Science of Computer Programming 74 (2009) 143–165

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A general technique for proving lock-freedom

Robert Colvin *, Brijesh Dongol

ARC Centre for Complex Systems, School of Information Technology and Electrical Engineering, The University of Queensland, Australia

ARTICLE INFO

Article history:
Received 14 May 2008
Received in revised form 29 August 2008
Accepted 22 September 2008
Available online 1 October 2008

Keywords:
Lock-free programs
Concurrency
Verification
Temporal logic

ABSTRACT

Lock-freedom is a property of concurrent programs which states that, from any state of the program, eventually some process will complete its operation. Lock-freedom is a weaker property than the usual expectation that eventually *all* processes will complete their operations. By weakening their completion guarantees, lock-free programs increase the potential for parallelism, and hence make more efficient use of multiprocessor architectures than lock-based algorithms. However, lock-free algorithms, and reasoning about them, are considerably more complex.

In this paper we present a technique for proving that a program is lock-free. The technique is designed to be as general as possible and is guided by heuristics that simplify the proofs. We demonstrate our theory by proving lock-freedom of two non-trivial examples from the literature. The proofs have been machine-checked by the PVS theorem prover, and we have developed proof strategies to minimise user interaction.

1. Introduction

Lock-freedom is a progress property of non-blocking concurrent programs which ensures that the system as a whole makes progress, even if some processes never make progress [14,15,8]. Lock-free programs tend to be more efficient than their lock-based counterparts because the potential for parallelism is increased [15]. However, compared to lock-based implementations, lock-free programs are more complex [15,12], and can be error-prone (see, e.g., [3] for discussion on an incorrect published algorithm). Although formal safety proofs for lock-free algorithms exist [7,3,4], formal proofs of progress have mostly been ignored.

Our definition of lock-freedom is based on a literature survey and formal definition given by Dongol [8] and states that from any state of the program, eventually some process executes the final line of code of its operation, i.e., executes a *completing* action. The initial step in proving this property is to identify a set of *progress* actions (which includes the *completing* actions), and prove that if eventually some process executes a progress action, it follows that eventually some process will execute a *completing* action. Having widened the set of actions of interest, we augment the program with an auxiliary variable which detects the execution of progress actions. We use this variable to construct a *well-founded relation* on states of the program, such that every transition of the program results in an improvement with respect to the relation, or is the execution of a progress action.

The steps outlined above are guided by heuristics, and in particular by inspection of the control-flow graphs of the program. The final step involves case analysis on the actions of the system and only requires simple propositional reasoning. The proofs are supported by the theorem prover PVS: we provide a theory for showing that the relations are well-founded, and strategies for minimising user interaction in the case analysis.

Overview. This paper is organised as follows. In Section 2 we present the theoretical background to the rest of the paper. In Section 3 we present the technique and apply it to a running example, the Michael & Scott lock-free queue [15]. In Section 4

* Corresponding author.
E-mail addresses: robert@itee.uq.edu.au (R. Colvin), brijesh@itee.uq.edu.au (B. Dongol).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.013

$$\mathcal{P} \;\widehat{=}\; \|\ p{:}\ procs(\mathcal{P}) \bullet \texttt{while true do}\ \{\ \sqcap\ op{:}\ OP(\mathcal{P}) \bullet op\ \}$$
$$op \;\widehat{=}\; op.preLoop;\ op.Loop;\ op.postLoop$$

Fig. 1. Typical structure of lock-free algorithms.

we apply the technique to a more complex example, a bounded array-based queue [3]. In Section 5 we describe how the proofs can be checked using the PVS theorem prover [18], using a number of strategies for guiding the proofs to minimise user interaction. We conclude and discuss related work in Section 6.

2. Preliminaries

We describe the programming model and the general structure of a lock-free program in Section 2.1; provide the lock-free queue by Michael and Scott as a concrete example in Section 2.2; briefly review well-founded relations in Section 2.3; and describe transition systems, trace-based reasoning and temporal logic in Section 2.4.

2.1. Lock-free programs

The general structure of a lock-free program is summarised by the program, $\mathcal{P}$, in Fig. 1, where $\mathcal{P}$ is formed by a finite set of parallel processes, $procs(\mathcal{P})$, that execute operations from $OP(\mathcal{P})$. Each process p selects some operation op for execution. After op completes, p makes another selection, and so on. This form of $\mathcal{P}$ is an abstraction of a real lock-free program, where processes are likely to perform other functions between calls to operations in $\mathcal{P}$. We assume that functions external to $\mathcal{P}$ do not alter its variables, and hence may be ignored for the purposes of defining $\mathcal{P}$. We say a process is *idle* if the process is not executing any operation, i.e., is between calls to operations. Note that because each process continually chooses new operations for execution, each execution of $\mathcal{P}$ is infinite in length.

Each operation $op \in OP(\mathcal{P})$ is of the form described in Fig. 1. They are sequential non-blocking statements, i.e., no atomic statement of op exhibits blocking behaviour. The main part of the operation occurs in $op.Loop$, which is a potentially infinite retry-loop. Given that $op.Loop$ modifies shared data G, the typical structure of $op.Loop$ is to take a snapshot of G, construct a new value for G locally, then attempt to update G to this new value. If no interference has occurred, i.e., G has not been modified (by a different process) since the snapshot was taken, $op.Loop$ terminates, whereas if interference has occurred $op.Loop$ is retried. Because $op.Loop$ is retried based on interference from external sources, a loop variant, for proving loop termination in the traditional way, cannot be derived.

An operation op may also require some pre- and post-processing tasks to $op.Loop$, as given by $op.preLoop$ and $op.postLoop$. Both $op.preLoop$ and $op.postLoop$ are assumed not to contain potentially infinite loops. Note that $op.preLoop$ and $op.postLoop$ may be empty for some operations. Because $op.Loop$ may contain multiple exit points, there could be multiple control flows within $op.postLoop$, i.e., the structure in Fig. 1 is simplified for descriptive purposes.

Within an operation, each atomic statement has a unique *label*, and each process p has a *program counter*, denoted pc_p, whose value is the label of the next statement p will execute. We assume the existence of a special label idle to identify idle processes. We use $PC(\mathcal{P})$ to denote the set of all labels in the program (including idle).

Lock-free programs are typically implemented using hardware primitives such as Compare-and-Swap (CAS), which combines a test and update of a variable within a single atomic statement. A procedure call CAS(G, ss, n) operates as follows: if (shared) variable G is the same as (snapshot) variable ss, then G is updated to the value of n and *true* is returned; otherwise no update occurs and *false* is returned. CAS instructions are available on many current architectures, including x86.

```
CAS(G, ss, n) ≙ if G = ss
                then G := n ; return true
                else return false
```

CAS-based implementations can suffer from the "ABA problem" [15], which can be manifested in the following way. A process p takes a snapshot, say ss_p, of the shared variable G when G's value is A; then another process modifies G to B, then back again to A. Process p has been interfered with, but the CAS that compares G and ss_p cannot detect the modification to G because $G = ss_p$ holds after the interference has taken place. If value A is a pointer, this is a potentially fatal problem because the contents of the location pointed to by A may have changed. A work-around is to store a *modification counter* with each shared variable, which is incremented whenever the variable is modified. If the modification counter is also atomically compared when executing the CAS, any interference will be detected. As discussed by Michael and Scott [15], modification counters do not constitute a full solution to the ABA problem in a real system because modification counters will be bounded. However, in practice, the likelihood of the ABA problem occurring is reduced to a tolerably small level [17].

2.2. Example: The Michael and Scott queue

To understand how typical lock-free operations are implemented, consider the program $\mathcal{MSQ}$ in Fig. 2, which is the lock-free queue algorithm presented by Michael and Scott [15]. The algorithm is used as the basis of the implementation