ON "A SIMPLE PROTOCOL WHOSE PROOF ISN'T": THE STATE MACHINE APPROACH

Mohamed G. Gouda

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

TR-84-21 July 1984

ABSTRACT

We discuss how to model a synchronous protocol (due to Aho, Ullman, and Yannakakis) using communicating finite state machines, and present a proof for its safety and liveness properties. Our proof is based on constructing a labeled finite reachability graph for the protocol. This reachability graph can be viewed as a sequential program whose safety and liveness properties can be stated and verified in a straightforward fashion.

ON
"A SIMPLE PROTOCOL WHOSE PROOF ISN'T": THE STATE MACHINE APPROACH

Mohamed G. Gouda

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

TR-84-21 July 1984

ABSTRACT

We discuss how to model a synchronous protocol (due to Aho, Ullman, and Yannakakis) using communicating finite state machines, and present a proof for its safety and liveness properties. Our proof is based on constructing a labeled finite reachability graph for the protocol. This reachability graph can be viewed as a sequential program whose safety and liveness properties can be stated and verified in a straightforward fashion.

1. INTRODUCTION

Hailpern [5] presented a correctness proof for a synchronous protocol (due to Aho, Ullman, and Yannakasis [1]) using the abstract-program approach, and temporal logic. He also considered modeling the protocol using "pure" communicating finite state machines (i.e. communicating finite state machines without local variables), but contended that this model can only exhibit the liveness properties (without the safety properties) of the protocol.

In this paper, we discuss how to model this protocol using "extended" communicating finite state machines (i.e. communicating finite state machines with local variables, as suggested by Bochmann [2] and Sunshine [7]). We also discuss a proof for its safety and liveness properties.

Our proof is based on constructing a labeled, finite reachability graph for the protocol. This reachability graph can be viewed as a sequential program that operates on the local variables of the communicating machines in the protocol. Hence, the safety and liveness properties of the protocol can be expressed as safety and liveness properties for this sequential program, and so can be verified in a straightforward fashion.

2. MODELING THE PROTOCOL USING COMMUNICATING FINITE STATE MACHINES

We consider a protocol where two communicating finite state machines A and B exchange bits, *in steps*, over two 1-bit registers x and y. In each step,

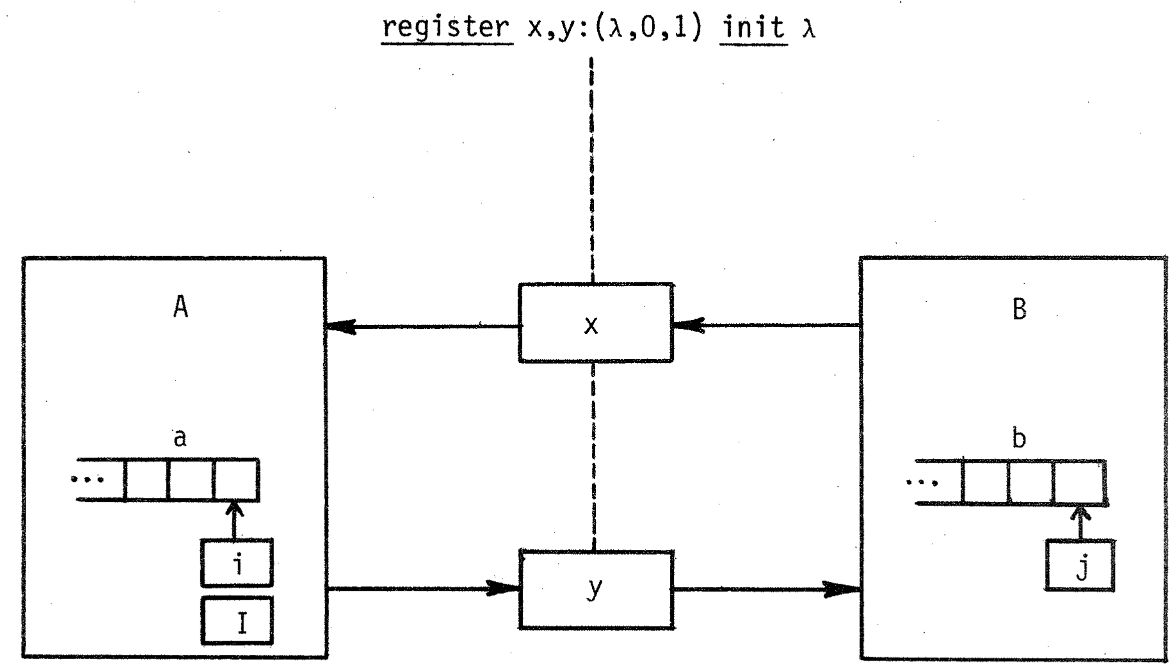
(a) machine A reads register x, and machine B reads register y, then

(b) based on the read values, machine A writes register y and machine B writes register x.

Each machine has some local variables that it can update in each step immediately after it writes in its register. The local variables in each machine, and their initial values, are defined in Figure 1 using a Pascal-like notation. Notice that machine A has a constant, infinite array "a" of bits, and machine B has a variable, infinite array "b" of bits. (Variables "i" and "j" are used as indices for arrays "a" and "b", respectively.)

The function of this protocol is to copy the bit values of array "a" into array "b" under the assumption that registers x and y can lose their values. (After a machine writes a value 0 or 1, in its register in some step, the written value may be changed into the lost, or empty, value λ before the next step.)

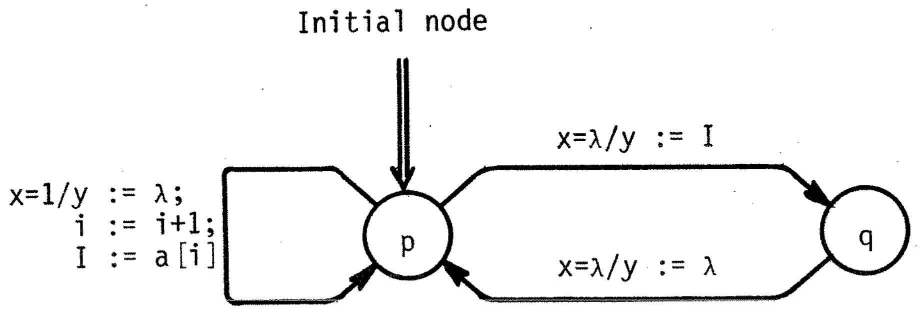
To achieve this function, Aho, Ullman, and Yannakakis [1] defined the control structures of the two machines to be as shown in Figure 2. (We have used a slightly different notation to be consistent with our data structure in Figure 1.) Each machine has two local states called *nodes*, and some local state transitions called *edges*. Each edge is



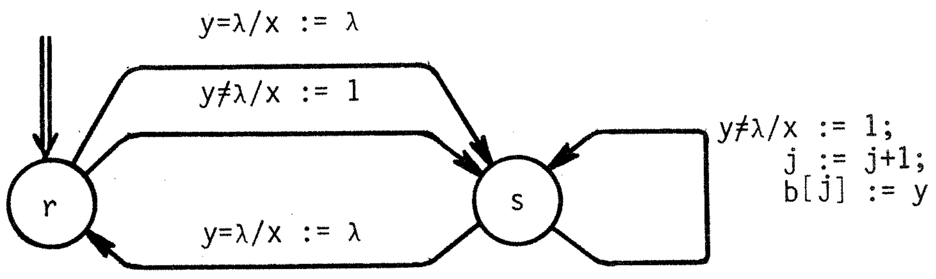
```
A: const a: array [integer]
            of (0,1);
   var   i: integer init 1;
         I: (0,1) init a 1

B: var b: array [integer]
          of (0,1);
       j: integer init 0
```

Figure 1. Data structure of the protocol.



(a) A



(b) B

Figure 2. Control structures of the protocol.

labeled with a pair: condition/action sequence. Initially each machine is at its initial node. In each step, each machine traverses exactly one edge to reach from its current node to its next node. The edge selected for traversal in each step is the outgoing edge, of the current node, whose condition is true. The traversal of an edge in a step implies the execution of its action sequence in that step. Notice that with the exception of the labeling action sequences that update the local variables of each machine, Figure 2 is essentially the same as Hailpern's Figure 2 in [5].

3. A REACHABILITY GRAPH FOR THE PROTOCOL

In this section, we construct a reachability graph, for the above protocol, whose vertices correspond to the reachable global states of the protocol, and whose arcs correspond to global state transitions. Later, we use this reachability graph to establish the safety and liveness properties of the protocol.

The first question that arises in constructing the reachability graph of a given protocol is how to define the global state of the protocol? In our case, the obvious answer of defining the global state by (a) one node in each machine, and (b) one value for each register or local variable in the data structure, will cause the reachability graph to be infinite, and so not useful for our purposes. Instead, we define the global state of the protocol by (a) one node in each machine, and (b) one value for each register (thus ignoring the values of the local variables of the two machines). Formally the *global state* (or *state* for short) of the protocol is defined to be a four-tuple [m,y,x,n], where

m is a node in machine A,
y is a value for register y,
x is a value for register x, and
n is a node in machine B.

(This notation is identical to Hailpern's [5].)

Based on this definition of a state, a *reachability graph* for the protocol can be constructed as shown in Figure 3. Each vertex in this graph is labeled with a reachable state of the protocol. (For instance, vertex 1 is labeled with the initial state [1,λ,λ,1], and vertex 2 is labeled with the state [2,λ,I,2], where the value of register y is identical to the current value of local variable I.) Each arc in the graph corresponds to one step. A *dashed arc* corresponds to a step followed by one of the two registers losing its value, and a *solid arc* corresponds to a step after which the values of the two registers are not lost. (For instance, the solid edge from vertex 1 to 2 corresponds to a step where A copies the value of I into y which is not lost. By contrast, the dashed edge from 1 to 3 corresponds to the same step followed by the loss of y's value.)

In this reachability graph, the values of the local variables are not recorded in the protocol's states. Therefore, each arc in the reachability graph must be labeled by some action sequence that updates the local variables of the two machines in accordance with the protocol's definition. To figure out the labeling sequence for each arc, notice that

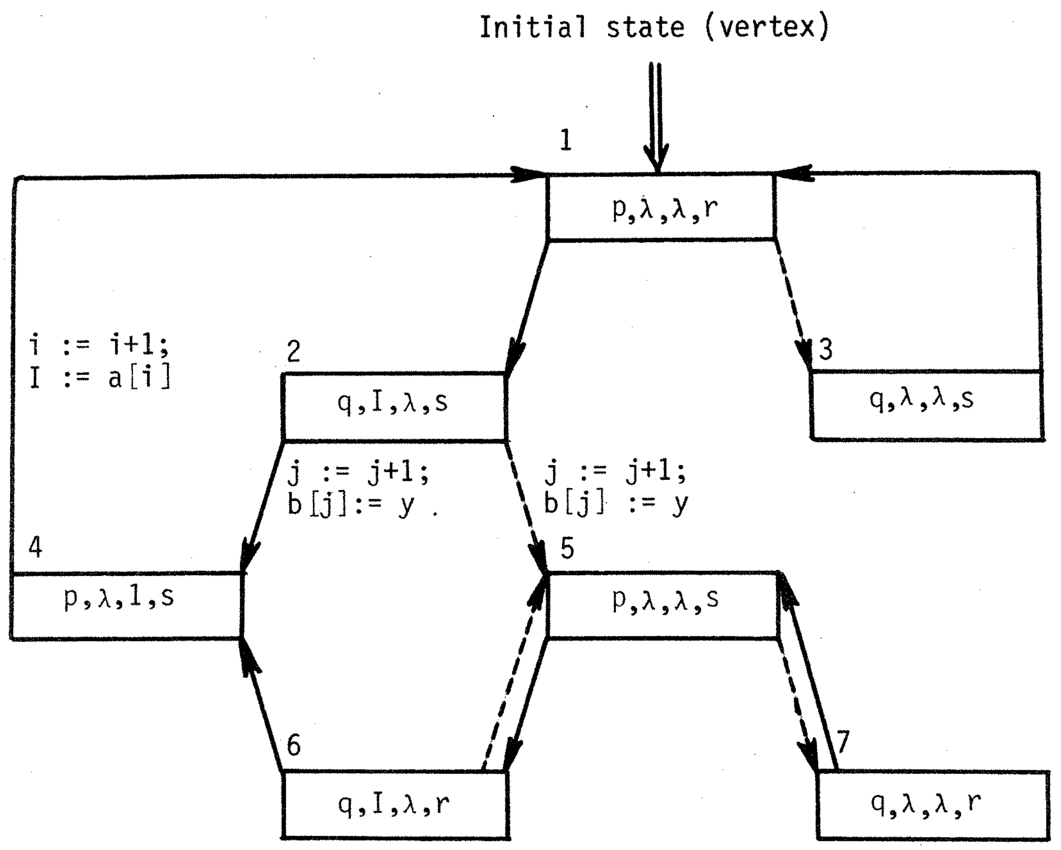


Figure 3. A labeled finite reachability graph for the protocol.

every arc c corresponds to two directed edges e_1 and e_2, one in machine A and the other in B. Thus, a concatenation of the two action sequences that update the local variables and label the two edges e_1 and e_2 should label arc c. (For example, the arc from vertex 1 to 2 in the reachability graph corresponds to two edges from nodes 1 to nodes 2 in machines A and B; since these two edges are not labeled with action sequences that update the local variables, the arc from vertex 1 to 2 is labeled with the empty sequence, or equivalently not labeled as shown in Figure 3.) Notice that with the exception of the labeling action sequences that update the local variables of the two machines, Figure 3 is identical to Hailpern's Figure 4 in [5].

On one hand, the reachability graph in Figure 3 is a complete definition of the protocol defined earlier in Figures 1 and 2. On the other hand, this reachability graph can be viewed as a *sequential program* that operates on the variables x, y, a, b, i, j, and I; hence proving its safety and liveness properties is straightforward as shown next.

4. PROVING SAFETY PROPERTIES

The *safety property* of this protocol can be stated as follows: After each step, the written string in array "b" is identical to the initial string, of equal length, in array "a". To prove this property, we need to prove that the following assertion P is *true* at each vertex in the reachability graph (i.e. it is an invariant):

$P :: \{i - j \geq 0 \wedge \forall\, k = 1..j,\ a[k] = b[k]\}$

We follow the style of Floyd [3] and Hoare [6] to annotate the reachability graph according to the following three rules:

i. Assign one assertion to each vertex in the reachability graph.

ii. The assertion assigned to vertex 1 (i.e. the initial state) should be computed to *true* when each variable in the assertion is assigned its initial value.

iii. If vertices v and w in the reachability graph are assigned assertions P and $Q \wedge Q^1$ respectively, and if there is an arc from v to w labeled with the action sequence S, then P{S}Q and Q^1 should be computed to *true* when x and y are assigned their values in the protocol's state at vertex w.

The following three assertions are selected to annotate the reachability graph:

$P_1 :: \{I = a[i] \wedge i = j + 1 \wedge \forall\, k = 1..j,\ a[k] = b[k]\}$

$P_2 :: \{y = I\}$

$P_3 :: \{I = a[i] \wedge i = j \wedge \forall\, k = 1..j,\ a[k] = b[k]\}$

In particular, assertion P_1 is assigned to vertices 1 and 3, assertion $P_1 \wedge P_2$ is assigned to vertex 2, and assertion P_3 is assigned to vertices 4, 5, 6, and 7. It is straightforward to show that this annotation satisfies the above three rules. (Guessing these assertions is no great mystery. Since P should be true at each vertex in the reachability graph, each vertex should be assigned an assertion of the form $P \wedge Q$. The additional assertions Q's should ensure that at vertex 2 just before y is assigned to b[j], $y = I = a[i]$.)

Since in this annotation each vertex in the reachability graph is assigned an assertion that implies P, then P is an invariant.

5. PROVING LIVENESS PROPERTIES

The *liveness property* of this protocol can be stated as follows: The arc from vertex 2 to vertex 4 in the reachability graph should be traversed *infinitely often* provided that the value loss from registers x and y is constrained in some way. There are three reasonable ways to constrain the value loss:

i. *Values can be lost up to a certain time instant:* After this instant no value can be lost from register x or y. Therefore from this instant on, the reachability graph is the same as in Figure 3 except that all dashed arcs are removed. In this new reachability graph, there is exactly one directed cycle; moreover this cycle contains the arc from vertex 2 to 4. Therefore, this arc will be traversed infinitely often. (This proof is due to Hailpern [5].)

ii. *Values can always be lost, but infinitely often the value loss will stop for at lease six steps:* As mentioned earlier stopping the value loss is equivalent to removing the dashed arcs from the reachability graph. Thus, every time the value loss is stopped for at least six steps, the arc from vertex 2 to 4 will be traversed at least once. Since this is guaranteed to occur infinitely often, the arc will be traversed infinitely often. (This notion of liveness along with the notion of safety discussed earlier corresponds to the notion of robustness discussed in [1].)

iii. *Values can always be lost, but in a fair fashion:* Under this constraint, if a vertex in the reachability graph is reached infinitely often, and if this vertex has two outgoing arcs, then each of the two arcs will be traversed infinitely often. It is straightforward to show that under this constraint, each arc in the reachability graph (including the arc from vertex 2 to 4) will be traversed infinitely often. For more details about these notions of liveness and fairness for networks of communicating finite state machines, we refer the reader to Gouda and Chang [4].

6. CONCLUDING REMARKS

The basic idea of our proof is to construct a finite (but labeled) reachability graph for the protocol. This reachability graph can be viewed as a sequential, nondeterministic program, and so can be verified in a straightforward manner. This approach can be used to verify many synchronous protocols, and some asynchronous protocols with bounded communications (for which finite reachability graphs can be constructed).

Acknowledgements: I wish to thank Brent Hailpern whose trials with this protocol have inspired this paper. I am also thankful to Carl Sunshine for his suggestions that have greatly improved the presentation.

REFERENCES

[1] A. V. Aho, J. D. Ullman, and M. Yannakakis, "Modeling communications protocols by automata," *Twentieth Annual Symposium on Foundations of Computer Science*, IEEE, October 1979, pp. 267-273.

[2] G. V. Bochmann, and J. Gescei, "Unified method for the specification and verification of protocols," *Information Processing Congress 77*, 1977, pp. 229-234.

[3] R. W. Floyd, "Assigning meanings to programs," *Proc. of Symposia in Applied Mathematics XIX*, American Math. Society, 1967, pp. 19-32.

[4] M. G. Gouda, and C. K. Chang, "A technique for proving liveness of communicating finite state machines with examples," *Proc. of the Third ACM Symposium on Principles of Distributed Computing*, August 1984.

[5] B. T. Hailpern, "A simple protocol whose proof isn't," *IEEE Trans. on Comm.*, to be published.

[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Comm. of the ACM*, Vol. 12, No. 10, May 1969, pp. 276-281.

[7] C. A. Sunshine, "Formal techniques for protocol specification and verification," *Computer*, Vol. 12, No. 9, Sept. 1979, pp. 20-27.