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

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.