Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 319 (2015) 3–18

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Steps in Modular Specifications for Concurrent Modules
(Invited Tutorial Paper)

Pedro da Rocha Pinto [b,1,3] Thomas Dinsdale-Young [a,2,4]
Philippa Gardner [b,1,5]

[a] *Department of Computer Science*
Aarhus University
Aarhus, Denmark

[b] *Department of Computing*
Imperial College London
London, United Kingdom

Abstract

The specification of a concurrent program module is a difficult problem. The specifications must be strong enough to enable reasoning about the intended clients without reference to the underlying module implementation. We survey a range of verification techniques for specifying concurrent modules, in particular highlighting four key concepts: auxiliary state, interference abstraction, resource ownership and atomicity. We show how these concepts combine to provide powerful approaches to specifying concurrent modules.

Keywords: Concurrency, specification, program verification.

1 Introduction

The specification of a concurrent program module is a difficult problem. When concurrent threads work with shared data, the resulting behaviour can be complex. Consequently, the specification of such modules requires effective abstractions for describing such complex behaviour. The amount of progress that has been made since the 1970s has been substantial. In this paper, we describe some of the key concepts

[1] This research was supported in part by the EPSRC Programme Grants EP/H008373/1 and EP/K008528/1.

[2] This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

[3] Email: pmd09@doc.ic.ac.uk

[4] Email: tyoung@cs.au.dk

[5] Email: pg@doc.ic.ac.uk

http://dx.doi.org/10.1016/j.entcs.2015.12.002

that have emerged over the last few decades. We restrict our exposition to those concepts which we find most important: auxiliary state, interference abstraction, resource ownership and atomicity.

We use a counter module to highlight the challenges of specifying a concurrent module. We require a specification to be expressive enough for verifying the intended clients of the module, such as a ticket lock. We also require that the specification to be opaque, in that the implementation details do not leak into the specification. Using the counter as illustration, we look at a range of historical verification techniques for concurrency:

- Owicki-Gries introduces *auxiliary state* to abstract internal state of threads;
- rely/guarantee introduces *interference abstraction* to abstract the interactions between different threads;
- concurrent separation logic introduces *resource ownership* to encode interference abstraction as auxiliary state;
- linearisability introduces *atomicity* as a way to abstract the effects of an operation.

We show how recent developments enable us to combine these techniques to provide expressive ways for specifying concurrent modules such as the counter.

2 A Concurrent Counter

We use a concurrent counter as a running example throughout this paper.

2.1 Implementation

Consider the following implementation of a concurrent counter:[6]

```
function read(x) {
  r := [x];
  return r;
}

function incr(x) {
  do {
    r := [x];
    b := CAS(x, r, r + 1);
  } while (b = 0);
  return r;
}

function wkincr(x) {
  r := [x];
  [x] := r + 1;
}
```

A specification should describe how each operation affects the value of the counter. Here, the `read` operation returns the value of the counter, the `incr` operation increments the value and returns the old value, and the `incr` just increments the value of the counter.

A specification should require the counter to exist as a precondition for each operation, since operations will not work unless the memory holding the counter is allocated. In this paper, we use the abstract predicate $\mathsf{C}(x, n)$ to denote the existence of a counter at memory location x with the value n.

[6] We assume that the primitive read, write and compare-and-swap (CAS) memory operations are atomic.

```
function lock(x) {              function unlock(x) {
  t := incr(x.next);              wkincr(x.owner);
  do {                          }
    v := read(x.owner)
  } while (v ≠ t);
}
```

Fig. 1. A ticket lock implementation using the counter module.

A specification 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[7]) `wkincr` requires that there is no concurrent operation which changes the value of the counter between the read and increment of the value.

2.2 Ticket Lock Client

The ticket lock algorithm [16] uses the counter module to provide synchronisation. The code for the lock is given in Fig. 1. The lock uses two counters, `next` and `owner`, which both initially have value 0. A thread acquires the lock by calling the `lock` 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 `unlock` operation. This increments the `owner` counter, passing the lock on to the next waiting thread. Intuitively, the use of `incr` for the `lock` operation is necessary, since it needs to be robust with respect to concurrent threads taking tickets. The use of `wkincr` for the `unlock` operation is possible since only the thread holding the lock should release it.

The challenge is to develop a concurrent specification of a counter module that is strong enough to reason about the ticket lock. This example requires a precise description of how each operation affects the value of the counter, and a detailed account of interference to capture the intuitive distinction between `incr` and `wkincr`.

The counter and its ticket lock client are realistic examples that illustrate key difficulties in specifying and reasoning about concurrent modules.

3 Sequential Specification

We can give a sequential specification for the counter module using Hoare triples [11]:

$$
\begin{gathered}
\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n) \land \mathsf{ret} = n\} \\
\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1) \land \mathsf{ret} = n\} \\
\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1)\}
\end{gathered}
$$

[7] In a quick and dirty experiment, `wkincr` was around 60% faster than `incr`.

With standard Hoare logic, we can use this specification to verify sequential clients that call the counter operations. However, this specification gives no information about the behaviour of the operations in a concurrent setting.

4 Auxiliary State

Owicki and Gries [19] 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 others' proofs. This is achieved by extending standard Hoare logic with the Owicki-Gries rule for parallel composition:

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\} \quad \mathsf{non\text{-}interference}}{\{P_1 \wedge P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \wedge Q_2\}}$$

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.

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* — an assertion that is preserved by each atomic action in the module. For the counter, the invariant $\exists n.\, \mathsf{C}(\mathtt{x}, n)$ asserts that the counter at `x` is allocated and has some value.

We can give the following specification for the counter module:

$$\begin{gathered}
\{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\exists n, m.\, \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = m\} \\
\{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\exists n, m.\, \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = m\} \\
\{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}
\end{gathered}$$

However, these specifications are too weak to specify such clients 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 it would 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 code is instrumented with *auxiliary code*, which updates the auxiliary variables. Since the auxiliary code only updates auxiliary variables, it has no effect on the program behaviour, and so can be erased — it is not required when the program is run.

$$
\begin{array}{c}
\{\mathsf{C}(\mathtt{x},0)\}\\
\mathtt{y} := 0;\ \ \mathtt{z} := 0;\\
\{\mathsf{C}(\mathtt{x},0) \land \mathtt{y}=0 \land \mathtt{z}=0\}\\
\begin{array}{l||l}
\{\mathsf{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \land \mathtt{y}=0\} & \{\mathsf{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \land \mathtt{z}=0\}\\
\mathtt{do}\ \{ & \mathtt{do}\ \{\\
\quad \mathtt{r} := [\mathtt{x}]; & \quad \mathtt{r}' := [\mathtt{x}];\\
\quad \langle \mathtt{b} := \mathtt{CAS}(\mathtt{x},\mathtt{r},\mathtt{r}+1); & \quad \langle \mathtt{b}' := \mathtt{CAS}(\mathtt{x},\mathtt{r}',\mathtt{r}'+1);\\
\qquad \mathtt{if}\ (\mathtt{b})\ \mathtt{y}{+}{+};\rangle & \qquad \mathtt{if}\ (\mathtt{b}')\ \mathtt{z}{+}{+};\rangle\\
\}\ \mathtt{while}\ (\mathtt{b}=0); & \}\ \mathtt{while}\ (\mathtt{b}'=0);\\
\{\mathsf{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \land \mathtt{y}=1\} & \{\mathsf{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \land \mathtt{z}=1\}
\end{array}\\
\{\mathsf{C}(\mathtt{x},2) \land \mathtt{y}=1 \land \mathtt{z}=1\}\\
\{\mathsf{C}(\mathtt{x},2)\}
\end{array}
$$

Fig. 2. Reasoning about concurrent increments using auxiliary state.

By way of example, consider two threads that both increment a counter, as in Fig. 2. The auxiliary variables $\mathtt{y}$ and $\mathtt{z}$, with initial values 0, are used to record the contribution (*i.e.* the number of increments) of each thread. For each thread, the code of the `incr` operation is instrumented with code that updates the auxiliary variables when the `CAS` operations succeed. The auxiliary variables 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 angle brackets, the $\langle \_ \rangle$, which indicate that the `CAS` and auxiliary code should be executed in a single atomic step.

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, each use of the `incr` operation requires the underlying implementation to be extended with auxiliary code to increment the appropriate auxiliary variable. A modular proof would not modify the module code for each use by the client.

Secondly, the `incr` operations require different specifications depending on the client's use: in our example, the counter predicate $\mathsf{C}(\mathtt{x},\mathtt{y}+\mathtt{z})$ uses auxiliary variables $\mathtt{y}$ and $\mathtt{z}$; with three threads, the specification requires three auxiliary variables. A modular proof would give a specification for the module that captures all use cases.

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. A modular proof would be explicit about such assumptions about the behaviour of the client.

Thesis.

The concept of *auxiliary state*, introduced in the Owicki-Gries method, is important in specifying concurrent modules. Auxiliary state abstracts the internal state of threads. It is more convenient to reason using auxiliary variables than to consider the program counter and local variables of each thread in describing invariants. This abstraction is a step towards compositional reasoning. As we shall see, various subsequent approaches have taken a more modular approach to auxiliary state than auxiliary variables provide in the Owicki-Gries method.

5 Interference Abstraction

Jones [13] introduced *interference abstraction*, providing the rely/guarantee method as a way to improve the compositionality of the Owicki-Gries approach. To avoid the global $\mathsf{non\text{-}interference}$ condition, specifications 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 derivation; each atomic update by the thread must be described by the guarantee.

Rely/guarantee specifications have the form $R, G \vdash \{P\}\ \mathbb{C}\ \{Q\}$, where R and G are the rely and guarantee relations respectively. When composing concurrent threads, the guarantee of each thread must be included in the rely of the other. The parallel composition rule is therefore adapted to:

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 \land P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \land Q_2\}}$$

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

$$A, \emptyset \vdash \{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\exists n.\, \mathsf{C}(\mathtt{x}, n) \land \mathsf{ret} \leq n\}$$
$$A, A \vdash \{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\exists n.\, \mathsf{C}(\mathtt{x}, n) \land \mathsf{ret} \leq n\}$$

where $A = \{\mathsf{C}(\mathtt{x}, n) \rightsquigarrow \mathsf{C}(\mathtt{x}, n+1) \mid n \in \mathbb{N}\}$. The `read` specification has an empty guarantee relation indicating that nothing is changed by the read. It has rely relation A stating that the other threads can only increment the counter, although they can do so as many times as they like. The `incr` relation 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 values n, because the context 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:

$$\emptyset, G \vdash \{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1)\}$$

where $G \triangleq \{\mathsf{C}(\mathtt{x}, n) \rightsquigarrow \mathsf{C}(\mathtt{x}, n+1)\}$. 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. Effectively, the increment will 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. Only one thread can call `unlock` at any one time, because only one thread can have the lock. However, suppose one thread calling `unlock` has executed the body of `wkincr`. Then, a second thread may correctly conclude that it now has the lock and release it, before the call of the first thread has returned. This results in a concurrent invocation of `wkincr`. By ruling out all concurrent updates to the counter with an empty rely relation, the above specification does not allow this concurrent behaviour.

By changing the rely, it is possible to allow such concurrent updates, but at the expense of weakening the specification:

$$R, G \vdash \left\{\mathsf{C}(\mathtt{x}, n)\right\}\ \mathtt{wkincr}(\mathtt{x})\ \left\{\exists n' \geq n+1.\, \mathsf{C}(\mathtt{x}, n')\right\}$$

where $R = \{\mathsf{C}(\mathtt{x}, m) \rightsquigarrow \mathsf{C}(\mathtt{x}, m+1) \mid m > n\}$ 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 weakening the rely, we must weaken the postcondition to make it stable.

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

Thesis.

The concept of *interference abstraction*, introduced in the rely/guarantee method, is important in specifying concurrent modules. By abstracting the interactions between different threads, specifications can 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 they may specify it differently, some form of interference abstraction is generally present in subsequent concurrency verification approaches.

6 Resource Ownership

In the Owicki-Gries and rely/guarantee approaches, auxiliary variables provide a mechanism for reasoning about which threads can do what and when. For instance, auxiliary variables can be used to reason about the contribution of individual threads to the counter, as we demonstrated in §4, or that one thread can increment a counter after another. O'Hearn introduced a style of reasoning based on *resource ownership*, developing concurrent separation logic [18] which provides an alternative, more modular approach to such reasoning.

Concurrent separation logic is a Hoare logic, with assertions describing data (such as heap cells or counter objects) treated as resources. Each operation acts on specific resources, with the precondition conferring ownership of the resources it represents. When threads operate on disjoint resources, they do not interfere and so their effects can be simply combined. This principle is embodied in the disjoint parallel composition rule:

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

where the separating conjunction $P_1 * P_2$ describes the disjoint combination of the resources of P_1 and P_2.

We can think of ownership as embodying specialised notions of auxiliary state and interference abstraction. Ownership is a form of auxiliary state: the program does not explicitly record which threads own what resources, it is an abstraction that we use for reasoning. Ownership implements a simple interference abstraction: threads may update the resources that they own, and disjointness of ownership enforces that they cannot interfere with other threads' resources.

In the original concurrent separation logic, it was only possible to reason about shared resource that had been transferred between threads through synchronisation. [8] Subsequent approaches [23,7,6] added support for reasoning about fine-grained concurrency by incorporating various styles of rely/guarantee reasoning over shared resources. Building on this work, the concurrent abstract predicates (CAP) [5] approach introduces abstractions over these shared resources that may be split, effectively allowing concurrent manipulation at the abstract level.

Treating the abstract predicate $\mathsf{C}(\mathtt{x}, n)$ 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,

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

does not hold.

Following Boyland [2], Bornat *et al.* [1] 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:

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

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

[8] In [18], conditional critical regions provide the synchronisation mechanism.

read access:

$$\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, \pi) \land \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1) \land \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1)\}$$

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 $\mathsf{C}(\mathtt{x}, n, \pi)$. Now the resource $\mathsf{C}(\mathtt{x}, n, \pi)$ 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:

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

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

$$\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, \pi) \land \mathsf{ret} \geq n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, 1) \land \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, \pi) \land \mathsf{ret} \geq n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1) \land \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1)\}$$

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

Fig. 3 shows how it can be used to verify the example of two concurrent increments. Whereas in Fig. 2 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 resources that are owned by the thread and encapsulated in the $\mathsf{C}(\mathtt{x}, n, \pi)$ predicate. This idea of subjective auxiliary state is at the core of subjective concurrent separation logic (SCSL) [15] (and the subsequent FCSL [17]).

This specification still has some weaknesses. The `wkincr` operation must still be synchronised with the other operations. Also, 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 [5], the proof depends on the

$$
\begin{array}{c}
\{\mathsf{C}(\mathtt{x},0,1)\}\\
\{\mathsf{C}(\mathtt{x},0,0.5) * \mathsf{C}(\mathtt{x},0,0.5)\}\\
\begin{array}{l||l}
\{\mathsf{C}(\mathtt{x},0,0.5)\} & \{\mathsf{C}(\mathtt{x},0,0.5)\}\\
\mathtt{incr}(\mathtt{x}) & \mathtt{incr}(\mathtt{x})\\
\{\mathsf{C}(\mathtt{x},1,0.5)\} & \{\mathsf{C}(\mathtt{x},1,0.5)\}
\end{array}\\
\{\mathsf{C}(\mathtt{x},1,0.5) * \mathsf{C}(\mathtt{x},1,0.5)\}\\
\{\mathsf{C}(\mathtt{x},2,1)\}
\end{array}
$$

Fig. 3. Ownership-based reasoning for concurrent increments.

atomicity of the underlying counter operations in order to synchronise access to shared resources. The proof does work with any of our abstract specifications, since they simply do not embody the necessary atomicity.

Thesis.

The concept of *resource ownership*, developed by concurrent separation logic and its successors, is important in specifying 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 [5,15,17,3,14]. While it is an effective tool, and can be used to give elegant specifications, something more is required to provide the strong specifications we are seeking.

7 Atomicity

Atomicity is the abstraction that an operation takes effect at a single, discrete instant in time. The concurrent behaviour of such atomic operations is equivalent to a sequential interleaving of the operations. A well-known correctness condition for atomicity, which identifies when the operations of a concurrent module *appear* to behave atomically, is *linearisability* [10]. A client can use such operations as if they were simple atomic operations.

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 it 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 §3, 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. By itself, 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.[9] 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.* [8] 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, $\texttt{incr}(\texttt{x})$ contextually refines $\langle\texttt{incr}(\texttt{x})\rangle$.

CaReSL [22] is a logic for proving contextual refinement of concurrent programs. CaReSL makes use of auxiliary state, interference abstraction and ownership in its proof technique. However, these concepts are not exposed in their 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 in specifying 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.

8 Synthesis

We now examine several approaches that bring together the ideas we have so far discussed to provide expressive specifications for concurrent modules.

8.1 A Higher-Order Approach

One way of overcoming the non-modularity of the Owicki-Gries method was introduced by Jacobs and Piessens [12]. 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.

[9] In general, the specification code need not be directly executable, although it does have a semantics.

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

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

Note that ρ is an auxiliary code parameter of the operation. When the atomic update to the counter occurs, the auxiliary code is run and can update auxiliary variables of the client.

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

$$\frac{I(\mathtt{x}) * S \Leftrightarrow \exists n.\, \mathsf{C}(\mathtt{x}, n) * R(n) \quad \forall n.\, \{R(n) * P\}\ \rho\ \{R(n+1) * Q(n)\}}{I(\mathtt{x}) \vdash \{S * P\}\ \mathtt{incr}(\mathtt{x}, \rho)\ \{S * Q(\mathsf{ret})\}}$$

In the conclusion of this rule, $I(\mathtt{x})$ 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:

(i) The equivalence from the premiss is used to convert the combination of the invariant $I(\mathtt{x})$ and the portion of the precondition S into the counter predicate $\mathsf{C}(\mathtt{x}, n)$ and $R(n)$ for some value of n.

(ii) The module performs the increment, updating $\mathsf{C}(\mathtt{x}, n)$ to $\mathsf{C}(\mathtt{x}, n+1)$.

(iii) The auxiliary code ρ is run, updating $R(n) * P$ to $R(n+1) * Q(n)$.

(iv) Together, $\mathsf{C}(\mathtt{x}, n+1)$ and $R(n+1)$ are converted back to recover the invariant $I(\mathtt{x})$ and S.

This specification now allows us to exploit the expressivity of auxiliary variables in a modular way. Fig. 4 shows how this technique can be used to prove two concurrent increments. The proof is very similar to the one shown in Fig. 2. The new specification allows us to abstract the atomic update performed by the `incr` and use the same module implementation for both threads.

In Fig. 4, the invariant $I(\mathtt{x})$ is instantiated as $\mathsf{C}(\mathtt{x}, \mathsf{y} + \mathsf{z})$. The predicate $R(n)$ is instantiated as $n = \mathsf{y} + \mathsf{z}$. The predicate S is True, while P and Q are instantiated with the pre- and postconditions of `incr` at each call site.

The `read` operation can be specified as:

$$\frac{I(\mathtt{x}) * S \Leftrightarrow \exists n.\, \mathsf{C}(\mathtt{x}, n) * R(n) \quad \forall n.\, \{R(n) * P\}\ \rho\ \{R(n) * Q(n)\}}{I(\mathtt{x}) \vdash \{S * P\}\ \mathtt{read}(\mathtt{x}, \rho)\ \{S * Q(\mathsf{ret})\}}$$

$$\begin{array}{c}
\{\mathsf{C}(\mathtt{x},0)\}\\
\mathtt{y} := 0;\ \mathtt{z} := 0;\\
\{\mathsf{C}(\mathtt{x},0) \land \mathtt{y} = 0 \land \mathtt{z} = 0\}\\
\begin{array}{c|c}
\begin{array}{c}\text{invariant:}\\ \mathsf{C}(\mathtt{x},\mathtt{y}+\mathtt{z})\end{array} &
\begin{array}{c}
\{\mathtt{y} = 0 \land \mathtt{z} = 0\}\\
\begin{array}{l||l}
\{\mathtt{y} = 0\} & \{\mathtt{z} = 0\}\\
\mathtt{incr}(\mathtt{x}, \mathtt{y++;}); & \mathtt{incr}(\mathtt{x}, \mathtt{z++;});\\
\{\mathtt{y} = 1\} & \{\mathtt{z} = 1\}
\end{array}\\
\{\mathtt{y} = 1 \land \mathtt{z} = 1\}
\end{array}
\end{array}\\
\{\mathsf{C}(\mathtt{x},2) \land \mathtt{y} = 1 \land \mathtt{z} = 1\}\\
\{\mathsf{C}(\mathtt{x},2)\}
\end{array}$$

Fig. 4. Reasoning about concurrent increments using parametrised auxiliary code.

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

$$\frac{I(\mathtt{x}) * P \Leftrightarrow \mathsf{C}(\mathtt{x}, n) * R \quad \{\mathsf{C}(\mathtt{x}, n+1) * R\}\ \rho\ \{I(x) * Q\}}{I(\mathtt{x}) \vdash \{P\}\ \mathtt{wkincr}(\mathtt{x}, \rho)\ \{Q\}}$$

A key difference with the `wkincr` specification is that n is not quantified in each of 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 proof rules, they are actually implications. If a client establishes the premisses then it can use the conclusion. The implementation must show that the conclusion follows from the premisses. The predicates I, P, Q, R and S, 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 [21], iCAP [20] and Iris [14]. In these logics, auxiliary state is not manipulated by auxiliary code, but by *view shifts* [4]. These view shifts serve essentially the same purpose — they can update auxiliary state, but have no effect on the concrete state — but do not involve instrumenting the code.

8.2 A First-Order Approach

An alternative way of providing specifications for concurrent modules was introduced in the program logic TaDA [3] using *atomic triples*. Rather than treating atomic specifications as a higher-order construct, atomic triples build such specifications in to TaDA as a first-order construct. An atomic triple has the following form:

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

Superficially, this can be read as "$\mathbb{C}$ atomically updates $P(x)$ to $Q(x)$ (for arbitrary $x \in X$)". What it actually means is a bit more subtle.

An implementation of the specification may assume that the assertion $P(x_0)$ holds initially (for some $x_0 \in X$). It must tolerate interference from the environment updating $P(x)$ to $P(x')$ (for any $x, x' \in X$). It is at liberty to update the state, providing that it preserves $P(x)$ (for the current value of x), until it updates it to $Q(x)$. After this update $Q(x)$ is no longer available to the implementation (another thread may be using it). Finally, the implementation cannot terminate without having updated $P(x)$ to $Q(x)$ at some point.

Using the atomic triple, we can specify the counter as:

$$\mathbb{V}n.\, \langle \mathsf{C}(\mathtt{x}, n) \rangle \ \mathtt{read(x)} \ \langle \mathsf{C}(\mathtt{x}, n) \land \mathsf{ret} = n \rangle$$
$$\mathbb{V}n.\, \langle \mathsf{C}(\mathtt{x}, n) \rangle \ \mathtt{incr(x)} \ \langle \mathsf{C}(\mathtt{x}, n+1) \land \mathsf{ret} = n \rangle$$
$$\langle \mathsf{C}(\mathtt{x}, n) \rangle \ \mathtt{wkincr(x)} \ \langle \mathsf{C}(\mathtt{x}, n+1) \rangle$$

Intuitively, the first two specifications state the value of the counter will be read and incremented atomically, even in the presence of concurrent updates by the environment that change the value of the counter — since the value n is bound by $\mathbb{V}$. However, the environment must preserve the counter, e.g. it cannot deallocate it. The last specification means that `wkincr(x)` will atomically update the counter from n to $n+1$, as long as the environment guarantees that the shared counter will not change value before the atomic update — since the value of n is not bound by $\mathbb{V}$.

Atomic triples specify operations with respect to an abstraction (*e.g.* $\mathsf{C}(x, n)$), which means that each operation can be verified independently. 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: adding new operations (*e.g.* `wkincr`) can break the linearisability.

In [3], we introduce a generalised version of the atomic triple that can combine atomicity with resource transfer. For example, we can 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 is transferred between the client and implementation. This is not possible with traditional linearisability, although Gotsman and Yang [9] have proposed an extension of linearisability that supports ownership transfer.

Evaluation.

The counter specifications shown in this section are strong: a client can derive the abstract disjoint specifications from them. Moreover, they are strong enough to support synchronisation: the correctness of the ticket lock can be justified from the counter specifications. These approaches to specification are expressive enough to enforce obligations on both the client and the implementation. By contrast, CAP specifications tend to unduly restrict the client (*e.g.* a counter specification cannot be used for synchronisation), while linearisability specifications tend to unduly restrict the implementation (*e.g.* a counter cannot provide a `wkincr` operation).

Atomic triples have been encoded in Iris [14] by interpreting them as specifications in the Jacobs-Piessens style. This captures the intensional meaning behind atomic triples — that is, what they can be used for — which in TaDA is expressed through the proof rules for using atomic triples.

9 Conclusions

We have considered a number of proof methods for verifying concurrent programs: Owicki-Gries, rely/guarantee, concurrent separation logics and linearisability. In each method, we have identified a particularly valuable contribution towards specifying concurrent modules. Finally, we have demonstrated how these ideas can be brought together to produce specifications that are both expressive and modular.

References

[1] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[2] John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, pages 207–231, 2014.

[4] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.

[5] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[6] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-Guarantee Reasoning. In *ESOP*, pages 363–377, 2009.

[7] Xinyu Feng. Local rely-guarantee reasoning. In *ACM SIGPLAN Notices*, volume 44, pages 315–327. ACM, 2009.

[8] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for Concurrent Objects. In *ESOP*, pages 252–266, 2009.

[9] Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. In *CONCUR*, pages 256–271, 2012.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

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

[12] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[13] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

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

[15] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574, 2013.

[16] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[17] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.

[18] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

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

[20] Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, pages 149–168, 2014.

[21] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, pages 169–188, 2013.

[22] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[23] Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, Computer Laboratory, 2008.