ATOMIC SEMANTICS OF NONATOMIC PROGRAMS*

James H. Anderson and Mohamed G. Gouda

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-20 May 1987
Revised March 1988

*Work supported in part by Office of Naval Research Contract N00014-86-K-0763.

Atomic Semantics of Nonatomic Programs*

James H. Anderson Mohamed G. Gouda

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

December 1987

Abstract

We argue that it is possible, and sometimes useful, to reason about nonatomic programs within the conventional atomic model of concurrency.

1 Introduction

Most of the proof methods that have been proposed for reasoning about concurrent programs are developed within the atomic model of concurrency [Ho 72,LS 84,MP 84,OG 76]. This model is based on the assumption that no two operations in a concurrent program are executed at the same time. Hence, the resulting proof theory may seem inadequate for reasoning about programs in which operations of different processes may overlap. In this paper, we show to the contrary that it *is* possible to reason about such programs within the atomic model of concurrency.

*Work supported in part by Office of Naval Research Contract N00014-86-K-0763.

2 Nonatomic Programs

A *concurrent program* consists of two or more sequential processes that access a set of program variables. A *program state* is a mapping from the program variables and program counters to values. A *process* is a sequence of operations, each of which is either atomic or nonatomic. An *atomic operation* is a relation on the set of program states; that is, an atomic operation causes a single transition between a pair of states. A *nonatomic operation* is a sequence of atomic operations; that is, a nonatomic operation may cause several state transitions. Intuitively, the execution of an atomic operation is instantaneous and does not overlap the execution of another atomic operation, whereas the execution of a nonatomic operation lasts for an arbitrary, but finite, period of time and may overlap the execution of other operations.

If each process in a concurrent program consists only of atomic operations, then the program is called *atomic*; otherwise, it is called *nonatomic*.

Each program variable is either global or local: a variable is *global* if it is accessed by more than one process, and *local* if it is accessed by only one process. Since each process is sequential, we may assume that an operation that accesses no global variable is atomic. By contrast, an operation that does access a global variable may be nonatomic.

In this note, we consider a class of nonatomic programs in which processes communicate *only* by reading and writing global variables. For now, we assume that the value of each global variable ranges over a finite domain $\{0, \ldots, M-1\}$, for some M called the *range* of the global variable. (Later, in Section 5, we relax this restriction.) We also assume that each variable is written by only one process. Furthermore, we assume that global variables are accessed according to the following rules.

1. *(Atomicity)* Each operation either reads or writes at most one global variable. An operation that reads a global variable is atomic, and an operation that writes a global variable is nonatomic.

2. *(Reading While Writing)* If a read of a global variable occurs while the variable is being written, then the read operation returns an arbitrary value from the value domain of the variable.

3. *(Exclusive Reading)* If a read of a global variable does not occur while the variable is being written, then the read operation returns the most recently written value.

For convenience, each nonatomic write operation is distinguished by the special syntax:

write v **to** x

where x is a global variable, and v is the value being written.

3 Semantics

In [La 77], Lamport defines the semantics of the nonatomic write operation "**write** v **to** x" by the atomic program fragment

$\langle x := ? \rangle;$
$\langle x := v \rangle$

where "⟨" and "⟩" enclose the atomic operations, and "?" is an indeterminate value. Furthermore, a read operation that reads x when $x = ?$ returns an arbitrary value from the value domain of x. Therefore, the semantics of a read operation that reads x must be augmented to allow the possibility of $x = ?$. In particular, the value of x must now be viewed as a relation instead of a function.

We would like to suggest an alternative approach to defining the semantics of the nonatomic write operation "**write** v **to** x". In this approach, it is unnecessary to redefine the semantics of a read operation. Instead, the semantics of the write operation is defined by a nondeterministic program fragment along with a fairness condition. The program fragment is as follows:

do $\langle true \rightarrow x := x + 1$ **modulo** $M \rangle$
$[\!]$ $\langle true \rightarrow x := v;$ **exit** $\rangle$
od

where M is the range of x, and the second branch of the **do-od** loop is called the *exit branch*. The fairness condition can be stated as follows.

> If the exit branch is continuously enabled, then it is eventually executed.

This condition guarantees that the program fragment terminates in a finite time, and, consequently, the duration of the corresponding nonatomic write operation is finite.

```
local var  k: 1..N;

while true do
            ⟨Noncritical Section⟩;
         1: write true to a[i];
         2: ⟨k := 1⟩;
         3: while ⟨k ≤ i − 1⟩ do
                     4: if ⟨a[k]⟩ then
                                  5: write false to a[i];
                                  6: while ⟨a[k]⟩ do 7: ⟨skip⟩ od;
                                  8: ⟨goto 1⟩
                        fi;
                     9: ⟨k := k + 1⟩
            od;
        10: ⟨k := i + 1⟩;
        11: while ⟨k ≤ N⟩ do
                     12: while ⟨a[k]⟩ do 13: ⟨skip⟩ od;
                     14: ⟨k := k + 1⟩
            od;
        15: ⟨Critical Section⟩;
        16: write false to a[i]
od
```

Figure 1: Process P_i of a nonatomic program.

4 Verification

The above semantics suggests the following method for verifying that a nonatomic program P satisfies some assertion, under some fairness condition F. First, translate the pair (P, F) into a pair (P', F'), where P' is an atomic program. Second, show that P' satisfies the required assertion under the fairness condition F'. Since P' is atomic, this step can be accomplished using traditional proof methods, i.e. invariants and well-founded sets [MP 84].

As an example, consider the one-bit mutual exclusion program given in [La 86a]. The program, call it P, consists of N processes, $P_1, \ldots, P_N$, that communicate via a global boolean array $a[1..N]$; each element in the array is initially false. The code for process P_i is shown in Figure 1, and the

fairness condition F associated with P is *true*.

As discussed earlier, the pair (P, F) can be translated into a pair (P', F'). The code for process P_i' in the resulting program P' is shown in Figure 2. The fairness condition F' is as follows:

> If any exit branch is continuously enabled, then it is eventually executed.

Now, to prove that program P satisfies the *mutual exclusion* property

$$S \equiv [\ \forall i, j : i \neq j : \neg(P_i \textbf{ at } \{15\} \wedge P_j \textbf{ at } \{15\})\ ]$$

at each of its reachable states, it is sufficient to show that the atomic program P' satisfies S at each of its reachable states. (Fairness is not needed in proving mutual exclusion, since it is a safety property.) This can be done by finding a suitable invariant of P'. To this end, let k_i denote the local variable k of process P_i', and let z_i be an auxiliary variable of process P_i' defined as follows:

$$z_i = \begin{cases} 1 & \text{if } P_i' \textbf{ at } \{9, 14\} \\ 0 & \text{otherwise} \end{cases}$$

Then, the required invariant is as follows. (Proving that it is indeed an invariant of P' is left to the reader.)

$$J \equiv S \wedge S_1 \wedge S_2 \wedge S_3 \wedge S_4$$

where

$$S_1 \equiv [\ \forall i :: P_i' \textbf{ at } \{2..4, 9..15\} \Rightarrow a[i]\ ]$$
$$S_2 \equiv [\ \forall i :: P_i' \textbf{ at } \{15\} \Rightarrow k_i > N\ ]$$
$$S_3 \equiv [\ \forall i :: P_i' \textbf{ at } \{10\} \Rightarrow k_i \geq i\ ]$$
$$S_4 \equiv [\ \forall i :: P_i' \textbf{ at } \{3, 4, 9..15\} \Rightarrow (\ \forall j : j \neq i \wedge j < k_i + z_i :$$
$$\neg a[j] \vee P_j' \textbf{ at } \{1, 2, 5, 16\} \vee (k_j + z_j \leq i)\ )\ ]$$

We used an interesting "heuristic" in order to deduce the invariant J. We first deduced an invariant I for the nonatomic program (Figure 1), under the assumption that each "**write** *true* **to** $a[i]$" and "**write** *false* **to** $a[i]$" is an atomic operation. We then "massaged" this invariant to get J. The required massaging was slight, since I was very close to J already; in fact, $I \equiv S \wedge S_1 \wedge S_2 \wedge S_3 \wedge R$, where

$$R \equiv [\ \forall i :: P_i \textbf{ at } \{3, 4, 9..15\} \Rightarrow (\ \forall j : j \neq i \wedge j < k_i + z_i :$$
$$\neg a[j] \vee P_j \textbf{ at } \{2\} \vee (k_j + z_j \leq i)\ )\ ]$$

```
local var  k: 1..N;


while ⟨true⟩ do
        ⟨Noncritical Section⟩;
    1:  do ⟨true → a[i] := ¬a[i]⟩
        [] ⟨true → a[i] := true; exit⟩
        od;
    2:  ⟨k := 1⟩;
    3:  while ⟨k ≤ i − 1⟩ do
                4:  if ⟨a[k]⟩ then
                            5:  do ⟨true → a[i] := ¬a[i]⟩
                                [] ⟨true → a[i] := false; exit⟩
                                od;
                            6:  while ⟨a[k]⟩ do 7: ⟨skip⟩ od;
                            8:  ⟨goto 1⟩
                    fi;
                9:  ⟨k := k + 1⟩
        od;
   10:  ⟨k := i + 1⟩;
   11:  while ⟨k ≤ N⟩ do
                12:  while ⟨a[k]⟩ do 13: ⟨skip⟩ od;
                14:  ⟨k := k + 1⟩
        od;
   15:  ⟨Critical Section⟩;
   16:  do ⟨true → a[i] := ¬a[i]⟩
        [] ⟨true → a[i] := false; exit⟩
        od
od
```

Figure 2: Process P_i' of an equivalent atomic program.

5 Concluding Remarks

Our approach can be extended to reason about nonatomic writes to unbounded global variables. For example, the semantics of a nonatomic write to an integer variable x can be defined by the program fragment

$$
\begin{array}{l}
\textbf{do } \langle true \rightarrow x := x + 1 \rangle \\
[\!] \quad \langle true \rightarrow x := x - 1 \rangle \\
[\!] \quad \langle true \rightarrow x := v;\ \textbf{exit} \rangle \\
\textbf{od}
\end{array}
$$

along with the obvious fairness condition.

The semantics that we proposed in Section 3 is, in fact, the semantics of a write operation of a *safe* register. A safe register is the most primitive register in a hierarchy of registers defined by Lamport [La 86b]; it satisfies only one constraint: a read of a safe register must return the most recently written value if it does not "overlap" a write of the register. Another register in Lamport's hierarchy is the *regular* register. A regular register is a safe register that satisfies one additional constraint: a read of a regular register that overlaps a write of the register must return either the "old" or the "new" value. The operation "**write** v **to** x", where x is a regular register, can be defined by the program fragment

$$
\begin{array}{l}
u := x; \\
\textbf{do } \langle true \rightarrow x := u \rangle \\
[\!] \quad \langle true \rightarrow x := v \rangle \\
[\!] \quad \langle true \rightarrow x := v;\ \textbf{exit} \rangle \\
\textbf{od}
\end{array}
$$

along with the usual fairness condition. This example illustrates the fact that our approach is general enough to reason about a variety of shared objects.

The semantics that we suggest is useful for proving safety properties (which specify that something will not occur) and progress properties (which specify that something will occur). However, it is not particularly useful for proving possibility properties (which specify that something may occur). For example, consider a read of a shared variable that occurs while the variable's value is being changed from 0 to 200. To prove that the read may

return the value 500, at least 500 atomic steps are required. An alternative semantics, which is more convenient for proving possibility properties, is obtained by using a nondeterministic selection function. In particular, "**write** v **to** x" can be defined by the following program fragment along with the usual fairness condition:

do $\langle true \rightarrow x := \textbf{select}(\textbf{domain}(x))\rangle$
[] $\langle true \rightarrow x := v;\ \textbf{exit}\rangle$
od

where **select**(...) is the selection function, and **domain**(x) returns the value domain of the variable x.

Recently, Lamport has proposed a proof theory for reasoning about nonatomic programs in which the implementation of the nonatomic operations in terms of atomic operations is left unspecified [La 83,La 87]. Thus, this proof theory allows implementation decisions to be deferred, in contrast to our approach in which implementation decisions are made *a priori*. On the other hand, our approach allows one to reason about program correctness within the conventional atomic framework, instead of appealing to a new theory.

Acknowledgements We are thankful to L. Lamport, C. Lengauer, M. Merritt, F. Schneider, and the referees for their helpful comments on this note.

References

[Ho 72] Hoare, C.A.R., "Towards a Theory of Parallel Programming," *Operating Systems Techniques*, Hoare and Perott (Eds.), Academic Press, New York, 1972.

[La 77] Lamport, L., "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 2, pp. 125-143, March 1977.

[La 83] Lamport, L., "Reasoning About Nonatomic Operations," *Proceedings of the 10th Annual ACM SIGACT-SIGPLAN Symposium on*

Principles of Programming Languages, pp. 28-37, 1983.

[La 86a] Lamport, L., "The Mutual Exclusion Problem, Parts I and II," *Journal of the ACM*, Vol. 23, No. 2, pp. 311-348, April 1986.

[La 86b] Lamport, L., "On Interprocess Communication, Parts I and II," *Distributed Computing*, Vol. 1, pp. 77-101, 1986.

[La 87] Lamport, L., "*win* and *sin*: Predicate Transformers for Concurrency," Technical Report, Systems Research Center, Digital Equipment Corporation, May 1987.

[LS 84] Lamport, L., and Schneider, F., "The Hoare Logic of CSP, and All That," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 2, pp. 281-296, April 1984.

[MP 84] Manna, Z., and Pnueli, A., "Adequate Proof Principles for Invariance and Liveness Properities of Concurrent Programs," *Science of Computer Programming*, Vol. 4, pp. 257-289, 1984.

[OG 76] Owicki, S., and Gries, D., "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, Vol. 6, pp. 319-340, 1976.