Journal of Logical and Algebraic Methods in Programming 98 (2018) 1–25

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A perspective on specifying and verifying concurrent modules

Thomas Dinsdale-Young [a,*,1], Pedro da Rocha Pinto [b,2], Philippa Gardner [b,2]

[a] Department of Computer Science, Aarhus University, Aarhus, Denmark
[b] Department of Computing, Imperial College London,London, United Kingdom

ARTICLE INFO

Article history:
Received 22 May 2017
Received in revised form 17 February 2018
Accepted 19 March 2018
Available online xxxx

Keywords:
Concurrency
Specification
Program verification

ABSTRACT

The specification of a concurrent program module, and the verification of implementations and clients with respect to such a specification, are difficult problems. A specification should be general enough that any reasonable implementation satisfies it, yet precise enough that it can be used by any reasonable client. We survey a range of techniques for specifying concurrent modules, using the example of a counter module to illustrate the benefits and limitations of each. In particular, we highlight four key concepts underpinning these techniques: auxiliary state, interference abstraction, resource ownership and atomicity. We demonstrate how these concepts can be combined to achieve two powerful approaches for specifying concurrent modules and verifying implementations and clients, which remove the limitations highlighted by the counter example.
© 2018 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

The specification of a concurrent program module and the verification of implementations and clients with respect to such a specification are difficult problems. When concurrent threads work with shared data, the resulting behaviour can be complex. Reasoning about such modules in a tractable fashion requires effective abstractions that hide this complexity. To be effective, an abstract specification of a module must balance two key requirements: it must be general enough that any reasonable implementation satisfies it; and it must be precise enough that any intended client can use it. A specification that is too precise will disallow some reasonable implementations, while one that is too general will disallow reasonable clients. The specification should support *modular* verification, in that the verification of the module implementation and clients should only reference the specification, and not each other's code. This requires the specification to be *modular*, in that it should capture the entire contract between a module and its clients. Since the 1970s, substantial progress has been made on reasoning techniques for concurrency, and recent developments have brought us closer than ever to a general approach to effective modular specification and verification.

In this survey paper, we describe some of the key techniques for reasoning about concurrency that have been developed in recent decades. We restrict our exposition to four concepts which are pervasive and underpin modern program logics for concurrency: auxiliary state, interference abstraction, resource ownership and atomicity. To illustrate these concepts, we consider a concurrent counter module, with an implementation using a spin loop (Section 2.1) and a ticket-lock client

* Corresponding author.
E-mail addresses: tyoung@cs.au.dk (T. Dinsdale-Young), pmd09@doc.ic.ac.uk (P. da Rocha Pinto), pg@doc.ic.ac.uk (P. Gardner).
[1] This research was supported in part by the "Automated Verification for Concurrent Programs" postdoc grant from The Danish Council for Independent Research for Technology and Production Sciences.
[2] This research was supported in part by the EPSRC Programme Grants EP/H008373/1 and EP/K008528/1.

https://doi.org/10.1016/j.jlamp.2018.03.003
2352-2208/© 2018 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```
function makeCounter() {
  x := alloc(1);              // Allocate a single cell
  [x] := 0;                   // Initialise the value at address x with 0
  return x;
}

function read(x) {
  v := [x];                   // Get value at address x
  return v;
}

function incr(x) {
  do {
    v := [x];                 // Get value at address x
    b := CAS(x, v, v + 1);    // Compare value at address x with v and
                              // set it to v + 1 if they are the same
  } while (b = 0);            // Retry if the CAS failed
  return v;
}

function wkIncr(x) {
  v := [x];                   // Get value at address x
  [x] := v + 1;               // Set value at address x to v + 1
  return v;
}
```

Fig. 1. A counter module given using a spin-counter implementation.

(Section 2.2). In Section 3, we look at a range of historical reasoning techniques for concurrency, and how they embody the key concepts:

- Owicki–Gries reasoning [1] introduces *auxiliary state* (Section 3.2) to abstract the internal state of threads;
- rely/guarantee reasoning [2] introduces *interference abstraction* (Section 3.3) to abstract the interactions between different threads;
- concurrent separation logic [3] introduces *resource ownership* (Section 3.4) to encode interference abstraction as auxiliary state;
- linearisability [4] introduces *atomicity* (Section 3.5) to abstract the effects of an operation so that it appears to take place instantaneously.

Modern program logics, such as TaDA [5,6], Iris [7] and FCSL [8,9], combine these techniques, allowing us to prove effective modular specifications for concurrent modules such as the counter. We compare two approaches: a first-order approach used in TaDA (Section 3.6.2), and a higher-order approach introduced by Jacobs and Piessens [10] and used in Iris (Section 3.6.1). In Section 4, we compare these approaches by showing how the spin-counter implementation can be verified against such a counter specification and how the ticket-lock client can be verified using the specification.

2. Concurrent modules

We use a concurrent counter module as the case study for this paper. This section describes a spin-counter implementation and a ticket-lock client.

2.1. A spin-counter implementation

Consider the spin-counter implementation of a concurrent counter shown in Fig. 1. We make use of three *primitive atomic* operations (*i.e.* operations that take effect at a single, discrete instant in time) for manipulating the heap. The load operation $\mathtt{x} := [\mathbb{E}]$; reads the value of the heap at the address given by $\mathbb{E}$ and assigns it to the variable $\mathtt{x}$. The store operation $[\mathbb{E}_1] := \mathbb{E}_2$; stores the value $\mathbb{E}_2$ in the heap at the address given by $\mathbb{E}_1$. Finally, the compare-and-set (CAS) operation $\mathtt{x} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3)$; checks if the value in the heap at the address given by $\mathbb{E}_1$ is equal to $\mathbb{E}_2$: if so, it replaces it with the value $\mathbb{E}_3$ and assigns 1 to $\mathtt{x}$; otherwise, $\mathtt{x}$ is assigned 0.

The counter module has three operations. The `read` operation returns the value of the counter. The `incr` operation increments the value of the counter and returns the old value, using the compare-and-set operation to do this atomically. The compare-and-set can fail if the value of the counter is changed concurrently, so the operation loops (or spins) until it

```
function makeLock() {
  next := makeCounter();     // Allocate ticket counter
  owner := makeCounter();    // Allocate serving counter
  x := alloc(2);             // Allocate two cells
  [x.next] := next;          // Initialise first cell with ticket counter address
  [x.owner] := owner;        // Initialise second cell with serving counter address
  return x;
}

function acquire(x) {
  next := [x.next];          // Get address of the ticket counter
  owner := [x.owner];        // Get address of the serving counter
  t := incr(next);           // Take a ticket
  do {
    v := read(owner);        // Get serving counter value
  } while (v ≠ t);           // Wait for serving value to match the ticket
}

function release(x) {
  owner := [x.owner];        // Get address of the serving counter
  wkIncr(owner);             // Increment the serving counter
}
```

where $\mathbb{E}.\texttt{next} \stackrel{\text{def}}{=} \mathbb{E}$ and $\mathbb{E}.\texttt{owner} \stackrel{\text{def}}{=} \mathbb{E} + 1$.

Fig. 2. A ticket lock implemented using the counter module from Fig. 1.

succeeds. The `wkIncr` operation also increments the value of the counter and returns the old value. However, this operation uses a store instead of a CAS, which can lead to different behaviour in a concurrent setting.

A specification of the counter module should describe how each operation affects the value of the counter. It should express that the counter must be allocated as a precondition of the operations that access it. It should also describe the permitted interference from the context of concurrent operations. Intuitively, the `read` and `incr` operations are robust with respect to concurrent operations that change the value of the counter. By contrast, the potentially faster `wkIncr` requires that no concurrent operation changes the value of the counter between the load and store operations, in order for it to behave as intended. This informal specification is subtle, and so it is an interesting case study to capture formally.

2.2. A ticket-lock client

Consider a ticket-lock client [11] that uses the counter module to provide synchronisation. The code for the ticket lock is given in Fig. 2. The lock uses two counters, the ticket counter `next` and the serving counter `owner`, which both initially have the value 0. A thread acquires the lock by calling the `acquire` operation. This operation increments the `next` counter to obtain a notional ticket. When the value of the `owner` counter agrees with this ticket, the thread has acquired the lock. It can then use whatever resources are protected by the lock, without interference from other threads. Control of these resources is relinquished by calling the `release` operation. This increments the `owner` counter, passing the lock on to the next waiting thread. Intuitively, the use of `incr` for the `acquire` operation is necessary, since it needs to be robust with respect to concurrent threads taking tickets. The use of `wkIncr` for the `release` operation is possible, since only the thread holding the lock should release it.

The spin-counter and its ticket-lock client provide a case study that illustrates some of the key difficulties in specifying concurrent modules, and verifying their implementations and intended clients. The challenge is to develop a concurrent specification of the counter module that is strong enough to allow us to reason about the ticket lock. This mandates a precise description of how each operation affects the value of the counter, and a detailed account of concurrent interference, which distinguishes between `incr` and `wkIncr`. We require a reasoning technique that can formally express such specifications, and verify implementations and clients using these formal specifications. In Section 3, we concentrate on how to specify the counter module, while in Section 4, we demonstrate how to verify the spin-counter implementation and ticket-lock client using our eventual specification.

3. Specification

Our objective is to give a specification for the counter module that, in particular, is satisfied by our spin-counter implementation and can be used to verify the ticket lock as a client. We start by considering a sequential specification, before exploring how different techniques can be used to give concurrent specifications for the counter module. In particular, we show how these techniques use the concepts of auxiliary state, interference abstraction, resource ownership, and atomicity.

We then show how recent logics combine these ideas in a way that can be used to give an effective specification for the counter.

3.1. Sequential specification

It is straightforward to give a sequential specification for the counter module using standard Hoare triples [12]. A Hoare triple, written $\vdash \{P\}\ C\ \{Q\}$, states that if program C is executed from a state satisfying assertion P, then it either does not terminate or terminates with the resulting state satisfying assertion Q. The assertions P and Q are given in first-order logic, with predicates of the form $\mathbb{E}_1 \mapsto \mathbb{E}_2$ describing those heaps containing a heap cell at address $\mathbb{E}_1$ with value $\mathbb{E}_2$. The sequential specification of the counter module is given by:

$$\vdash \{\mathsf{True}\}\ \texttt{makeCounter()}\ \{\mathsf{ret} \mapsto 0\}$$
$$\vdash \{\mathtt{x} \mapsto n\}\ \texttt{read(x)}\ \{\mathtt{x} \mapsto n \wedge \mathsf{ret} = n\}$$
$$\vdash \{\mathtt{x} \mapsto n\}\ \texttt{incr(x)}\ \{\mathtt{x} \mapsto n+1 \wedge \mathsf{ret} = n\}$$
$$\vdash \{\mathtt{x} \mapsto n\}\ \texttt{wkIncr(x)}\ \{\mathtt{x} \mapsto n+1 \wedge \mathsf{ret} = n\}$$

With Hoare logic, it is possible to verify that the spin-counter implementation satisfies the sequential specification, and to verify the correctness of sequential clients that use the counter module. However, this specification gives no information about the behaviour of the operations in a concurrent setting.

3.2. Auxiliary state

Owicki and Gries [1] developed the first tractable proof technique for concurrent programs, identifying the importance of reasoning about *interference* between threads and of using *auxiliary state*. With the Owicki–Gries method, each thread is given a sequential proof. When the threads are composed, we must check that they do not interfere with each other's proofs. This is achieved by extending standard Hoare logic with the Owicki–Gries rule for parallel composition:

$$\textsc{OG-Parallel}\ \frac{\vdash_{\mathsf{OG}} \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad \vdash_{\mathsf{OG}} \{P_2\}\ \mathbb{C}_2\ \{Q_2\} \qquad \text{non-interference}}{\vdash_{\mathsf{OG}} \{P_1 \wedge P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \wedge Q_2\}}$$

The non-interference side-condition constrains the proof derivations for $\mathbb{C}_1$ and $\mathbb{C}_2$. It requires that every intermediate assertion between atomic actions in the proof of $\mathbb{C}_1$ must be preserved by every atomic action in the proof of $\mathbb{C}_2$, and vice-versa. This side-condition leads to non-compositional reasoning, in the sense that it refers to details of the proof derivations that are not represented in the specifications.

An abstract specification for the counter needs to be robust with respect to the non-interference condition. However, in general, the condition will vary depending on the concurrent context. Let us assume that the client may invoke any of the counter operations concurrently, but will not directly interact with the state of the counter. That is, we will only consider interference caused by the counter operations themselves. To this end, we can use an *invariant:* that is, an assertion that is preserved by each atomic action in the module. For the counter specification, the invariant $\exists n.\, \mathtt{x} \mapsto n$ asserts that the counter at x is allocated and has some value. Using this invariant, we can give the following specification for the counter module:

$$\vdash_{\mathsf{OG}} \{\mathsf{True}\}\ \texttt{makeCounter()}\ \{\exists n.\, \mathsf{ret} \mapsto n\}$$
$$\vdash_{\mathsf{OG}} \{\exists n.\, \mathtt{x} \mapsto n\}\ \texttt{read(x)}\ \{\exists n, m.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} = m\}$$
$$\vdash_{\mathsf{OG}} \{\exists n.\, \mathtt{x} \mapsto n\}\ \texttt{incr(x)}\ \{\exists n, m.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} = m\}$$
$$\vdash_{\mathsf{OG}} \{\exists n.\, \mathtt{x} \mapsto n\}\ \texttt{wkIncr(x)}\ \{\exists n, m.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} = m\}$$

However, these specifications are too weak to verify clients such as the ticket lock. They lose all information about the value of the counter, and give no information about how the operations change this value. In fact, the `read` operation could change the value of the counter and still satisfy the specification! Unfortunately, assertions that describe the precise value of the counter are not invariant.

The Owicki–Gries method is able to provide stronger specifications by using *auxiliary state*, which records extra information about the execution history via auxiliary variables. The auxiliary state is updated by *auxiliary code*, which instruments the program code. Since the auxiliary code only updates auxiliary variables, it has no effect on the program behaviour and so can be erased. The auxiliary code is not required when the program is run; it is only used for the static logical reasoning.

By way of example, consider two threads that both increment a counter, as in Fig. 3. The auxiliary variables `y` and `z`, with initial values 0, are used to record the contribution (that is, the number of increments) of each thread. For each thread, the code of the `incr` operation is instrumented with code that updates the appropriate auxiliary variable when the `CAS`

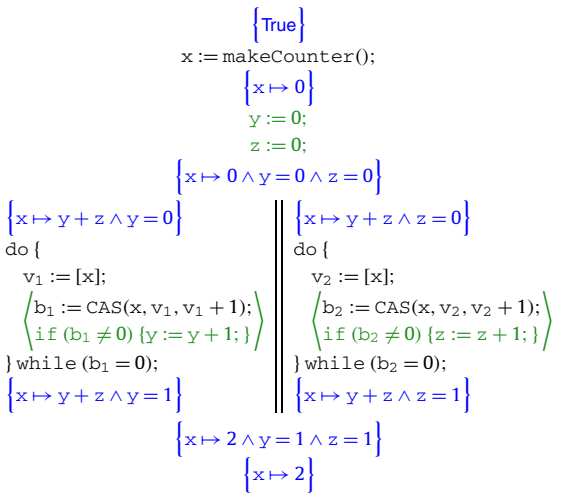


Fig. 3. Reasoning about concurrent increments using auxiliary state.

operation succeeds. This auxiliary variable must be updated at the same instant as the counter, so that the counter always holds the sum of the two contributions — our invariant. This is expressed by the angle brackets, $\langle _ \rangle$, which indicate that the `CAS` and auxiliary code should be executed in a single atomic step. Note that the auxiliary state can be seen as an abstraction of the internal state of the threads. That is, we can recover all of the relevant information about the auxiliary variables by knowing each thread's program counter and local variables. The auxiliary state captures just the information about the internal state that we need for our reasoning.

The resulting specification of the two-increment program is strong, with precise information about the initial and final value of the counter. However, it comes at the price of modularity. Firstly, the `incr` operations require different specifications depending on the client's use: in our example, the assertion $\mathtt{x} \mapsto \mathtt{y} + \mathtt{z}$ uses auxiliary variables `y` and `z`; with three threads, the specification requires three auxiliary variables. A modular specification would be one that captures all use cases. Secondly, each use of the `incr` operation requires the underlying implementation to be extended with auxiliary code to increment the appropriate auxiliary variable. Modular verification would not modify the module code for each use by the client. Thirdly, and more subtly, the Owicki–Gries method requires the global non-interference condition. To meet this, we made the implicit assumption that the client only interacts with the state of the counter through the counter operations. Modular specification would be explicit about such assumptions regarding the behaviour of the client.

Thesis The concept of *auxiliary state*, introduced in the Owicki–Gries method, is important for the specification of concurrent modules. Auxiliary state abstracts the internal state of threads, and is a powerful mechanism for giving precise specifications. However, auxiliary variables can violate modularity; module code may be instrumented with different auxiliary code and its specification may give a different description of the auxiliary state, depending on how the client uses the code. As we shall see, various subsequent approaches have taken a more modular approach to auxiliary state than that provided by auxiliary variables in the Owicki–Gries method.

3.3. Interference abstraction

Jones [2] introduced *interference abstraction*, providing the rely/guarantee method as a way to improve the compositionality of the Owicki–Gries approach. To avoid the global non-interference condition, specifications both explicitly constrain the interference from the concurrent context and describe the interference that a thread may cause. To this end, each specification incorporates two relations, the *rely* and *guarantee* relations, that abstract the interference between threads. The rely relation abstracts the actions of other threads; each assertion in the derivation must be *stable* under all of these actions. The guarantee relation abstracts the actions in the current derivation; each atomic update by the thread must be described by the guarantee. (See Fig. 4.)

Rely/guarantee specifications have the form $R, G \vdash_{\mathsf{RG}} \{P\}\ \mathbb{C}\ \{Q\}$, where the additional relations, R and G, are the rely and guarantee relations respectively. We denote the elements of the rely and guarantee relations as actions $p \rightsquigarrow q$. The actions of the rely relation describe the changes that may be made by the concurrent environment, while the actions of the guarantee relation describe the changes that may be made by the thread (or threads) under consideration. When composing concurrent threads, each thread belongs to the environment of the other and, hence, the guarantee of each thread must be included in the rely of the other. The parallel composition rule is therefore adapted to:

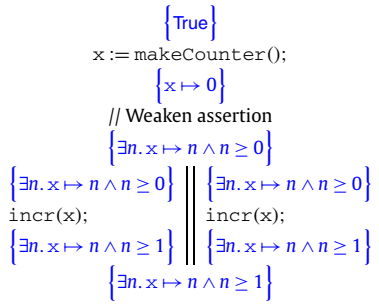


Fig. 4. Reasoning about concurrent increments using interference abstraction.

$$\textsc{RG-Parallel}\quad \frac{R \cup G_2, G_1 \vdash_{\mathsf{RG}} \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad R \cup G_1, G_2 \vdash_{\mathsf{RG}} \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash_{\mathsf{RG}} \{P_1 \wedge P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \wedge Q_2\}}$$

The rely for $\mathbb{C}_1$ consists of the actions that may be performed by the wider environment, namely R, together with the actions that may be performed by $\mathbb{C}_2$, namely G_2. The guarantee for $\mathbb{C}_1 \parallel \mathbb{C}_2$ consists of both the actions of $\mathbb{C}_1$ and the actions of $\mathbb{C}_2$, namely $G_1 \cup G_2$.

The rely/guarantee specifications for the `read` and `incr` operations are:

$$A, \emptyset \vdash_{\mathsf{RG}} \{\exists n.\, \mathtt{x} \mapsto n\}\ \mathtt{read(x)}\ \{\exists n.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} \leq n\}$$

$$A, A \vdash_{\mathsf{RG}} \{\exists n.\, \mathtt{x} \mapsto n\}\ \mathtt{incr(x)}\ \{\exists n.\, \mathtt{x} \mapsto n + 1 \wedge \mathsf{ret} \leq n\}$$

where $A = \{\mathtt{x} \mapsto n \rightsquigarrow \mathtt{x} \mapsto n + 1 \mid n \in \mathbb{N}\}$. The `read` specification has an empty guarantee relation indicating that nothing is changed by the read. It has the rely relation A, stating that other threads can only increment the counter, and that they can do so as many times as they like. The `incr` specification has the same rely relation. Its guarantee relation is also A, stating that the increment can increase the value of the counter. The guarantee must be defined for all n, as the environment can change the counter value. This means that we cannot express that the `incr` operation only does a single increment.

The rely/guarantee specification for the `wkIncr` operation is subtle. Recall that, intuitively, the `wkIncr` operation is intended to be used when no other threads are concurrently updating the counter. As a first try, we can give a simple specification with a rely condition that enforces this constraint:

$$\emptyset, G \vdash_{\mathsf{RG}} \{\mathtt{x} \mapsto n\}\ \mathtt{wkIncr(x)}\ \{\mathtt{x} \mapsto n + 1 \wedge \mathsf{ret} = n\}$$

where $G \triangleq \{\mathtt{x} \mapsto n \rightsquigarrow \mathtt{x} \mapsto n + 1\}$. The rely relation is empty, so this specification cannot be used in a context where concurrent updates may occur. This means that the guarantee relation can be very precise, consisting of a single action.With this specification, the `wkIncr` operation will effectively appear as a single atomic operation.

Although this specification captures some of the intended behaviour of `wkIncr`, it is insufficient to reason about the ticket lock. With the ticket lock, it is possible for two invocations of the `wkIncr` operation to be executing concurrently. Consider a situation in which one thread currently holds the lock, while another thread is attempting to acquire the lock (that is, it is in the loop of the `acquire` operation). Suppose that the first thread executes the `release` operation, in which it calls `wkIncr`. After the body of `wkIncr` has executed, but before the call has returned, the second thread can perform its read and observe that it holds the lock. Moreover, it can then call the `release` operation, executing `wkIncr` *before* the first thread's call to `wkIncr` has returned. Thus we have two concurrent invocations of `wkIncr`.

The above specification does not allow this concurrent behaviour, since the empty rely relation rules out all concurrent updates to the counter. It is possible to allow such concurrent updates by changing the rely, but at the expense of weakening the postcondition:

$$R, G \vdash_{\mathsf{RG}} \{\mathtt{x} \mapsto n\}\ \mathtt{wkIncr(x)}\ \{\exists n' \geq n + 1.\, \mathtt{x} \mapsto n' \wedge \mathsf{ret} = n\}$$

where $R = \{\mathtt{x} \mapsto m \rightsquigarrow \mathtt{x} \mapsto m + 1 \mid m \in \mathbb{N}, m > n\}$ and G is as before. Notice that the rely states that concurrent increments can only happen when the value of the counter is above n. Also notice that, in generalising the rely, we must weaken the postcondition to make it stable.

In summary, this specification is too weak to reason about the ticket lock. It is possible to instrument the code with auxiliary variables, as with the Owicki–Gries method, again leading to a loss of modularity.

Thesis The concept of *interference abstraction*, introduced in the rely/guarantee method, is important for the specification of concurrent modules. By abstracting the interactions between different threads, specifications are able to express constraints on their concurrent contexts. This abstraction leads to more compositional reasoning: since the interference is part of the specification, we do not need to examine proofs in order to justify parallel composition. While it may be specified differently, some form of interference abstraction is generally present in subsequent approaches to verifying concurrent programs.

3.4. Resource ownership

O'Hearn and Brookes developed a new style of Hoare-logic reasoning for concurrency based on *resource ownership*, extending the ideas of separation logic [13,14] to concurrency. They introduced concurrent separation logic [3,15], which provides a highly compositional approach to reasoning about concurrency. Resource ownership can be seen as a specialised form of auxiliary state and interference abstraction. Resource ownership provides auxiliary information that a thread has the right to access some resource; the program does not explicitly record which threads own which resources. Ownership also provides the simple interference abstraction that only a thread that owns a resource can update that resource.

Concurrent separation logic uses an assertion language based on the Bunched Logic of O'Hearn and Pym [16]. Assertions in separation logic treat data, such as heap cells or counter objects, as resources. Each operation acts on some specific resource, with the precondition requiring ownership of the resource it represents. When threads operate on disjoint resources, they do not interfere with each other and so their effects can be combined simply. This principle is embodied in the disjoint parallel composition rule:

$$\textsc{Parallel}\quad \frac{\vdash_{\mathsf{CSL}} \{P_1\}\,\mathbb{C}_1\,\{Q_1\} \qquad \vdash_{\mathsf{CSL}} \{P_2\}\,\mathbb{C}_2\,\{Q_2\}}{\vdash_{\mathsf{CSL}} \{P_1 * P_2\}\,\mathbb{C}_1 \parallel \mathbb{C}_2\,\{Q_1 * Q_2\}}$$

where the assertion $P_1 * P_2$ in the conclusion describes the disjoint combination of the resources described by the assertions P_1 and P_2.

Remark 1. (Disjoint Resources) The fundamental idea behind separation logic [13,14] is to treat the heap as a resource, which can be subdivided into separate disjoint sub-heaps also treated as resources. Heap operations only require parts of the heap for their execution. For example, the update of a heap cell only requires the resource of the heap cell. The rest of the heap is not needed for the update. Such operations are *local* to the specific resource on which they operate, as they do no affect other parts of the heap.

Locality is expressed by the frame rule[3]:

$$\textsc{Frame}\quad \frac{\vdash_{\mathsf{CSL}} \{P\}\,\mathbb{C}\,\{Q\}}{\vdash_{\mathsf{CSL}} \{P * R\}\,\mathbb{C}\,\{Q * R\}}$$

The frame rule allows us to reason about programs in a local way. We can focus our reasoning on the resource that the program uses; any additional resource, which would not be affected by the program, can be added using the frame rule. In particular, the premiss states that, if a program is run in a state described by precondition P then it will not fault and, if it terminates, the resulting state will be described by postcondition Q. The conclusion states that the program has the same behaviour if the disjoint resource R is added to the precondition and postcondition. This is possible because the separating conjunction $*$ enforces disjointness. □

In the original concurrent separation logic, resources can be shared between threads by using invariants. The resource associated with an invariant can only be accessed by a thread during a conditional critical region [3,15], which enforces coarse-grained synchronisation between accesses to the shared resource. We have seen that invariants with auxiliary state allow for precise reasoning, but they are less compositional than the interference abstraction provided by rely/guarantee reasoning. Subsequent developments in concurrent separation logic [17–19] incorporate various forms of rely/guarantee reasoning over shared resources in order to support reasoning about fine-grained concurrency, where threads typically make multiple accesses to shared resources through atomic operations. The concurrent abstract predicate (CAP) [20] approach builds on this with abstractions that hide the shared resources, effectively allowing disjoint concurrent reasoning at the abstract level.

Let us illustrate this CAP reasoning on the counter module. Consider the concurrent abstract predicate $\mathsf{Counter}(\mathtt{x}, n)$ which denotes the existence of a counter at address $\mathtt{x}$ with value n. With our spin counter implementation, this abstract predicate simply describes the concrete sub-heap that is the heap cell $\mathtt{x} \mapsto n$. Treating $\mathsf{Counter}(\mathtt{x}, n)$ as a resource, we could use the original sequential specification as a concurrent one. However, for multiple threads to use the counter, they would have to transfer the resource between each other using some form of synchronisation. Such a specification effectively enforces sequential access to the counter. This is because the client has no mechanism for dividing the resource: in particular,

$$\mathsf{Counter}(\mathtt{x}, n) \implies \mathsf{Counter}(\mathtt{x}, n) * \mathsf{Counter}(\mathtt{x}, n)$$

just does not hold since it is not possible to split the concrete heap into two parts which both contain the cell $\mathtt{x} \mapsto n$.

[3] In many presentations, the frame rule has a side-condition stating that program variables modified by $\mathbb{C}$ do not occur free in R. An alternative is to treat variables as resource, in which case the frame rule does not need a side-condition.

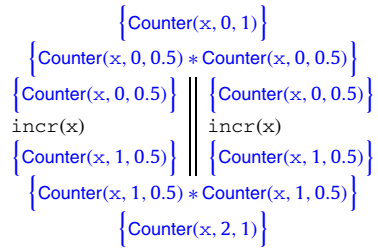


Fig. 5. Reasoning about concurrent increments using resource ownership.

Following Boyland [21], Bornat et al. [22] introduced *permission accounting* to separation logic. This allows shared resources to be divided by associating with them a fraction in the interval $(0, 1]$. Shared resources may be subdivided by splitting this fraction. For instance, we may associate fractions with our counter resource and declare the logical axiom:

$$\mathsf{Counter}(\mathtt{x}, n, \pi_1 + \pi_2) \iff \mathsf{Counter}(\mathtt{x}, n, \pi_1) * \mathsf{Counter}(\mathtt{x}, n, \pi_2)$$

for $\pi_1 + \pi_2 \leq 1$. We can now modify our counter specification to give concurrent read access:

$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, \pi)\}\ \mathtt{read(x)}\ \{\mathsf{Counter}(\mathtt{x}, n, \pi) * \mathsf{ret} = n\}$$
$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, 1)\}\ \mathtt{incr(x)}\ \{\mathsf{Counter}(\mathtt{x}, n+1, 1) * \mathsf{ret} = n\}$$
$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, 1)\}\ \mathtt{wkIncr(x)}\ \{\mathsf{Counter}(\mathtt{x}, n+1, 1) * \mathsf{ret} = n\}$$

Notice that we require full permission (the 1) in order to perform either increment operation. This means that only concurrent reads are permitted; concurrent updates must be synchronised with all other concurrent accesses (both increments and reads). If only partial permission were necessary, then the specification for `read` would be incorrect, since it could no longer guarantee that the value being read matched the resource it had.

It is possible to specify concurrent increments by changing how we interpret the counter predicate $\mathsf{Counter}(\mathtt{x}, n, \pi)$. Now, the resource $\mathsf{Counter}(\mathtt{x}, n, \pi)$ no longer asserts that the value of the counter is n, except if $\pi = 1$. Instead, it asserts that the thread is contributing n to the value of the counter; other threads may also have contributions. We can split this counter resource by declaring the logical axiom:

$$\mathsf{Counter}(\mathtt{x}, n_1 + n_2, \pi_1 + \pi_2) \iff \mathsf{Counter}(\mathtt{x}, n_1, \pi_1) * \mathsf{Counter}(\mathtt{x}, n_2, \pi_2)$$

for $n_1, n_2 \in \mathbb{N}$ and $\pi_1, \pi_2 \in (0, 1]$. We then specify our counter operations as:

$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, \pi)\}\ \mathtt{read(x)}\ \{\mathsf{Counter}(\mathtt{x}, n, \pi) * \mathsf{ret} \geq n\}$$
$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, 1)\}\ \mathtt{read(x)}\ \{\mathsf{Counter}(\mathtt{x}, n, 1) * \mathsf{ret} = n\}$$
$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, \pi)\}\ \mathtt{incr(x)}\ \{\mathsf{Counter}(\mathtt{x}, n+1, \pi) * \mathsf{ret} \geq n\}$$
$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, 1)\}\ \mathtt{incr(x)}\ \{\mathsf{Counter}(\mathtt{x}, n+1, 1) * \mathsf{ret} = n\}$$
$$\vdash_{\mathsf{CAP}} \{\mathsf{Counter}(\mathtt{x}, n, 1)\}\ \mathtt{wkIncr(x)}\ \{\mathsf{Counter}(\mathtt{x}, n+1, 1) * \mathsf{ret} = n\}$$

At last, we have a specification that allows concurrent reads and increments.

Fig. 5 shows how this specification can be used to verify the example of two concurrent increments. Whereas in Fig. 3 each thread was instrumented with different auxiliary code, here the code has not been changed. Rather than each thread having an auxiliary variable to record its contribution to the counter, the contribution is recorded in auxiliary resource that is owned by the thread and encapsulated in the $\mathsf{Counter}(\mathtt{x}, n, \pi)$ predicate. This idea of subjective auxiliary state is at the core of Subjective Concurrent Separation Logic (SCSL) [23] (and the subsequent Fine-grained Concurrent Separation Logic (FCSL) [8,9]).

This specification still has weaknesses. It requires the `wkIncr` operation to be synchronised with the other operations. It also does not guarantee that sequenced reads will never see decreasing values of the counter (since the contribution is not changed and only provides the lower bound). It is possible to describe a more elaborate permission system that allows `wkIncr` in the presence of reads, and to extend the predicate to record the last known value as a lower bound for reads. This would give us a more useful, if somewhat cumbersome, specification. However, it would still not handle the ticket lock. While a ticket lock has been verified using CAP reasoning [20], the proof depends on the atomicity of the underlying counter operations in order to synchronise access to shared resources. The proof does not work with any of our abstract specifications, since they simply do not embody the necessary atomicity.

Thesis The concept of *resource ownership*, developed in the work on concurrent separation logic and its successors, is important for the specification of concurrent modules. The idiom of ownership can be seen as a form of auxiliary state, which critically embodies a notion of disjointness and interference abstraction. Various approaches have explored the power of ownership for reasoning about concurrency [20,23,24,8,25,26,9,7]. While it is an effective concept, and can be used to give elegant specifications, something more is required to provide the strong specifications required for our ticket-lock example.

3.5. Atomicity

Atomicity is the abstraction that an operation takes effect at a single, discrete instant in time. The concurrent behaviour of such operations is equivalent to a sequential interleaving of the operations. A client can use such operations as if they were simple atomic operations.

Herlihy and Wing introduced *linearisability* [4], a well-known correctness condition for atomicity, which identifies when the operations of a concurrent module *appear* to behave atomically. Using the linearisability approach, each operation is given a sequential specification. The operations are then proved to behave atomically *with respect to each other*. One way of seeing this is that there is an instant during the invocation of each operation at which that operation appears to take effect. This instant is referred to as the *linearisation point*. With linearisability, the interference of every operation is tolerated at all times by any of the other operations. Consequently, the interference abstraction is deemed to be the module boundary.

Given our sequential specification for the counter in Section 3.1, is our implementation linearisable? If we only consider the `read` and `incr` operations, then yes, it is. However, the addition of the `wkIncr` operation breaks linearisability. The problem with `wkIncr` is that, for instance, two concurrent calls can result in the counter only being incremented once. This is not consistent with atomic behaviour.

The essence of the problem is that we only envisage calling `wkIncr` in a concurrent context where there are no other increments. In such a case, it would appear to behave atomically. However, the sequential specification cannot express this constraint. We need an interference abstraction that constrains the concurrent context.

Linearisability is related to the notion of contextual refinement. With contextual refinement, the behaviour of program code is described by (more abstract) specification code. (In general, this specification code need not be directly executable.) Contextual refinement asserts that the specification code can be replaced by the program code in any context, without introducing new observable behaviours; we say that the program code contextually refines the specification code. Filipović et al. [27] have shown that, under certain assumptions about a programming language, linearisability implies contextual refinement for that language. For a linearisable module, each operation contextually refines the operation itself executed atomically. For instance, the code for `incr(x)` contextually refines the atomic command $\langle$`incr(x)`$\rangle$. Conversely, contextual refinement implies linearisability.

CaReSL [24] is a logic for proving contextual refinement of concurrent programs. CaReSL makes use of auxiliary state, interference abstraction and ownership in the technical proofs. However, these concepts are not exposed in specifications. This means that it is not obvious what a suitable specification of `wkIncr` in CaReSL should be.

Thesis The concept of *atomicity*, put forward by linearisability, is important for the specification of concurrent modules. Atomicity can be seen as a form of interference abstraction: it effectively guarantees that the only observable interference from an operation will occur at a single instant in its execution. This is a powerful abstraction, since a client need not consider intermediate states of an atomic operation (which, for non-atomic operations, might violate invariants) but only the overall transformation it performs.

3.6. Synthesis

We now examine two approaches, a higher-order approach and a first-order approach, that bring together the ideas we have so far discussed to provide expressive modular specifications for concurrent modules.

3.6.1. A higher-order approach

One way of overcoming the non-modularity of the Owicki–Gries method was introduced by Jacobs and Piessens [10]. Their key idea is to give higher-order specifications for operations, which are parametrised by auxiliary code that is performed when the abstract atomic operation appears to take effect (the linearisation point). Where previously we instrumented the code of the `incr` operation differently for different call sites, here it is instrumented uniformly; the auxiliary code is a parameter that is determined at the call site.

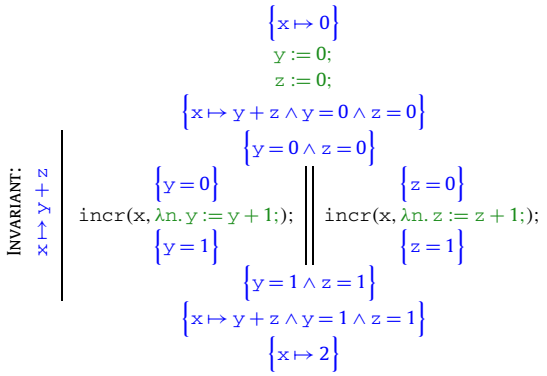


Fig. 6. Reasoning about concurrent increments using parametrised auxiliary code.

Applying this idea to the `incr` operation we have the following code:

```
function incr(x, ρ) {
  do {
    v := [x];
    ⟨b := CAS(x, v, v + 1);
      if (b ≠ 0) { ρ(v); }⟩
  } while (b = 0);
  return v;
}
```

Note that the function ρ is an auxiliary code parameter of the operation. When the atomic update to the counter occurs, ρ is invoked, which can update the client's auxiliary state. The function ρ is parametrised by the value of the counter immediately before the update occurs, which allows the update to the auxiliary state to depend on this value.

The specification of `incr` is parametrised by the specification of the auxiliary code. Written as a proof rule, the specification is as follows:

$$\frac{I * P \Leftrightarrow \exists n \in \mathbb{N}.\, \mathtt{x} \mapsto n * R(n) \qquad \forall n \in \mathbb{N}.\, \vdash \{\mathtt{x} \mapsto n+1 * R(n)\}\ \rho(n)\ \{I * T(n)\}}{I \vdash \{P\}\ \mathtt{incr}(\mathtt{x}, \rho)\ \{T(\mathsf{ret})\}}$$

In the conclusion of this rule, the assertion I is an invariant; it is disjoint from the pre- and postcondition, and must be preserved by atomic updates of all threads. At the point where the counter is atomically incremented, the following steps conceptually take place:

1. the first equivalence from the premiss is used to convert the disjoint combination of the invariant I and the precondition P into the disjoint combination of the counter heap assertion $\mathtt{x} \mapsto n$ and $R(n)$ for some value of n;
2. the module performs the increment, updating $\mathtt{x} \mapsto n$ to $\mathtt{x} \mapsto n+1$; and
3. the auxiliary code $\rho(n)$ is run, updating the combination of $\mathtt{x} \mapsto n+1$ and $R(n)$ to the combination of I and $T(n)$.

This specification enables us to exploit the expressivity of auxiliary variables in a modular way. In particular, Fig. 6 shows how this technique can be used to prove two concurrent increments. The proof is very similar to the one shown in Fig. 3. The new specification allows us to abstract the atomic update performed by the `incr` and use the same module implementation for both threads. The invariant I is instantiated as $\mathtt{x} \mapsto \mathtt{y} + \mathtt{z}$. The predicate $R(n)$ is instantiated as $n = \mathtt{y} + \mathtt{z}$. The predicates P and T are instantiated with the pre- and postconditions of `incr` at each call site. Since the lifetime of the threads is syntactically scoped, we can create an invariant that holds for this scope: we require it to hold before we enter the scope and assume that it holds after the scope; outside the scope, it is no longer invariant. (When threads are created by a `fork` operation, their lifetimes are not syntactically scoped, and so a different approach to invariants is required.)

The `read` operation can be specified as:

$$\frac{I * P \Leftrightarrow \exists n \in \mathbb{N}.\, \mathtt{x} \mapsto n * R(n) \qquad \forall n \in \mathbb{N}.\, \vdash \{\mathtt{x} \mapsto n * R(n)\}\ \rho(n)\ \{I * T(n)\}}{I \vdash \{P\}\ \mathtt{read}(\mathtt{x}, \rho)\ \{T(\mathsf{ret})\}}$$

Finally, recall that the `wkIncr` operation is intended to be used when there are no updates from the environment. This can be specified as:

$$\frac{I * P \Leftrightarrow \mathtt{x} \mapsto n * R \qquad \vdash \{\mathtt{x} \mapsto n+1 * R\}\ \rho(n)\ \{I * T(n)\}}{I \vdash \{P\}\ \mathtt{wkIncr}(\mathtt{x}, \rho)\ \{T(\mathsf{ret})\}}$$

A key difference with the `wkIncr` specification is that n is not quantified in the premisses. This is because the value of the counter must be preserved by other threads before the update.

Note that, although these specifications are written in the form of a proof rules, they are actually implications: the implementation must show that the conclusion follows from the premisses, and a client can use the conclusion if it establishes the premisses. The predicates I, P, T and R, as well as the ghost code ρ, are universally quantified: the client can instantiate them as necessary.

This higher-order specification approach has been adopted in other higher-order logics such as HOCAP [25], iCAP [26] and Iris [7]. In these logics, auxiliary state is not manipulated by auxiliary code, but by *view shifts* [28]. These view shifts serve essentially the same purpose: they are able to update auxiliary state, but have no effect on the concrete state.

3.6.2. A first-order approach

An alternative way of providing specifications for concurrent modules was introduced in the program logic TaDA [5, 6] using *atomic triples*. Rather than treating atomic specifications as a higher-order construct, atomic triples build such specifications into TaDA as a first-order construct. An atomic triple has the form:

$$\vdash \mathbb{\forall} x \in X. \langle P(x) \rangle \, \mathbb{C} \, \langle Q(x) \rangle$$

Abstractly, this can be read as: $\mathbb{C}$ atomically updates $P(x)$ to $Q(x)$, under the assumption that the environment ensures that, before the atomic update, $P(x)$ holds continuously for some $x \in X$ which may change over time. The "pseudo-quantifier" $\mathbb{\forall}$ is part of the syntax of the TaDA specification, and not a quantifier in the underlying assertion language. If x were bound by the standard universal quantifier $\forall$ instead, $\mathbb{C}$ would still update $P(x)$ to $Q(x)$ for arbitrary x, but the environment would not be permitted to change the value of x. The $\mathbb{\forall} x$ binding combines the arbitrary nature of x (hence the resemblance to $\forall$) and the changeable nature of x.[4]

This abstract description hides the subtle behaviour permitted by an implementation. The implementation may assume that the assertion $P(x_0)$ holds initially for some $x_0 \in X$. The implementation must tolerate continual interference from the environment updating $P(x)$ to $P(x')$ for any $x, x' \in X$. The implementation may make updates, but it must preserve $P(x)$ at each step; it cannot change the value of x itself. At some point, the implementation must update $P(x)$ to $Q(x)$ for the current choice of x, if the implementation terminates. After this update, $Q(x)$ is no longer available to the implementation and another thread may be using it.

In TaDA, resources may belong to a particular thread or be shared between all threads. Shared resources are encapsulated by *shared regions*, a kind of invariant which establish protocols for threads to use the encapsulated resources. To ensure that the protocol is followed, a thread can only access the contents of a shared region for the duration of an atomic operation, and the atomic update must conform to the region's protocol. To use $\mathbb{C}$ with the above atomic specification to update a shared region, the region's protocol must ensure that $P(x)$ currently holds, and will continue to do so for arbitrary, changeable $x \in X$. Moreover, the thread must have the right to perform the update from $P(x)$ to $Q(x)$ according to the protocol.

With the counter example in TaDA, the counter operations are specified in terms of an abstract predicate [30] that represents the state of a counter: the abstract predicate $\mathsf{Counter}(s, x, n)$ asserts the existence of a counter at address x with value n. The first parameter s ranges over an *abstract type* $\mathbb{T}_1$, which captures implementation-specific information about the counter. To the client, the type is opaque; the implementation realises the type appropriately. The predicate confers ownership of the counter: it is not possible to have more than one $\mathsf{Counter}(s, x, n)$ for the same value of x.

The specification for the `makeCounter` operation is a simple Hoare triple:

$$\vdash \{\mathsf{True}\} \; \texttt{makeCounter}() \; \{\exists s \in \mathbb{T}_1 . \, \mathsf{Counter}(s, \mathsf{ret}, 0)\}$$

The operation creates a new counter, which is initially set to value 0, and returns its address. The specification says nothing about the granularity of the operation. In fact, the granularity is hardly relevant, since no concurrent environment can meaningfully observe the effects of `makeCounter` until its return value is known: that is, once the operation has been completed.

Remark 2 *(On the abstractly-typed parameters).* Typically, a proof of a specification concludes with a step that existentially quantifies over some fixed parameters in the representation of the data structure. With the atomic specifications of TaDA, this approach is not possible as the following rule is unsound:

$$\frac{\vdash \langle P(s) \rangle \, \mathbb{C} \, \langle Q(s) \rangle}{\vdash \langle \exists s. \, P(s) \rangle \, \mathbb{C} \, \langle \exists s. \, Q(s) \rangle}$$

This rule is unsound in because, in the conclusion, the environment is able to change the value of s, while in the premiss the value cannot be changed by the environment. (A limited form of atomic existential rule is sound [29], where the parameter

[4] In Ntzik's thesis [29], which combines TaDA with refinement, $\mathbb{\forall} x$ is interpreted as a combination of universal quantification with stuttering and mumbling refinement rules that account for x changing over time.

in the premiss is bound by $\forall$.) Since we cannot abstract in this fashion, we instead expose the parameter s. The client does not need to know any particular information about s, only that it should not be changed; hence, the type of s can be abstracted.

In contrast, a higher-order logic can avoid this additional parameter. This is done by specifying in the postcondition of the constructor (`makeCounter`) that there exists some predicate Counter for which the counter operations satisfy their specifications:

$$\begin{array}{c}\{\mathsf{True}\}\\ \mathtt{x} := \mathtt{makeCounter}()\\ \{\exists \mathsf{Counter}.\ \mathsf{Counter}(0) * (\vdash \forall n \in \mathbb{N}. \langle \mathsf{Counter}(n)\rangle\ \mathtt{read(x)}\ \langle \mathsf{Counter}(n) * \mathsf{ret} = n\rangle) * \ldots\}\end{array}$$

Since the parameter is specific to the particular instance of the counter, it is abstracted in the existentially-quantified Counter predicate. (Indeed, the address of the counter can also be abstracted in the predicate.) The parameter s can be viewed as an artefact of defunctionalising the higher-order specification. □

The specification for the `read(x)` operation is the atomic triple:

$$\vdash \forall n \in \mathbb{N}. \langle \mathsf{Counter}(s, \mathtt{x}, n)\rangle\ \mathtt{read(x)}\ \langle \mathsf{Counter}(s, \mathtt{x}, n) * \mathsf{ret} = n\rangle$$

Intuitively, this specification states that the `read` operation will read the state of the counter *atomically*, even in the presence of concurrent updates by the environment that may change the value of the counter, which are possible as n is bound by $\forall$. However, the environment must preserve the counter and cannot, for instance, deallocate it.

This atomicity means that the resources in the specification may be shared: that is, concurrently accessible by multiple threads. Sharing in this way is not possible with ordinary Hoare triples, since they make no guarantee that intermediate steps preserve invariants on the resources. The atomic triple, by contrast, makes a strong guarantee: as long as the concurrent environment guarantees that the (possibly) shared resource $\mathsf{Counter}(s, \mathtt{x}, n)$ is available for some n, the `read` operation will preserve $\mathsf{Counter}(s, \mathtt{x}, n)$ until it reads it; after reading, the operation no longer requires $\mathsf{Counter}(s, \mathtt{x}, n)$, and is consequently oblivious to subsequent transformations by the environment such as another thread incrementing the counter.

It is significant that the notion of atomicity is tied to the abstraction in the specification. The predicate $\mathsf{Counter}(s, \mathtt{x}, n)$ can abstract multiple underlying states in the implementation. If we were to observe the underlying state, the operation might no longer appear to be atomic.

The specification of the `incr` is similar:

$$\vdash \forall n \in \mathbb{N}. \langle \mathsf{Counter}(s, \mathtt{x}, n)\rangle\ \mathtt{incr(x)}\ \langle \mathsf{Counter}(s, \mathtt{x}, n+1) * \mathsf{ret} = n\rangle$$

The specification states that `incr` operation will increment the state of the counter atomically and return its previous value, even in the presence of concurrent updates by the environment that may change the value of the counter, which are possible as n is bound by $\forall$.

The specification of the `wkIncr` operation is slightly different:

$$\forall n \in \mathbb{N}. \vdash \langle \mathsf{Counter}(s, \mathtt{x}, n)\rangle\ \mathtt{wkIncr(x)}\ \langle \mathsf{Counter}(s, \mathtt{x}, n+1) * \mathsf{ret} = n\rangle$$

The specification states that the `wkIncr` will increment the state of the counter atomically and return the previous value, as long as the environment guarantees that the shared counter will not change the value before the atomic update. This specification holds for arbitrary n, but this n cannot be changed by the environment as it is universally quantified in the standard sense. This means that, if the counter is shared, other threads can concurrently only perform `read` operations until the counter has been incremented. It is, however, possible for other `incr` or `wkIncr` operations to occur between the update and the return of the operation.

Atomic triples specify operations with respect to an abstract assertion, such as the $\mathsf{Counter}(s, x, n)$. This means that each operation can be verified independently of the modules of the library. This makes it possible to extend modules with new operations without having to verify the existing operations again. Linearisability, by contrast, is a whole module property: the addition of new operations such as `wkIncr` can break the linearisability of the module.

TaDA [5] introduced a generalised version of the atomic triple that combines atomic updates to shared resources with non-atomic updates to resources owned by the thread. For example, we can use this to specify an operation that reads the value of the counter into a buffer: the read happens atomically, but the write to the buffer does not, and so ownership of the buffer must be transferred between the client and implementation. Such specifications are not possible with traditional linearisability, although Gotsman and Yang [31] have proposed an extension of linearisability that supports ownership transfer.

Remark 3 *(On relating first-order and higher-order approaches).* We can relate the first-order approach to the higher-order approach by encoding atomic triples in the higher-order setting. This approach was taken with Iris [7]. In the Jacobs–Piessens logic, the atomic triple

$$\vdash \forall x \in X. \langle P(x) \rangle \mathbb{C} \langle Q(x, \mathsf{ret}) \rangle$$

can be encoded as the "rule"

$$\frac{I * S \Leftrightarrow \exists x \in X.\, P(x) * R(x) \qquad \forall x \in X.\, \vdash \{Q(x, \mathsf{ret}) * R(x)\}\ \rho(x)\ \{I * T(\mathsf{ret})\}}{I \vdash \{S\}\ \mathbb{C}\ \{T(\mathsf{ret})\}}$$

This encodes how an atomic triple may be used which, in TaDA, is expressed through the proof rules for the atomic triples. The first premiss is used to guarantee that the environment maintains $P(x)$ initially. The second premiss establishes that it is legal to update $P(x)$ to $Q(x)$.

In Section 4, we show proofs using both approaches. While the details differ, the essence of the proofs are the same in each approach. □

Evaluation A combination of auxiliary state, interference abstraction, resource ownership and atomicity makes it possible to specify modules in a way that is both precise and modular. The specifications described in this section are precise enough to derive the earlier specifications we have considered. As we shall see in Section 4, they are also precise enough to verify a client such as the ticket lock, which uses counters to provide synchronisation. Furthermore, the specifications do not expose implementation details. This makes it possible to vary the implementation without changing the specification, which would require updating proofs of client modules. The result is an expressive, modular approach to specification and verification of concurrent modules.

4. Verification

We have discussed both higher-order and first-order approaches to giving expressive, modular specifications for concurrent programs. We now show how these approaches can be used to verify the spin-counter implementation given in Section 2.1, and the ticket-lock client of the counter given in Section 2.2. We begin with the first-order approach of TaDA, before comparing it with a higher-order approach in the style of Jacobs and Piessens.

Remark 4 *(Constructors).* We omit the proofs for the constructors `makeCounter` and `makeLock`, focusing instead on the operations that specifically involve abstract atomicity.

4.1. First-order approach

In Section 3.6.2, we used TaDA's first-order approach to specify the spin counter using atomic triples. TaDA has proof rules for establishing atomic triples. These rules involve shared regions, which are TaDA's mechanism for providing interference abstraction over shared state.

4.1.1. Spin counter

Recall the spin counter implementation from Section 2.1 and the counter specification from Section 3.6.2. To verify the implementation against the specification, we must give an interpretation of the abstract predicate $\mathsf{Counter}(s, x, n)$ including an interpretation of the abstract type $\mathbb{T}_1$ of its first parameter, and prove the implementation against the specification under this interpretation. Since the specifications are atomic, we cannot simply interpret $\mathsf{Counter}(s, x, n)$ as $x \mapsto n$. Instead, TaDA requires that $x \mapsto n$ be encapsulated by a *shared region*, which determines the interference abstraction associated with the counter.

A shared region encapsulates resource that is available to multiple threads when they perform atomic operations. The region enforces a *protocol* that determines how threads can mutate the encapsulated resource. Rather than expressing the protocol for a region directly in terms of the resource it encapsulates, the region is associated with a set of *abstract states*, and the protocol is specified in terms of these. An *interpretation function* determines the concrete resource that is associated with each abstract state.

The region is associated with abstract resources called *guards* — a form of auxiliary state — that determine the role that a thread can play in the protocol. The protocol is defined as a labelled transition system on the abstract states of the region, where the labels are guards. To change the state of the shared region, a thread needs to own the guard associated with the transition it will perform. The guard that a thread owns determines the possible guards that the environment can own, thus limiting the transitions that are available to the environment. Consequently, the guards determine what knowledge a thread can have about the region that is stable: that is, continues to hold under the actions of other threads. In TaDA, the guards for a region are specified as a partial commutative monoid (PCM). This gives us the flexibility to specify complex usage patterns for regions.

Remark 5 *(Partial commutative monoids as auxiliary state).* Since concurrent separation logic, there has been extensive work [32,33,28,7] on using partial commutative monoids to model auxiliary state. Partial commutative monoids allow us

to express complex patterns for subdividing resources. A partial commutative monoid $(\mathsf{G}, \bullet, \mathbf{0})$ consists of a carrier set G equipped with a partial binary operator $\bullet : \mathsf{G} \times \mathsf{G} \rightharpoonup \mathsf{G}$ and a neutral (or identity) element[5] $\mathbf{0} \in \mathsf{G}$ satisfying:

- Commutativity: $x \bullet y = y \bullet x$ when either is defined.
- Associativity: $x \bullet (y \bullet z) = (x \bullet y) \bullet z$ when either is defined.
- Identity: $x \bullet \mathbf{0} = \mathbf{0} \bullet x = x$. □

The guard PCM, protocol and interpretation function associated with a region are determined by a *region type*. We define a region type **Counter** for counter regions. Multiple regions can have the same region type; for example, the ticket lock uses two counters, and hence two instances of the **Counter** region type. We distinguish instances by giving each region a distinct region identifier. A region type can be parametrised: **Counter** is parametrised by the address of the heap cell representing the counter. Region type parameters do not change during the lifetime of the region, unlike the region's abstract state. For a **Counter** region, the abstract state is a natural number representing the current value of the counter. To specify a **Counter** region with region identifier a, parameter x and current state n, we write $\mathbf{Counter}_a(x, n)$.

The guard PCM associated with **Counter** regions simply comprises an indivisible guard Inc, which is used to increment the counter, and the empty guard $\mathbf{0}$. The composition $\textsc{Inc} \bullet \textsc{Inc}$ is undefined, and all other compositions are determined by $\mathbf{0}$ being the neutral element.

The set of abstract states for **Counter** regions is the set of natural numbers $\mathbb{N}$, representing the possible values of the counter. The labelled transition system for the region enables the counter to be incremented using the guard Inc. This is specified by:

$$\textsc{Inc} : \forall n.\, n \rightsquigarrow n + 1$$

The region interpretation function I for **Counter** regions is:

$$I(\mathbf{Counter}_a(x, n)) \triangleq x \mapsto n.$$

With this interpretation, the heap cell that contains the value of the counter is always in the region, and its value corresponds to the abstract state of the region.

Remark 6 *(Protocols).* Protocols enforce a set of rules governing how threads can mutate and exchange resources. Protocols exist in many forms, the simplest form being unary invariants such as the ones used in concurrent separation logic. With a unary invariant, all updates must preserve the invariant assertion. Another form of a protocol is the relational invariants used in rely/guarantee reasoning. With a relational invariant, updates can only change the state in accordance with the invariant relation: for the environment, this is the rely relation; for the thread, this is the guarantee relation. Various approaches, such as RGSep [17], LRG [18], CAP [20], VCC [34], Verifast [10] and HOCAP [25], have localised the notion of protocols to specific shared resources, often as regions or other similar constructs. CaReSL [24], SCSL [23] and iCAP [26] extended the concept of regions with a notion of abstract state and a transition system over those abstract states: in CaReSL these protocols are called islands; in SCSL they are called concurroids; and in iCAP and, following iCAP, TaDA[5], Total-TaDA [35] and Caper [36], they are called shared regions. Iris [7] encodes regions with state transition systems using unary invariants and partial commutative monoids. Finally, some logics support additional abstraction over the protocol actions, such as LRG [18], SCSL [23] and CoLoSL [37]. □

Having defined the **Counter** region type, we can now give a concrete interpretation to the abstract predicate $\mathsf{Counter}(s, x, n)$ and the abstract type $\mathbb{T}_1$:

$$\begin{aligned} \mathbb{T}_1 &\triangleq \mathsf{RId} \\ \mathsf{Counter}(a, x, n) &\triangleq \mathbf{Counter}_a(x, n) * [\textsc{Inc}]_a \end{aligned}$$

Here, RId is the set of region identifiers. The *region assertion* $\mathbf{Counter}_a(x, n)$ asserts that there exists a **Counter** region with identifier a and parameter x in state n. The *guard assertion* $[\textsc{Inc}]_a$ asserts ownership of guard Inc for region a. Notice that the $\mathsf{Counter}(a, x, n)$ predicate encapsulates ownership of both a **Counter** shared region and the guard Inc required to update the region.

We are now in a position to prove that the counter implementation satisfies the specification using the above definitions. The proof outlines for the `read`, `incr` and `wkIncr` operations are given in Figs. 7, 8 and 9, respectively. These proofs use TaDA's core proof rules for deriving atomic specifications: MakeAtomic and UpdateRegion. The MakeAtomic rule allows us to derive an atomic specification that updates a shared region, provided evidence that the code performs a single atomic update on the region, under suitable constraints on how the environment can update the region. The UpdateRegion rule is used to perform the update at the linearisation point, and produces the evidence of this update.

[5] An alternative definition of PCMs uses a *set* of neutral elements. The definition given here is typically simpler to use.

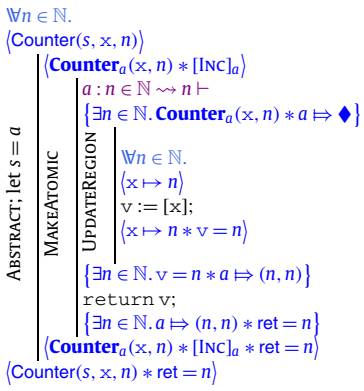


Fig. 7. Proof outline for the `read` operation.

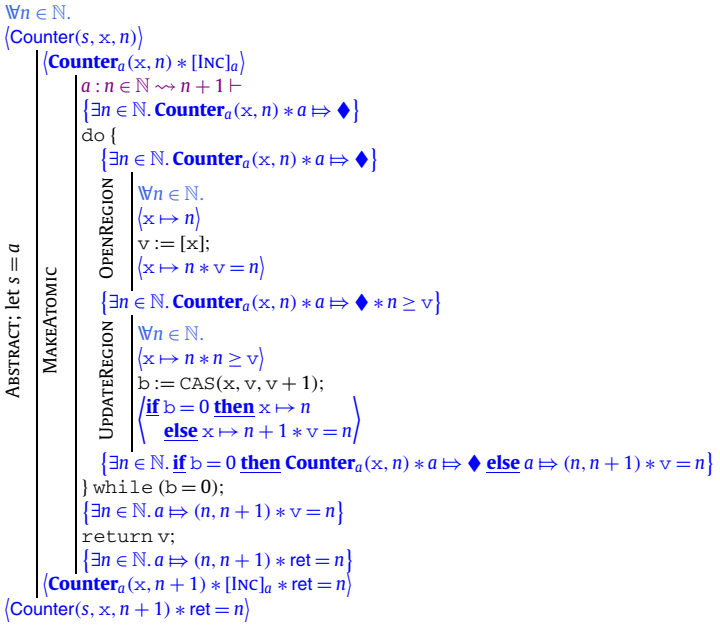


Fig. 8. Proof outline for the `incr` operation.

We consider a simplified version of TaDA's MAKEATOMIC rule:

$$\frac{\{(x,y) \mid x \in X,\, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad R \text{ is pure} \qquad a : x \in X \rightsquigarrow Q(x) \vdash \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * a \mapsto \blacklozenge\right\}\ \mathbb{C}\ \left\{\exists x \in X,\, y \in Q(x).\, a \mapsto (x,y) * R(x,y)\right\}}{\vdash \mathbb{V}x \in X. \langle \mathbf{t}_a(\vec{z}, x) * [\mathrm{G}]_a \rangle\ \mathbb{C}\ \langle \exists y \in Q(x).\, \mathbf{t}_a(\vec{z}, y) * [\mathrm{G}]_a * R(x,y) \rangle}$$

The region assertion $\mathbf{t}_a(\vec{z}, x)$ asserts that the region with identifier a is of type $\mathbf{t}$ (for example **Counter**), with parameters $\vec{z}$, and is in state x. The conclusion of the rule establishes that $\mathbb{C}$ effectively atomically updates this region from some state $x \in X$ to some state $y \in Q(x)$ using the guard G for the region.

The first premiss requires that this update is permitted by the transition system given the guard G. Here, $\mathcal{T}_{\mathbf{t}}(\mathrm{G})$ is the set of transitions for region type $\mathbf{t}$ that are labelled by G. We assume that this is closed upwards under adding resource: if $(a,b) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})$ then $(a,b) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G} \bullet \mathrm{H})$ for all H with $\mathrm{G} \bullet \mathrm{H}$ defined. Then $\mathcal{T}_{\mathbf{t}}(\mathrm{G})^*$ is the reflexive-transitive closure of this relation. The second premiss requires the assertion R to be pure: that is, independent of resources and regions. The final premiss captures the notion of atomicity of $\mathbb{C}$, with respect to the abstraction in the conclusion, as a proof obligation. Specifically, the region must be in the state x for *some* $x \in X$, which may be changed by the environment, until at some point the thread updates it to some $y \in Q(x)$. This obligation is expressed using two new technical concepts that are used in the premiss. The first, $a : x \in X \rightsquigarrow Q(x)$, is called the *atomicity context*. The atomicity context records the actual abstract atomic action that is to be performed: from some state $x \in X$ to a state in $Q(x)$. The second, $a \mapsto -$, is the *atomic tracking resource*. The atomic tracking resource indicates whether the atomic update has occurred (the $a \mapsto \blacklozenge$ indicates it has not)

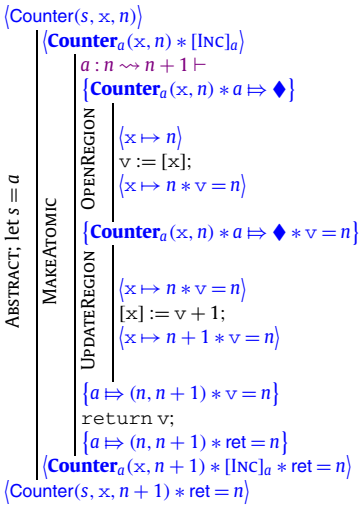


Fig. 9. Proof outline for the `wkIncr` operation.

and, if it has, the state of the shared region immediately before and after (the $a \Rightarrow (x, y)$ indicates an update from x to y). The resource $a \Rightarrow \blacklozenge$ also plays two special roles that are normally filled by guards. Firstly, it limits the interference on region a: the environment may only update the state so long as it remains in the set X, as specified by the atomicity context. Secondly, it confers permission for the thread to update the region from state $x \in X$ to any state $y \in Q(x)$; in doing so, the thread also updates $a \Rightarrow \blacklozenge$ to $a \Rightarrow (x, y)$. This permission is expressed by the UPDATEREGION rule (described below), and ensures that the atomic update only happens once.

In the proof of the `read` operation given in Fig. 7, the MAKEATOMIC rule is instantiated as follows:

$$\frac{\{(n,n) \mid n \in \mathbb{N}\} \subseteq \mathcal{T}_{\mathbf{Counter}}(\text{INC})^* = \{(n, n+1) \mid n \in \mathbb{N}\}^* = \{(n,m) \mid n, m \in \mathbb{N} \wedge n \leq m\} \quad a : n \in \mathbb{N} \rightsquigarrow n \vdash \left\{\exists n \in \mathbb{N}.\, \mathbf{Counter}_a(\mathtt{x}, n) * a \Rightarrow \blacklozenge\right\} \mathtt{v} := [\mathtt{x}]; \mathtt{return}\ \mathtt{v}; \left\{\exists n \in \mathbb{N}.\, a \Rightarrow (n,n) * \mathsf{ret} = n\right\}}{\vdash \forall n \in \mathbb{N}. \left\langle \mathbf{Counter}_a(\mathtt{x}, n) * [\text{INC}]_a \right\rangle \mathtt{v} := [\mathtt{x}]; \mathtt{return}\ \mathtt{v}; \left\langle \mathbf{Counter}_a(\mathtt{x}, n) * [\text{INC}]_a * \mathsf{ret} = n \right\rangle}$$

Here, we choose the pure assertion $R(x, y) \triangleq (\mathsf{ret} = x)$. Since $Q(x) \triangleq \{x\}$, we simplify the postconditions by eliminating the existentially quantified variable y.

The second key TaDA proof rule is the UPDATEREGION rule, which uses the atomicity tracking resource to update a region. A simplified version of this rule is as follows (where $*$ binds tighter than $\vee$):

$$\frac{\vdash \forall x \in X. \left\langle I(\mathbf{t}_a(\vec{z}, x)) * P(x) \right\rangle \mathbb{C} \left\langle \left(\exists y \in Q(x).\, I(\mathbf{t}_a(\vec{z}, y)) * Q_1(x, y)\right) \vee I(\mathbf{t}_a(\vec{z}, x)) * Q_2(x) \right\rangle}{a : x \in X \rightsquigarrow Q(x) \vdash \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * P(x) * a \Rightarrow \blacklozenge\right\} \mathbb{C} \left\{ \begin{array}{c} \exists x \in X.\, (\exists y \in Q(x).\, Q_1(x, y) * a \Rightarrow (x, y)) \\ \vee\ \mathbf{t}_a(\vec{z}, x) * Q_2(x) * a \Rightarrow \blacklozenge \end{array} \right\}}$$

In the premiss of this rule, either the atomic operation updates the state of the region to some $y \in Q(x)$, or the state is unchanged. In the conclusion, this is represented by either updating the atomic tracking resource to $a \Rightarrow (x, y)$, or leaving it as $a \Rightarrow \blacklozenge$. Note that, if $y = x$ in the postcondition, the abstract state of the region is not changed and we can either perform the atomic update or not.

In the proof of the `read` operation in Fig. 7, the UPDATEREGION rule is instantiated as follows:

$$\frac{\vdash \forall n \in \mathbb{N}. \left\langle \mathtt{x} \mapsto n \right\rangle \mathtt{v} := [\mathtt{x}]; \left\langle \mathtt{x} \mapsto n * \mathtt{v} = n \right\rangle}{a : n \in \mathbb{N} \rightsquigarrow n \vdash \left\{\exists n \in \mathbb{N}.\, \mathbf{Counter}_a(\mathtt{x}, n) * a \Rightarrow \blacklozenge\right\} \mathtt{v} := [\mathtt{x}]; \left\{\exists n \in \mathbb{N}.\, \mathtt{v} = n * a \Rightarrow (x, y)\right\}}$$

Here, we choose $P(x) \triangleq \mathsf{True}$, $Q_1(x, y) \triangleq (\mathtt{v} = x)$ and $Q_2(x) \triangleq \mathsf{False}$, and simplify the assertions.

The proof of the `read` implementation (Fig. 7) first rewrites the specification using the definition of the Counter predicate. It is then possible to apply the MAKEATOMIC rule. The atomicity context allows the region a to be in any abstract state $n \in \mathbb{N}$. The UPDATEREGION rule performs the atomic action, which leaves the region in the same state and records the state in the atomic tracking resource.

The proof of the `incr` implementation (Fig. 8) follows a similar style. The main difference is that, when entering the loop, it first performs a read operation and stores the current value of the counter in `v`. The OPENREGION rule allows a region to be opened for an atomic operation, provided that the abstract state is unchanged. Here, it is used to read the value of the counter. The UPDATEREGION rule is then used to perform the atomic action conditionally. If the atomic compare-and-set operation succeeds, the region transitions from state n to $n + 1$ and the atomic tracking component is updated. If it fails,

the region remains in the same state and the atomic tracking component is not updated. The loop will repeat until the compare-and-set succeeds, with the loop invariant ensuring that the region has not yet been updated. After the loop, the region is guaranteed to have been updated.

The proof of the `wkIncr` implementation (Fig. 9) is somewhat similar to `incr`, except that the atomicity context does not allow the environment to change the abstract state of the region before the atomic update occurs. Consequently, the update to the region is always successful.

4.1.2. Ticket lock

We give a specification for a lock based on ownership transfer. We show how to prove that the ticket lock implementation from Section 2.2 satisfies this specification using the atomic specification of the counter.

Lock specification We start by specifying the lock module using a specification based on ownership transfer, first given in the work on CAP reasoning [20]. The specification provides two abstract predicates: $\mathsf{IsLock}(x)$, which is a non-exclusive resource that allows a thread to compete for the lock; and $\mathsf{Locked}(x)$, which is an exclusive resource that represents that the thread has acquired the lock, and allows it to release the lock. The lock is specified as follows:

$$\vdash \{\mathsf{True}\}\ \texttt{makeLock()}\ \{\mathsf{IsLock(ret)}\}$$

$$\vdash \{\mathsf{Locked(x)}\}\ \texttt{release(x)}\ \{\mathsf{True}\}$$

$$\vdash \{\mathsf{IsLock(x)}\}\ \texttt{acquire(x)}\ \{\mathsf{IsLock(x)} * \mathsf{Locked(x)}\}$$

$$\mathsf{IsLock}(x) \iff \mathsf{IsLock}(x) * \mathsf{IsLock}(x)$$

$$\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$$

When a thread acquires the lock, it gets the $\mathsf{Locked}(x)$ predicate that can subsequently be used to release the lock. The last two axioms respectively allow us to duplicate the non-exclusive resource describing the existence of a lock, and guarantee that two threads cannot hold the $\mathsf{Locked}(x)$ resource at the same time.

Implementation To verify this implementation against the atomic specification, we first define a shared region for the ticket lock. Recall that the ticket lock comprises two counters: the first counter records the next available ticket, while the second counter records the ticket which currently holds the lock. The lock is considered unlocked when the two counters are equal. In order for a thread to acquire the lock, it must obtain a ticket by incrementing the first counter and then must wait until the second counter reaches the value of the obtained ticket. To release the lock, a thread simply increments the second counter.

To verify the implementation, we introduce a new shared region type, **TLock**. The abstract state of the region will be a natural number n, representing the ticket that currently holds the lock. The guard PCM is generated by the guards $\textsc{Pending}(n, m)$, for $n \leq m \in \mathbb{N}$, and $\textsc{Key}(n)$, for $n \in \mathbb{N}$, subject to the following rules:

$$\textsc{Pending}(n_1, m_1) \bullet \textsc{Pending}(n_2, m_2) \text{ undefined}$$

$$\textsc{Key}(n) \bullet \textsc{Key}(n) \text{ undefined}$$

$$\textsc{Pending}(n, m) \bullet \textsc{Key}(v) \text{ defined} \implies n \leq v < m$$

$$\textsc{Pending}(n, m) = \textsc{Pending}(n, m+1) \bullet \textsc{Key}(m)$$

$$\textsc{Pending}(n, m) \bullet \textsc{Key}(n) = \textsc{Pending}(n+1, m)$$

Conceptually, $\textsc{Key}(n)$ will represent ownership of ticket n for the lock. The guard $\textsc{Pending}(n, m)$ tracks the yet-to-be-used tickets that are currently held by threads, namely those between n and $m-1$ inclusive. The last two rules allow us to extract and merge tickets with the $\textsc{Pending}$ guard. This PCM can be seen as an instance of the authoritative monoid of Iris [7]. (Specifically, it is the authoritative monoid on the monoid of finite subsets of $\mathbb{N}$ under disjoint union. Here, $\textsc{Pending}(n, m) = \bullet\{i \mid n \leq i < m\}$ and $\textsc{Key}(n) = \circ\{n\}$.) An alternative approach would be to have only $\textsc{Key}$ guards, one for each natural number, and define $\textsc{Pending}$ as an infinite combination of $\textsc{Key}$ guards, as in [38,20].

The labelled transition system is given by:

$$\textsc{Key}(n) \ :\ n \rightsquigarrow n+1$$

It ensures that a thread must hold the ticket $\textsc{Key}(n)$ in order to pass ownership of the lock to the next ticket.

We define the region interpretation for **TLock** regions by:

$$I(\mathbf{TLock}_a(x, s, t, owner, next, n)) \triangleq \exists m.\, x.\texttt{owner} \mapsto owner * x.\texttt{next} \mapsto next * \mathsf{Counter}(s, owner, n) * \mathsf{Counter}(t, next, m) * [\textsc{Pending}(n, m)]_a * n \leq m$$

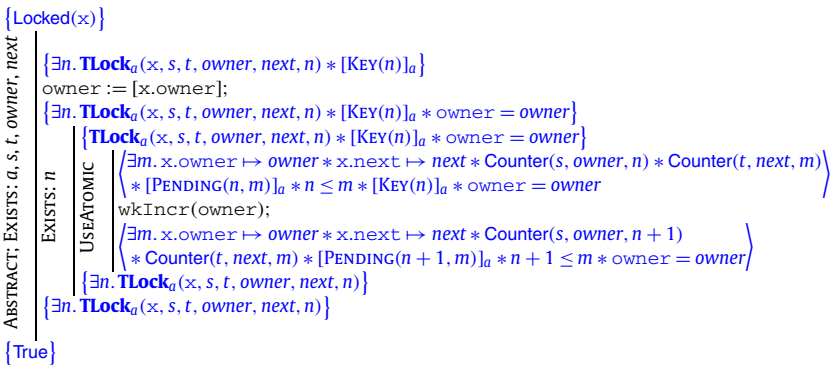


Fig. 10. Proof outline for the `release` operation.

In the region, x is the address of the lock and enables us to retrieve both counter addresses, located at *owner* and *next* respectively, and n is the value of the owner counter. In addition, the logical variables s and t denote parameters associated with the two counter abstract predicates.

We now define the interpretation of the predicates as follows:

$$\begin{aligned} \mathsf{IsLock}(x) &\triangleq \exists a, s, t, \mathit{owner}, \mathit{next}, n.\, \mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n) \\ \mathsf{Locked}(x) &\triangleq \exists a, s, t, \mathit{owner}, \mathit{next}, n.\, \mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n) * [\textsc{Key}(n)]_a \end{aligned}$$

The abstract predicate $\mathsf{IsLock}(x)$ asserts there is a ticket lock region with address x in some state n. $\mathsf{Locked}(x)$ additionally asserts ownership of the guard $[\textsc{Key}(n)]_a$ which can be used to update the region. Note that by holding the $\mathsf{Locked}(x)$ exclusively, we guarantee that the region abstract state cannot be changed by the environment, as no other thread holds the guard ($\textsc{Key}(n)$) necessary to perform the update action.

It remains to prove the specifications for the operations and the axioms. The last key TaDA rule that we mention is the UseAtomic rule. A simplified version of the rule is as follows:

$$\frac{\forall x \in X.\,(x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \vdash \mathbb{V} x \in X. \left\langle I(\mathbf{t}_a(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \right\rangle \mathbb{C} \left\langle I(\mathbf{t}_a(\vec{z}, f(x))) * Q(x) \right\rangle}{\vdash \left\{ \exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * P(x) * [\mathrm{G}]_a \right\} \mathbb{C} \left\{ \exists x \in X.\, \mathbf{t}_a(\vec{z}, f(x)) * Q(x) \right\}}$$

This rule allows a region a, with region type $\mathbf{t}$, to be opened so that it may be updated by $\mathbb{C}$, from some state $x \in X$ to state $f(x)$. In order to do so, the precondition must include a guard G that is sufficient to perform the update to the region, in accordance with the labelled transition system as is established by the first premiss.

The proofs of the `release` and `acquire` operations are given in Fig. 10 and Fig. 11. The interesting part of `release` is the call to `wkIncr`. Here, the thread has the $\textsc{Key}(n)$ guard for the current state of the lock n. The UseAtomic rule is applied choosing $X = \{n\}$ and $f(x) = n + 1$. When the region is opened, the guards $\textsc{Pending}(n, m)$ and $\textsc{Key}(n)$ combine to give $\textsc{Pending}(n + 1, m)$. Together with the update to the owner counter, the region is closed in state $n + 1$. In the postcondition of the UseAtomic rule, we must stabilise the assertion to account for the environment's possible changes to the region. Ultimately, we weaken the postcondition to True, as required by the specification.

The `acquire` proof uses the $\mathbb{V}$ quantifier in the premiss of the UseAtomic rule to account for the fact that the state of the lock is not stable. The first use of the UseAtomic rule increments the counter and retrieves a $\textsc{Key}(\mathtt{t})$ for the value read. After the read, because we own $\textsc{Key}(\mathtt{t})$, we can guarantee that the state of the region cannot be larger than `t`, *i.e.* that the environment does not have the necessary guards to perform such a transition. The loop then simply waits until the state of the region matches the ticket. When that happens, we know it cannot change as long as we own the guard $\textsc{Key}(\mathtt{t})$ and as such we can satisfy the $\mathsf{Locked}(x)$ predicate.

The axiom $\mathsf{IsLock}(x) \iff \mathsf{IsLock}(x) * \mathsf{IsLock}(x)$ follows from the duplicability of region assertions: that is, $\mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n) \iff \mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n) * \mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n)$. Finally, the axiom $\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$ follows from the fact that $\textsc{Key}(n) \bullet \textsc{Key}(n)$ is undefined.

4.2. Higher-order approach

We now consider how the spin-counter implementation and the ticket-lock client can be verified in a higher-order approach based on that of Jacobs and Piessens [10]. Similar proofs are possible in other higher-order separation logics, such as HOCAP [25], iCAP [26] and Iris [7], although the details vary.

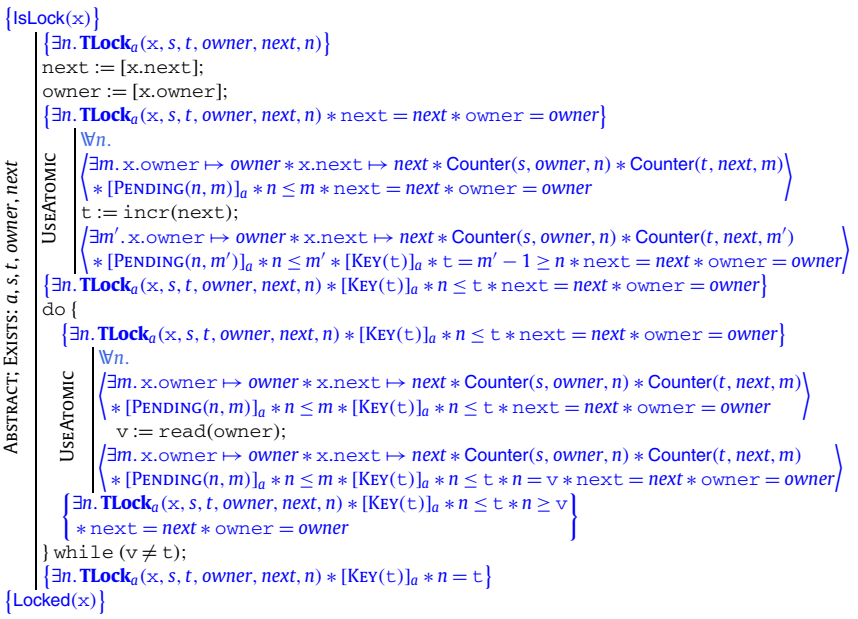


Fig. 11. Proof outline for the `acquire` operation.

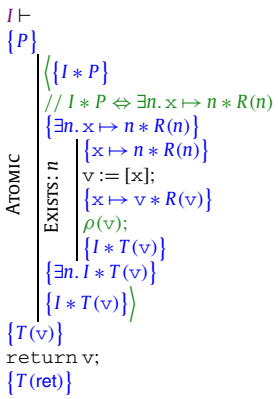


Fig. 12. Proof outline for the `read` operation.

4.2.1. Spin counter

Recall the higher-order specification of `read`, given in Section 3.6.1 in the form of a rule:

$$\frac{I * P \Leftrightarrow \exists n \in \mathbb{N}.\, \mathtt{x} \mapsto n * R(n) \qquad \forall n \in \mathbb{N}.\, \vdash \left\{\mathtt{x} \mapsto n * R(n)\right\}\ \rho(n)\ \left\{I * T(n)\right\}}{I \vdash \left\{P\right\}\ \mathtt{read}(\mathtt{x}, \rho)\ \left\{T(\mathsf{ret})\right\}}$$

This should be read as a logical implication between the premisses and conclusion, with the predicates I, P, R, T, and parameters $\mathtt{x}, \rho$ universally quantified. A proof outline for this specification is given in Fig. 12.

In the `read` operation, the concrete memory read is sequenced with the auxiliary code ρ in a single atomic operation (delimited by $\langle_\rangle$). Since it is atomic, we can transfer the invariant I into the local state for the duration, reestablishing it at the end. The assumed bi-implication $I * P \Leftrightarrow \exists n.\, \mathtt{x} \mapsto n * R(n)$ is used to obtain the $\mathtt{x} \mapsto n$ resource, which is then read into $\mathtt{v}$. The auxiliary code ρ then updates $\mathtt{x} \mapsto n * R(n)$ to $I * T(n)$. Once the invariant is closed, we are thus left with the required postcondition.

Compare this proof with the corresponding TaDA proof (Fig. 7). We may look at the MAKEATOMIC rule as establishing $a \Mapsto \blacklozenge$ and $a \Mapsto (n, n)$ as proxy resources for P and $T(n)$ respectively. The **Counter** region plays a similar role to the invariant, in that both are opened for the duration of atomic operations. The interpretation of the region plays the role of the bi-implication $I * P \Leftrightarrow \exists n.\, \mathtt{x} \mapsto n * R(n)$. The UPDATEREGION both opens the region for the duration of the atomic update and plays the role of ρ in updating the auxiliary state.

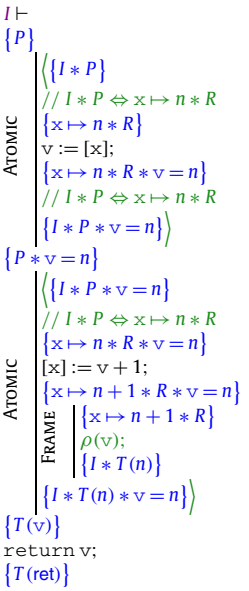


Fig. 13. Proof outline for the `wkIncr` operation.

The specification for `incr` is as follows:

$$\frac{I * P \Leftrightarrow \exists n \in \mathbb{N}.\, \mathtt{x} \mapsto n * R(n) \qquad \forall n \in \mathbb{N}.\, \vdash \{\mathtt{x} \mapsto n + 1 * R(n)\}\ \rho(n)\ \{I * T(n)\}}{I \vdash \{P\}\ \mathtt{incr}(\mathtt{x}, \rho)\ \{T(\mathtt{ret})\}}$$

The proof outline for this operation is given in Fig. 14. By comparison with `read`, the operation involves multiple atomic steps and so exploits the bi-implication in both directions: at all atomic steps where the update does not occur, the bi-implication is used to restore the invariant and precondition. The linearisation point occurs when the `CAS` succeeds, so the auxiliary code ρ is executed conditionally at this point.

Note that after the first atomic read, no relationship between the value that was read and the current value is retained. This is necessary, since it may in fact change arbitrarily before the `CAS` operation. This contrasts with the `wkIncr` operation, where the value cannot change.

Recall the `wkIncr` specification:

$$\frac{I * P \Leftrightarrow \mathtt{x} \mapsto n * R \qquad \vdash \{\mathtt{x} \mapsto n + 1 * R\}\ \rho(n)\ \{I * T(n)\}}{I \vdash \{P\}\ \mathtt{wkIncr}(\mathtt{x}, \rho)\ \{T(\mathtt{ret})\}}$$

The proof outline for this operation is given in Fig. 13. Here, the value of the cell must be fixed at some n, and so when the update is performed $\mathtt{v} + 1$ will be $n + 1$.

As for `read`, the higher-order proofs of `incr` and `wkIncr` somewhat resemble their TaDA counterparts, but with abstract predicates taking on the roles of the region and atomic tracking resources. The higher-order approach does lead to specifications that obscure the atomic update. However, we can see them as encoding a notion of atomic triple, as per Remark 3.

4.2.2. Ticket lock

We now show how to verify the ticket lock in the higher-order setting, using the above specifications for the counter. We give the ticket lock a specification where the lock itself (represented by the IsLock predicate) belongs to the invariant:

$$\vdash \{\mathsf{True}\}\ \mathtt{makeLock}()\ \{\mathsf{IsLock}(\mathtt{ret})\}$$

$$\mathsf{IsLock}(\mathtt{x}) \vdash \{\mathsf{Locked}(\mathtt{x})\}\ \mathtt{release}(\mathtt{x})\ \{\mathsf{True}\}$$

$$\mathsf{IsLock}(\mathtt{x}) \vdash \{\mathsf{True}\}\ \mathtt{acquire}(\mathtt{x})\ \{\mathsf{Locked}(\mathtt{x})\}$$

$$\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$$

Other than treating IsLock as an invariant rather than a duplicable assertion, this specification is essentially the same as the TaDA specification given in Section 4.1.2.

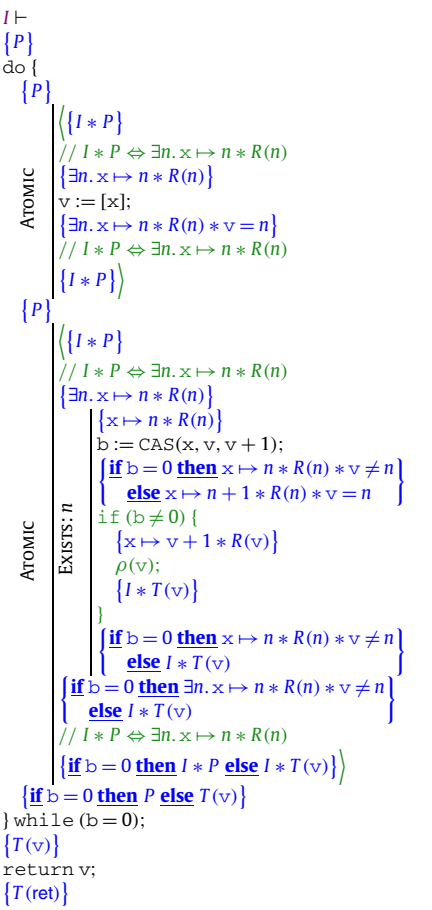


Fig. 14. Proof outline for the `incr` operation.

We define the IsLock and Locked predicates, along with auxiliary predicate LockDescr, as follows:

$$\mathsf{LockDescr}(x, o, n) \triangleq \exists \pi_1, \pi_2.\, x.\mathtt{owner} \xmapsto{\pi_1} o * x.\mathtt{next} \xmapsto{\pi_2} n$$

$$\mathsf{IsLock}(x) \triangleq \exists owner, next, n, m.\, \mathsf{LockDescr}(x, owner, next) * n \leq m * owner \mapsto n * next \mapsto m * x.\mathtt{ghost} \xmapsto{1/3} n * \begin{pmatrix} (\mathsf{gbag}(x.\mathtt{ts}, \{n+1, \ldots, m-1\}) * n < m) \vee \\ (\mathsf{gbag}(x.\mathtt{ts}, \{n, \ldots, m-1\}) * x.\mathtt{ghost} \xmapsto{2/3} n) \end{pmatrix}$$

$$\mathsf{Locked}(x) \triangleq \exists n.\, x.\mathtt{ghost} \xmapsto{2/3} n$$

Remark 7 *(Fractional Permissions).* Fractional permissions are associated with heap locations [21,22] to indicate partial ownership of the heap location. The assertion $x \xmapsto{\pi} y$ denotes partial ownership of heap cell x with fractional permission $\pi \in (0, 1]$; a heap cell can be read with any fractional permission. The assertion $x \xmapsto{1} y$ (or simply $x \mapsto y$) denotes full ownership of the heap cell x; a heap cell can only be written with full permission. □

The invariant for the lock, IsLock(x), establishes that the x.owner and x.next cells point to *owner* and *next*, respectively. It only holds fractional permission for x.owner and x.next, allowing all threads to be able to obtain knowledge of *owner* and *next* and to be sure that they will not change; when a thread reads either of the cells, it will take a fractional permission from the invariant. The invariant holds full ownership of the *owner* and *next* cells, which represent the owner and next counters, and asserts that the value of the owner counter is at most the value of the next counter. The invariant also uses auxiliary (or ghost) resources, which can only be accessed by auxiliary code. The auxiliary heap cell, x.ghost, tracks the owner counter. A $1/3$ permission to this cell always belongs to the invariant. When a thread holds the lock, the remaining $2/3$ permission belongs to the thread holding the lock, which is encapsulated by the Locked(x) predicate. Since permissions cannot exceed 1, the axiom $\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$ holds. Holding the Locked(x) predicate ensures that no other

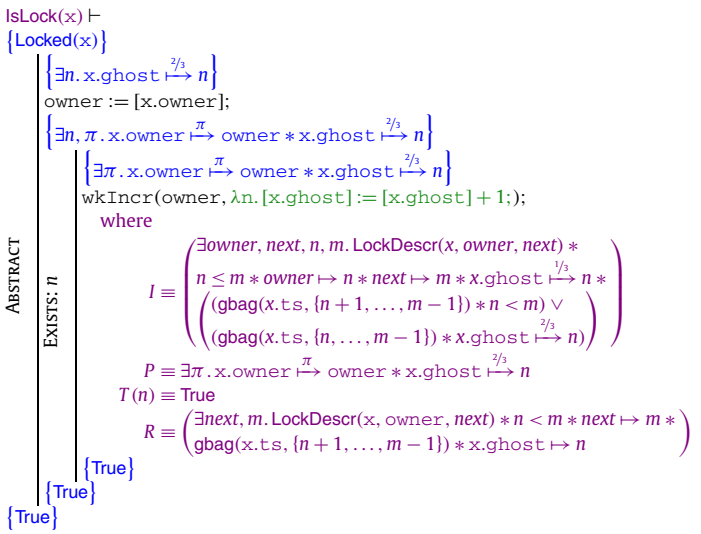


Fig. 15. Proof outline for the `release` operation.

thread can change the value of the ghost cell, and hence the value of the owner counter, which it tracks. When no thread owns the lock, the remaining $2/3$ permission belongs to the invariant. (It might seem that we could use fractional permissions on the *owner* counter instead of this auxiliary heap cell. However, the specification of the counter `read` operation requires full permission, so full permission for the *owner* counter must belong to the invariant.)

The other auxiliary resource in the invariant is a *ghost bag* [10], represented by the gbag predicate. The ghost bag is an abstract data type that represents a bag (or multiset). Ghost bags have two associated abstract predicates: $\mathsf{gbag}(b, B)$, which represents a ghost bag with identifier b and contents B; and $\mathsf{gbagh}(b, v)$, which represents the knowledge that the ghost bag with identifier b contains element v and the permission to remove this element. The two operations for updating the ghost bags are specified as follows:

$$\{\mathsf{gbag}(b, B)\}\ \texttt{gbag_add}(b, v)\ \{\mathsf{gbag}(b, B \uplus \{v\}) * \mathsf{gbagh}(b, v)\}$$

$$\{\mathsf{gbag}(b, B) * \mathsf{gbagh}(b, v)\}\ \texttt{gbag_remove}(b, v)\ \{v \in B \wedge \mathsf{gbag}(b, B \setminus \{v\})\}$$

Here, $\uplus$, $\in$ and $\setminus$ are bag join, membership and difference operations, respectively. We treat a set as a bag where the elements have multiplicity 1.

The invariant uses a ghost bag with identifier $x.\texttt{ts}$ to track the pending tickets. When the `acquire` operation takes a ticket, it adds the ticket to the bag using `gbag_add`. In doing so, it obtains a $\mathsf{gbagh}(x.\texttt{ts}, v)$ resource, representing ownership of ticket v. When it successfully obtains the lock, it removes the ticket from the bag using `gbag_remove`, losing the resource in the process.

The invariant allows for two cases, depending on whether or not the lock is currently held. If the lock has been successfully acquired by some thread, the disjunct

$$\mathsf{gbag}(x.\texttt{ts}, \{n + 1, \ldots, m - 1\}) * n < m$$

holds. In this case, the thread holding the lock owns the fractional permission $x.\texttt{ghost} \overset{2/3}{\longmapsto} n$, and the pending tickets run from $n + 1$ to $m - 1$. (If $n + 1 = m$ then there are no pending tickets.)

Alternatively, if the lock has not been acquired, the disjunct

$$\mathsf{gbag}(x.\texttt{ts}, \{n, \ldots, m - 1\}) * x.\texttt{ghost} \overset{2/3}{\longmapsto} n$$

holds. If $n = m$ then there are no pending tickets. Otherwise, the thread with ticket n (in the form of the $\mathsf{gbagh}(x.\texttt{ts}, n)$ resource) can acquire the lock by removing n from the ghost bag and obtaining the $x.\texttt{ghost} \overset{2/3}{\longmapsto} n$ permission.

Fig. 15 gives the proof outline for the `release` operation. When the invariant is opened, the first disjunct must hold, since otherwise there would be too much ownership of the `x.ghost` heap cell. We can thus establish that $I * P \Leftrightarrow \texttt{owner} \mapsto n * R$. When the auxiliary code is executed, it updates `x.ghost` to match `owner` at $n + 1$. We establish $\vdash \{\texttt{owner} \mapsto n + 1 * R\}\ [\texttt{x.ghost}] := [\texttt{x.ghost}] + 1;\ \{I * T(n)\}$ as follows:

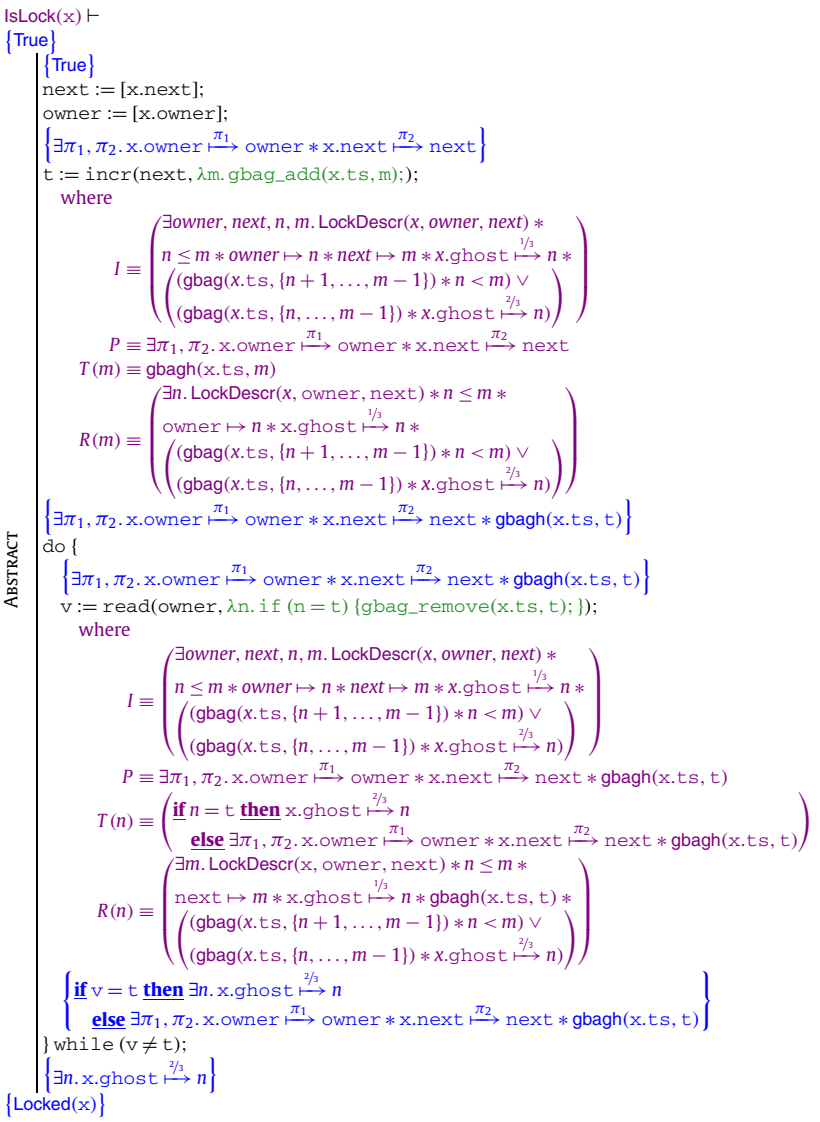


Fig. 16. Proof outline for the `acquire` operation.

$$
\begin{array}{l}
\left\{\begin{array}{l}\texttt{owner} \mapsto n+1 * \exists next, m.\, \mathsf{LockDescr}(\texttt{x}, \texttt{owner}, next) * n < m * next \mapsto m * \\ \qquad\qquad \mathsf{gbag}(\texttt{x.ts}, \{n+1, \ldots, m-1\}) * \texttt{x.ghost} \mapsto n\end{array}\right\} \\
[\texttt{x.ghost}] := [\texttt{x.ghost}] + 1; \\
\left\{\begin{array}{l}\texttt{owner} \mapsto n+1 * \exists next, m.\, \mathsf{LockDescr}(\texttt{x}, \texttt{owner}, next) * n+1 \leq m * next \mapsto m * \\ \qquad\qquad \mathsf{gbag}(\texttt{x.ts}, \{n+1, \ldots, m-1\}) * \texttt{x.ghost} \mapsto n+1\end{array}\right\} \\
//\ n' := n+1 \\
\left\{\begin{array}{l}\exists owner, next, n', m.\, \mathsf{LockDescr}(x, owner, next) * n' \leq m * owner \mapsto n' * next \mapsto m * x.\mathsf{ghost} \overset{1/3}{\mapsto} n' * \\ \left((\mathsf{gbag}(x.\mathsf{ts}, \{n'+1, \ldots, m-1\}) * n' < m) \vee (\mathsf{gbag}(x.\mathsf{ts}, \{n', \ldots, m-1\}) * x.\mathsf{ghost} \overset{2/3}{\mapsto} n')\right)\end{array}\right\}
\end{array}
$$

Fig. 16 gives the proof outline for the `acquire` operation. When `incr` is used to acquire a ticket, the auxiliary code adds the new ticket m to the ghost bag, reestablishing the invariant, while obtaining the ticket resource $\mathsf{gbagh}(\texttt{x.ts}, m)$ for the thread. The proof outline for this auxiliary code is as follows:

$$\left\{\begin{array}{l}\exists n.\,\mathsf{LockDescr}(x,\mathtt{owner},\mathtt{next}) * n \le m * \\ \mathtt{owner} \mapsto n * \mathtt{x.ghost} \overset{1/3}{\longmapsto} n * \\ \left(\begin{array}{l}(\mathsf{gbag}(x.\mathtt{ts}, \{n+1,\ldots,m-1\}) * n < m) \vee \\ (\mathsf{gbag}(x.\mathtt{ts}, \{n,\ldots,m-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} n)\end{array}\right) * \mathtt{next} \mapsto m+1\end{array}\right\}$$

$\mathtt{gbag_add}(\mathtt{x.ts}, m);$

$$\left\{\begin{array}{l}\exists n.\,\mathsf{LockDescr}(x,\mathtt{owner},\mathtt{next}) * n \le m * \\ \mathtt{owner} \mapsto n * \mathtt{x.ghost} \overset{1/3}{\longmapsto} n * \mathtt{next} \mapsto m+1 * \\ \left(\begin{array}{l}(\mathsf{gbag}(x.\mathtt{ts}, \{n+1,\ldots,m\}) * n < m) \vee \\ (\mathsf{gbag}(x.\mathtt{ts}, \{n,\ldots,m\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} n)\end{array}\right) * \mathsf{gbagh}(\mathtt{x.ts}, m)\end{array}\right\}$$

$// \; m' := m+1$

$$\left\{\begin{array}{l}\exists owner, next, n, m'.\,\mathsf{LockDescr}(x, owner, next) * \\ n \le m' * owner \mapsto n * next \mapsto m' * x.\mathtt{ghost} \overset{1/3}{\longmapsto} n * \\ \left(\begin{array}{l}(\mathsf{gbag}(x.\mathtt{ts}, \{n+1,\ldots,m'-1\}) * n < m') \vee \\ (\mathsf{gbag}(x.\mathtt{ts}, \{n,\ldots,m'-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} n)\end{array}\right) * \mathsf{gbagh}(\mathtt{x.ts}, m)\end{array}\right\}$$

In the `read` operation, the ghost code conditionally removes the ticket `t` from the bag, when $n = \mathtt{t}$: *i.e.* it is now the thread's turn to hold the lock. It can do so because it holds the $\mathsf{gbagh}(\mathtt{x.ts}, \mathtt{t})$ resource. When it does so, the second disjunct must apply (since $n \notin \{n+1,\ldots,m-1\}$), and so the resource $\mathtt{x.ghost} \overset{2/3}{\longmapsto} n$ is available to be transferred to the thread. When $n \neq \mathtt{t}$, the invariant and thread resources are unchanged. The proof outline for this auxiliary code is as follows:

$$\left\{\begin{array}{l}\exists m.\,\mathsf{LockDescr}(\mathtt{x},\mathtt{owner},\mathtt{next}) * n \le m * \\ \mathtt{next} \mapsto m * \mathtt{x.ghost} \overset{1/3}{\longmapsto} n * \mathsf{gbagh}(\mathtt{x.ts}, \mathtt{t}) * \\ \left(\begin{array}{l}(\mathsf{gbag}(x.\mathtt{ts}, \{n+1,\ldots,m-1\}) * n < m) \vee \\ (\mathsf{gbag}(x.\mathtt{ts}, \{n,\ldots,m-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} n)\end{array}\right) * \mathtt{owner} \mapsto n+1\end{array}\right\}$$

$\mathtt{if}\ (n = \mathtt{t})\ \{$

$$\left\{\begin{array}{l}\exists m.\,\mathsf{LockDescr}(\mathtt{x},\mathtt{owner},\mathtt{next}) * \mathtt{t} \le m * \\ \mathtt{next} \mapsto m * \mathtt{x.ghost} \overset{1/3}{\longmapsto} \mathtt{t} * \mathsf{gbagh}(\mathtt{x.ts}, \mathtt{t}) * \\ \left(\begin{array}{l}(\mathsf{gbag}(x.\mathtt{ts}, \{\mathtt{t}+1,\ldots,m-1\}) * \mathtt{t} < m) \vee \\ (\mathsf{gbag}(x.\mathtt{ts}, \{\mathtt{t},\ldots,m-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} \mathtt{t})\end{array}\right) * \mathtt{owner} \mapsto \mathtt{t}+1\end{array}\right\}$$

CASES

$$\left\{\begin{array}{l}\mathsf{gbag}(x.\mathtt{ts}, \{\mathtt{t}+1,\ldots,m-1\}) * \\ \mathtt{t} < m * \mathsf{gbagh}(\mathtt{x.ts}, \mathtt{t})\end{array}\right\}$$

$\mathtt{gbag_remove}(\mathtt{x.ts}, \mathtt{t});$

$\{\mathsf{False}\}$

$$\left\{\begin{array}{l}\mathsf{gbag}(x.\mathtt{ts}, \{\mathtt{t},\ldots,m-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} \mathtt{t} * \\ \mathsf{gbagh}(\mathtt{x.ts}, \mathtt{t})\end{array}\right\}$$

$\mathtt{gbag_remove}(\mathtt{x.ts}, \mathtt{t});$

$$\left\{\mathsf{gbag}(x.\mathtt{ts}, \{\mathtt{t}+1,\ldots,m-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} \mathtt{t} * \mathtt{t} < m\right\}$$

$$\left\{\begin{array}{l}\exists m.\,\mathsf{LockDescr}(\mathtt{x},\mathtt{owner},\mathtt{next}) * \mathtt{t}+1 \le m * \mathtt{next} \mapsto m * \mathsf{gbag}(\mathtt{x.ts}, \{\mathtt{t}+1,\ldots,m-1\}) * \\ \mathtt{x.ghost} \overset{1/3}{\longmapsto} \mathtt{t} * \mathtt{x.ghost} \overset{2/3}{\longmapsto} \mathtt{t} * \mathtt{owner} \mapsto \mathtt{t}+1\end{array}\right\}$$

$\}$

$$\left\{\begin{array}{l}\left(\begin{array}{l}\exists owner, next, n, m.\,\mathsf{LockDescr}(x, owner, next) * \\ n \le m * owner \mapsto n * next \mapsto m * x.\mathtt{ghost} \overset{1/3}{\longmapsto} n * \\ \left((\mathsf{gbag}(x.\mathtt{ts}, \{n+1,\ldots,m-1\}) * n < m) \vee (\mathsf{gbag}(x.\mathtt{ts}, \{n,\ldots,m-1\}) * x.\mathtt{ghost} \overset{2/3}{\longmapsto} n)\right)\end{array}\right) * \\ \underline{\mathbf{if}}\ n = \mathtt{t}\ \underline{\mathbf{then}}\ \mathtt{x.ghost} \overset{2/3}{\longmapsto} n \\ \quad \underline{\mathbf{else}}\ \exists \pi_1, \pi_2.\,\mathtt{x.owner} \overset{\pi_1}{\longmapsto} \mathtt{owner} * \mathtt{x.next} \overset{\pi_2}{\longmapsto} \mathtt{next} * \mathsf{gbagh}(\mathtt{x.ts}, \mathtt{t})\end{array}\right\} *$$

Compared with the TaDA approach, the invariant $\mathsf{IsLock}(x)$ plays a similar role to the **TLock** shared region. Here, we use fractional permissions to establish the fact that $x.\mathtt{owner}$ and $x.\mathtt{next}$ hold unchangeable values, while in TaDA this was a consequence of these values being parameters of the region. In TaDA, the counters were abstracted by the $\mathsf{Counter}(s, owner, n)$ and $\mathsf{Counter}(s, next, m)$ predicates, whereas here they are represented directly as $owner \mapsto n$ and $next \mapsto m$ respectively. We could have used abstract predicates in the higher-order counter specification, but chose to expose the heap cell directly, which is not possible in the TaDA approach. The auxiliary resource $\mathsf{gbag}(\mathtt{x.ts}, \{n,\ldots,m-1\})$ is a close analogue of $[\textsc{Pending}(n,m)]_a$. Similarly $\mathsf{gbagh}(\mathtt{x.ts}, t)$ is analogous to $[\textsc{Key}(t)]_a$. One difference, however, is that whereas the assertion $[\textsc{Key}(t)]_a * [\textsc{Key}(t)]_a$ is inconsistent, the assertion $\mathsf{gbagh}(\mathtt{x.ts}, t) * \mathsf{gbagh}(\mathtt{x.ts}, t)$ is not. This is since a bag may, in principle, contain more than one copy of a given element. We thus cannot use $\mathsf{gbagh}(\mathtt{x.ts}, t)$ to represent

ownership of the lock in the case that t matches the owner counter. Instead, the $\mathsf{gbagh}(\mathtt{x.ts}, t)$ resource is exchanged for $\mathtt{x.ghost} \overset{2/3}{\longmapsto} t$ when a thread acquires the lock.

5. Conclusions

We have examined four major techniques for specifying and verifying concurrent modules: Owicki–Gries, rely/guarantee, concurrent separation logic, and linearisability. For each technique, we identified a particularly valuable contribution. We demonstrated how a synthesis of these contributions can be used to produce effective modular specifications for concurrent modules, using a counter module as a case study. We gave specifications for the counter module in both the first-order approach of TaDA and the higher-order approach of Jacobs and Piessens. With each approach, we demonstrated the expressivity and modularity of the counter specification by proving that it is satisfied by a spin-counter implementation and sufficient to verify a ticket-lock client.

References

[1] S. Owicki, D. Gries, Verifying properties of parallel programs: an axiomatic approach, Commun. ACM 19 (5) (1976) 279–285, https://doi.org/10.1145/360051.360224.
[2] C.B. Jones, Specification and design of (parallel) programs, in: IFIP Congress, vol. 83, 1983, pp. 321–332.
[3] P.W. O'Hearn, Resources, concurrency, and local reasoning, Theor. Comput. Sci. 375 (1–3) (2007) 271–307, https://doi.org/10.1016/j.tcs.2006.12.035.
[4] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492, https://doi.org/10.1145/78969.78972.
[5] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, TaDA: a logic for time and data abstraction, in: ECOOP, 2014, pp. 207–231.
[6] P. da Rocha Pinto, Reasoning with Time and Data Abstractions, Ph.D. thesis, Imperial College London, Department of Computing, 2016.
[7] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, D. Dreyer, Iris: monoids and invariants as an orthogonal basis for concurrent reasoning, in: POPL, 2015, pp. 637–650.
[8] A. Nanevski, R. Ley-Wild, I. Sergey, G.A. Delbianco, Communicating state transition systems for fine-grained concurrent resources, in: ESOP, 2014, pp. 290–310.
[9] I. Sergey, A. Nanevski, A. Banerjee, Specifying and verifying concurrent algorithms with histories and subjectivity, in: ESOP, 2015, pp. 333–358.
[10] B. Jacobs, F. Piessens, Expressive modular fine-grained concurrency specification, in: POPL, 2011, pp. 271–282.
[11] J.M. Mellor-Crummey, M.L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. Comput. Syst. 9 (1) (1991) 21–65, https://doi.org/10.1145/103727.103729.
[12] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580, https://doi.org/10.1145/363235.363259.
[13] S.S. Ishtiaq, P.W. O'Hearn, BI as an assertion language for mutable data structures, in: POPL, 2001, pp. 14–26.
[14] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: LICS, 2002, pp. 55–74.
[15] S. Brookes, A semantics for concurrent separation logic, Theor. Comput. Sci. 375 (1) (2007) 227–270, https://doi.org/10.1016/j.tcs.2006.12.034, http://www.sciencedirect.com/science/article/pii/S0304397506009248.
[16] P.W. O'Hearn, D.J. Pym, The logic of bunched implications, Bull. Symb. Log. 5 (2) (1999) 215–244, https://doi.org/10.2307/421090, http://www.jstor.org/stable/421090.
[17] V. Vafeiadis, Modular Fine-Grained Concurrency Verification, Ph.D. thesis, University of Cambridge, Computer Laboratory, 2008.
[18] X. Feng, Local rely-guarantee reasoning, in: POPL, 2009, pp. 315–327.
[19] M. Dodds, X. Feng, M. Parkinson, V. Vafeiadis, Deny-guarantee reasoning, in: ESOP, 2009, pp. 363–377.
[20] T. Dinsdale-Young, M. Dodds, P. Gardner, M.J. Parkinson, V. Vafeiadis, Concurrent abstract predicates, in: ECOOP, 2010, pp. 504–528.
[21] J. Boyland, Checking interference with fractional permissions, in: SAS, 2003, pp. 55–72.
[22] R. Bornat, C. Calcagno, P. O'Hearn, M. Parkinson, Permission accounting in separation logic, in: POPL, 2005, pp. 259–270.
[23] R. Ley-Wild, A. Nanevski, Subjective auxiliary state for coarse-grained concurrency, in: POPL, 2013, pp. 561–574.
[24] A. Turon, D. Dreyer, L. Birkedal, Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency, in: ICFP, 2013, pp. 377–390.
[25] K. Svendsen, L. Birkedal, M. Parkinson, Modular reasoning about separation of concurrent data structures, in: ESOP, 2013, pp. 169–188.
[26] K. Svendsen, L. Birkedal, Impredicative concurrent abstract predicates, in: ESOP, 2014, pp. 149–168.
[27] I. Filipović, P. O'Hearn, N. Rinetzky, H. Yang, Abstraction for concurrent objects, in: ESOP, 2009, pp. 252–266.
[28] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, H. Yang, Views: compositional reasoning for concurrent programs, in: POPL, 2013, pp. 287–300.
[29] G. Ntzik, Reasoning About POSIX File Systems, Ph.D. thesis, Imperial College London, Department of Computing, 2016.
[30] M.J. Parkinson, G.M. Bierman, Separation logic and abstraction, in: POPL, 2005, pp. 247–258.
[31] A. Gotsman, H. Yang, Linearizability with ownership transfer, in: CONCUR, 2012, pp. 256–271.
[32] C. Calcagno, P.W. O'Hearn, H. Yang, Local action and abstract separation logic, in: LICS, 2007, pp. 366–378.
[33] R. Dockins, A. Hobor, A.W. Appel, A fresh look at separation algebras and share accounting, in: APLAS, 2009, pp. 161–177.
[34] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, W. Schulte, VCC: contract-based modular verification of concurrent C, in: ICSE, 2009, pp. 429–430.
[35] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, J. Sutherland, Modular termination verification for non-blocking concurrency, in: ESOP, 2016, pp. 176–201.
[36] T. Dinsdale-Young, P. da Rocha Pinto, K.J. Andersen, L. Birkedal, Caper: automatic verification for fine-grained concurrency, in: ESOP, 2017, pp. 420–447.
[37] A. Raad, J. Villard, P. Gardner, CoLoSL: concurrent local subjective logic, in: ESOP, 2015, pp. 710–735.
[38] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, M. Wheelhouse, A simple abstraction for complex concurrent indexes, in: OOPSLA, 2011, pp. 845–864.