Automatic Verification of a Class of Symmetric Parallel Programs*

Boleslaw K. Szymanski and Jose M. Vidal
Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA

This paper presents an efficient method of verification of a class of symmetric parallel programs characterized by a limited usage of communication variables. The limitation requires that any process identifier appearing in a condition must be a logical variable bounded by the universal or existential quantifier. Hence, programs in this class are only able to test whether a set of processes in a certain state is empty or not. Our approach to automatic verification is to reduce the verified program to a directed graph. Nodes of this diagraph represent different states of the program. Each edge of the diagraph is labeled with a predicate that must be satisfied if this edge is to be selected by a state transition. We use standard reachability analysis to verify the properties of the program. The efficiency of the approach results from the limitations imposed on the form of the conditions in the verified program. Using this method, we prove that a novel mutual exclusion program enforces mutual exclusion and is deadlock-free.

Keyword Codes: C.1.2; C.4; I.6
Keywords: Multiprocessors; Performance of Systems; Simulation and Modeling

1. Introduction

Parallel programs, even those published in the literature, are often given without any formal proof that they work as advertised. The demonstration of their correctness is usually done by informal methods. While such a demonstration might be sufficient for some cases, it is always desirable and sometimes necessary to prove that the program does indeed work in all possible executions. In [4], Manna and Pnueli proposed a system of temporal logic constructs that could be used when proving properties of parallel programs. However, construction of the proof is not easy; it takes some creativity and effort to come up with the right set of invariants that will lead to a proof. In [3], the same authors use similar machinery to provide temporal logic proof of the novel mutual exclusion program proposed by Szymanski in [6]. This program has interesting properties that allowed us to mechanize its proof of correctness. In this paper, we first define a class of parallel programs with such properties and then we describe an automated algorithm for proving correctness of the programs in this class.

*The research presented in this paper was done partially during the first author's stay at the Weizmann Institute of Science, Rehovot, Israel, in 1993. This work was also partially supported by the Office of Naval Research under grant N00014-93-1-0076 and by an IBM Corp. Development Grant. The content of this entry does not necessarily reflect the position or policy of the U.S. Government—no official endorsements should be inferred or implied.

Automatic program verification is often based on the exploration of underlying state transition systems. In this method, programs are represented by a set of states and program execution corresponds to state transitions. In such a representation, a terminating program execution corresponds to a finite path in the state diagram.

The main difficulty in applying this method to a real system is the large and sometimes infinite number of states that need to be created in order to represent all feasible executions. The number of states is exponentially dependent on the level of loop nesting and the number of concurrent tasks in the system. Such a technique is used, for example, in [1,5], where a process is represented by a Finite State Automaton (FSA) and each global state in the global automaton consists of the combination of the local states in FSAs of all processes.

The traditional method of hand or automatic verification is to show that states in which desirable properties are not satisfied cannot be reached from the feasible initial states. Although techniques such as reducing the search space or searching the space probabilisticly alleviate the state explosion partially, they are not sufficient for massively parallel programs.

The focus of this paper is on symmetric parallel programs in which a number of processes execute the same program. Depending on the used architecture, such symmetric parallel programs are often referred to as SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) programs. For such programs, it is desirable to have proofs independent of the number of executing processes. This requirement is not satisfied in traditional automatic verification methods in which a state space depends on the number of processes involved.

In this paper, we propose the automatic verification method that is independent of the number of involved processes and that also avoids the state explosion for a class of symmetric parallel programs. In addition, if a program is found not to satisfy the verified property, then a sequence of execution steps that led to this error is also produced. Such a sequence is invaluable in understanding where and why the verified program fails.

2. Symmetric Parallel Programs with Membership Tests

In the following, we consider symmetric parallel programs satisfying the following properties:

1. There is a finite but unlimited number of processes executing the same program.

2. Programs are either reliable or can fail to abortions only [2].

3. All communication variables are uniquely owned; i.e., each communication variable can only be changed by the one process that owns it; however, communication variables can be read by all processes. We will refer to such variables as specific communication variables.

4. Each condition appearing in an **if** or **while** statement can contain only the following Boolean factors:

   - any expression with private variables and those specific communication variables that are owned by the process,

specific communication boolean $a_{pid}, s_{pid}, w_{pid}$:=**false**
loop forever S0: Non Critical Section
S1: a_{pid}:=**true**;
S2: **while** $(\exists j : 0 \leq j < n : \; s_j)$;
S3: w_{pid}, s_{pid}:=**true**; a_{pid}:=**false**;
S4: **if** $(\exists j : 0 \leq j < n : \; a_j)$ **then**
S5.1: **while** $(\forall j : 0 \leq j < n : \; \neg s_j$ **or** $w_j)$ s_{pid}:=**false**; S5.2: s_{pid}:=**true endif**;
S6: w_{pid}:=**false**;
S7: **while** $(\exists j : 0 \leq j < n : \; w_j)$;
S8: **while** $(\exists j : 0 \leq j < i : \; s_j)$;
Critical Section
S9: s_{pid}:=**false**;

Figure 1. Mutual Exclusion Algorithm with Linear Wait.

- an expression of the form $(\forall j \in \Pi : P(j))$ or $(\exists j \in \Pi : P(j))$, where $P(j)$ is a predicate defined over communication variables owned by the process j, and Π is either the set of all processes or a set of processes that precede the testing process in some total order of all processes.

Conditions that satisfy these requirements are referred to here as *membership tests*. They can be used to test if a set of processes with certain properties defined by the communication variables is empty or not. On the other hand, such a test cannot be used to check if the particular process is a member of such a set of processes.

As an example of a parallel program satisfying these conditions, we consider the mutual exclusion algorithm from [6] given in Figure 1. We assume that there are n executing processes and that the process identifiers are numbered from 0 to $n-1$ and processes are ordered by their identifiers. The variable *pid* denotes the identifier of the process executing the given instance of the program. Properties of this program are discussed elsewhere [6]. Below, we outline a methodology for translating such programs into state diagraphs and an algorithm for state reachability that is used to verify the properties of the translated program. As an example, we also show how mutual exclusion enforcement and deadlock freedom can be proved in this manner.

3. Program Verification

3.1. Program Diagraph Representation

The first step in automatic verification is to build the program state diagraph. A node in this diagraph is created for each block of statements ended by either assignment to a specific communication variable or a condition using such a variable. Let j be a subscript with the range equal to the set of processes participating in the program execution (e.g., $j \in [0, n-1]$). Let $P(g_j)$ denote a predicate over that instance of the communication variable g that is owned by the process j. Edges of the diagraph show the possible flow of control in the program. They can be of one of the following four types:

- unconditional, if the predecessor node is a block ended by assignment to a communication variable,
- conditional-while, if the predecessor node corresponds to a block ended by a while statement with a condition formed as $(\forall j :\ P(g_j))$ or $(\exists j :\ P(g_j))$,
- conditional-if, if the predecessor node is a block ended by an if-then or if-then-else statement with a condition in the form of $(\forall j :\ P(g_j))$ or $(\exists j :\ P(g_j))$,
- sequentializing, if the predecessor node corresponds to a block ended with the while statement with a condition $(\forall j :\ 0 \leq j \leq pid :\ P(g_j))$. A set of nodes for which predicate $P(g_j)$ is true is called a region of the sequentializing edge.

A conditional edge can be open (when the corresponding condition is satisfied), or closed(otherwise). Each node from which a conditional edge originates creates two or three states in the state diagraph. One state represents processes before successful completion of the corresponding test and the others represent the processes afterwards. Unconditional edges are always open and a sequentializing edge is open for exactly one process at a time. Thus, each node from which outgoing edges are of the two last kinds creates a single state in the final diagraph.

As an example, the program diagraph for the mutual exclusion algorithm presented in Figure 1 is shown in Figure 2. Conditional edges are marked with circles and the sequentializing edge is marked with an oval. The nodes are marked with the labels from the original program. The number of states in a diagraph depends linearly on the number of statements in the program from which it was derived. The diagraph is completely independent of the number of processes executing the program in parallel.

3.2. Global State Enumeration and Traversal

The global state of execution must encapsulate the states of the individual processes. The most straightforward method for such encapsulation is to simply list the state of each participating process. However, for an unbounded number of processes, this method will generate an infinite number of global states.

For the programs satisfying the conditions listed in the previous section, a global state can be defined by a list of program diagraph states that have at least one process in it (we will refer to such a state as *populated*). The global state will, therefore, have one bit for each state in the program diagraph.

The initial global state corresponds to just a single program state populated; namely, the initial state of the program diagraph. It describes the execution stage at which all processes are in their own initial state.

The enumeration algorithm takes a global state and generates all the global states that can be reached from it by transition of a single process. This enumeration step is repeated until all the global states have been processed, including all newly created global states. For each populated local state on the global state list, the algorithm checks the associated edge and if it is open, it creates the new global states that can be obtained by:

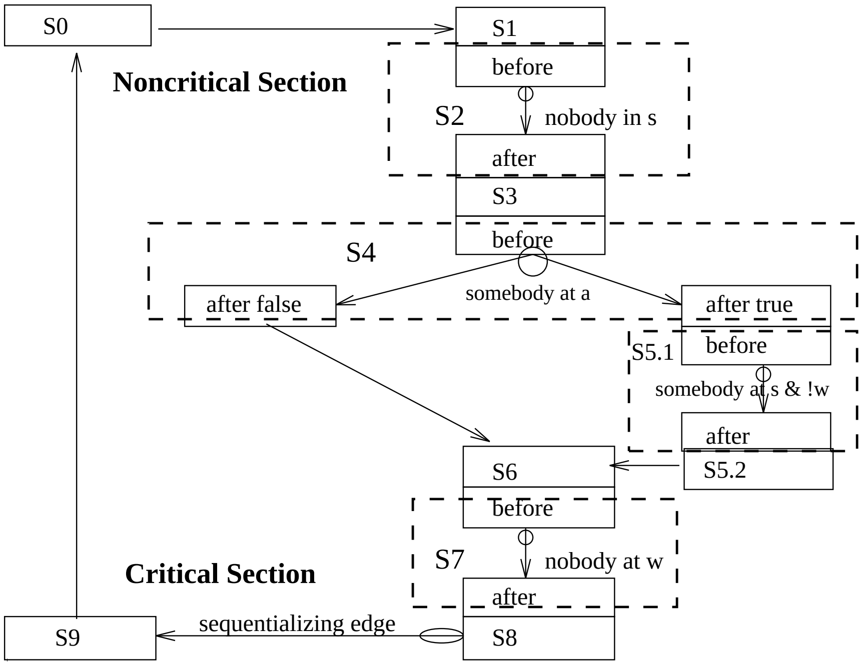

Figure 2. Program Diagraph.

1. populating the local successor state,

2. emptying the local predecessor state.

The presented version of the enumeration algorithm assumes that the execution of a membership test is an atomic operation. It also assumes an interleaving model of execution in which a process reading from a memory location always obtains the value that was written there last. Under the stated assumptions, it is straightforward to prove by induction that the presented algorithm does generate all the global states reachable from the initial state.

The deadlock in a parallel program occurs when there is a reachable global state in which all populated local states have all edges closed. For the considered program, mutual exclusion is violated when a process enters the sequentializing edge region while the local state from which the sequentializing edge originates is already populated. Hence, both deadlock and mutual exclusion violation can be easily detected by monitoring the states that are generated by the algorithm. If such a detection occurs, then the algorithm can also produce a sequence of reachable global states that led to this event.

An extension of the enumeration algorithm supports non-atomic membership tests. This support also requires an extended state definition for conditional statements as follows. As long as there is an empty after-edge for a conditional state, that state is kept as unprocessed. If the edge condition requires that some other state is empty, this condition becomes satisfied if the state in question becomes open; i.e., if there is an open incoming and outgoing edge to this state. Hence, the modified algorithm is capable of proving whether or not the given program has verified properties if the testing is done in any order whatsoever. The modification creates and reaches more global states than the original algorithm. For example, for the program in Figure 1, the basic algorithm produced 157 global states, whereas the modified algorithm produced 6211 states (and found the mutual exclusion violation for non-atomic membership tests).

Another simple extension of the algorithm enables verification in the presence of abortions. Following Lamport [2], by the process abortion we mean an atomic operation that makes the process (i) stop executing its program at an arbitrary point of execution, (ii) set all its variables to initial values and, (iii) after an unspecified but finite delay, proceed again from the initial state. Hence, the traversal algorithm's extension can be achieved by adding to each global state additional successors that are created by the transition of a process to its initial state.

4. Conclusion

There are three important directions for future research connected with the presented method. First, we are currently working on an extension to C language that will enable the user to write membership tests conveniently. We are also implementing a parser for this extension and the program diagraph generation code. The second important direction is to find out what class of parallel algorithms can be written using the membership tests as the only form of conditions over the specific communication variables. Finally, there is an important open question of whether the method could be extended to synchronized non-atomic membership tests. Synchronized tests access communication variables in the same total order of the owner processes, but the whole test is otherwise a non-atomic operation. It is a simple generalization of the Lemma from [3] that synchronized non-atomicity is the weakest limitation on the membership tests necessary for the algorithm shown in Figure 1 to work correctly.

REFERENCES

1. S.M. German and A.P. Sistala, "Reasoning about Systems with Many Processes," *Journal ACM,* Vol. 39, No. 3, 1992, pp. 675–735.
2. L. Lamport, "The Mutual Exclusion Problem," *Journal ACM,* Vol. 33, No. 2, 1986, pp. 313–326.
3. Z. Manna and A. Pnueli, "An Exercise in the Verification of Multi-Process Programs," *Beauty is our Business,* Springer-Verlag, Berlin, 1991, pp. 289-301.
4. Z. Manna and A. Pnueli, "Tools and Rules for the Practicing Verifier," *Carnegie Mellon University, Computer Science: A 25th Anniversary Commemorative*, ACM Press, New York, NY, 1991.
5. A. Pnueli and L. Zuck, "Verification of Multiprocess Probabilistic Protocols," *Distributed Computing,* Vol. 1, No. 1, 1986, pp. 53–72.
6. B. K. Szymanski, "Mutual Exclusion Revisited," *Proc. Fifth Jerusalem Conference on Information Technology,* IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 110-117.