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.


## 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

```
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