Synthesizing Code for Resource Controllers

KRITHIVASAN RAMAMRITHAM

Abstract—A distributed system is viewed as a set of objects and processes utilizing the objects. If a shared object, known as a *resource*, is accessed concurrently, some mechanism is necessary to control use of the resource in order to satisfy the consistency and fairness requirements associated with the resource. These mechanisms are termed *resource controllers.*

In this paper, we address two issues: specification of resources and their controllers and the synthesis of code for the controllers from these specifications. In an object-oriented model, user processes access the resource through specific operations defined on the resource. It is the responsibility of the controller of a resource to service requests for access only when appropriate conditions hold. In addition, the controller should service requests for access with fairness. Our temporal logic-based language permits the specification of safety properties of resources and their controllers, such as mutual exclusion and resource invariants, as well as liveness properties such as fairness. Using meaning-preserving transformation rules, the synthesis algorithm transforms given specifications into necessary conditions for servicing requests and also formulates the strategy for achieving the specified fairness. The synthesis algorithm is target language independent.

Index Terms—Shared resources, specification, synchronization, synthesis, temporal logic.

I. Introduction

A DISTRIBUTED software system is one that is executed on an architecture in which several processors are connected via a communication network. It seems attractive to structure such a system based on the resource-server model in conjunction with object-orientation [9]. Conceptually, an object is a collection of information and a set of operations defined on that information. This model views a distributed system as a set of objects and processes utilizing the objects. A shared object, known as a *resource*, is controlled by a *resource controller* which maintains the consistency and fairness requirements associated with the resource. Most proposed programming languages for distributed systems have facilities for encapsulating resources along with their operations, and hence for supporting the notion of a resource controller, for example, *MOD [4], Ada [5], SR [1], and Argus [14].

In this paper, we address two issues: specification of resources and their controllers and the synthesis of code for the controllers from these specifications. Given the object-oriented model, user processes access the resource through specific access operations defined on the resource. It is the responsibility of the controller to service user requests for access only when appropriate conditions hold. In addition, the controller should service requests for access with fairness. Our temporal logic-based [12], [16] language permits the specification of safety properties of resources and the operations on them, such as mutual exclusion and precedence relationships among operations, as well as liveness properties such as fairness. Using meaning-preserving transformation rules, the synthesis algorithm transforms given specifications into necessary conditions for servicing requests and determines the strategies for achieving the specified fairness. Except for its code generation phase, the synthesis algorithm is target language independent.

In the next section we define our model for a resource controller. Section III introduces the language used for specifying the behavior of resource controllers. Section IV deals with the synthesis of code for resource controllers and discusses rules for the derivation of necessary conditions for service, rules for guaranteeing different types of priority and fairness, and strategies for improving the efficiency of synthesized code. Extensions to our specification and synthesis techniques as well as related work are discussed in Section V.

Manuscript received September 6, 1983; revised March 15, 1985. This work was supported by the National Science Foundation under Grants MCS 77-09369, MCS 82-02586, and DCR-8403097.

The author is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

II. A Model for Resource Controllers

This section formalizes the notion of a *resource controller.* We start with a brief introduction to temporal logic which provides the formalism for our approach. We then present an operational model for resource controllers and define the terms associated with the domain of resource control.

A. Temporal Logic

Pnueli first applied temporal logic for reasoning about safety and liveness properties of concurrent programs [16]. Following along those lines, concurrency is modeled by a nondeterministic interleaving of computations of individual processes. Each computation changes the system *state* which consists of values assigned to program variables and the instruction pointer of each of the processes. Using temporal logic operators, one can specify and reason about the properties of the sequence of states that results from the execution of the concurrent processes.

Since temporal logic is an extension of predicate calculus, a temporal logic statement can involve the usual logical operators $\vee$ (or), $\wedge$ (and), $\sim$ (not) and $\Rightarrow$ (implication) besides the temporal operators $\Box$, $<>$ and UNTIL. The operator $\Box$ is called *always.* $\Box P$ states that assertion P is true now and will remain true throughout the future. The operator $<>$ is called *eventually* and is the dual of $\Box$ in that

$$<>P \text{ IFF } \sim \Box \sim P.$$

Thus, $<>P$ if P is true now or will be true sometime in the future. A requirement such as "every request *will* be serviced"

0098-5589/85/0800-0774$01.00

can be specified as

□{"request for service exists" => <> "request serviced"}.

The operation UNTIL has the following interpretation:

(P UNTIL Q) IFF P will be true as long as Q is false.

(The truth value of P once Q becomes true is not defined by UNTIL. Also, our definition of UNTIL does not assert that eventually Q becomes true and hence differs from the definition of UNTIL given in [6].) The UNTIL operator is typically used for expressing temporal orderings. For example, the fact that a service cannot be provided until there is a request for that service can be stated as

~("request serviced") UNTIL ("request for service exists").

We also use the following derived operator:

P ONLYAFTER Q ($\sim P$ UNTIL Q), P can become true only after Q.

B. A Formal Model for a Resource Controller

By a *resource controller*, we mean a *sequential* process which guarantees disciplined access to a *shared* resource. Thus, constraints essential for maintaining the integrity of a resource are built into its resource controller. Each distinct type of access on the resource is called *operation class*. All instances of a particular type are said to be *operations* in that operation class. Any access to the resource is through the execution of one of the operations. Furthermore, each of the operations can execute only when the resource controller permits it to do so.

Execution of an operation goes through four distinct phases in sequence: Request, Service, Active, and Termination. The *request phase* for an operation begins after a resource controller recognizes that a user program requests execution of that operation. The state of the shared resource, priority associated with the request, invariant properties of the resource, etc., determine the *necessary conditions* for executing an operation. The *service phase* begins when and if the necessary conditions hold and the resource controller decides to permit the execution of the operation. By requiring that the necessary conditions must hold when the operation is serviced, the resource controller guarantees that the specified properties are maintained. The resource controller services only one operation at a time. The *active phase* begins after the service phase ends. It is in this phase that the resource access defined by the operation takes place. The active phase ends when access is complete. The *termination phase* begins after the active phase ends.

In general, the resource controller itself executes the request, service, and termination phases of an operation. In order to simplify the model, we assume that a resource controller executes these three phases of an operation as indivisible actions. To permit the concurrent execution of access operations, the active phase of an operation is not constrained to be executed by the resource controller itself.

We use "p|op" to refer to phase p of operation op. To specify the state of each operation, we introduce predicates whose truth values are depicted in Fig. 1.

In addition to these primitive predicates, we use a predicate

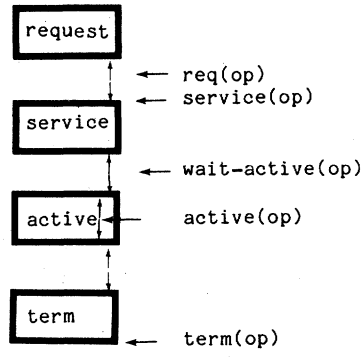


Fig. 1. Sequence of phases in an operation.

enabled(op) which is true when the necessary conditions for servicing *op* hold. (How enabled(op) is determined for each access operation op is the subject of Section IV-A). In this paper, we assume that enabled(op) for every operation op is such that its truth value can be changed only by some action of the resource controller. Thus, actions outside the resource controller cannot change the value of enabled(op).

Specifications in our language will involve the above predicates as well as predicates on the resource state. The model does not introduce any major restrictions on the class of resource control problems that can be solved, or the mechanisms that can be used for resource control, but is motivated by a desire to achieve a suitable abstraction of resource control. We believe that this model is applicable to structured resource control mechanisms such as Monitors [7], Serializers [2], Sentinels [10], Ada Tasks [5], and Synchronizing Resources [1].

III. Specification of Resource Controllers

The specifications that are input to the synthesis system are in a high level temporal logic based language with constructs to specify both invariant and time-dependent properties of a shared resource, the operations on the resource, and the controller of the resource. The design of the specification language is based on the following premise: Since there appears to be only a limited number of *characteristic* properties affecting resource control, the formal semantics of these properties can be *packaged* by employing appropriate keywords. For instance, exclusion between operations in two classes is specified via a statement, involving the names of the two classes and the keyword *excludes*. This statement is endowed with the semantics of the temporal expression that asserts exclusion between the two classes. A salient feature of specifications in our language is that distinct requirements are specified independently of each other. A complete description of the specification language may be found in [19].

The following resource control problem, that of controlling accesses to a disk, will serve as the running example for the paper. Access requests are serviced using the *elevator* algorithm. The disk head moves in one direction until there are no more requests for tracks in that direction, at which time the direction of movement of the head is reversed. If the head is moving *up* (*down*), that is, towards higher-numbered (lower-numbered) tracks, then requests for higher-numbered (lower-

numbered) tracks should be serviced when the head reaches that track. Requests for the current track or lower-numbered (higher-numbered) tracks must wait for the reverse sweep of the disk head. The formal specification for this problem using our language is given in Fig. 2.

The language permits reference to the state of a resource during different phases of an operation. For example, in the above specifications, the term "dir AT req|a", signifies the value of "dir" at the beginning of the request phase of operation a. Quantifiers, FOR EACH and FOR SOME, are permitted only over the domain of operations in a particular class. The resource-state specifications declare that "dir" and "current_track" constitute the resource state variables. The LET clause facilitates the abbreviation of complex predicates.

The elevator algorithm itself is expressed by the specifications of changes to the resource state, the specifications of constraints on these changes, specifications of service-constraints (conditions which should hold when control is at the service phase), and intraclass priority specifications. Specifications of resource state changes state the following: the track being serviced becomes the current_track; if at the beginning of the termination phase of an operation there are no more requests for tracks in the direction of movement of the disk head but there are requests for tracks in the opposite direction, then the direction changes at the end of the termination phase. The resource constraint specifications state the requirement that direction of disk head movement should not change until there are no more requests for tracks in that direction. Service constraint specifications express the dependence of the direction of disk head movement when a request is serviced on the conditions that exist at the request phase. Intraclass priority specifications apply only to enabled requests, that is, those that have satisfied their necessary conditions. The exclusion specification expresses the mutually exclusive execution of accesses whereas the fairness specification states the requirement that every request should be eventually serviced. The formal temporal semantics of the above specifications can be found in [19].

In addition to the language constructs that appear in the above example, our language has constructs to specify precedence relationships, resource state invariants and interclass priority. It should be pointed out that we permit the specification of four types of fairness requirements:

- that operations be serviced *first-come-first*
- that an operation op be serviced
  - –if predicate(op) *is true*
  - –if predicate(op) *is repeatedly true*
  - –if predicate(op) *is always true.*

Predicate(op) is typically dependent on the necessary conditions for servicing op. Subsequently we refer to these four types of fairness as fairness-1, fairness-2, fairness-3, and fairness-4, respectively.

In [19] we discuss certain types of "erroneous" specifications, for instance, inconsistent, incomplete, or deadlock-prone specifications, and discuss techniques to detect their presence in a set of specifications. Using these techniques, for instance, it is possible to determine if the fairness specified for a problem is consistent with the rest of the specifications. Such techniques should be implemented as part of a specifications preprocessor, so that synthesis is undertaken only if the above mentioned errors are absent.

```
OPERATION-CLASS Access(track_#);

RESOURCE-STATE-SPECIFICATION
 tracks_per_disk CONSTANT 100;
 current_track: [0..tracks_per_disk] INITIALLY 0;
 dir: {up, down} INITIALLY up;

LET upreq(a) BE req(a) ∧
                [(a.track_# > current_track AT reqIa) ∨
                 ((a.track_# = current_track AT reqIa) ∧
                  (dir AT reqIa = down))];
    downreq(a) BE req(a) ∧
                [(a.track_# < current_track AT reqIa) ∨
                 ((a.track_# = current_track AT reqIa) ∧
                  (dir AT reqIa = up))];

RESOURCE-STATE-CHANGES
 FOR EACH access OPERATION a:
  current_track=a.track_# AFTER serviceIa;
  IF [dir=up ∧ FOR EACH  access OPERATION a1, (req(a1) => downreq(a1))
             ∧ FOR SOME  access OPERATION a2, downreq(a2)]
       AT termIa
  THEN dir=down AFTER termIa;
  IF [dir=down ∧ FOR EACH  access OPERATION a1, (req(a1) => upreq(a1))
             ∧ FOR SOME  access OPERATION a2, upreq(a2)]
       AT termIa
  THEN dir=up AFTER termIa;

RESOURCE-CONSTRAINTS
 IF (dir=up)
 THEN (dir=up) UNTIL
      [FOR EACH access OPERATION a, (req(a) => downreq(a))];
 IF (dir=down)
 THEN (dir=down) UNTIL
      [FOR EACH access OPERATION a, (req(a) => upreq(a))];

SERVICE-CONSTRAINTS
 FOR EACH a IN access:
  IF (a.track_# < current_track) AT reqIa  THEN (dir=down);
  IF (a.track_# > current_track) AT reqIa  THEN (dir=up);
  IF [(a.track_# = current_track) ∧ dir=up] AT reqIa  THEN (dir=down);
  IF [(a.track_# = current_track) ∧ dir=down] AT reqIa  THEN (dir=up);

EXCLUSION
 Access OPERATIONS EXCLUDE EACH OTHER;

INTRA CLASS PRIORITY
 FOR EACH ENABLED access OPERATION a IS
  a.track_# WHEN dir=down;
  (tracks_per_disk - a.track_#) WHEN dir=up;

FAIRNESS
 SERVICE EACH a IN access IF nec-cond(a) IS TRUE;
```

Fig. 2. Formal specification of the Disk Controller Problem.

IV. Generation of Resource Controller Code

A resource controller takes two types of actions: it services requests that have satisfied conditions necessary for service and it makes appropriate changes to the resource state. Components of the synthesis algorithm which deal with the derivation of necessary conditions for service, the choice of next request to be serviced, and the changes to be made to the resource state are detailed below. Code generation in a specific target language is then discussed. Issues concerning the efficiency of synthesized code are dealt with in the final subsection.

A. Derivation of Necessary Conditions for Servicing Operations

Constraints on servicing an operation are imposed by the

specifications of exclusion, resource state invariants, and precedence relationships and are determined via transformation rules listed below. These rules are used to derive the necessary conditions for servicing operations in order to satisfy the specifications. (Proof of correctness of these rules in given in [17].)

The Mutual Exclusion Transformation Rule: The specification "C OPERATIONS EXCLUDE EACH OTHER" translates to

$$\forall c \in C, \Box \{\text{service}(c) \Rightarrow \forall c1 \in C, (\sim\text{active}(c1) \wedge \sim\text{wait\_active}(c1))\}.$$

The specification "$C1$ OPERATIONS EXCLUDE $C2$ OPERATIONS" expresses the exclusion between operations in class $C1$ and those in $C2$ and translates to

$$\forall c2 \in C2, \Box \{\text{service}(c2) \Rightarrow \forall c1 \in C1, (\sim\text{active}(c1) \wedge \sim\text{wait\_active}(c1))\} \wedge$$
$$\forall c1 \in C1, \Box \{\text{service}(c1) \Rightarrow \forall c2 \in C2, (\sim\text{active}(c2) \wedge \sim\text{wait\_active}(c2))\}$$

The Precedence Transformation Rule: Precedence specifications are transformed into 1) a set of changes to auxiliary variables local to the resource controller and 2) a set of service-constraints which depend on the values of these variables. Suppose that according to the precedence specifications, an operation op2 should be preceded by op1. An auxiliary variable, say $v12$, is introduced such that $v12$ becomes true during the termination phase of op1, op2 is serviced only if $v12$ is true, and $v12$ becomes false during the service phase of op2.

The Resource-State-Invariant Transformation Rule: Using the specifications of exclusion, verify that all operations which change the resource state exclude each other. Suppose that according to the specifications, I is the invariant on the resource state and that operations in class C change the resource state. With I as the post condition, using the specification of changes to the resource state, derive the weakest precondition for operations in C. The weakest precondition derived becomes a constraint on servicing operations in C. Verify that if an operation is serviced when the precondition is true, the invariant will hold during the active phase of the operation.

The constraints on servicing on operation are expressed by the conjunction of

1) the implicit requirement that

$$\forall op, \Box \{\text{service}(op) \Rightarrow \text{req}(op)\},$$

2) the service-constraint specifications, and

3) the constraints derived by the application of the above rules.

If these constraints are true for an operation *op*, then *enabled*(*op*) is said to be true.

Example: By applying the above transformation rules to the disk-scheduler problem and performing logical simplification, we have the following:

$$\{[(a.\text{track\_\#} < \text{current\_track}) \vee ((a.\text{track\_\#} = \text{current\_track}) \wedge \text{dir=up})]\} \text{ AT req}|a$$
$$\Rightarrow$$
$$\{\text{service}(a) \Rightarrow [\text{req}(a) \wedge (\text{dir=down}) \wedge \forall a1 \in \text{access}, (\sim\text{active}(a1) \wedge \sim\text{wait\_active}(a1))]\}$$
$$\{[(a.\text{track\_\#} > \text{current\_track}) \vee ((a.\text{track\_\#} = \text{current\_track}) \wedge \text{dir=down})]\} \text{ AT req}|a$$
$$\Rightarrow$$
$$\{\text{service}(a) \Rightarrow [\text{req}(a) \wedge (\text{dir=up}) \wedge \forall a1 \in \text{access}, (\sim\text{active}(a1) \wedge \sim\text{wait\_active}(a1))]\}$$

B. Rules for Choosing the Next Request to be Serviced

In this section, we present rules for choosing the next request to be serviced, from among those that have satisfied their necessary conditions, in order to achieve the specified priority and fairness.

1) Priority Translation Rule: We define two function *intercp* (interclass prority) and *intracp* (intraclass priority). The specification

INTERCLASS PRIORITY

$C2 > C1$, WHEN R

stands for

$\Box$ [R => $C2$ OPERATIONS HAVE HIGHER PRIORITY THAN $C1$ OPERATIONS]

where the consequent implies

$$\forall op1 \in C1, \forall op2 \in C2, \{[\text{req}(op1) \wedge \text{req}(op2)] \Rightarrow [\text{intercp}(op2) > \text{intercp}(op1)]\}.$$

The specification

INTRACLASS PRIORITY

FOR EACH C OPERATION c IS PRRULE(c), WHEN R

stands for

$\Box$ [R => PRIORITY FOR EACH C OPERATION c IS PRRULE(c)]

where the consequent implies

$$\forall op1, op2 \in C, \{[\text{req}(op1) \wedge \text{req}(op2) \wedge \text{PRRULE}(op2) > \text{PRRULE}(op1)] \Rightarrow [\text{intracp}(op2) > \text{intracp}(op1)]\}.$$

It is possible, as in the case of the disk-scheduler problem, to restrict priority specifications to apply only to enabled operations. It this case, enabled(op) should be substituted for req(op) in the above statements.

Suppose an operation is serviced only when no operations with higher priority exist. This will mean that a lower priority operation will be serviced only after all higher priority operations have been serviced, thereby satisfying priority requirements. Thus, to satisfy priority

$\forall op1, service(op1) \Rightarrow$

$\forall op2 \{req(op2) \Rightarrow [(intercp(op2) \leqslant intercp(op1)) \wedge (intracp(op2) \leqslant intracp(op1))]\}$

that is, the interclass or intraclass priority of existing requests should be less than or equal to that of the serviced operation. Note that any two operations are related by intracp, intercp or neither, but not both. When two operations op1 and op2 are not related by intraclass priority, any relation between intracp(op1) and intracp(op2) is vacuously true. Similarly, when two operations op1 and op2 are not related by inter-class priority, any relation between intercp(op1) and intercp(op2) is vacuously true. Thus, either or both predicates in the above conjunction may be vacuously true.

Let us now consider the transformation of interclass priority specifications. For this purpose, we define the set "higher_pr_classes".

IF $C2$ OPERATIONS HAVE HIGHER PRIORITY THAN $C1$ OPERATIONS

THEN $C2 \in higher\_pr\_classes(C1)$

If no requests exist for operations in higher priority classes then no requests exist with higher interclass priority. Therefore

$\forall C2 \in higher\_pr\_classes(C1), \forall op2 \in C2, \sim req(op2)$

$\Leftrightarrow$

$\forall op1 \in C1, \forall op2 \neq op1, [req(op1) \wedge req(op2)] \Rightarrow [intercp(op2) \leqslant intercp(op1)]$.

We thus have the following interclass priority transformation rules for deriving the constraints on servicing operations.

If interclass priority applies to all requests,

$\forall C1, \forall op1 \in C1,$

$\square \{service(op1) \Rightarrow$

$\forall C2 \in higher\_pr\_classes(C1), \forall op2 \in C2,$

$\sim req(op2)\}$

If interclass priority applies only to enabled requests,

$\forall C1, \forall op1 \in C1,$

$\square \{service(op1) \Rightarrow$

$\forall C2 \in higher\_pr\_classes(C1), \forall op2 \in C2,$

$\sim enabled(op2)\}$

Example: Suppose we have the following interclass priority specification for a controller of a pool of buffers.

Release > Acquire, WHEN cond

where

cond=true IFF "The buffer pool is almost empty".

This specification is intended to expedite the freeing of resources whenever the supply of buffers is depleting quickly. From the semantics of the above specification, we have

cond =>

release operations have higher priority than acquire operations =>

$release \in higher\_priority\_classes(acquire)$.

From the above interclass priority transformation rules, we have the following servicing constraint when "cond" holds.

$\forall a \in acquire, \square [service(a) \Rightarrow \forall r \in release, \sim req(r)]$.

Since release operations have higher priority than acquire operations only if "cond" holds, i.e., $release \in higher\_priority\_classes(acquire)$ only if "cond" holds, the above constraint is not applicable when "cond" is false.

Let us now consider intraclass priority. For this purpose, we define the set "highest_pr_op" for a class C:

If $op1 \in C \wedge$

"INTRACLASS PRIORITY FOR EACH C OPERATION c *IS* PRRULE(c)" $\wedge$

$\forall op2 \in C, op1 \neq op2, [req(op1) \wedge req(op2) \Rightarrow PRRULE(op2) \leqslant PRRULE(op1)]$

then $op1 \in highest\_pr\_op(C)$

Thus an operation in class C belongs to the set highest_priority_op(C) if it has the highest priority within C. That is

$\forall op1 \in highest\_pr\_op(C), \forall op2 \in C, op2 \neq op1,$

$[req(op1) \wedge req(op2)] \Rightarrow [intracp(op2) \leqslant intracp(op1)]$.

We thus have the following transformation rules applicable to intraclass priority specifications:

If intraclass priority in class C applies to all requests,

$\forall op \in C, \square \{service(op) \Rightarrow op \in highest\_pr\_op(C)\}$

If intraclass priority in class C applies only to enabled requests,

$\forall op \in C, \square \{service(op) \Rightarrow op \in highest\_pr\_enabled\_op(C)\}$

where highest_pr_enabled_op(C) is similar to highest_pr_op(C) except that it applies only to enabled operations in class C.

Example: The specification for the disk scheduler problem has an intraclass priority specification which applies to enabled operations. Also, one priority rule applies when dir=up and another when dir=down. The above intraclass priority transformation rules are such that an operation can be serviced only if it has the highest_priority (based on the priority rule that applies, depending on whether dir=up or dir=down) among the enabled operations.

In the following section we discuss how these priority-imposed constraints on service are realized through the use of queues.

2) Use of Queues for Ordering Requests: Queues are used by requests waiting for service. For our purposes, a queue is an ordered set with provision for a new element to be inserted

(enqueued) anywhere within the set and for an element to be removed (dequeued) from anywhere within it. (Some languages, such as Ada [5] have built-in queues with predefined methods for enqueueing and dequeueing. In general, we are assuming that the queues needed for our purposes can be implemented in the target language.)

Associated with each resource controller is an enqueuing module, known as the *enqueuer*, containing the code required to determine the queue in which a given request should wait. The enqueuer constructs a token for each request and enqueues the token in the appropriate queue. Information embedded in a token for a particular request depends in general on the information used for servicing the request. Thus the token for a request can contain the following: its operation class, its arguments, its priority, its necessary conditions, and the resource state at the beginning of its request phase.

Enqueueing a Request: A queue can be a *simple queue* or a *priority queue*. In the case of a simple queue, a new request is enqueued at the end of the queue. If it is a priority queue, the request will be enqueued according to its priority relationship with respect to requests already in the queue.

Dequeuing a Request: For this purpose, we define four selector functions which return the identity of a unique request in the queue. They are

First_req(Q): The first request (from the front) of Q.

Highest_pr_req(Q): The request in Q with the highest priority.

First_enabled_req(Q):
The first request (from the front) in Q which is enabled.

Highest_pr_enabled_req(Q):
That request in Q with the highest priority and which is enabled.

These have been listed in the increasing order of complexity of selecting a request from a simple queue. These procedures use the information stored with each request by the enqueuer.

One of the tasks in implementing priority and fairness is to determine the number and type of queues required and the dequeueing procedures appropriate for them. One queue could be allocated per operation class, multiple queues could be allocated for a single operation class or multiple classes could have one queue allocated for them. The choice is motivated by the need to reduce the amount of search required to select the next request to be serviced. In this context, we infer the following:

If Q is a simple queue and all class C operations enqueue into Q then
Highest_pr_op(C) = Highest_pr_req(Q)
Highest_pr_enabled_op(C) = Highest_pr_enabled_req(Q)

If Q is a priority queue, all class C operations enqueue into it, and the priority of an operation does not change then
Highest_pr_op(C) = First_req(Q)
Highest_pr_enabled_op(C) = First_enabled_req(Q)

In addition, the following inferences can be made.

- If in a simple queue Q the first request is serviced before the rest, then requests in that queue will be serviced in FIFO order.
- If the first enabled request in Q is always serviced before the rest, then requests in that queue which are repeatedly enabled will be serviced.

The truth of the first statement should be obvious from the definition of FIFO. The rationale for the second is as follows: since the first enabled request is always serviced, every request, if it is not serviced before it reaches the front of the queue, will eventually reach the front of the queue. Due to a request being repeatedly enabled, the first request will eventually be enabled at which time it will be serviced. Based on these, we now present the algorithm for the allocation of queues and for the selection of the next element to be dequeued.

Case 0: No priority specifications exist. A single fairness criterion applies to all requests.

Case 0.1: Fairness-1 is specified. Allocate a single simple queue for all requests. Service the first_req from the queue after it is enabled.

Case 0.2: Fairness-2, 3, or 4 is specified. Allocate a simple queue for all requests. Service the first_enabled_req from the queue.

If a single fairness criterion does not apply to all requests, then requests in each class will be enqueued into queues allocated for each class. The following discussion pertains to requests in a single class.

Case 1: There is no intraclass priority specified for the class.

Case 1.1: All requests in the class have the same necessary condition or fairness-1 is specified. Allocate a simple queue for that class. Service the first_req from the queue after it is enabled.

Case 1.2: All requests in the class do not have the same necessary conditions and fairness-2, 3 or 4 is specified. Allocate a simple queue. Service the first_enabled_req from the queue.

Case 2: There is an intraclass priority specified for the class.

Case 2.1: A unique intraclass priority rule is specified for the class.

Case 2.1.1: Priority rule applies to all requests in the class.

Case 2.1.1.1: Priority for an operation does not vary with resource state. Allocate a priority queue for the class. Service the first_req in the queue.

Case 2.1.1.2: Priority for an operation varies with resource state. Allocate a simple queue for the class. Service the highest_pr_req in the queue.

Case 2.1.2: Priority rule applies only to enabled requests in the class.

Case 2.1.2.1: Priority for an operation does not vary with resource state. Allocate a priority queue for the class. Service the first_enabled_req in the queue.

Case 2.1.2.2: Priority for an operation varies with resource state. Allocate a simple queue for the class. Service the highest_pr_enabled_req in the queue.

Case 2.2: Different intraclass priority rules apply depending on resource state.

Case 2.2.1: Priority rule applies to all operations in the class. Allocate a simple queue for that class. Service the highest_pr_req in the queue.

Case 2.2.2: Priority rule applies only to enabled operations.

Case 2.2.2.1: The class can be divided into a finite number of

disjoint subclasses with disjoint necessary conditions. Unique intraclass priority specifications apply to each subclass. For each subclass proceed as in case 2.1.1.

Case 2.2.2.2: Case 2.2.2.1 does not apply. Allocate a simple queue for that class. Service the highest_pr_enabled_req in the queue.

Case 2 considers operations in a single class for which intraclass priority is specified and thus the transformation rules for intra-class priority specifications are embedded in the above algorithm. Interclass priority specifications have to be considered separately. Following our rules for translating interclass priority, the controller is coded so as to service operations in higher priority classes before attempting to service operations in a class with lower priority.

A resource controller services requests when they become enabled. It is conceivable to build the resource controller as a set of guarded commands where a command would be executed only if the associated guard is true. If more than one guard is true, one of these commands is chosen nondeterministically. For our purposes however, when more that one request is enabled, the resource controller should choose the request to be serviced next in accordance with the specified fairness and priority.

Some of the ordering information for achieving priority and fairness is embedded in the queues. Our enqueueing and dequeueing strategies include machinery required to achieve a single type of fairness for all requests (Case-0) or to achieve a single type of fairness for all requests within a particular class for which intraclass priority specifications do not exist (Case 1). In Cases 0.2 and 1.2, it is said that choosing the first_enabled request from a queue will satisfy fairness-2, 3, or 4. Thus the resource controller examines every request beginning at the front of the queue until it finds one which is enabled.[1]

Note that fairness specified is not taken into account while allocating queues for classes with intraclass priority specifications. This is because intraclass priority specifications by themselves express the order in which users expect requests in that class to be serviced. However, it has to be verified that the fairness specified for the class will indeed be satisfied (see Section IV-D for an example).

We now discuss how a resource controller should examine requests from different classes to achieve fairness specified for those classes. If only one class requires fairness-2 or 3 then requests in this class are considered for service before requests in other classes. If operations in more than one class of operations require fairness-2 or 3 then all requests from these classes should be examined in the order in which they arrive, (say by enqueuing the requests into one simple queue) and the oldest enabled request (first_enabled_req) in this queue should always be serviced. Fairness-4 is the easiest type of fairness to guarantee. It suffices to examine the requests requiring this type of fairness after testing all operations requiring other forms of fairness.

Example: Applying the above rules for choosing requests for service to the disk-scheduler problem we find that Case 2.2.2.1 of the queue allocation algorithm applies to this problem. As result we have two queues allocated: one for requests which should be serviced when "dir=up" and another for requests which need to be serviced when "dir=down".

C. Resource State Changes

Our language permits the specification of changes that occur to the state of the resource when different phases of operations are executed. In addition to the specified changes, the need for resource state changes may arise due to the application of transformation rules of the algorithm. For instance, the precedence transformation rule introduces auxiliary variables which are modified during the service and termination phases of operations.

Example: In the disk-scheduler problem, only the specified changes, namely to current_track and dir, are needed. These specifications of resource state changes are carried over to the next phase in which code is generated.

D. Generation of Code for a Specific Target Language

We use the term *target language* to refer to the language in which code for the resource controller should be synthesized. Thus this step of synthesis has to take into account the synchronization primitives and data structuring facilities available in the target language.

From the discussion so far it should be clear that the following are required to realize a resource controller in a specific target language:

- determination of how the notions of operation request, service, execution, and termination translate in the target language;
- transformation of the necessary conditions into a form that can be evaluated using primitives in the language; and
- implementation of different types of queues and dequeueing techniques using data structures in the language.

As discussed earlier, we utilize queues for requests to wait before serviced. To test the presence or absence of requests in in these queues, a function, such as "empty(q)", is needed. Translation of mutual exclusion specifications requires information about active operations in a particular class. Counters of active operations in each class are employed for this purpose.

The synthesis algorithm is designed to handle general resource control scenario since

1) operations in a class can have different enabling conditions,

2) operations in a queue can have different enabling conditions,

3) enqueuing into a queue and dequeueing from a queue can occur in arbitrary order, and

4) the priority for an operation can vary dynamically.

Evidently, such generality is unnecessary for a number of problems and leads to inefficient solutions. In general, the synthesis algorithm generates resource controller code such that it creates processes to execute the active phase of opera-

[1] Here it is assumed that the fairness specified for an operation is *admissible* [19]. A given type of fairness is said to be *admissible* if some resource controller for the problem can guarantee such a fairness. For example, the fairness "Service op IF enabled(op) is True" is admissible only if one of the following hold. 1) When enabled(op1) is true, no other operation is enabled. 2) Even if another operation op2 is enabled, enabled(op1) remains true until op1 is serviced. 3) Even if another operation op2 is serviced, enabled(op1) will eventually become true again.

tions so that, if possible, operations can be active in parallel. However, whenever all operations on a resource exclude each other, the synthesis system generates optimized code so that the resource controller itself executes the active phases of operations. For example, this happens in the case of access operations in the disk-scheduler problem. Since the resource controller itself executes the active phase of an operation "*a*" soon after the operation's service phase is completed

$$\sim\text{active}(a) \wedge \sim\text{wait\_active}(a)$$

will always be true at the beginning of the service phase of any access operation. Hence the above constraint on servicing disk access requests, resulting from the exclusion specification, (see Section IV-A) is not explicitly checked in the derived code (see Fig. 3). Other strategies used for optimization are discussed in the next section.

In [18] we describe a synthesis system geared towards a specific target machine, namely, the applicative multiprocessor system (AMPS) and a specific target language, namely Function Graph language (FGL) [11]. The system uses the simplifier component of the Stanford Pascal Verifier to simplify necessary conditions and conditions that cause changes to resource state. In addition, the simplifier is also used to perform logical deductions. For instance, in the disk-scheduler problem it is used to determine that there are two distinct subclasses with disjoint necessary conditions. Implementation of a synthesis system for generating resource controller tasks in Ada is currently nearing completion [20].

Since the emphasis of this paper is on the algorithm underlying the synthesis system, we do not discuss details of this system but instead give, in Fig. 3, the code generated for the disk-scheduler problem in an abstract target language. In the derived code, all disk access requests are assumed to exist in the "disk_controller_*q*". The enqueuer examines these and enqueues them (using the enqueue operation) in either the "upq" or the downq". In this case, enqueue has three arguments: the element being enqueued, the queue into which it is enqueued, and the priority of the element. The resource controller uses the function "first_req" to remove the first element from the queues. Function "empty" is used to check for emptiness of queues.

As noted earlier, the fairness specification is not utilized in generating the above code, whereas the priority specification is. Thus it is necessary to verify that the fairness requirement that every request will eventually be serviced is indeed true for the generated resource controller. This can be proved formally using the following reasoning: Every request is enqueued either into upq or downq. Based on the assumption that in any practical system there are only a finite number of waiting requests, it can be proved that all requests in a given queue will eventually be serviced, after which point in time, due to the change in direction of the disk head, requests in the other queue will be serviced.

The generated disk-scheduler changes disk-head direction only if there are no more waiting requests in the queue being serviced *and* there are requests waiting in the other queue. Thus the specifications constraining the change in direction of movement of the disk head, which are not utilized during code

```
Disk_Controller is
cons tracks_per_disk   = 100;
var current_track : 0..tracks_per_disk := 0;
   dir : {up, down} := up;
   upq: priority_queue := "empty";
   downq: priority_queue := "empty";
   a : request_element;

Procedure enqueuer;
begin
 loop
  if not empty(disk_controller_q)
   then begin
          a := first_req(disk_controller_q);
          if [dir=up ∧ (a.track_# = current_track)] ∨
              (a.track_# < current_track)
           then enqueue(a, downq, a.track_#);
          if [dir=down ∧ (a.track_# = current_track)] ∨
              (a.track_# > current_track)
           then enqueue(a, upq, tracks_per_disk - a.track_#)
          end
end loop
end enqueuer;

Begin
loop
 enqueuer;
 If dir=up ∧ not empty(upq)
  then begin
         a := first_req(upq);
         current_track := a.track_#;
         - access current_track;
         if empty(upq) ∧ not empty(downq)
           then dir=down
       end
  else if dir=down ∧ not empty(downq)
   then begin
          a := first_req(downq);
          current_track := a.track_#;
          - access current_track;
          if empty(downq) ∧ not empty(upq)
           then dir=up
          end
 end loop
end
end Disk_Controller;
```

Fig. 3. Synthesized code for the disk Scheduler.

generation, can be shown to be satisfied by the generated code. Given the undecidability of predicate temporal logic, it may become necessary for the synthesis system to interact with the user to ensure that the unutilized specifications have been met.

E. Improving the Efficiency of Synthesized Code

It is our experience [17] that synthesized resource-controllers are in general less efficient than hand-coded resource-controllers. The reason for this is that efficiency improvement techniques used in programming are essentially based on heuristics. Thus, any practical synthesis system ought to be able to build a set of heuristics in the process of synthesizing code.

In general, the efficiency of a resource-controller can be improved via the efficient evaluation of constraints and the efficient selection of operations.

Efficient Evaluation of Enabling Conditions: The strategy adopted to achieve efficiency in evaluating enabling conditions is to examine an enabling condition only if there exists a possibility of its being true. Following is a list of strategies which can be used in this regard.

- The enabling condition for a class of operations should be evaluated only if some minimum constraint is satisfied. For example, it is not necessary to evaluate enabling conditions for a particular class of operations unless there are requests in that class.
- Information on the current state of the resource should be used to avoid unnecessary evaluations.
- Simplify all enabling conditions so as to reduce the number of terms in an enabling condition.

Efficient Selection of Operations: Another strategy to improve the efficiency of resource controllers is to construct them so that the next operation to be serviced is selected with the minimum of computation. Thus, for instance, in the case of the disk-scheduler problem, the synthesis system decides to order requests according to their priority so that it can always service the first request from a queue. In contrast, if requests were enqueued in the order they arrive, we have to search for the highest-priority request, thus increasing the selection time. It should be pointed out that the suggested efficiency improvement strategies should be incorporated only after ensuring that they are consistent with the order of service mandated by the fairness and priority specifications.

V. Conclusions

Statements in a temporal logic based specification language are used to express resource control requirements which are supplied to a synthesis system that constructs code for resource controllers. Our synthesis algorithm is based on the recognition of the following.

- Resource controller code is essentially a set of condition-action pairs; when certain conditions hold, certain actions are performed.
- Conditions are evaluated and actions performed according to a certain order; this order guarantees the priority and fairness specified.
- There are basically two types of actions a resource controller is involved in: servicing a request for access to a shared resource, and making changes to local variables.

The algorithm derives the information needed to code the condition-action pairs and orders them to satisfy priority and fairness: from the specifications of mutual exclusion, resource state invariance, and precedence relationships, the constraints for starting an operation are determined; from the specifications of resource state changes and precedence relationships, modifications to variables local to the resource controller are derived; from the specifications of priority and fairness, the basis for ordering the condition-action pairs is extracted. This provides the infrastructure to generate resource controller code. In addition, heuristics are employed to improve the efficiency of the resulting resource controller. Finally, utilizing the primitives available in the target language, code is constructed for a specific target.

The algorithm is indeed practical, as evidenced by the implementation of a synthesis system to generate resource controllers for AMPS in FGL. We have used this sytem to automatically generate resource controllers for a variety of problems found in literature. Also, as mentioned earlier, a synthesis system for resource controller tasks in Ada is nearing completion.

Since our interest is in synthesizing resource controllers from the specifications, if necessary, the specification language should be extended to allow other types of specifications only after its implication from the viewpoint of synthesis is well understood. This area, namely increasing the flexibility of the language while retaining the capability to automatically synthesize resource controller code, needs to be further investigated. In practice, resource controllers should be capable of handling exception conditions. A way to specify exception situations along with the actions to be taken when they arise, and the incorporation of exception handling code, are extensions that need to be incorporated in a full-fledged synthesis system.

Manna and Wolper [15] as well as Clarke and Emerson [3] have also attempted the synthesis of synchronization code starting from propositional temporal logic specifications. Manna and Wolper specify synchronizing processes in a CSP model [8] by abstracting concurrent computation to sequence of events. Clarke and Emmerson utilize branching time logic for specifying synchronization requirements in a shared resource environment. Our human-engineered specification language for resource controllers is based on predicate temporal logic of linear time.

Also, the techniques described in [15] and [3] are confined to synchronization aspects and require the specification for the processes that use the resources. Our scheme, on the other hand, utilizes the specifications of the resource, operations on the resource and the controller, and generates resource controllers that can then be used by any process. Although both permit arbitrary propositional temporal logic statements as specifications, cycles in their model graphs can lead to unsatisfiability of eventualities. In addition, these schemes, due to their adoption of the finite-state model, may not always be applicable.

Synthesis of synchronization code for data abstractions has been reported in [13]. In that approach, properties such as exclusion, priority, etc., are specified through suitable orderings of key events pertaining to an access operation. The language lacks the expressiveness to specify eventualities and synchronization properties dependent on resource state.

Acknowledgment

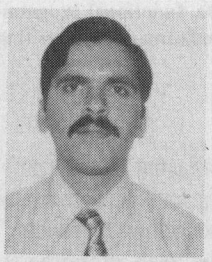
Thanks to Prof. R. M. Keller, who directed the thesis which formed the basis for this work.

References

[1] G. R. Andrews, "Synchronizing resources," *ACM Trans. Program. Lang. Syst.*, vol. 3, pp. 405-430, Oct. 1981.

[2] R. R. Atkinson and C. E. Hewitt, "Specification and proof techniques for serializers," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 10-23, Jan. 1979.

[3] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop Logics of Programs* (Lecture Notes in Computer Science, Vol. 131). New York: Springer-Verlag, 1981.
[4] R. P. Cook, "*MOD-A language for distributed programming," in *Proc. 1st Int. Conf. Distributed Comput. Syst.*, Oct. 1979, pp. 233-241.
[5] *Reference Manual for the Ada Programming Language*, U.S. Dep. Defense, July 1980.
[6] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal analysis of fairness," in *Proc. 7th Annu. Symp. POPL*, Jan. 1980, pp. 163-173.
[7] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, pp. 540-557, Oct. 1974.
[8] ——, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666-677, Aug. 1978.
[9] A. K. Jones, The Object Model: A Conceptual Tool for Structuring Software (Lecture Notes in Computer Science, Vol. 60). New York: Springer-Verlag, 1978, pp. 3-19.
[10] R. M. Keller, "Sentinels: A concept for multiprocess coordination," Univ. Utah, Rep. UUCS-78-104, June 1978.
[11] R. M. Keller, G. Lindstrom, and S. Patil, "A loosely-coupled applicative multi-processing system," in *Proc. AFIPS*, June 1979.
[12] L. Lamport, "'Sometime' is sometimes 'not never'," in *Proc. 7th Annul. Symp. POPL*, Jan. 1980, pp. 174-185.
[13] M. S. Laventhal, "Synthesis of synchronization code for data abstractions," Massachusetts Inst. Technol., Cambridge, Tech. Rep. MIT/LCS/TR-203, June 1978.
[14] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," in *Proc. 9th Symp. Principles of Program. Lang.*, Jan. 1982, pp. 7-19.
[15] Z. Manna and P. Wolper, "Synthesis of communicating processes from temporal lobic specifications," in *Proc. Workshop Logics of Programs* (Lecture Notes in Computer Science, Vol. 131). New York: Springer-Verlag, 1981.
[16] A. Pnueli "The temporal semantics of concurrent programs", in *Semantics of Concurrent Computation* (Lecture Notes in Computer Science, Vol. 70). New York: Springer Verlag, June 1979, pp. 1-20.
[17] K. Ramamritham, "Specification and synthesis of synchronizers," Ph.D. dissertation, Univ. Utah, Salt Lake City, Aug. 1981.
[18] K. Ramamritham and R. M. Keller, "Automatic generation of synchronization code," Dep. Comput. Inform. Sci., Univ. Massachusetts, Amherst, Tech. Rep. 81-21, Sept. 1981.
[19] —— "Specification of synchronizing processes," *IEEE Trans. Software Eng.*, vol. SE-9, Nov. 1983.
[20] K. Ramamritham and P. Sunderrajan, "Automatic generation of code for resource controller tasks in Ada," in *Proc. Symp. Application and Assessment of Automated Tools for Software Eng.*, San Francisco, CA, Nov. 1983.



Krithivasan Ramamritham received the B.Tech. degree in electrical engineering and the M.Tech. degree in computer science from the Indian Institute of Technology, Madras, India, in 1976 and 1978, respectively, and the Ph.D. degree in computer science from the University of Utah, Salt Lake City, in 1981.

Currently he is an Assistant Professor in the Department of Computer and Information Science at the University of Massachusetts at Amherst. His research interests include software engineering, operating systems, and distributed computing. Specifically, his current work covers the specification, verification, and synthesis of resource controllers, scheduling algorithms for real-time systems, mechanisms for access control, i.e., protection in distributed systems, and the incorporation of transactions in distributed object-oriented systems.

Dr. Ramamritham is a member of the Association for Computing Machinery and the IEEE Computer Society.