15

Serializable Programs, Parallelizable Assertions: A Basis for Interleaving

Mohamed G. Gouda

1 Introduction

Many formal models of concurrent programs are based on the notion of *interleaving*: in executing a concurrent program, only one enabled action is executed at each step. This notion has both its advantages and disadvantages.

The main advantage of interleaving is that it tends to simplify the verification of concurrent programs. For instance, to verify that a given predicate is an invariant of some program requires showing that each action of the program preserves the invariant when executed in isolation. Were it possible for actions to be executed in parallel, checking the same invariant would require showing that each set of actions preserves the invariant when executed in parallel. In other words, the number of cases needed to verify an invariant would have increased from n to $2^n - 1$, where n is the number of actions in the program.

The main disadvantage of interleaving is that it does not reflect our understanding that multiple actions of a concurrent program may in fact be executed at the same time. The discrepancy between our assumption of how concurrent programs are executed (interleaving) and how they are "actually" executed can lead to programs that, although provably correct under interleaving, do not perform as expected when executed. This point is made clearer by the next two examples.

Consider the following program with two boolean variables *left* and *right* and two actions:

$$\begin{array}{lll} left \neq right & \rightarrow & left := right \\ left \neq right & \rightarrow & right := left \end{array}$$

If this program starts in a state satisfying $left \neq right$, its next state, under interleaving, satisfies $left = right$. On the other hand, if the two actions are allowed to execute in parallel, then the program may remain within

the states satisfying $left \neq right$ indefinitely. This discrepancy is caused by the fact that the program can make a transition from $left \neq right$ to $left \neq right$ by executing its two actions in parallel, but cannot make a similar transition by executing the two actions in some sequence.

In order to avoid such a discrepancy, concurrent programs should be designed so that any state that can be reached by executing some actions in parallel can still be reached by executing the same actions in some sequence. We refer to such programs as serializable. (A more formal definition of serializable programs is given in Section 4.)

Consider the following serializable program with three boolean variables and two actions.

$$\begin{array}{lll} left \neq middle & \rightarrow & left := middle \\ right \neq middle & \rightarrow & right := middle \end{array}$$

This program satisfies the assertion

$$(left \neq middle \ \wedge \ right \neq middle) \quad leads\text{-}to \quad left \neq right$$

under interleaving; this is because if the program starts at a state where $left \neq middle$ and $right \neq middle$, which implies $left = right$, then after executing exactly one action the next state satisfies $left \neq right$. On the other hand the program does not satisfy the assertion if the two actions are allowed to execute in parallel. Thus, the correctness of this program should not be based on this or similar assertions. Rather, it should be based on what we call parallelizable assertions, i.e. assertions that if satisfied under interleaving are also satisfied when actions are executed in parallel. (A more formal definition of parallelizable assertions is given in Section 3.)

In summary, as long as verification of concurrent programs is based on interleaving (and as mentioned earlier, there are good reasons for continuing this practice), one should design only serializable programs and base their correctness only on parallelizable assertions.

In the remainder of this note, we define the notions of parallelizable assertions and serializable programs and identify a reasonable set of parallelizable assertions for the family of serializable programs.

2 Concurrent programs

Consider a *program* that consists of a set of *variables* and a set of *actions*; both sets are finite and nonempty. Each action is of the form

$$G \rightarrow x.1, \ldots, x.n := F.1, \ldots, F.n$$

where G is a predicate called the *guard* of the action, the $x.i$'s are distinct variables, and the $F.i$'s are total functions of the program variables.

The set of actions is partitioned into one or more (mutually exclusive) subsets such that each variable is written by the actions of at most one subset. We call these subsets *processes*.

A *state* is defined by one value for each variable.

A *transition* is a nonempty set of actions with at most one action from each process. A transition that has exactly one action is called a *serial transition*.

A state q *follows* a state p *over* a transition t iff the guard of each action in t is true at p, and q can be computed starting from p by parallel execution of all the actions in t.

This notion of "follows-over" can be extended to finite sequences of transitions. A state q *follows* a state p *over* $(t.1, \ldots, t.r)$ iff there are states $p.1, \ldots, p.(r+1)$ such that $p = p.1$, $q = p.(r+1)$, and for each i, $p.(i+1)$ follows $p.i$ over $t.i$.

A (*serial*) *computation* is a maximal sequence

$$p.1, t.1, p.2, t.2, \ldots, p.r, t.r, p.(r+1)$$

where each $p.i$ is a state, each $t.i$ is a (serial) transition, and for each i, $p.(i+1)$ follows $p.i$ over $t.i$. The maximality condition means that either the sequence is infinite or it is finite and the guard of every action in the program is false in the last state of the computation.

3 Parallelizable assertions

Logical properties of programs are expressed as assertions. An assertion *holds* for a program iff program states, transitions, and computations satisfy some condition called the *holding condition* for the assertion. Each assertion has a "serial version", which is also an assertion. The holding condition for the serial version of an assertion is the same as that for the assertion except that all occurrences of "transition", "computation", and any other assertion are respectively replaced by "serial transition", "serial computation", and the serial version of that assertion. In this note, we identify three classes of assertions: closure, activity, and convergence. The holding conditions for the assertions in these classes are defined next.

Closure under execution: A closure assertion has the form (P is closed); it holds for a program iff P is a set of program states and for every p in P and every program transition t, if q follows p over t, then q is in P. The serial version of this assertion is (P is serially closed); it has the same holding condition as that of its assertion except that "transition" is replaced by "serial transition".

Activity within a closure: An activity assertion has the form (u is active in P); it holds for a program iff u is a set of program actions, P is a closed set of program states, and for every program computation that starts in a state in P there is at least one action in u that occurs infinitely many times in the transitions of the computation. The serial version of this assertion is (u is serially active in P); it has the same holding condition as that of its assertion except that "closed" and "computation" are replaced by "serially closed" and "serial computation", respectively.

Convergence to a Closure: A convergence assertion has the form (P is convergent to Q); it holds for a program iff both P and Q are closed sets of program states and every program computation that starts in a state in P has a state in Q. The serial version of this assertion is (P is serially convergent to Q); it has the same holding condition as that of its assertion except that "closed" and "computation" are replaced by "serially closed" and "serial computation", respectively.

One program property that can be expressed as a closure assertion is "mutual exclusion". A property that can be expressed as an activity assertion is "freedom from starvation". Examples of properties that can be expressed as convergence assertions are "termination" and "stabilization".

A class of assertions is called *parallelizable* for a program iff for each assertion in the class, if the serial version of the assertion holds for the program, then the assertion holds for the program. For example, closure is parallelizable for a program iff for each P, if (P is serially closed) holds for the program, then (P is closed) holds for the program.

We mentioned in the introduction that program correctness should be based on parallelizable assertions. We can now be more explicit about the reason for making this statement. The serial version of an assertion is what we prove, under interleaving, about program execution; the assertion itself is what actually is maintained during program execution. Thus, basing program correctness on parallelizable assertions guarantees that what we prove is what actually is maintained (or occurs) during execution.

It is straightforward to show that none of our assertion classes —closure, activity, or convergence— is parallelizable for arbitrary programs. In the next section, we identify a large family of programs for which these three classes of assertions are parallelizable.

4 Parallelizable assertions for serializable programs

A program is called *P-serializable* iff P is a serially closed set of program states, and for every p in P and every program transition t, if q follows p

over t, then t can be partitioned into serial transitions $t.1, \ldots, t.r$ such that q follows p over $(t.1, \ldots, t.r)$.

The family of serializable programs is reasonably large. For example, if program variables are partitioned into shared and private, where a shared variable is one that is read or written by the actions of two or more processes, then any program in which no action both reads and writes shared variables is P-serializable, where P is the (closed) set of all program states. This shows that serial programs, i.e. those programs that consist of single processes, are serializable.

The next theorem, whose proof follows from the above definitions, states that the assertion classes, closure, activity and convergence, are parallelizable for serializable programs.

Theorem 1 If a program is P-serializable, then the following three statements are satisfied for every subset Q of P and every set u of program actions.

a. If (Q is serially closed) holds for the program,
then (Q is closed) holds for the program.

b. If (u is serially active in Q) holds for the program,
then (u is active in Q) holds for the program.

c. If (P is serially convergent to Q) holds for the program,
then (P is convergent to Q) holds for the program.

□

As a result of Theorem 1, verification of serializable programs can be carried out under interleaving, provided that all derivations are based solely on the assertion classes: closure, activity, and convergence.

So far, serializable programs is the largest family of programs for which rich classes of assertions are known to be parallelizable. This should explain our earlier recommendations: as long as verification of concurrent programs is based on interleaving, one should design only serializable programs and base their correctness only on parallelizable assertions.

The main result in this note (Theorem 1) is based on the assumption that any pair of program actions, belonging to different processes, are executed either in sequence or in exact parallel. The validity of this assumption hinges on the actions being "small", with each of them accessing the shared variables in a "minimal" way. This requirement can be achieved by resorting to the well-known automicity condition of Gries and Owicki, namely that each action has at most one reference to a shared variable. (A recent result of my student Jim Anderson shows this restriction can be relaxed somewhat.)

Acknowledgements: I would like to thank Anish Arora and James Burns for helpful discussions concerning this work. The comments of James Anderson, David Gries and Jayadev Misra on earlier drafts of this note are greatly appreciated.

Mohamed G. Gouda,
Department of Computer Sciences,
The University of Texas at Austin,
Taylor Hall 2.124,
Austin, Texas 78712–1188,
U.S.A.