Journal of Logical and Algebraic Methods in Programming 85 (2016) 972–984

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Possible values: Exploring a concept for concurrency

Cliff B. Jones [a,*], Ian J. Hayes [b]

[a] School of Computing Science, Newcastle University, UK
[b] School of Information Technology and Electrical Engineering, The University of Queensland, Australia

ARTICLE INFO

Article history:
Received 15 June 2015
Received in revised form 10 September 2015
Accepted 8 January 2016
Available online 13 January 2016

Keywords:
Concurrent programming
Rely/guarantee conditions
Possible values

ABSTRACT

An important issue in concurrency is interference. This issue manifests itself in both shared-variable and communication-based concurrency — this paper focuses on the former case where interference is caused by the environment of a process changing the values of shared variables. Rely/guarantee approaches have been shown to be useful in specifying and reasoning compositionally about concurrent programs. This paper explores the use of a "possible values" notation for reasoning about variables whose values can be changed multiple times by interference. Apart from the value of this concept in providing clear specifications, it offers a principled way of avoiding the need for some auxiliary (or ghost) variables whose unwise use can destroy compositionality.

1. Introduction

High on the list of issues that make the design of concurrent programs difficult to get right is 'interference'. Reproducing a situation that exhibited a 'bug' can be frustrating; attempting to reason informally about all possible interleavings of interference can be exasperating; and designing formal approaches to the verification of concurrent programs is challenging.

Recording post conditions for sequential programs applies the only real tool that we have: abstraction is achieved by winnowing out what is inessential in the relationship between the initial and final states of a computation. Post conditions record the required relationship without fixing an algorithm to bring about the transformation; furthermore, they record required properties only of those variables which the environment will use. The rely/guarantee approach (see Section 1.1) uses abstraction in the same way to provide specifications of concurrent software components that are more abstract than their implementations: for any component, rely conditions are relations that record interference that the component must tolerate and guarantee conditions document the interference that the environment of the component must accept.

This paper explores a concept that fits well with rely/guarantee reasoning but probably has wider applicability. In relational post conditions, it is necessary to be able to refer to the initial value x and final value x' of a variable x (e.g. $x \le x' \le x + 9$). If however it is necessary to record something as simple as the fact that a local variable x captures one of the values of a shared variable y, it is inadequate to write $x' = y \vee x' = y'$ in the case where y might be changed many times by the environment. Enter 'possible values': the suggested notation is that $\widehat{y}$ denotes the set of values which variable y contains during the execution of the operation in whose specification $\widehat{y}$ is written. So, (assuming the access to read the value of y is atomic):

$$\textit{post-Op} : x' \in \widehat{y}$$

is satisfied by a simple assignment of y to x.

* Corresponding author.
E-mail address: cliff.jones@ncl.ac.uk (C.B. Jones).

http://dx.doi.org/10.1016/j.jlamp.2016.01.002
2352-2208/© 2016 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```
                                  $f \leftarrow \mathsf{wr}$;
while true do                         ||   while true do
  $\cdots$ produce $v \cdots$          ||     while $f = \mathsf{wr}$ do skip od;
  while $f = \mathsf{rd}$ do skip od;  ||     $r \leftarrow b$;
  $b \leftarrow v$;                    ||     $f \leftarrow \mathsf{wr}$
  $f \leftarrow \mathsf{rd}$           ||     $\cdots$ consume $r \cdots$
od                                    ||   od

guar $(f = \mathsf{rd} \Rightarrow b' = b) \wedge$      rely $(f = \mathsf{rd} \Rightarrow b' = b) \wedge$
     $(f = \mathsf{rd} \Rightarrow f' = \mathsf{rd})$           $(f = \mathsf{rd} \Rightarrow f' = \mathsf{rd})$
rely $f = \mathsf{wr} \Rightarrow f' = \mathsf{wr}$           guar $f = \mathsf{wr} \Rightarrow f' = \mathsf{wr}$
```

Fig. 1. A one-place buffer.

1.1. Rely/Guarantee thinking

Before going into more detail on the possible values notation (see Section 2), a brief overview of background work is offered. The specifications given in Section 3 are written in the notation of VDM [12,15]. It is unlikely that they will present difficulties even to readers unfamiliar with that specific notation because similar ideas for sequential programs are present in Z [6], B [1], Event-B [2], and TLA [23]. The basic idea is of state-based specifications with operations (or events) transforming the state and being specified by something like pre and post conditions. Pre conditions are predicates over states that indicate what can be assumed about states in which an operation can be initiated. Post conditions are relations over initial and final states that specify the required relations between the initial and final values of state components. Good sequential specifications eschew any details of implementation algorithms: they do not specify anything about intermediate states; in fact an implementation might use a state with more components. At first sight, it might appear surprising that there is not a precise functional requirement on the final state but using non-determinism in specifications turns out to be an extremely useful way of postponing design decisions.

The use of abstract objects in specifications is a crucial tool for larger applications. Moreover, datatype invariants can make specifications clearer: restricting types by predicates simplifies pre/post conditions and also offers a way for the specifier to record the intention of a specification. Another useful aspect of VDM is the ability to define more tightly the ‘frame’ of an operation by recording whether access to state components is for (only) reading or for both reading and writing.[1]

The basic rely/guarantee [13,14] idea[2] is simple: interference is documented and proof rules are given which support reasoning about interference in concurrent threads. Just as in sequential specifications, the role of a state is central to recording rely/guarantee specifications. For concurrency, it is accepted that the environment of a process can change values in the state during execution of an operation.[3] Such changes are however assumed to be constrained by a rely condition. In order to reason about the combined effect of operations, the interference that a process can inflict on its environment is also recorded; this is done in a guarantee condition. Both rely and guarantee conditions are, for obvious reasons, relations over states. In the original form – and after many experiments – both conditions are reflexive and transitive covering the possibility of zero or many steps. Such relations often indicate monotonic evolution of variables.

It is useful to compare the roles of rely and guarantee conditions with the better known pre/post conditions. Pre conditions are essentially an invitation to the designer of a specified component to ignore some starting states; in the same way, the developer can ignore the possibility that interference will make state changes that do not satisfy the rely condition. In neither case should a developer include code to test these assumptions; there is an implicit requirement to prove that the component is only used in an appropriate context. In contrast, post conditions and guarantee conditions are obligations on the running code that the developer has to create; these conditions record properties on which the deployer can depend.

The simplest form of relation that could be used in rely or guarantee conditions is to state that the value of a variable remains unchanged (e.g. $b' = b$). Such unconditional constraints are normally better handled by marking an operation (or part thereof) as having only read access. There is however an important way to combine ‘monotonic’ changes to flags with assertions about variables remaining unchanged. Consider a simple one-place buffer in which a producer process places a value in a buffer variable b from which a consumer process extracts values. Testing and setting flag f in Fig. 1 ensures that the producer and consumer alternate their access to b. During its read phase, the consumer needs to rely on the fact that the value of b cannot change but this is too strong as a rely condition for the whole of the consumer process — the producer process could never insert anything into the buffer if it were required to achieve a guarantee condition of $b' = b$. But the consumer process can instead rely on $f = \mathsf{rd} \Rightarrow b' = b$, which in turn is easy for the producer to guarantee. The ‘monotonic’ behaviour of the flags means that the producer has also to guarantee that $f = \mathsf{rd} \Rightarrow f' = \mathsf{rd}$ and the consumer must guarantee $f = \mathsf{wr} \Rightarrow f' = \mathsf{wr}$. This example shows one way in which rely/guarantee conditions can be used

[1] Most of the literature on rely/guarantee conditions is limited to normal (or ‘scoped’) variables; [21] shows how ‘heap’ variables can be viewed as representations of more abstract states.

[2] The literature on rely/guarantee approaches continues to expand; see [9,11] for further references. For a reader who is completely unfamiliar with rely/guarantee concepts, a useful brief presentation can be found in [16].

[3] Notice that there is an essential difference here from ‘actions’ [5] or ‘events’ [2] which view execution of a guarded action as atomic.