A Proof Lattice-Based Technique for Analyzing Liveness of Resource Controllers

Ugo Buy

Department of Electrical Engineering
and Computer Science
University of Illinois
Chicago, Illinois 60610
E-mail: buy@uicbert.eecs.uic.edu

Robert Moll

Department of Computer
and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
E-mail: moll@cs.umass.edu

1 Introduction and Background

The automatic synthesis of programs is one of the most challenging activities in computational logic. Deductive approaches to the synthesis of sequential programs have met with only limited success, in part because of the relative inadequacy of proposed specification languages for such programs. That is, specifications often turn out to be less enlightening and more difficult to construct than the code they specify. This is not the state of affairs with synchronization code. Desired program properties, such as liveness or freedom from deadlock, are relatively easy to include in a specification but are quite difficult to discern in code. Indeed, it is precisely this feature of concurrent programming that makes automated program synthesis an attractive goal. In this paper we discuss an aspect of the automatic synthesis of synchronization code for asynchronous processes.

Our synthesis approach conforms to the following paradigm: 1) first a specification is written in a nonconstructive specification language; 2) that specification is analyzed in an attempt to establish that crucial concurrency properties are respected; and 3) if the concurrency properties of the specifications are established, then Ada code is generated.

Here we report on the most difficult part of this process: establishing that a program specification has a crucial concurrency property, namely liveness. (Other related aspects of our proposed approach, such as safety properties and deadlock avoidance, are discussed elsewhere [1]). By liveness we mean a condition that must be satisfied at some point during the execution of a program. Liveness contrasts with a safety property, a condition that must hold continuously throughout program execution.

Underlying our approach to synthesis is a model of a concurrent program in which processes communicate by accessing and modifying shared resources. This *resource-oriented model* uses temporal logic for specification and analysis. By a *process* we mean an abstract state machine defining an active computation. A process computation generally requires allocation of shared resources. By a *synchronizer* we mean an abstract state machine that

defines a set of encapsulated resources [8]. A synchronizer specification is expressed in terms of the safety and liveness properties that the synchronizer must satisfy. Our approach to program specifications is similar to that presented in [8], which, however, does not address the issue of liveness. Synchronizer liveness is the main focus of this paper.

We believe our approach to synthesis and, in particular, our approach to liveness analysis can be automated for two reasons. First, by identifying indiscernible synchronizer computation states, we drastically reduce the size of the resulting synchronizer state space graph. Second, by using this graph to drive the formal analysis of program specifications, we limit the need for general theorem proving capabilities. At present a partial implementation has been completed.

This paper is organized as follows. Synchronizer specifications are described in section 2. A graph model of synchronizer behavior is discussed in section 3. A proof lattice-based method for liveness analysis is given in section 4 along with the analysis of the familiar readers/writers example. Some conclusions are presented in section 5.

2 Synchronizer Specifications

Our specification language is built around the concept of the state of the resources encapsulated in a synchronizer. Specifications are formulated in terms of a set of state variables and a set of operations that can access and modify those variables. Linear-time temporal logic is used to define the semantics of the constructs appearing in a synchronizer specification. Thus, a synchronizer specification consists of a set of clauses defining the state transitions performed by each synchronizer operation. A specification also gives the safety and liveness conditions governing the execution of synchronizer operations. An example, the specification of synchronizer *buffer* from the readers/writers problem, is given in Figure 1. In this problem a set of reader and a set of writer processes must access a shared memory area simultaneously. Synchronizer *buffer* controls access and modifications to the shared memory on behalf of the reader and writer processes.

The **control_resources** clause of the specification defines resources *read_permission* and *write_permission* for synchronizer *buffer*. The clause asserts that, in order for a reader (or a writer) process to access the shared memory, the resource *read_permission* (or *write_permission*) must be allocated first. Allocation is performed by operations *start_read* and *start_write*. Operations *end_read* and *end_write* perform the corresponding deallocations.

The **state_variables** clause defines the type and initial value of the variables *reader_#* and *writer_#* used by the synchronizer to represent the state of the encapsulated resources. These two variables track the total number of reader and writer processes that are in the synchronizer at any point in time. The **operations** clause declares the operations that

Synchronizer Buffer
Control_resources write_permission **allocated_by** start_write;
deallocated_by end_write;
read_permission **allocated_by** start_read;
deallocated_by end_read;
State_variables
reader_#: 0..10 **initially** 0 ;
writer_#: 0..1 **initially** 0 ;
Operations
start_read, start_write, end_read, end_write ;
Operation_preconditions
$\forall$s$\in$start_write: (reader_# = 0) $\wedge$ (writer_# = 0)
$\forall$s$\in$start_read: (writer_# = 0)
Operation_state_changes
$\forall$s$\in$start_read: reader_# $\leftarrow$ reader_# + 1
$\forall$s$\in$start_write: writer_# $\leftarrow$ 1
$\forall$e$\in$end_read: reader_# $\leftarrow$ reader_# $-$ 1
$\forall$e$\in$end_write: writer_# $\leftarrow$ 0
Operation_priorities
start_write > start_read
Liveness
$\forall$s$\in${start_write, start_read} s@request $\rightarrow$ $\Diamond$s@terminate

Figure 1: Specification of synchronizer *buffer* in the readers-writers problem

units outside synchronizer *buffer* can invoke to perform allocation and deallocation of the resources encapsulated in this synchronizer. Whenever an operation is invoked, a model of operation execution is used in which the invocation goes through a sequence of three phases. In the *request* phase the operation has been invoked and is waiting to be selected for execution; in the *service* phase the operation has been selected for execution and will be executed next; and in the *terminate* phase the operation has completed execution. For a given invocation o of a synchronizer operation O, the propositions *o@request*, *o@service* and *o@terminate* are true when o is in the corresponding phase and false otherwise.

The **operation_preconditions** clause specifies the conditions to be satisfied in order for each operation request to be executed. Temporal operators are not allowed in this clause. A request for operation *start_write* can be serviced only if the synchronizer state satisfies:

$$(reader\_\# = 0) \wedge (writer\_\# = 0)$$

The **operation_state_changes** clause specifies the changes in the value of the state variables resulting from the execution of a given operation. Whenever a request for operation *start_read* is executed, the value of the state variable *reader_#* is incremented. The **operation priorities** clause defines the order in which instances of invocations of

different operations are to be considered for service. In synchronizer *buffer*, operation *start_write* has higher priority than operation *start_read*.

Finally, the **liveness** clause specifies that invocations of operations *start_read* and *start_write* must eventually be serviced. As with most temporal logic systems, the formula $\Box p$ asserts that predicate p is true in the current state and any subsequent state. The formula $\Diamond p$ asserts that p is true in the current state or in some subsequent state.

Linear time temporal logic underlies the computational model of a synchronizer specification. Thus, time is viewed as a totally ordered sequence of *time instants*. Each time instant corresponds to a well-defined synchronizer state (i.e. the assignment of a value to every synchronizer state variable). In general a synchronizer operation O can be invoked an arbitrary number of times by the units contained in the concurrent program. Each operation invocation o can be in the *request* phase for an arbitrary (and possibly indefinite) number of consecutive time instants, while waiting for the appropriate preconditions and priority conditions to become true. When these conditions become true, invocation o is *enabled*, in which case the predicate *enabled(o)* is true.

When an invoked operation o is selected for execution, o goes into the *service* phase for exactly one time instant. At the following instant operation o is in the *terminate* phase. At this instant the synchronizer state variables have been modified to reflect the changes caused by the execution of o. Operation execution in a synchronizer is strictly sequential: only one operation invocation can be in the service phase at any time instant. Once an operation request reaches the *terminate* phase it remains in that phase forever.

3 Graph Model of Synchronizer Behavior

We model synchronizer behavior using an augmented finite state machine called a Reduced State Transition Graph or RSTG. This model underlies the proof lattice mechanism we introduce to establish liveness, and our proof lattice inferences are aimed at identifying valid transition sequences between the states in an RSTG.

The RSTG of a synchronizer is a graph $G = (N, E)$. Nodes N in this graph represent sets of synchronizer states and arcs E represent transitions resulting from the execution of enabled operation requests. Each node N_i is associated with a set of synchronizer states X_i, and each edge E_i is associated with a synchronizer operation O_i and a predicate P_i on the state of the synchronizer, subject to the following conditions. First, the states associated with the same node are indiscernible in that they enable the same operations. Second, if E_i is an edge labeled (P_i, O_i) leading from node N_j to N_k, if operation O_i is executed when the synchronizer is in one of the states associated with N_j, and if predicate P_i is true, then the synchronizer is in a state associated with N_k after O_i has been executed. Third, state sets associated with distinct nodes are disjoint.

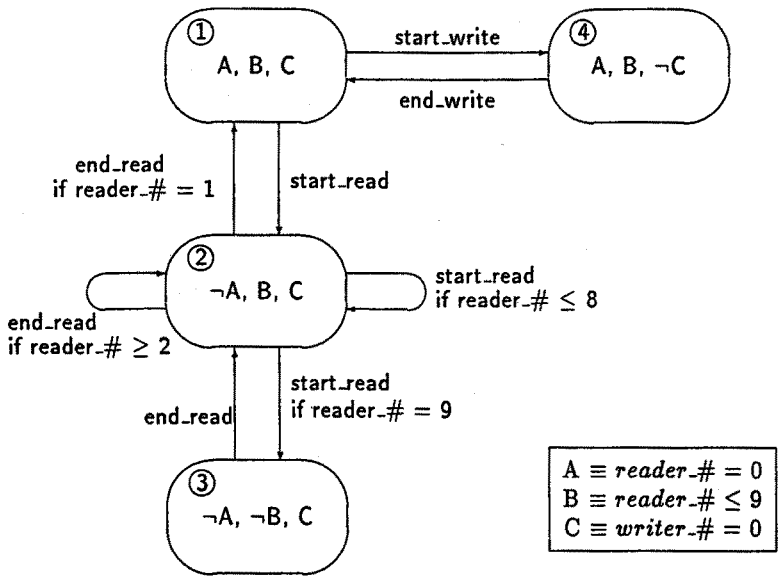

Figure 2: RSTG of synchronizer *buffer*

Construction of an RSTG is carried out in two phases. First, the nodes in the graph are defined. Each node corresponds to a combination of truth values of the predicates appearing in a program specification. Then edges are defined by identifying the operations that are enabled at each node and by tracking the state transitions caused by the execution of each enabled operation. The RSTG construction is discussed in detail in [1].

The RSTG for synchronizer *buffer* is shown if Figure 2, along with the predicate truth values corresponding to each node. Node 1 corresponds to the initial synchronizer state. Operations *start_read* and *start_write* are enabled in this state, as shown by the edges leaving node 1. Node 2 represents the synchronizer states in which resource *read_permission* has been allocated to a number of reader processes that is less than the maximum number of readers allowed. Consequently both *start_read* and *end_read* are enabled in the states corresponding to this node. The execution of the former operation leads to node 3 when the variable *reader_#* is one short of the maximum reader number; otherwise it leaves the synchronizer in a state still contained in node 2. Likewise, the execution of operation *end_read* can either lead to node 1 or back to node 2, depending on whether variable *reader_#* is one or greater than one. Node 3 corresponds to the synchronizer state in which *reader_#* equals the maximum number of readers. Only operation *end_read* is enabled in this state, leading to state 2. Finally, node 4 represents the state in which a writer process is in the synchronizer. Only operation *end_write* is enabled in this state.

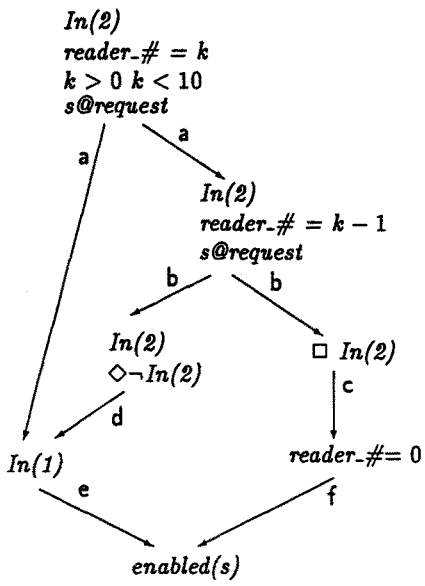

Figure 3: Proof lattice for operation *start_write* for RSTG node 2

4 Proof Lattice-based Analysis

We now describe our deductive method for establishing the following liveness condition:

$$\forall o \in O \ \Box(o@request \rightarrow \Diamond \ o@service) \tag{1}$$

We prove liveness using the notion of a *proof lattice*. Proof lattices were introduced in [5] to prove first-order formulas. They were used in [7] to analyze a subset of first-order temporal logic. A proof lattice is a finite directed acyclic graph in which each node is labeled with an assertion and such that:

1. There is a single entry node with no incoming edges
2. There is a single exit node with no outgoing edges
3. If a proof lattice node labeled P has outgoing arcs to nodes labeled R_1, R_2, ... R_n, the following formula holds for the synchronizer:

$$\Box(P \rightarrow \Diamond(R_1 \vee R_2 \vee ... \vee R_n))$$

Our approach to liveness is based on several assumptions. First we assume that the implementation of a synchronizer uses a *fair scheduler*, that is, a scheduler that does not ignore an operation invocation that is enabled infinitely often:

$$\Box\Diamond enabled(o) \rightarrow \Diamond o@service \tag{2}$$

Second, we assume that a resource allocation is always followed by the corresponding resource deallocation. This assumption is realistic because deadlock prevention is addressed independently of the liveness analysis described here [1]. Thus, if operations A and D allocate and deallocate a given resource, any invocation a of operation A is eventually followed by an invocation d of operation D:

$$\Box(a@request \rightarrow \Diamond d@request)$$

Third, we make use of the notion of *conformity* between a state variable x and a resource R. We say x is *conformal* with R if the value of x reflects the number of open allocations of R (i.e. the number of executions of the operation that allocates R that have not been followed by matching deallocations). In the *buffer* example variable *reader_#* is conformal with resource *read_permission*.

The following theorem establishes the validity of the proof lattice-based approach [1]. The predicate $In(N_i)$ indicates whether the current synchronizer state belongs to an RSTG node N_i.

Fundamental Theorem
Given an operation O defined in synchronizer S and an RSTG for S, if for every node N in the RSTG a proof lattice can be built whose entry node is labeled $In(N) \wedge o@request$ and whose exit node is labeled *enabled(o)*, then the following assertion is true for every synchronizer state

$$\Box(o@request \rightarrow \Box\Diamond enabled(o))$$

The conclusion of the theorem is that, under the stated hypotheses, every invocation o of operation O is enabled infinitely often. In the presence of the fair scheduler assumption, this guarantees that the liveness condition (1) above is satisfied. Thus, every invocation of operation O is eventually serviced. Consequently the liveness analysis of synchronizer operations can be performed by constructing a suitably labeled proof lattice for each node N in the RSTG of the synchronizer.

Proof lattice construction rules fall into three categories. Rules in the first group tie proof lattice deductions to RSTG transitions. For example, if an RSTG node N has arcs leading to nodes N_1, ... , N_k, a proof lattice node L labeled $In(N)$ has descendants labeled $In(N_1), \ldots, In(N_k)$.

Rules in the second group are based on axioms of linear-time temporal logic. For example, suppose there is a proof lattice node L labeled $\Box p$. In this case, predicate p can be added to the label of any descendant of node L, based on these two axioms of temporal logic: $(\Box p \rightarrow p)$ and $(p \rightarrow \Diamond p)$.

Rules in the third group reflect the behavior of resource allocation and deallocation induced by our synchronizer model. As an example, consider the *allocation completion* rule. Suppose that a state variable x of synchronizer S is conformal with resource R, which

is allocated and deallocated by operations A and D, respectively. Then a proof lattice node L labeled $\Box\neg$*enabled*(A) has a descendant labeled $(x = \tilde{x})$, where $\tilde{x}$ is the initial value of x. The validity of this rule is proved by use of the assumption of resource deallocation. Since operation A is never enabled, eventually all executions of this operation will be followed by executions of operation D. Roughly speaking, the effects of each execution of operation A on variable x will eventually be "undone" by the corresponding execution of operation D. Consequently, variable x will eventually return to its initial value.

A schematic description of a proof lattice for operation *start_write* is given in Figure 3. This proof lattice is aimed at proving that the operation is eventually serviced, assuming that it is invoked when synchronizer *buffer* is in a state corresponding to RSTG node 2. Thus, the entry node is labeled by the predicates $In(2)$ and *s@request*, assuming s is an invocation of operation *start_write*.

Step (a) is performed by applying the first set of construction rules to the entry node in the proof lattice. Note that four edges leave node 2 in the RSTG; however, the edges labeled *start_read* are temporarily disabled, because this operation cannot be serviced due to the pending invocation of higher-priority operation *start_write*. Consequently, only two edges are created in the proof lattice, corresponding to the RSTG edges labeled *end_read*.

Step (b) is performed by applying the second set of rules and temporal logic axiom $(\Box p \vee \Diamond\neg p)$, where p is instantiated to the predicate $In(2)$. So the formula $(In(2) \rightarrow (\Box In(2) \vee \Diamond\neg In(2)))$ is also valid. Consequently, the node under consideration has two descendants, one for each of the disjuncts in the temporal logic axiom.

Step (c) is performed by noting that *end_read* is enabled when $In(2)$ is true, whereas *start_read* is not enabled. Moreover, these two operations deallocate and allocate resource *read_permission*, respectively. Since variable *reader_#* is conformal with this resource, the allocation completion rule can be applied to produce a descendant labeled $(reader\_\# = 0)$. This predicate is in contradiction with the earlier assumption $\Box In(2)$. As a result, a rule leading directly to the exit node, whose label is *enabled(s)*, can be applied in the following proof step (f).

Step (d) is performed by applying a construction rule from the first group. This rule is similar to the rule applied in step (a); however, here the label of a proof lattice node has the additional predicate $\Diamond\neg In(2)$. The execution of an enabled operation can lead from node 2 to node 1 or node 3; however, the edge leading to node 3 is disabled due to the priority specification. Consequently, the formula $(\neg In(2) \rightarrow \Diamond In(1))$ is valid and a descendant labeled $In(1)$ is created for the proof lattice node under consideration. Finally, a rule leading to the exit node can be applied in the step (e), concluding the proof lattice construction.

The above example has shown the liveness of operation *start_write*, assuming this operation is invoked when synchronizer *buffer* is in a given state subset. Similar proofs

have been made for the other synchronizer operations and states, in an effort to complete the liveness analysis of synchronizer *buffer*. These proofs have been generally successful. However, difficulties have arisen in the proof of liveness for operation *start_read*, which has a lower priority than operation *start_write* in the specification of synchronizer *buffer*. Because of this priority condition invocations of *start_read* may "starve", due to the continuous presence of *start_write* invocations. To avoid this phenomenon, the specification of synchronizer *buffer* could be modified to allow invocations of *start_read* and *start_write* to be placed in the same FIFO queue. In this case, all the proof lattices for synchronizer *buffer* can be constructed successfully, thus establishing the liveness of this synchronizer.

Our approach to liveness analysis has been applied successfully to a variety of traditional examples in the domain of concurrent programs, such as the producers/consumers, the sleepy barber [4] and a simplified version of a memory controller. While the full expressive power of this approach is still under active investigation, a prototype that partially automates the proof lattice construction shown here has been implemented [1].

Several features of our approach have made an implementation feasible. First, we have wedded our proof lattice machinery to the RSTG construction. This means that proof flow is directed by RSTG state transitions, thus controlling the size of the proof search space. Second, the absence of loop constructs in operation state changes means that loop invariants need not be generated during proof lattice construction. Third, our assumptions about the behavior of resource allocation and deallocation simplify proof lattice construction.

5 Conclusions

In this paper we have briefly sketched the workings of a proposed automatic synthesis system for synchronization code and we have described one particular crucial feature, a deductive system for establishing the liveness of synchronizer operations.

We believe our method is automatable, and is even potentially practical, for several reasons. First of all, the RSTG construction is relatively straightforward. This is crucial because the RSTG reduces the excessive state information present in system behavior to a small, tractable body. Second, the absence of temporal operators in the specification of operation preconditions means that existing theorem provers can be used in the construction of the RSTG of a synchronizer. Third, our liveness analysis is rather stylized, thus sidestepping the need to use the full deductive power of a temporal logic system during proof lattice construction. Finally, proof lattice construction is facilitated by the absence of loop constructs in operation state transitions, by the absence of temporal operators in operation preconditions, and by the stated assumptions about the behavior of resource allocation and deallocation. As a result, only a limited subset of temporal logic has been

required to build the proof lattices in the example set considered so far.

We believe that the practicality of the proposed approach is enhanced by the structure of program specifications. First, our approach to synchronizer specification is close to the way people think about resources (i.e. in terms of operations performing resource allocations and deallocations). Second, high-level support is provided, so that specifications can include such features as operation preconditions and priorities. Third, these properties are clearly separated in a program specification. We believe that these aspects of our approach compare favorably with previous work that does not support the specification of these properties explicitly [2, 6].

References

[1] U. Buy. *Automatic Synthesis of Resource Sharing Concurrent Programs*, PhD Dissertation, Computer Science Department, University of Massachusetts, Amherst, MA, September 1990.

[2] E. M. Clarke, E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic, *Lecture Notes in Computer Science 131,* Springer-Verlag, New York, 1981.

[3] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the Association for Computing Machinery, 18,* 8, August 1975.

[4] E. W. Dijkstra. *Cooperating Sequential Processes,* Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[5] L. Lamport. Proving the Correctness of Multiprocess Programs, *IEEE Trans. on Software Engineering, SE-3,* 2, March 1977.

[6] Z. Manna, P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications, *ACM Transactions on Programming Languages and Systems, 6,* 1, January 1984.

[7] S. S. Owicki, L. Lamport. Proving Liveness Properties of Concurrent Programs, *ACM Transactions on Programming Languages and Systems, 4,* 3, July 1982.

[8] K. Ramamritham. Synthesizing Code for Resource Controllers, *IEEE Transactions on Software Engineering, SE-11,* 8, August 1985.