Q) \quad P' \Rightarrow P \quad R' \Rightarrow R \quad G \Rightarrow G' \quad Q \Rightarrow Q'}{C\ \text{sat}_{\text{RG}}\ (P',R',G',Q')}\ \text{(RG-WEAKEN)}$$

The following rule exploits the relational nature of the postcondition and allows us to strengthen it. In the postcondition, we can can always assume that the precondition held at the starting state, and that the program's effect was just some arbitrary interleaving of the program and environment actions.

$$\frac{C\ \text{sat}_{\text{RG}}\ (P,R,G,Q)}{C\ \text{sat}_{\text{RG}}\ (P,R,G,Q \wedge \overleftarrow{P} \wedge (G \vee R)^*)}\ \text{(RG-ADJUSTPOST)}$$

Then, we have a proof rules for each type of command, C. The rules for `skip`, sequential composition, non-deterministic choice and looping are straightforward. In the sequential composition rule, note that the total effect, $Q_1;Q_2$, is just the relational composition of the two postconditions.

$$\frac{}{\texttt{skip}\ \text{sat}_{\text{RG}}\ (\mathsf{true},R,G,\text{ID})}\ \text{(RG-SKIP)}$$

$$\frac{C_1\ \text{sat}_{\text{RG}}\ (P_1,R,G,(Q_1 \wedge R_2)) \quad C_2\ \text{sat}_{\text{RG}}\ (P_2,R,G,Q_2)}{(C_1;C_2)\ \text{sat}_{\text{RG}}\ (P_1,R,G,(Q_1;Q_2))}\ \text{(RG-SEQ)}$$

$$\frac{C_1 \text{ sat}_{\text{RG}}\ (P,R,G,Q) \quad C_2 \text{ sat}_{\text{RG}}\ (P,R,G,Q)}{(C_1 + C_2) \text{ sat}_{\text{RG}}\ (P,R,G,Q)}\ \text{(RG-CHOICE)}$$

$$\frac{C \text{ sat}_{\text{RG}}\ (P,R,G,(Q \wedge P))}{C^* \text{ sat}_{\text{RG}}\ (P,R,G,Q^*)}\ \text{(RG-LOOP)}$$

The rules for atomic blocks and parallel composition are more interesting. The atomic rule checks that the specification is well formed, namely that P and Q are stable under interference from R, and ensures that the atomic action satisfies the guarantee condition G and the postcondition Q. Because $\langle C \rangle$ is executed atomically, we do not need to consider any environment interference within the atomic block. That is why we check C with the identity rely condition:

$$\frac{(P;R) \Rightarrow P \quad (R;Q) \Rightarrow Q \quad (Q;R) \Rightarrow Q \quad C \text{ sat}_{\text{RG}}\ (P, \text{ID}, \text{true}, (Q \vee G))}{\langle C \rangle \text{ sat}_{\text{RG}}\ (P,R,G,Q)}\ \text{(RG-ATOMIC)}$$

When composing two threads in parallel, we require that each thread is immune to interference by all the other threads. So, the thread C_1 can get interfered by the thread C_2 or by environment of the parallel composition. Hence, its rely condition must account for both possibilities, which is represented as $R \vee G_2$. Conversely, C_2's rely condition is $R \vee G_1$. Initially, the preconditions of both threads must hold; at the end, if both threads terminate, then both postconditions will hold. This is because both threads will have established their postcondition, and as each postcondition is stable under interference, so both will hold for the entire composition. Finally, the total guarantee is $G_1 \vee G_2$, because each atomic action belongs either to the first thread or the second.

$$\frac{C_1 \text{ sat}_{\text{RG}}\ (P_1,R,G,(Q_1 \wedge R_2)) \quad C_2 \text{ sat}_{\text{RG}}\ (P_2,R,G,Q_2)}{(C_1 \| C_2) \text{ sat}_{\text{RG}}\ (P,R,(G_1 \vee G_2),(Q_1 \wedge Q_2))}\ \text{(RG-PAR)}$$

An additional operation that is addressed in this thesis is *compare and swap*. CAS takes takes three arguments: a memory address, an expected value and a new value. It *atomically* reads the memory address and if it contains the expected value, it updates it with the new value; otherwise, it does nothing:

```
bool CAS(value_t *addr, value_t exp, value_t new){
  atomic {
    if (*addr == exp) {
      *addr = new;
      return true;
    } else {
      return false;
    }
  }
}
```

Soundness and completeness

In line with the rest of the thesis this section presented rely/guarantee proof rules for partial correctness.

The rely/guarantee rules are sound [28], but intentionally incomplete: they model interference as a relation, ignoring the environment's control flow. Hence, they cannot directly prove properties that depend on the environment's control flow. Nevertheless, we can introduce auxiliary variables to encode the implicit control flow constraints, and use these auxiliary variables in the proof. Modulo introducing auxiliary variables, rely/guarantee is complete. The various completeness proofs [15] introduce an auxiliary variable that records the entire execution history. Of course, introducing such an auxiliary variable has a global effect on the program to be verified. Therefore, the completeness result does not guarantee that a modular proof can be found for every program.

Chapter 4

Concurrency-Aware Linearizability (CAL)

Linearizability [10] relates (the observable behavior of) an implementation of a concurrent object with a sequential specification. Both the implementation and the specification are formalized as *prefix-closed sets of histories*. A history $H = \psi_1 \psi_2 \ldots$ is a sequence of method *invocation* and *response* (return) actions. Specifications are given using *sequential histories*, histories in which every response is immediately preceded by its matching invocation. Implementations, on the other hand, allow for arbitrary interleaving of actions by different threads, as long as the subsequence of actions of every thread is sequential. Informally, a concurrent object OS_C is linearizable with respect to a specification OS_A if every history H in OS_C can be *explained* by a history S in OS_A that "looks similar" to H. The similarity is formalized by a real-time relation $H \sqsubseteq_{RT} S$, which requires S to be a permutation of H preserving the per-thread order of actions and the order of non-overlapping operations (execution of methods) on objects.

We claim that it is *impossible* to provide a useful sequential specification for the exchanger. Consider the client program P shown in Figure 4.1(P) which uses an exchanger object. In the same figure, we show three histories (H_1, H_2, and H_3) where an `exchange`(n) operation returning value n' is depicted using an interval bounded by an "inv(n)" and a "res(n')" actions. Note that histories H_1 and H_3 might occur when P executes, but H_2 cannot. History H_1 corresponds to the case where threads t_1 and t_2 exchange items 3 and 4, respectively, and t_3 fails to pair up. History H_2 is one possible sequential explanation of H_1. Using H_2 to explain H_1 raises the following problem: if H_2 is allowed by the specification then every prefix of H_2 must be allowed as well. In particular, history H_2' in which only t_1 performs its operation should be allowed. Note that in H_2', a thread exchanges an item without finding a partner. Clearly, H_2' is an *undesired* behavior. In fact, any sequential history that attempts to explain H_1 would allow for similar undesired behaviors. Indeed, sequential histories can explain only executions in which all `exchange` operations fail. We conclude that any sequential specification of the exchanger is either too restrictive or too loose.

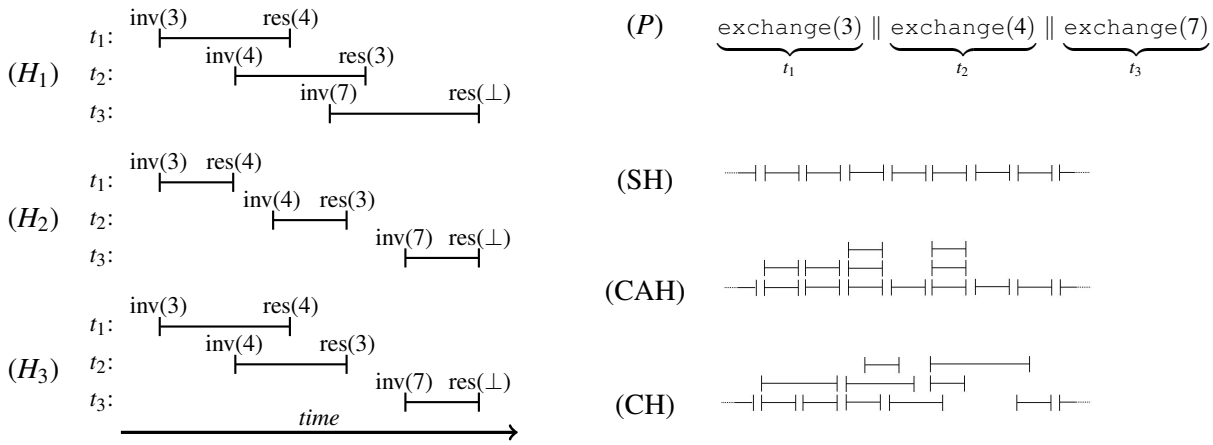


Figure 4.1: (P) a client program, (H_1) a concurrent history, (H_2) an undesired sequential history, (H_3) a CA-history, a graphical depiction of a (SH) sequential history, a (CAH) CA-history, and a (CH) concurrent history.

A Formal Definition of Concurrency-Aware Linearizability We now formalize the notion of *concurrency-aware linearizability*. We assume infinite sets of object names $o \in \mathsf{O}$, method names $f \in \mathsf{F}$, and threads identifiers $t \in \mathsf{T}$.

Definition 2 *An **object action** is either an **invocation** $\psi = (t, \mathrm{inv}\, o.f(n))$ or a **response** $\psi = (t, \mathrm{res}\, o.f \rhd n)$. We denote the* thread, object, *and* method *of ψ by $tid(\psi) = t$, $oid(\psi) = o$, and $fid(\psi) = f$, respectively.*

Intuitively, an invocation $\psi = (t, \mathrm{inv}\, o.f(n))$ means that thread t started executing method f on object o passing n as a parameter, and a *response* $\psi = (t, \mathrm{res}\, o.f \rhd n)$ means that the execution of method f' terminated with a return value n. As in [10], the observable behavior of a concurrent object is represented by a set of *histories*.

Definition 3 *A **history** H is a finite sequence of invocations and responses. We use H_i to denote the i-th action of H and $H|_t$ to denote the projection of H onto actions of thread t. A history is **sequential** if every response action is immediately preceded by a matching invocation. A history H is **well-formed** if invocations and responses are properly matched: for every thread t, $H|_t$ is sequential. A history is **complete** if it is well-formed and every invocation has a matching response (i.e, for every thread t, if $H|_t = \_\psi$ then ψ is a response). History H^c is a **completion** of a well-formed history H if it is complete and can be obtained from H by (possibly) extending H with some response actions and (possibly) removing some invocation actions. We denote by $complete(H)$ the set of all completions of H. An **object system** is a prefix-closed set of well-formed histories.*

Linearizability is a relation between **object systems**, defined using the notion of *real-time order*. In the following, we denote by _ an irrelevant expression that is implicitly existentially quantified.

Definition 4 *The **real-time order** between actions of a well-formed history H is an irreflexive partial order $\prec_H$ on (indices of) object actions: $H_i \prec_H H_j$ if there exists $i \leq i' < j' \leq j$ such that $tid(H_i) = tid(H_{i'})$, $tid(H_j) = tid(H_{j'})$, and either $tid(H_i) = tid(H_j)$ or $H_{i'} = (\_, \text{res}\_)$ and $H_{j'} = (\_, \text{inv}\_)$). A history H **agrees** with the real-time order of a history S, denoted by $H \sqsubseteq_{RT} S$, if there is a bijection $\pi : \{1, \cdots, |H|\} \to \{1, \cdots, |S|\}$ such that $\forall i.(H_i = S_{\pi(i)}) \wedge (\forall i, j.H_i \prec_H H_j \Longrightarrow S_{\pi(i)} \prec_S S_{\pi(j)})$.*

Intuitively, history S agrees with H if in both histories every thread performs the same sequence of actions, i.e., for every thread t, $H|_t = S|_t$, and the real-time order induced by H is a subset of that of S, i.e., $\prec_H \subseteq \prec_S$.

Definition 5 (Linearizability [10]) *Let OS_C and OS_A be object systems. OS_C is **linearizable** with respect to OS_A if every history $H \in OS_A$ is* sequential *and*

$$\forall H \in OS_C.\, \exists H^c \in complete(H).\, \exists S \in OS_A.\, H^c \sqsubseteq_{RT} S\ .$$

The key aspect in the definition of concurrency-aware linearizability is the notion of a *concurrency-aware history*. The latter is the main building block of a new class of specifications which strictly extend sequential ones.

Definition 6 (Concurrency-aware history) *A history H is **concurrency-aware** (**CA-History**) if for any history H_1 such that $H = H_1 \psi' \psi H_2$:*

1. *if ψ' is a response action and ψ is an invocation action then $H_1 \psi'$ is a complete history, and*
2. *if ψ' and ψ are both invocations or are both responses then $oid(\psi) = oid(\psi')$.*

*An object system is OS **concurrency-aware** if each $H \in OS$ is a concurrency-aware history.*

A concurrency-aware history allows for some operations to be executed concurrently by multiple threads on the same object. Moreover, it ensures that out of the set of threads that are operating concurrently, no thread will return before all other threads have invoked the operation (i.e., all operations must overlap pairwise). Figure 4.1 illustrates a sequential history (SH), a concurrency-aware history (CAH), and a concurrent history (CH). Note that while every sequential history is a CA-history, the opposite does not hold.

Following Definition 5, concurrency-aware linearizability of an object system is described by relating it to a concurrency-aware object system using the real-time order $\sqsubseteq_{RT}$:

Definition 7 (Concurrency-Aware Linearizability) *Let OS_C and OS_A be object systems. OS_C is **concurrency-aware linearizable (CAL)** with respect to OS_A if*

$$\forall H \in OS_C.\, \exists H^c \in complete(H).\, \exists S \in OS_A.\, H^c \sqsubseteq_{RT} S$$

and every history $H \in OS_A$ is concurrency-aware.

Thus, a *CA-linearizable object* is one that every interaction with it can be "explained" by a CA-history of some concurrency-aware object system OS_A. Note that both linearizability and concurrency-aware linearizability are defined using the same real-time order and notion of completions. The term concurrency-aware linearizability emphasizes that the specification is comprised of *concurrency-aware* histories, rather than *sequential* ones.

For example, any prefix of history H_3, depicted in Figure 4.1, is a concurrency-aware history. Furthermore, H_3 can explain the observable behavior of H_1 by requiring that `exchange(3)` and `exchange(4)` execute concurrently and, seemingly, at the same point in time. Note that every prefix of H_3 describes a possible behavior of the implementation of the exchanger. Indeed, the behavior of exchanger objects can be specified precisely using CA-histories.

Concurrency-aware traces (CA-traces) In Definition 6, we defined CA-histories using sequences of actions. In our correctness proofs, however, it is more convenient to work with an equivalent alternative presentation of complete CA-histories that is insensitive to the order of actions of overlapping operations. Thus, we use the following formulation.

Definition 8 (CA-traces) *An **operation** of a concurrent object o, denoted by $(t, f(n) \rhd n')$, is a pair of an invocation $(t, \mathrm{inv}\, o.f(n))$ and its matching response $(t, \mathrm{res}\, o.f \rhd n')$. A **concurrency-aware trace** T is a sequence of **CA-elements** where each CA-element is a pair $o.S$ of an object o and a non-empty set S of operations of o.*

CA-traces essentially provide a canonical representation of complete CA-histories, in which each CA-element represents overlapping operations on one object. For example, the CA-element $o.\{(t_1, f_1(n_1) \rhd r_1), \ldots, (t_k, f_k(n_k) \rhd r_k)\}$ represents, among others, the histories:

$$((t_1, \mathrm{inv}\, o.f_1(n_1)) \cdot \ldots \cdot (t_k, \mathrm{inv}\, o.f_k(n_k)) \cdot (t_1, \mathrm{res}\, o.f_1 \rhd r_1) \cdot \ldots \cdot (t_k, \mathrm{res}\, o.f_k \rhd,) r_k)$$

$$\text{and } ((t_k, \mathrm{inv}\, o.f_k(n_k)) \cdot \ldots \cdot (t_1, \mathrm{inv}\, o.f_1(n_1)) \cdot (t_1, \mathrm{res}\, o.f_1 \rhd r_1) \cdot \ldots \cdot (t_k, \mathrm{res}\, o.f_k \rhd,) r_k)$$

.

Given a CA-trace T, the projection of T to a thread t (resp., to an object o), denoted $T|_t$ (resp., $T|_o$), is the sequence of CA-elements of T mentioning t (resp., o). Note that the projection of a trace T to thread t returns not only the operations of t but also *all operations of other threads that are concurrent with some operation of thread t*.

Chapter 5

Specifying Concurrency-Aware Concurrent Objects

In this chapter, we gradually develop our approach for providing logical (syntactic) specifications of CA-objects by applying it to the exchanger. An accurate specification of the exchanger is one where every successful exchange corresponds to the overlapping of exactly the two operations that participated in the exchange, while an unsuccessful exchange, i.e., one that returns $(\mathsf{false}, \_)$, does not overlap with any other operation. Formally, the specification of an exchanger object $\mathtt{E}$ can be given using an object system OS_E defined as the set of prefixes of complete histories S such that $S = S^1 S^2 S^3 \cdots$ and the form of S^i is either

- $(t, \mathrm{inv}\,\mathtt{E}.\mathtt{ex}(n))(t', \mathrm{inv}\,\mathtt{E}.\mathtt{ex}(n'))(t', \mathrm{res}\,\mathtt{E}.\mathtt{ex} \rhd (\mathsf{true}, n))(t, \mathrm{res}\,\mathtt{E}.\mathtt{ex} \rhd (\mathsf{true}, n'))$, where $t \neq t'$, or
- $(t, \mathrm{inv}\,\mathtt{E}.\mathtt{ex}(n))(t, \mathrm{res}\,\mathtt{E}.\mathtt{ex} \rhd (\mathsf{false}, n))$.

It is easy to see that every history $S \in OS_E$ is a *CA-history*. Given in CA-Traces, it would be

- $E.\{(t, \mathtt{ex}(n) \rhd (\mathsf{true}, n')), (t', \mathtt{ex}(n') \rhd (\mathsf{true}, n))\}$, or
- $E.\{(t, \mathtt{ex}(n) \rhd (\mathsf{false}, n))\}$.

This specification, however, has a very global nature and is therefore cumbersome to use when reasoning about a particular exchange.

What we would like is a simpler way to describe CA-histories by focusing on the individual operations that is amenable for logical (syntactic) treatment. In the sequential setting, Hoare triples [11] are often used to specify partial correctness. Indeed, this approach was adopted to describe the set of histories in the sequential specification of concurrent objects [10]. Thus, we ask ourselves whether we can provide such a specification to the exchanger. As a first attempt we can consider the straw-man *concurrent* specification that we have already shown in Section 1:

$$\begin{array}{lll} \{\mathsf{true}\} & t_1 : x = \mathtt{exchange}(v_1) \;||\; t_2 : y = \mathtt{exchange}(v_2) & \{x = (\mathsf{true}, v_2) \wedge y = (\mathsf{true}, v_1)\} \\ \{\mathsf{true}\} & t : x = \mathtt{exchange}(v) & \{x = (\mathsf{false}, v)\} \end{array}$$

In this specification, the notation $t : r = \texttt{exchange(v)}$ is used to state that the operation `exchange` is invoked by thread t. This specification is quite intuitive as it emphasizes that *only* two threads that execute `exchange()` concurrently can match and successfully swap elements, while a thread that failed to find a partner fails to swap. The problem, however, is that it is difficult to give this specification a formal meaning which can be used efficiently in proofs as discussed below.

Interpreting the specification as normal Hoare triples is insufficient, because it precludes thread-modular compositional reasoning: Firstly, it is not possible to reason about the body of one thread in a sequential manner as the specification explicitly contains the parallel composition operator. Secondly, using it leads to proofs where the local variables of one thread are mentioned in the body of another. Thirdly, a less obvious problem is that it is not easy to adapt the concurrent specification of the exchange operations to an *agreed asymmetric view* in the context in which it is used. For example, when verifying the elimination stack, we would like to pretend that the exchange operation of the pushing thread happens right before that of the popping thread. This would allow to correctly interpret the simultaneous exchange operations as an elimination of a `push(`n`)` operation by a `pop()` which returns n (see Section 2.2).

Our first attempt to overcome the first two problems is to extend the specification with a variable $\mathscr{A}$ recording the *abstract state* of the exchanger, which we can take to be the set of exchanges that have happened. This allows us to specify the exchange method from the point of view of the thread calling it:

$$\{\mathsf{true}\}\ \ t\colon x = \texttt{exchange}(v)\ \ \left\{\begin{array}{l}(\exists r, t'.\ x = (\mathsf{true}, r) \wedge \{t : v,\ t' : r\} \in \mathscr{A} \wedge t' \neq t)\\ \vee\ (x = (\mathsf{false}, v) \wedge \{t : v\} \in \mathscr{A})\end{array}\right\}$$

This specification says that the exchange records itself in the abstract value $\mathscr{A}$, and in the successful case some other exchange, the one whose value is returned, is also recorded in $\mathscr{A}$. The new specification overcomes the first two problems, it does not handle the third problem. The specification should account for clients who wish to interpret concurrent exchanges as if they happened one after the other.[1]

We are now ready to present the specification of the exchanger in which we finally address the third problem. But before, we make one final observation regarding the use of abstract values in reasoning about (concurrent) data structures. These values are merely means to record the sequence of operations applied on the concurrent objects. In certain cases, using this technique might be undesirable. Thus, we adopt a more general approach where we symbolically record the CA-trace witnessing that the execution is valid. Technically, we provide specifications in the standard form of Hoare triple to record the local view of every thread, and rely-guarantee conditions, R and G, to specify the interaction. This allows to describe the behavior of the exchanger using the following specification, which we gradually explain

[1] Formally, this specification suffers from an ABA-like problem as it allows more than two threads to match with each other. This can be readily fixed by either assuming that the exchanged values are unique or by adding temporal modalities. (This deficiency is not problematic as we do not use this specification.) In fact, the history variable can be seen a simple specialized way for using temporal reasoning.

below.

$$R^{\mathtt{tid}}, G^{\mathtt{tid}} \Vdash \{\mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T\} \quad \mathtt{tid}\colon \mathit{ret} = \mathrm{E}.\mathtt{exchange}(v) \left\{ \begin{array}{l} (\exists t', v'.\ \mathit{ret} = (\mathsf{true}, v') \wedge \mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \rhd \mathsf{true}, v'), (t', \mathtt{ex}(v') \rhd \mathsf{true}, v)\}) \wedge t' \neq \mathtt{tid}) \\ \vee\, (\mathit{ret} = (\mathsf{false}, v) \wedge \mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \rhd \mathsf{false}, v)\})) \end{array} \right\}$$

5.1 Logging the object interaction using an auxiliary history variable

To specify and verify CAL, we instrument the program with an auxiliary variable $\mathscr{T}$ that records the CA-trace that is equivalent to a given concurrent history. Our idea is to add auxiliary assignments to the programs that append CA-elements to $\mathscr{T}$ at the appropriate points.

Since multiple objects can manipulate $\mathscr{T}$, the specification of an object o should not directly mention $\mathscr{T}$, but rather its view on $\mathscr{T}$, which we denote as $\mathscr{T}_o$. A simple choice would be to define this view to be $\mathscr{T}|_o$, the projection of the trace to the CA-elements of object o. While this works for objects that do not depend on subobjects, it does not enable compositional verification of higher-level objects. The reason is that the desired equivalent CA-trace of a higher-level object is typically determined by the CA-traces of its subobjects. If, however, we want to verify an object compositionally, we are not allowed to peek into the implementations of its subobjects in order to add auxiliary assignments to $\mathscr{T}$.

To allow for object-compositional reasoning, we require for each object o to provide a function F_o from the CA-elements of its immediate subobjects to CA-traces containing only operations for o. Given such a function F_o, we define its total extension $\hat{F}_o$ as the function that given an element a returns $F_o(a)$ if this is defined or a otherwise. Note that $\hat{F}_o$ is idempotent and that for disjoint objects o and o', $\hat{F}_o \circ \hat{F}_{o'} = \hat{F}_{o'} \circ \hat{F}_o$. Next, we define $\overline{F_o}$ to recursively apply $\hat{F}_{o_i}$ for all objects o_i encapsulated by o. This is defined by induction on the object nesting depth. At each level, if o depends on objects $o_1, \ldots, o_n$, we define

$$\overline{F_o} \triangleq \hat{F}_o \circ (\overline{F_{o_1}} \circ \ldots \circ \overline{F_{o_n}})$$

Again, because of encapsulation, the order in which $\overline{F_{o_1}}$ to $\overline{F_{o_n}}$ are composed does not matter. Finally, we define

$$\mathscr{T}_o \triangleq \overline{F_o}(h)$$

To summarize, our solution is to specify the operations using the auxiliary variable logging the operation that took affect. Introducing $\mathscr{T}$ makes the specification sensible because it only mentions code of the thread we reason about. However, the trace does not allow to interpret operations on lower-level objects as affecting higher level ones. In order to achieve compositional reasoning, we augment every object with an adaptation function which interprets the effect of operations of subcomponents as affect on its own state.

Example 1 *Consider the trace captured the CA-Trace*

$$E.\{(t, ex(n) \rhd (\mathsf{true}, n')), (t', ex(n') \rhd (\mathsf{true}, n))\}$$

The adaptation function of the elimination stack can interpret this trace in order to formally express the effect of an exchange done by a pushing thread and a popping thread:

$$F_{\mathrm{ES}}\left(E.\left\{\begin{array}{l}(t, ex(n) \rhd \mathsf{true}, \infty),\\ (t', ex(\infty) \rhd \mathsf{true}, n)\end{array}\right\}\right) \triangleq (\mathrm{ES}.(t, push(n) \rhd \mathsf{true})) \cdot (\mathrm{ES}.(t', pop() \rhd \mathsf{true}, n)) \quad \textit{provided } n \neq \infty$$

All other traces provided by the exchanger to the elimination stack can be ignored: $F_{\mathrm{ES}}(\mathrm{E}.\_) \triangleq \varepsilon$*, e.g. unsuccessful exchanges have no effect.*

To the best of our knowledge this adaptation function, as a formalization of the ability to expose and use interfaces of subobjects and superobjects in a formal proof, is novel.

5.2 Encoding interference and cooperation using rely-guarantee conditions

Next, since the exchange operations are concurrent, we cannot merely give a sequential specification in Hoare logic, but instead use rely/guarantee reasoning [12], a more expressive formalism that allows expressing concurrent specifications. In rely/guarantee, each program C is specified not only by a precondition P and a postcondition Q, but also by a rely condition R and a guarantee condition G, which we have written as $R, G \Vdash \{P\}\ C\ \{Q\}$. These rely/guarantee conditions are parameterized by thread identifiers and describe the interaction between threads. For a thread t, the rely condition R^t records the interference that t might incur from the other threads, while the guarantee G^t records the effect t is allowed to have on other threads. Rely/guarantee gives thread-modular reasoning as it exposes the interaction between threads without referring to the code of other threads (see Section 3.2).

Internally, in the verification of the exchanger, these conditions will correlate the concrete state manipulated by the algorithm and the recorded history. For example, they require that when a thread successfully modifies the `g.hole` to point to its own offer, it also logs in $\mathscr{T}$ a CA-element which records the successful exchange (see Section 6.2).

From the client's perspective, however, the internal definitions of $R^{\mathtt{tid}}$ and $G^{\mathtt{tid}}$ are irrelevant. For them to be usable, however, they should adhere to a few minimal constraints, which are common for any object o:

- For every two distinct threads $t \neq t'$, we should have $G^t \Rightarrow R^{t'}$. This is the standard requirement in rely/guarantee reasoning ensuring that multiple methods of o may be invoked in parallel.

- The methods of o may only modify the auxiliary history variable, $\mathscr{T}$, the parts of the memory used in its own representation, and (via method calls) the state of its concurrent subobjects. Moreover, they may only append onto $\mathscr{T}$ entries corresponding to o and its encapsulated objects, and pertaining only to threads currently executing one of its methods. Formally, this is

$$G^t \Rightarrow (\exists T.\ \ \mathscr{T} = \overleftarrow{\mathscr{T}} \cdot T \wedge T = T|_o = T|_t \wedge \forall x \notin \{h\} \cup \mathsf{Vars}(o).\ x = \overleftarrow{x}\,)$$

  We use the hook arrow notation to represent the value of a program variable in prior state.
- The object o does not assume anything about the private state of other objects, and allows them to extend the auxiliary history variable, $\mathscr{T}$. Formally, we require that $\text{IRRELEVANT}_o^t \Rightarrow R^t$ where

$$\text{IRRELEVANT}_o^t \triangleq \exists T.\ \ \mathscr{T}_o = \overleftarrow{\mathscr{T}_o} \cdot T \wedge T|_t = T|_o = \varepsilon \wedge (\forall x \in \mathsf{Vars}(o).\ x = \overleftarrow{x}\,)$$

Finally, since there are may be multiple threads running concurrently, the precondition and postcondition of the `exchange` method, we take the projection of $\mathscr{T}_\text{E}$ to the thread of interest (i.e., $\mathscr{T}_\text{E}|_{\texttt{tid}}$). As is standard in Hoare logic, we use the logical variable T to record the initial value of $\mathscr{T}_\text{E}|_{\texttt{tid}}$.

5.3 Stack specification

The specification of the elimination stack as well as the ordinary concurrent stack it contains is expressed in a similar style. Technically, we say that a sequential history of stack operations is *well-defined* over an initial stack, if executing the (successful) operations in order is possible and yields the same results for the *pop* operations. A history is *well-formed* with respect to the stack object, denoted $\text{WF}_\text{S}(H)$, if $H|_\text{S}$ is a sequential well-defined history over the empty initial stack. The specifications for the stack methods $f \in \{push, pop\}$ are:

$$R^t, G^t \Vdash \{\text{WF}_\text{S}(\mathscr{T}_\text{S}) \wedge \mathscr{T}_\text{S}|_t = H\}\ \ t\colon r := \text{S}.f(n)\ \ \{\text{WF}_\text{S}(\mathscr{T}_\text{S}) \wedge \mathscr{T}_\text{S}|_t = H \cdot (\text{S}.\{(t, f(n) \rhd r)\})\}$$

The abstract value of a concurrent object, if needed (e.g., to determine the result of a *pop()* operation), can be "computed" by replaying the logged actions.

In the following chapter, we exemplifies our approach by applying it to modularly reason about our three verification challenges (see Chapter 2)

Chapter 6

Overcoming the Verification Challenges

In this section, we prove that the elimination stack is linearizable in a *modular* way 1by verifying each of objects—the exchanger, the elimination array, the central stack, and the elimination stack—separately.

6.1 Formal Verification of the Elimination Stack

We start with the elimination array, whose correctness is the simplest to demonstrate. The elimination array, AR, encapsulates an array of exchanger objects $\mathrm{E}[1],\ldots,\mathrm{E}[K]$ and exposes the same specification as a single exchanger. To verify that it conforms to its specification, we define the F_{AR} function as

$$F_{\mathrm{AR}}(\mathrm{E}[i].\mathscr{S}) \triangleq (\mathrm{AR}.\mathscr{S})$$

i.e., an exchange done by any of AR's exchanger subobjects is converted to look like an exchange on the elimination array. Note that this hides the implementation of the elimination array from its clients, in our case, the elimination stack. To verify the implementation of the elimination array, we pick $R^t_{\mathrm{AR}} \triangleq \bigwedge_i R_{\mathrm{E}[i]}$ and $G^t_{\mathrm{AR}} \triangleq \bigvee_i G^t_{\mathrm{E}[i]}$. The postcondition of AR.*exchange* follows directly from the postcondition of the $\mathrm{E}[slot]$.*exchange* by observing that $h_{AR} = \overline{F_{AR}}(h_{E[slot]})$.

Verifying that the central stack is a standard proof of linearizability, and we omit it for brevity (see, e.g. [28]). Next, we consider the elimination stack assuming that the central stack, S, and the elimination array, AR, satisfy their specifications. Given our setup, this proof is also straightforward. The key step is to define the function F_{ES} correctly:

$$F_{\mathrm{ES}}((\mathrm{S}.(t,\mathtt{push}(n) \rhd \mathsf{true}))) \triangleq ((\mathrm{ES}.(t,\mathtt{push}(n) \rhd \mathsf{true})))$$

$$F_{\mathrm{ES}}((\mathrm{S}.(t,\mathtt{pop}() \rhd \mathsf{true},n))) \triangleq ((\mathrm{ES}.(t,\mathtt{pop}() \rhd \mathsf{true},n)))$$

$$F_{\mathrm{ES}}\left(AR.\left\{\begin{array}{l}(t,ex(n) \rhd \mathsf{true},\infty),\\(t',ex(\infty) \rhd \mathsf{true},n)\end{array}\right\}\right) \triangleq (\mathrm{ES}.(t,\mathtt{push}(n) \rhd \mathsf{true})) \cdot (\mathrm{ES}.(t',\mathtt{pop}() \rhd \mathsf{true},n)) \quad \text{provided } n \neq \infty$$

$$F_{\mathrm{ES}}(\mathrm{S}.\_) \triangleq \varepsilon \qquad F_{\mathrm{ES}}(\mathrm{AR}.\_) \triangleq \varepsilon$$

```
1  class ElimArray {
2    Exchanger[] E = new Exchanger[K];
3    (bool, int) exchange(int data) {
4      int slot = random(0,K-1);
5      return E[slot].exchange(data);
6  } }
7  class Stack {
8    class Cell {Cell next; int data;}
9    Cell top = null;
10   bool push(int data){
11     Cell h = top;
12     Cell n = new Cell(data, h);
13     return CAS(&top, h, n);
14   }
15   (bool, int) pop(){
16     Cell h = top;
17     if (h == null)
18       return (false, 0); // EMPTY
19     Cell n = h.next;
20     if (CAS(&top, h, n))
21       return (true, h.data);
22     else
23       return (false, 0);
24 } }
25 class EliminationStack {
26   final int POP_SENTINAL = INFINITY;
27   Stack S = new Stack();
28   ElimArray AR = new ElimArray();
```

```
29   {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T ∧ v < ∞}
30   bool push(int v) { int d;
31     while(true){
32       {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T ∧ v < ∞}
33       bool b = S.push(v);
34       {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T·F_ES(S.(tid, push(v) ▷ b)) ∧ v < ∞}
35       if (b) return true;
36       {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T ∧ v < ∞}
37       (b,d) = AR.exchange(v);
38       {b ∧ d = ∞ ∧ 𝒯_ES|_t = T·(ES.(t, push(v) ▷ true)) ∧ WF_ES(𝒯_ES)
          ∨ d ≠ ∞ ∧ WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T ∧ v < ∞}
39       if (d == POP_SENTINAL) return true;
40   } }
41   {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T·(ES.(t, push(v) ▷ ret))}
42   {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = H}
43   (bool,int) pop() { (bool, int) (b,v);
44     while(true){
45       (b,v) = S.pop();
46       {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T·F_ES(S.(t, pop() ▷ b, v))}
47       if (b) return (true,v);
48       {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T}
49       (b,v) = AR.exchange(POP_SENTINAL);
50       {b ∧ v ≠ ∞ ∧ 𝒯_ES|_t = T·(ES.(t, pop() ▷ true, v)) ∧ WF_ES(𝒯_ES)
          ∨ v = ∞ ∧ WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T}
51       if (v != POP_SENTINAL) return (true,v);
52   } }
53   {WF_ES(𝒯_ES) ∧ 𝒯_ES|_t = T·(ES.(t, pop() ▷ ret))}
54 }
```

Figure 6.1: Elimination stack implementation and proof outline.

This function picks as linearization points the successful pushes and pops of S, as well as a successful exchange where the exchanged values are ∞ and $n \neq \infty$. In the latter case, the push is linearized before the pop. All other operations are ignored. Similar to the previous case, we take $G^t_{\mathrm{ES}} \triangleq G^t_{\mathrm{S}} \vee G^t_{\mathrm{AR}}$ and $R^t_{\mathrm{ES}} \triangleq R^t_{\mathrm{S}} \wedge R^t_{\mathrm{AR}}$.

The proof outline of the elimination stack is fairly straightforward and shown in Figure 6.1. The only interesting part there is the translation of `exchange` operations to `push` and `pop` operations (lines 38 and 50), using $F_{\mathrm{ES}}(\cdot)$. Recall that the specification of the exchanger is

$$R^{\mathtt{tid}}, G^{\mathtt{tid}} \Vdash \{\mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T\} \quad \mathtt{tid}\colon ret = \mathrm{E}.\mathtt{exchange}(v)$$
$$\left\{\begin{array}{l}(\exists t', v'.\ ret = (\mathsf{true}, v') \wedge \mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \rhd \mathsf{true}, v'), (t', \mathtt{ex}(v') \rhd \mathsf{true}, v)\}) \wedge t' \neq \mathtt{tid}) \\ \vee\, (ret = (\mathsf{false}, v) \wedge \mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \rhd \mathsf{false}, v)\}))\end{array}\right\}$$

$$\text{INIT}^t \triangleq [\overleftarrow{g} = \mathit{null} \wedge \exists n.\ n.\mathsf{tid} = t \wedge n.\mathsf{hole} = \mathit{null} \wedge g = n]_g$$

$$\text{CLEAN}^t \triangleq [\overleftarrow{g}.\mathsf{hole} \neq \mathit{null} \wedge g' = \mathit{null}]_g$$

$$\text{PASS}^t \triangleq [g.\overleftarrow{\mathsf{hole}} = \mathit{null} \wedge g.\mathit{tid} = t \wedge g.\mathsf{hole} = \mathit{fail}]_{g.\mathsf{hole}}$$

$$\text{XCHG}^t \triangleq \left[\begin{array}{l}\exists n \neq \mathit{fail}.\ n.\mathsf{tid} = t \wedge g.\overleftarrow{\mathsf{hole}} = \mathit{null} \wedge g.\mathsf{tid} \neq t \wedge g.\mathsf{hole} = n \wedge \\ \mathscr{T} = \overleftarrow{\mathscr{T}} \cdot (\mathrm{E}.\{(g.\mathsf{tid}, \mathtt{ex}(g.\mathsf{data}) \rhd \mathsf{true}, n.\mathsf{data}), (t, \mathtt{ex}(n.\mathsf{data}) \rhd \mathsf{true}, g.\mathsf{data})\})\end{array}\right]_{g.\mathsf{hole},\mathscr{T}}$$

$$\text{FAIL}^t \triangleq \left[\exists d.\ \mathscr{T} = \overleftarrow{\mathscr{T}} \cdot (\mathrm{E}.\{(t, \mathtt{ex}(d) \rhd \mathsf{false}, d)\})\right]_{\mathscr{T}}$$

$$G^t_{\mathrm{E}} \triangleq (\text{INIT}^t \vee \text{CLEAN}^t \vee \text{PASS}^t \vee \text{XCHG}^t \vee \text{FAIL}^t) \qquad R^t_{\mathrm{E}} \triangleq (\text{IRRELEVANT}^t_{\mathrm{E}} \vee \exists t' \neq t.\ G^{t'}_{\mathrm{ex}})$$

$$J \triangleq \forall t.\ \ g \neq \mathit{null} \wedge g.\mathsf{hole} = \mathit{null} \implies \mathrm{In}_{\mathrm{E}}(g.\mathsf{tid})$$

$$A \triangleq \mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \wedge (g = \mathit{null} \vee g.\mathsf{hole} \neq \mathit{null} \vee g.\mathsf{tid} \neq \mathtt{tid}) \wedge n \mapsto \mathtt{tid}, p, \mathit{null}$$

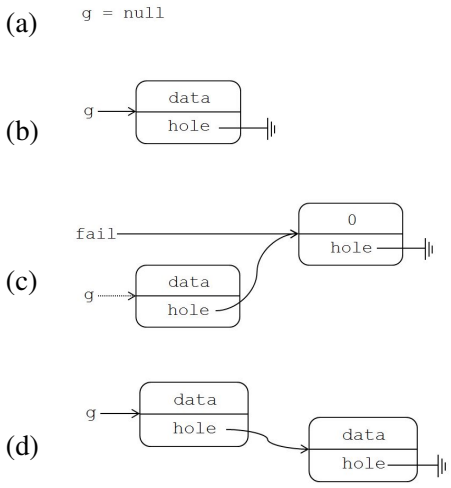
$$B(k) \triangleq (k \neq \mathit{null} \wedge k.\mathsf{tid} \neq \mathtt{tid} \wedge \mathscr{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(p) \rhd \mathsf{true}, k.\mathsf{data}), (k.\mathsf{tid}, \mathtt{ex}(k.\mathsf{data}) \rhd \mathsf{true}, p)\}))$$

Figure 6.2: Rely/guarantee conditions and assertions used for the exchanger proof. $\mathrm{In}_{\mathrm{E}}(t)$ implies that thread t is executing a function of E.

6.2 Formal Verification of the Exchanger

We move on to the verification of the exchanger, which is more challenging than that of its clients. As the exchanger does not encapsulate other objects besides memory cells, we take F_{E} to be the completely undefined function, which means that $\mathscr{T}_{\mathrm{E}} = \mathscr{T}$. The proof outline is shown in Figure 2.2. The proof uses two forms of auxiliary state. Firstly, we instrument the code with assignments to the history variable, $\mathscr{T}$, which appears in the specification of the exchanger. We instrument the code with assignments to $\mathscr{T}$ at the successful CAS on line 29 marked by the XCHG action and on the return statements annotated with the FAIL action. (The exact assignments we add can be read from the corresponding actions in Figure 6.2.) Secondly, we extend the `Offer` class with an auxiliary field `tid` to record the identifier of the thread that allocated the `Offer` object. This field is used to ensure that the auxiliary assignment to $\mathscr{T}$ in the XCHG action records the correct thread identifiers.

Figure 6.2 defines the rely/guarantee conditions that are used in the proof. Following the trend in modern program logics [5, 28], the rely/guarantee conditions are defined in terms of actions corresponding to the individual shared state updates performed. Here, actions are parametrized by the thread t performing the action. The first four actions describe the effects of the algorithm's CAS operations to the shared state, when they succeed. They modify g or g.hole and in the case of XCHG also the auxiliary history variable h. The FAIL action records the auxiliary assignments to h for failed exchanges, while IRR is a 'frame' action allowing other objects to append their events to h. Discarding the effects to the memory cells encapsulated by the exchanger (i.e., restricting attention to the variable h), the actions match those




```
14 class Offer {
15   thread_id tid;
16   int data;
17   Offer hole;
18   Offer(thread_id t, int d)
19   { tid = t; data = d; hole = null; }
20 }
21 class Exchanger {
22   private Offer g = null;
23   private Offer fail = new Offer(0,0);
```

11 $\{\mathscr{T}_E|_{tid} = T\}$
12 `(bool,int) exchange(int v) {`
13 `  Offer n = new Offer(tid,v);`
14 $\{A\}$
15 `  if (CAS(g, null, n)){ // INIT`
16 $\{(\mathscr{T}_E|_{tid} = T \wedge n \mapsto \mathtt{tid}, v, null \wedge g = n) \vee B(n.\mathsf{hole})\}$
17 `    sleep(50);`
18 `    if (CAS(n.hole, null, fail)) // PASS`
19 $\{\mathscr{T}_E|_{tid} = T\}$
20 `      return (false,v); // FAIL`
21 `    else` $\{B(n.\mathsf{hole})\}$
22 `      return (true,n.hole.data);`
23 `  }`
24 $\{A\}$
25 `  Offer cur = g;`
26 $\{A \wedge (g = cur \vee cur.\mathsf{hole} \neq null)\}$
27 `  if (cur != null) {`
28 $\{A \wedge (g = cur \vee cur.\mathsf{hole} \neq null) \wedge cur \neq null \wedge \neg s\}$
29 `    bool s = CAS(cur.hole, null, n); // XCHG`
30 $\{(\neg s \wedge A \vee s \wedge B(cur)) \wedge cur \neq null \wedge cur.\mathsf{hole} \neq null\}$
31 `    CAS(g, cur, null); // CLEAN`
32 `    if (s)` $\{B(cur)\}$
33 `      return (true,cur.data);`
34 `  }`
35 `  return (false,v); // FAIL`
36 `}`
37 $\left\{\begin{array}{l}(\exists t', v'.\ ret = (\mathsf{true}, v') \wedge t' \neq \mathtt{tid} \wedge \\ \qquad \mathscr{T}_E|_{tid} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \triangleright \mathsf{true}, v'), (t', \mathtt{ex}(v') \triangleright \mathsf{true}, v)\})) \\ \vee (ret = (\mathsf{false}, v) \wedge \mathscr{T}_E|_{tid} = T \cdot (\mathrm{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \triangleright \mathsf{false}, v)\}))\end{array}\right\}$
38 `}`

Figure 6.3: Implementation of the exchanger with a proof outline.

in the exchanger specification.

Figure 6.2 also defines the global invariant J saying that g cannot contain an unsatisfied offer of a thread not currently participating in the exchange, and two assertions A and B that will be used in the proof outline and are explained later on. For conciseness, we write $n \mapsto t, d, m$ as an abbreviation for $n.\mathsf{tid} = t \wedge n.\mathsf{data} = d \wedge n.\mathsf{hole} = m$. We note that J is stable both under the rely and guarantee conditions and we implicitly assume it to hold throughout execution.

We now proceed to the proof outline in Figure 6.3. Thanks to the encapsulated nature of concurrent objects in our programming language, we may assume that just before the start of the function $\neg \mathrm{In}_E(\mathtt{tid})$ holds, i.e., that thread `tid` is not executing a function of E. Hence, from invariant J, we can deduce that $g = null \vee g.\mathsf{hole} = null \vee g.\mathsf{tid} \neq \mathtt{tid}$. Thus, after allocating the offer object, we have the assertion A.

The assertion states that the thread has not performed its operation yet, which is implied by $\mathscr{T}_{\mathrm{E}}|_{\texttt{tid}} = T$, and that no other thread can access the newly allocated offer.

If the initialization CAS succeeds at line 15, we know that $g = n \wedge g.\mathsf{hole} = \mathit{null} \wedge \mathscr{T}_{\mathrm{E}}|_{\texttt{tid}} = T$. This assertion, however, is not stable because another thread can come along and modify g.hole, i.e., performs the XCHG action. If this happens, then it would have made n.hole non-null and extend the history appropriately (i.e., $B(n.\mathsf{hole})$ will hold). Therefore, at line 16, the disjunction of these two assertions holds: Either an exchange has not happened, and then $n.\mathsf{hole} = \mathit{null}$, or that it was done by some other thread, and then $B(b.\mathsf{hole})$ holds.

The CAS in line 18 checks which of the above cases hold: If it succeeds, it means that waiting passively for a partner thread did not pan out. This failure, indicated by the ability to set n.hole to `fail`, is manifested in the history by extending it with the failed operation. (Action PASS^t). If the CAS failed than the wait did work out. Specifically, because a thread can modify the `hole` field of an offer of another thread only when it can justify it using the XCHG action, which implies that the partner thread has also (actively) logged the successful exchange in the history variable.

Otherwise, if the initialization CAS fails, the algorithm reads g into the local variable *cur* at line 25. After this, we cannot assert that $g = \mathit{cur}$ because another thread may have modified g in the meantime. For this to happen, however, we know that *cur*.hole must be non-null; thus the disjunction $g = \mathit{cur} \vee g.\mathsf{hole} \neq \mathit{null}$ is stable. Then, if *cur* is non-null, the algorithm performs a CAS at line 29 trying to satisfy the exchange offer made by *cur*.tid. If the CAS succeeds, we know that $\mathit{cur} = g$ at the point that the CAS succeeded, and thus we can perform action XCHG and get the postcondition $B(\mathit{cur})$. Whether the CAS succeeds or not, afterwards at line 30, we know that $\mathit{cur}.\mathsf{hole} \neq \mathit{null}$, which allows us to satisfy the precondition of the CLEAN action corresponding to the final CAS operation.

6.3 Formal Verification of the Synchronous Queue

In the section we provide a formal proof of the concurrency-aware linearizabiliy of the synchronous messaging queue seen in Section 2.3. This data structure, sometimes referred to as "zero-length queues" [21], is used to transfer items between *producer* and *consumer* threads, via a standard queue API. Synchronous queues are especially efficient in high-contention systems in which data is constantly transferred and there is no need to maintain a buffer.

Our messaging queue uses a variant of the exchanger object, in which the offered value is an object with 'type' (i.e. header) and 'body' fields. For brevity, we implicitly convert this object to an integer pointer when passed to the exchanger. Leveraging the compositionality nature of our proof technique, the only thing we need to do is to define how the adaptation function of the messaging queue interprets the trace given by the exchanger:

```
1  class MessagingQueue {
2    private Exchanger E = new Exchanger();
3    {WF_MQ(T_MQ) ∧ T_MQ|_t = T}
4    public bool send(Message m){
5      (bool, Message) (r,d);
6      m.type = "INFO";
7      {WF_MQ(T_MQ) ∧ T_MQ|_t = T ∧ m.type == "INFO"}
8      do {
9        {WF_MQ(T_MQ) ∧ T_MQ|_t = T ∧ m.type == "INFO"}
10       (r,d) = E.exchange(m);
         { r ∧ d.type == "ACK"
         ∧ T_MQ|_t = T·(MQ.(t, send(m) ▷ true))
11       ∧ WF_MQ(T_MQ)
         ∨ d.type ≠ "ACK" ∧ WF_MQ(T_MQ)
         ∧ T_MQ|_t = T ∧ m.type == "INFO" }
12     } while (!r || d.type != "ACK")
         { r ∧ d.type == "ACK"
13       ∧ T_MQ|_t = T·(MQ.(t, send(m) ▷ true))
         ∧ WF_MQ(T_MQ) ∧ m.type == "INFO" }
14     return true;
15   }
16   {WF_MQ(T_MQ) ∧ T_MQ|_t = T·(MQ.(t, send(m) ▷ true))}
```

```
1  {WF_MQ(T_MQ) ∧ T_MQ|_t = T}
2    public Message receive(){
3      (bool, Message) (r,d);
4      Message m = new Message();
5      m.type = "ACK";
6      {WF_MQ(T_MQ) ∧ T_MQ|_t = T ∧ m.type == "ACK"}
7      do {
8        {WF_MQ(T_MQ) ∧ T_MQ|_t = T ∧ m.type == "ACK"}
9        (r,d) = E.exchange(m);
           { r ∧ d.type == "INFO"
           ∧ T_MQ|_t = T·(MQ.(t, receive() ▷ m))(r,d)
10         ∧ WF_MQ(T_MQ)
           ∨ d.type ≠ "INFO" ∧ WF_MQ(T_MQ)
           ∧ T_MQ|_t = T ∧ m.type == "ACK" }
11     } while (!r || d.type != "INFO")
         { r ∧ d.type == "INFO"
12       ∧ T_MQ|_t = T·(MQ.(t, receive() ▷ d))
         ∧ WF_MQ(T_MQ) ∧ m.type == "ACK" }
13     return d;
14   }
15   {WF_MQ(T_MQ) ∧ T_MQ|_t = T·(MQ.(t, receive() ▷ d))}
16 }
```

Figure 6.4: Synchrounous Messaging Queue implementation and proof outline.

$$F_{\mathrm{MQ}}\left(E.\left\{\begin{array}{l}(t, ex(m) \rhd \mathsf{true}, m'),\\ (t', ex(m') \rhd \mathsf{true}, m)\end{array}\right\}\right) \triangleq \begin{array}{l}(\mathrm{MQ}.(t, \mathtt{send}(m) \rhd m'))\cdot\\ (\mathrm{MQ}.(t', \mathtt{receive}() \rhd m))\end{array}$$

$$(\text{provided } m'.\mathtt{type} = \text{“ACK”} \text{ and } m.\mathtt{type} = \text{“INFO”})$$

$$F_{\mathrm{MQ}}(\mathrm{E}.\_) \triangleq \varepsilon$$

This function picks as linearization points the successful exchanges of values via the exchanger where the message type is ‘INFO’ and ‘ACK’ in correspondence to the invoking higher level function. In such case, the send is linearized before the receive. All other operations are ignored. For the Synchronous Messaging Queue we take $G^t_{\mathrm{MQ}} \triangleq G^t_{\mathrm{E}}$ and $R^t_{\mathrm{MQ}} \triangleq R^t_{\mathrm{E}}$.

The proof outline of the messaging queue is shown in Figure 6.4.

Chapter 7

Related Work

In this chapter we discuss some closely related work.

7.1 Concurrency-Aware linearizabilty

Neiger [14] proposed *set-linearlizability* as a means to unify specification of concurrent objects with task solutions, e.g., to relate linearizability with Borowsky and Gafni's immediate atomic snapshot objects [2]. The idea is to linearize concurrent operations against a *set* of concurrent operations. The notion of CA-traces is indeed very similar to set-linearizability. Neiger, however, did not provide a formal definition of set-linearizability, a syntactic approach to define concurrent specifications, and neither a formal nor an informal proof method. In contrast, we develop all of the above and employ them to produce the first compositional formal proof of an elimination stack [9].

The idea of elimination was introduced in [25], where it was used to construct pools and queues using trees. Scherer et. al. [20] present an exchanger objet equipped with a sophisticated adaptivity mechanism. This mechanism, however, does not affect the linearizability of the exchanger, and thus irrelevant in the context of our work.

Scherer et al. present a family of *dual-data structures* [22] which support "operations that must wait for some other thread to establish a precondition". Linearizability of dual-data structures is established by explicitly specifying a "request" and "follow-up" *observable* checkpoints within the object's purview, each with its own linearization point. Dual-data structures are in fact CA-objects. We believe that using CA-histories to describe the behavior of dual data structure would help streamline their specification as it would obviate the need to specify two linearization points.

Additional examples of CA-objects include dual-data structures [22], asymmetric elimination ring [1], and the Lock-Free FIFO elimination queues [21]. Other tasks with no sequential specification include the immediate-snapshot, the write-snapshot object, k-set agreement, safe-consensus and more.

7.2 Verification of linearizable objects

Vafeiadis [28] gives a thread modular proof for a variant of the HSY stack using RGSep [28], an extension of separation logic [16] to reason about fine-grained concurrency. There, he exploits the modularity of RGSep to reason about the shared stack and separately about the elimination module, in a thread modular way. He notes that the algorithm has a non-trivial linearization point in the elimination scheme, where a single CAS performed by one of the threads, if successful, linearise two threads, doing the `push` before the `pop`. Therefore, the elimination module is coupled in the context of a stack. His proof is not compositional as the reasoning about the elimination module is coupled with the reasoning about the stack. In particular, the elimination module is not given a context-independent specification. Dragoi et al. [6] present a technique for automatically verifying linearizability for concurrent objects are where the linearization points may be in the body of another thread. Their technique rewrites the program to introduce combined methods whose linearization points are easy to find. They verified the elimination stack by introducing a new method `push+pop`, which simulates the elimination. As a result, their proof is inherently non compositional.
In contrast, we allow for compositional proofs by

1. providing usage-context specifications for CA-object objects,

2. allowing clients to interpret operations that seem to happen in the same point in time as an imaginary sequence of abstract operations,

3. hiding operations on subobjects from clients of their containing object.

Sergey et al. [23] present a framework for verifying linearizability of highly concurrent data structures using time-stamped histories and subjective states, and used it to verify Hendler et al.'s flat combining algorithm. There approach allows to hide the inter-thread interaction in the algorithm, but does not allow, at least by its current instantiations, to verify CA-linearizability. Schellhorn et al. [19] proved that backward simulation is complete for verification linearizability; it would be interesting to see if their result extends to CAL.

A novel feature of our proof technique is that it allows to relate a single concrete atomic step done by *one thread* with a sequence of abstract steps done by *multiple threads*. Our approach stands in contrast with the standard technique of using *atomicity abstraction* [4, 13, 23, 26], which allows to relate several concrete atomic actions with a single abstract step executed by *one thread*.

Chapter 8

Conclusions and Future Work

The main observation in this thesis is that for certain concurrent objects it is *impossible* to give sequential specification, and as a result cannot be proven to be Linearizable [10]. Such objects operate in the concurrent setting in a way which is *observably different* from their behavior in the sequential setting. As a result, prior to our work, it was not possible to verify these objects in a modular fashion. We aimed to rectify this unfortunate state of affairs by addressing a number of issues:

1. We identified and formalized an interesting class of objects in which certain operations should "seem to take effect *simultaneously*" and provided formal means to specify them. Technically, we defined a new correctness condition—*concurrency-aware linearizability* (CAL)—a generalized notion of linearizability which allows to specify the behavior of objects using a restricted form of concurrent specifications.

2. We presented a formal method for verifying that the implementation of a CAL object adheres to it specification. Technically, we showed that a stylized form of classic rely/guarantee proofs, where the verifier instrument the program with an auxiliary variable that logs the CA-history of operations on concurrent objects, provides a simple and effective proof technique. The unique aspects of our approach are:

   (a) The ability to treat a *single* atomic action as an *sequence of operations* by *different* threads which must execute completely and without interruptions, thus providing the illusion of simultaneity, and,

   (b) Allowing CA-objects built over other CA-objects to define their CA-history as a function over the history of their encapsulated objects, which makes reasoning about clients straightforward.

3. We applied our technique to verify the correctness of several concurrent objects that utilize CA-objects in their implementation, thus showing that it is possible to verify CA-objects in a modular fashion. Most notably, we presented the first *modular* proof of linearizability for the

elimination stack of Handler, Shavit, and Yerushalmi [9] in which (i) the elimination subcomponent is verified independently of its particular usage by the stack, and (ii) the stack is verified using an implementation-independent *concurrency-aware specification* of the elimination module.

We believe that we have demonstrated that the class of CA-objects is indeed an interesting an important class. In the future, we would like identify other CA-objects, to apply our technique to verify them. We would also like to look into designing tools that can help verify CAL either automatically, or by helping to discharge certain proof obligations. Finally, we are curious to find out whether there are other classes of concurrent objects whose specification requires using concurrent histories more general than CA-histories but more restrictive than general concurrent histories, and design verification techniques for such classes similar to the methods we developed for CA-objects.

Bibliography

[1] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *Distributed Computing*. 2011.

[2] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, 1993.

[3] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 2007.

[4] Pedro de Rocha Pinto, Thomas Dinsdale-Yang, and Philippa Gardner. Tada: A logic for time and data abstraction. In *ECOOP*, 2014.

[5] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[6] Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *Computer Aided Verification (CAV)*, pages 174–190, 2013.

[7] Nir Hemed and Noam Rinetzky. Brief announcement: Concurrency-aware linearizability. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2014.

[8] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In *International Symposium on Distributed Computing (DISC)*, 2015. Available at http://www.cs.tau.ac.il/~nirh/disc15-ext.pdf.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[10] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

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

[13] 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 *Symposium on Principles of Programming Languages (POPL)*, pages 637–650. ACM, 2015.

[14] Gil Neiger. Set-linearizability (brief announcements). In *PODC*, page 396, 1994.

[15] Leonor Prensa Nieto. The rely-guarantee method in isabelle/hol. In *Proceedings of the 12th European Conference on Programming*, ESOP'03, pages 348–362, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

[17] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6(4):319–340, December 1976.

[18] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[19] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4):31:1–31:37, September 2014.

[20] William N Scherer III, Doug Lea, and Michael L Scott. A scalable elimination-based exchange channel. *SCOOL*, 2005.

[21] William N Scherer III, Doug Lea, and Michael L Scott. Scalable synchronous queues. In *PPoPP*, 2006.

[22] William N Scherer III and Michael L Scott. Nonblocking concurrent data structures with condition synchronization. In *Distributed Computing*. 2004.

[23] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *European Symposium in Programming (ESOP)*, pages 333–358, 2015.

[24] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks, 1995.

[25] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30(6):645–670, 1997.

[26] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2014.

[27] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[28] Viktor Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. University of Cambridge, 2008.