AUTOMATIC GENERATION OF SYNCHRONIZATION CODE[#]

Krithivasan Ramamritham[+]
Robert M. Keller[*]

COINS Technical Report 81-21
September 1981

[+]Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

[*]Department of Computer Science
University of Utah
Salt Lake City, UT 84112

[#]This material is based upon work supported by the National Science Foundation under grant MCS 77-09369.

SUMMARY

An automatic program synthesis system for synchronization code is described. This system accepts specifications that characterize a synchronization problem and generates a program that conforms to the problem description. It is geared towards a specific target machine, namely, the Applicative Multi Processor System (AMPS) and the generated code is in a specific target language, namely Function Graph Language (FGL). However, the synthesis algorithm is general enough to be applicable to other target mechanisms.

KEYWORDS

Synchronization, specification language, automatic programming, target language.

1. INTRODUCTION

In recent years, problems involved in the production of reliable and correct software have attracted the attention of numerous researchers. The problems assume greater importance today, given the ever increasing cost of software production and maintenance, and considerable decrease in the cost of hardware. Various methodologies have been proposed to improve the quality of programs produced and alleviate problems associated with software maintenance. These efforts have resulted in concepts such as structured programming [4] to aid in the construction of better programs, and verification [8] to prove the correctness of programs already written.

In parallel, research has been progressing in the direction of "automatic programming". The viewpoint in this case has been that if it is possible to formalize the principles involved in writing programs, then one should be able to embed this knowledge into a program synthesis system. Such a system would then be able to accept a description of the required program and based on the knowledge built into it, would automatically produce code that reflects the description.

The disadvantage of the synthesis approach is that since it has not been possible to completely "internalize" the process of programming, one all-encompassing system with such knowledge is yet to be produced. A synthesis system must embody a broad range of knowledge about the principles of programming languages and programming, as well as knowledge pertaining to different domains. As a result, algorithms that automatically produce code are difficult to devise. These algorithms must be verified, though this is a one time cost, if correct programs are to result.

One major shortcoming of synthesis systems is that the results they produce are sometimes less efficient than desired. Nonetheless, the advantages of the synthesis paradigm have lead to burgeoning efforts in the direction of producing viable program synthesis systems [6]. One of these advantages is that such systems eliminate the tedious task of program construction and a posteriori validation. Of significance is the fact that the programs generated are guaranteed to be correct. In addition, the specifications are required to be sufficiently free of ambiguity. Presence of such ambiguities is one of the woes of current programming techniques.

A pragmatic solution to the shortcomings mentioned earlier, while retaining the advantages of the synthesis paradigm, is to limit the scope of synthesis systems. The rationale behind this is the recognition of the differences in the reasoning involved in conceptually different domains. Also, the knowledge required to solve problems in a single domain are likely to be more manageable than that required for a general synthesis system. Thus by building systems with domain-specific knowledge, it becomes feasible to synthesize solutions for problems in that domain. Of course, such an approach lacks the generality required of a general synthesis system. Nevertheless, such software, if carefully designed, can be integrated to produce a cohesive and general automatic programming system.

In an operating system, where user programs concurrently share resources, there is a need for protection and synchronization mechanisms to ensure the integrity of the resources. Program modules that synchronize shared resource access play an indispensible role in the correct functioning of operating systems. In this paper, we refer to such modules as <u>synchronizers</u>. Monitors [10], sentinels [14] and serializers [1] are examples of mechanisms which perform the functions of a synchronizer. This paper concerns our experience with the construction and use of an automatic programming system which produces synchronizers. This system has been designed and implemented with the view of demonstrating the feasibility of our theory [23, 25] of automatic synthesis of synchronizers. Hence we chose as a target vehicle, a machine that allowed us to experiment with various primitives needed for synchronization. At the University of Utah, ongoing research on the Applicative Multi-Processor System provided us a suitable environment for this purpose. Hence the implementation is geared towards the Applicative Multi Processor System (AMPS) and the synchronization code is in a specific target language, namely Function Graph Language (FGL).

Section two explicates the concept of synchronization by developing a synchronization model and enumerating considerations that go into constructing a synchronizer. In section three, we briefly describe our specification language for expressing such considerations. In the next three sections we enter into the specifics of the implemented system. In section four, features of AMPS and FGL that are relevant to this paper are characterized. In section five, the structure of the synthesized code is explained. Section six discusses the implementation of each phase of the general synthesis algorithm. Section seven is devoted to a discussion of issues related to implementing the synthesis algorithm to produce code for other synchronization mechanisms such as monitors, serializers, or Ada tasks. In the final section, we summarize this work and discuss its limitations. A complete example of synthesized code appears in the Appendix.

2. WHAT IS A SYNCHRONIZER?

2.1. The Synchronization Model

This section is intended to clarify the notion of _synchronization_. To maintain the integrity of a shared resource, an answer to the question, "_Who_ is to access the resource, _when_, and _how_?", is essential. A protection mechanism is responsible for _who_ accesses the resource and _how_ the resource is accessed. On the other hand, the synchronizer is responsible for _when_ the access actually takes place.

A shared resource can be considered to be an abstract data type [4] comprised of the following:

- the _data_ that is shared,

- the operations on the data, and
- the synchronizer of the operations.

Any access to the data is through the execution of one of the operations. Furthermore, each of the operations can execute only when the synchronizer permits it to do so. A synchronizer is that sequential process which guarantees disciplined access to shared resources, i.e., accesses to a shared resource are controlled by a synchronizer. Constraints essential for maintaining the integrity of a resource are built into it. Concurrent processes access a shared resource by requesting execution of any of the specified operations. After ensuring that none of the constraints is violated, the synchronizer for that resource subsequently services the requests. We say that an operation becomes active when it is serviced by the synchronizer. The constraints on a resource access arise due to the following considerations.

2.2. Properties Affecting Synchronization

The properties of the shared resource and the properties of operations on the resource that affect the integrity and hence synchronization, is the subject of this section.

Mutual Exclusion of Resource Access Very often, when a process modifies the state of a resource, it should be provided exclusive access. For instance, a write access to a disk file should be permitted only when no other process is accessing the file. A resource that does not require exclusive access can be concurrently accessed by multiple processes.

Invariant Behavior of the Shared Resource Suppose a synchronizer were controlling accesses to a bounded stack. Correct use of the stack demands that the stack neither overflow nor underflow. Thus, accesses to the stack should be permitted only if this invariant property is not violated.

Sequencing Accesses to the Resource Often resource accesses are required to occur according to some strict sequence. Consider a one slot buffer, with operations "put" for leaving a message in the buffer and "get" for retrieving it. For correct use of the buffer, once a process puts a message, another "put" can be performed only after a "get" is executed by some process.

Priority to a Resource Access Returning to the disk file example, we may want processes to read the file as soon as possible, or require that the file always reflect the latest information. The former requires processes requesting read access to be served before the rest, while the latter would favor write accesses. In other words, some predefined priority criterion determines the request which requires faster response.

Fairness to processes requiring access The responsiveness of the synchronizer to requests depends on how "fair" it is in handling requests. When multiple processes are accessing a resource in parallel, the accesses should be permitted in such a way that no single process is made to wait indefinitely. So accesses should be controlled according to some fairness criterion.

The above list, though not meant to be exhaustive, is indicative of considerations that should go into the design of languages for specifying and constructing synchronizers.

2.3. Constituents of a Synchronizer

The purpose of this section is to motivate the various steps in the synthesis algorithm. Towards this end, we examine the constituents of a synchronizer. The most general synchronization scenario will be assumed for this purpose, in order to cater to all possible synchronization situations.

The synthesis procedure we envisage will generate synchronizers which accept requests for operations on a shared resource and grant the requests only if they conform to the prescribed behavior. In essence, for every type of resource access, the knowledge required to infer whether a given request of that type can be allowed to execute is built into the synchronizer for that resource. We refer to this knowledge as constraints on a resource access.

Such constraints are only one aspect of what comprises a synchronizer. The other relates to the ordering of multiple requests that satisfy the constraints. Hence, the algorithm which generates synchronization code has to transform a given set of specifications into a set of constraints and information required to order access requests.

The code for a synchronizer exists in some programming language. We use the term target language to refer to the language in which synchronization code should be synthesized. Also, synchronization code uses the synchronization primitives provided in the machine which executes the synchronization code. We use the term target machine to refer to this machine. It is necessary to obtain the constraints on servicing operations in a form that can be evaluated by the target machine. In this regard, two predominant considerations are:

- How do pending requests manifest in the target machine?
- How does the synchronizer know which operations are currently active?

The answers to these are required since very often constraints on servicing operations are expressed in terms of the predicates on the state of operations.

In every synchronization technique proposed so far, some mechanism is provided for requests to wait until the synchronizer services them. In most cases, queues are used for this purpose. Here we will assume the most general setting for the use of queues, i.e., one in which a user is provided the flexibility to choose the queue in which a request waits and the queue from which a request is chosen for service.

As far as the second question is concerned, the identity of individual requests is rarely required. More often it suffices to know whether some operation in a class is active or not. A counter of active operations is a natural choice for this purpose.

In addition to queues and counters, "synchronizer variables" are necessary to "mirror" the state of the resource. The state of a synchronizer variable is indicative of the state of the resource. Their advantage lies in the fact that the interactions with the shared resource which would otherwise be necessary, are avoided.

Now we are in position to catalogue the constituents of a shared resource.

1. A set of data structures -- queues, counters and synchronizer variables.
2. A procedure which accepts requests and enqueues them onto appropriate queues.
3. A procedure which dequeues requests when requisite conditions are satisfied.

In addition, to obtain modular code, information relating to the operations, such as constraints for their service, their priority, etc., are encapsulated in distinct procedures. So we have the following for every operation class.

1. A procedure which evaluates the enabling condition for operations in that class, hence returns true or false when evaluated.

2. A procedure which is a wait-until version of (1). So this procedure returns true when the enabling condition becomes true.

3. A procedure which incorporates all the changes effected to the synchronizer variables by operations in the class.

4. A procedure which evaluates the priority for an operation based on the priority rule applicable to that class.

The term "class procedure" is used to refer to a procedure in this set.

For reasons which will become apparent subsequently, associated with each queue are two procedures. They contain information on the minimum condition required to service an operation in the queue.

1. A procedure which evaluates the minimum condition to determine if it is true or false.

2. A procedure which is a wait until version of (1). So this procedure returns true when the minimum condition is true.

The term "queue procedure" is used to refer to a procedure in this set.

Synthesizing a synchronizer is then equivalent to synthesizing the necessary data structures, the enqueuing and dequeuing procedures, the class procedures and the queue procedures. (Recall our desire to provide for the most general synchronization scenario. Later we will explain how synchronization code for problems which do not require such generalities can be "optimized".)

This is achieved through the following phases of the synthesis algorithm. There are six steps in automatically generating synchronization code. They are

Parsing phase	A given set of specifications (expressed in the language described in the following section) is parsed.
Translation phase	High-level specifications are translated into a set of constraints on each class of operations.
Simplification Phase	The constraints derived in the previous phase are simplified.
Resource allocation phase	Queues, counters, and synchronizer variables are allocated.

Substitution phase	The constraints are derived in a form that is evaluable in the target language.
Code generation phase	Code for the synchronizer is generated.

These are explained in detail in section 6.

3. THE SPECIFICATION LANGUAGE

The language is designed to specify properties of a shared resource and the operations on it. As we just listed, some of these properties are invariant in nature, e.g. mutual exclusion, whereas fairness is a dynamic or time-dependent property. Hence the specification language should be able to express all properties in a uniform manner. Temporal logic [22] is a tool which facilitates this.

3.1. Specification Language Primitives

Always P	This means condition P will remain true from now on, i.e., P is true now and throughout the future.
Eventually P	This means condition P is true now or will eventually become true.
P UNTIL Q	To be read as "P remains true until Q becomes true". This means if Q eventually becomes true, then P remains true from now until Q becomes true; otherwise ALWAYS P.

In addition to the above temporal operators, the ordinary logic operators V (or), & (and), ~ (not) and => (implies) can also appear in the specifications. To enhance the readability of the specifications, certain operators are derived from the above primitive operators. They are defined as follows:

P ONLYIF Q	P can be true only if Q is true.
P ONLYAFTER Q	P can become true only after Q becomes true.
P AFTER Q	P will become true after Q does.
P TRIGGERS Q	Truth of P causes the truth of Q.

These are derived from ALWAYS, EVENTUALLY and UNTIL, and are formally defined in [25].

A specification statement can involve arbitrary predicates. However, since operations and their synchronization is the domain of interest, specific predicates are used to refer to different phases of an operation.

req(a)	There is a request for operation "a".

start(a)	Operation "a" is permitted to execute.
exec(a)	Operation "a" is executing now.
term(a)	Execution of operation "a" has terminated.

Using the above primitive predicates, we define the following:

sat(a,cond)	there exists a request for operation "a" which satisfies "cond".
req$A	there exists a request for an operation in class A.
exec$A	an operation of class A is active.

In essence, a specification statement is constructed using

1. Temporal logic operators,

2. Operators of predicate calculus,

3. Predicates associated with an operation, and

4. Arbitrary predicates.

3.2. Specification Language Constructs

The specifications that are input to the synthesis system are in a high-level language. The semantics of statements in the language is expressed in terms of the primitives just described. As an example of a complete set of specifications, consider the following synchronization problem and its specification.

Statement of the Problem A fixed number of similar resources is managed by an operating system. User processes acquire a resource by executing the operation "allocate", and the operation "free" releases the resource. Number of resources free at any given time is maintained by "avail". "Maxavail" gives the total number of available resources. In addition, to expedite the freeing of resources, free requests are given higher priority than allocate requests. This is a typical problem which arises in the context of resource management.

High Level Specification of the Problem

```
SYNCHRONIZER Resource_manager IS
OPERATION_CLASSES Allocate;
                  Free ;

OPERATIONS a: allocate;                                          (S1)
           f: free;

RESOURCE_STATE_INFORMATION
```

```
RESOURCE_STATE_SPECIFICATION                                    (S2)
maxavail : CONSTANT 2 ;
avail : [0,maxavail] INITIALLY maxavail;
RESOURCE_STATE_CHANGES                                          (S3)
Allocate : avail <- avail - 1;
Free     : avail <- avail + 1;
RESOURCE_STATE_INVARIANCE                                       (S4)
(avail ≤ maxavail ) & (avail ≥ 0);

SERVICING CONSTRAINTS                                           (S5)
always {start(a) ONLYIF req(a)};
always {start(f) ONLYIF req(f)};

OPERATION EXCLUSION                                             (S6)
 Allocate EXCLUDES Free;
 Free EXCLUDE;
 Allocate EXCLUDE;

INTER-CLASS PRIORITY AMONG ENABLED OPERATIONS                   (S7)
         free > allocate

SCHEDULING DISCIPLINE                                           (S8)
always{Req(f) => eventually Start(f)}
always{[Req(a) & (~enabled(f) UNTIL Start(a))]
                             => eventually Start(a)}
END
```

Following are significant aspects of the specification language:

- (S1) Instances of operations in a class can be referred to by using generic operation names, such as f and a.

- (S2, S3) Data structures constituting the state of the resource, and modifications to the resource state by the operations, can be specified.

- (S4) There is a construct to specify invariance of a resource state predicate.

- (S5) SERVICING CONSTRAINTS specifications express the conditions that should exist when an operation is serviced.

- (S6) Exclusion among operations belonging to the same class or different classes can be specified.

- (S7) Priority among operations within a class (INTRA-CLASS) and between operations of different classes (INTER-CLASS) can be specified.

- Using the specification of sequences, for example,

  A FOLLOWS B

  it is possible to state the order in which a set of operations should be serviced. In the above specification, A and B are two operation classes. Similar to operation exclusion, it is possible to specify the exclusion of sequences.

- (S8) Scheduling discipline statements specify the fairness that is expected of the synchronizer. An op is said to be "enabled" if it satisfies servicing constraint, exclusion, sequence and invariance specifications.

The specifications require that every free request be eventually serviced. Due to the presence of priority specifications, a weaker form of fairness is acceptable for allocate operations. Since requests are made by processes outside the synchronizer, the sequential model assumed precludes the immediate recognition of the presence of requests. This implies that, although at a given time an allocate request may be eligible for service, a free request may have arrived before the synchronizer recognizes this fact, thus preventing the synchronizer from servicing the former. Hence it is required that a allocate request eventually be serviced provided no free request is enabled until the allocate request is serviced. (Other types of fairness are specifiable, based on the behavior of the conditions that enable an operation. For a complete description of the specification language see [24].)

4. THE TARGET ENVIRONMENT

The motivation behind implementing the synthesis algorithm for a specific target is to demonstrate the feasibility of synthesis besides exhibiting the practical issues involved. In choosing a specific target, two considerations were predominant:

1. The target environment should permit experimenting with various synchronization primitives.

2. It should be possible to execute the synthesized synchronizer in order to study its performance.

The existence of a simulator for AMPS along with the data-structuring features of FGL made the combination an attractive target environment.

4.1. AMPS and FGL

The Applicative Multi-Processor System features a loosely-coupled architecture incorporating a large number of processors functioning relatively independently and interacting when necessary. The internal program representation in AMPS is essentially a coded directed graph known as a <u>function graph</u>. Function Graph Language (FGL), the language in which these graphs are defined, is an applicative language in the sense that the underlying basis of computation is the application of functions to data objects to form new data objects. Hence the name Applicative Multi Processor System. AMPS is based on a demand driven data flow model of computation.

Though an FGL program is a collection of graphs, a textual form of FGL exists. Programs in this textual form are compiled into a lower level textual form which is loaded into a simulation of the AMPS system and executed in a manner which simulates parallel execution. Description of the architecture of AMPS appears in [17]. Function graphs are developed in [15] while the precise syntax of FGL is defined in [16]. For the sake of brevity, here we describe only those features of FGL which are essential to follow this exposition.

An FGL program is essentially a composition of FGL functions and procedures. A function always produces the same output for a given input and is specified by giving it a name and a set of arguments. The result of evaluating a function is determined only by its arguments. A procedure is defined and evaluated similar to a function, but may have side-effects in its execution. Using the IMPORTS facility, certain arguments can be implicitly specified, their value determined by context, i.e., by their textual nesting. A textually nested function is defined by including the function in a WHERE .. END declaration within an outer function. The LET facility in FGL is used to abbreviate an expression by an identifier. In summary, a function is defined by

- Its name and list of arguments.
- A list of imported identifiers.
- A list of abbreviations.
- The code which is evaluated to return the value of the function.
- Nested functions.

With this prelude, we are now in a position to discuss the primitives and operators used in coding synchronizers.

4.2. FGL Primitives for Synchronization

In this section, we describe the operators available in FGL to program synchronization code. Some of these are built-in while others have been programmed using the built-in functions of FGL.

<u>Operators for Scheduling</u> In general, in a demand driven model of execution, there are no assumptions made regarding the sequencing of function executions.

Thus, concurrency is achieved "naturally". However, for synchronization control, some mechanism is required to "control concurrency". The following operators in FGL provide the facility to order the evaluation of functions and procedures.

SEQ($f_1,\ldots,f_n$)	evaluates the arguments sequentially; returns the result of evaluating f_n.
PAR($f_1,\ldots,f_n$)	evaluates all the arguments concurrently; returns the result of evaluating f_1 as soon as it is evaluated.
SPAR($f_1,\ldots,f_n$)	evaluates all the arguments concurrently; returns the result of f_n after all the arguments are evaluated.
ARBIT(f_1,f_2)	evaluates f_1 and f_2 concurrently; returns true (nil) if f_1 (f_2) is evaluated before f_2 (f_1); favors f_1 in case of a tie.

These operators are employed to control the execution of functions.

Operators to Avoid Busy Waiting In the demand driven model of execution, as the name indicates, a function is executed only if there is a demand for the result of that function. Under certain circumstances, there is a need to execute a function only when there are two demands for its value. Such a situation arises when "busy-waiting" is to be avoided by the synchronizer. To be more specific, a synchronizer should not have to proceed with its execution unless some condition necessary for it to take an action is true. Suppose a predicate function returned a value only if

1. A synchronizer is waiting for the predicate to become true, and

2. The predicate is true.

Then busy waiting can be avoided. This is made possible by the operator "DJOIN". A DJOIN evaluates its argument only when it has two demands for its result. In the above example, one demand originates from the synchronizer and another from the operator which makes the predicate true.

Using the DJOIN operator, it is possible to have two versions of every procedure that evaluates a predicate.

1. Status determining version: This version returns true if the predicate is true, false otherwise.

2. Wait until version: This version, which uses the DJOIN operator, always returns true; this result is returned when the predicate becomes true.

The above operators are built-in FGL functions. Now we introduce the programmed functions used in the synthesized synchronization code.

<u>Operators for Handling Requests</u> As was indicated earlier, requests waiting for the attention of a synchronizer are enqueued into appropriate queues. A queue has been implemented as a set of tokens ordered according to some criterion, e.g. the order of arrival, or priority of the request in the token. Associated with every queue are the following:

1. A unique name
2. Names of classes (enqueued classes) of operations which wait in the queue.
3. A semaphore which provides exclusive access to the queue.
4. A composite condition (queue condition); this is the minimum condition necessary for a token in the queue to be enabled.
5. A "wait until" version of (4), (wu queue condition).

A queue is created by executing

```
GQUEUE(<q>,
       <enqueued classes>,
       <queue condition>,
       <wu queue condition>).
```

This returns a reference to the created queue which is used for subsequent manipulation of the queue.

A token is enqueued into a queue by executing

```
ENQ(<q>, <token>).
```

Later in the chapter we shall elaborate on the constitution of a <token>. Here it suffices to note that one of the components of a <token> is a reference to the access operation which the enqueueing process desires to execute. ENQ returns the result of executing that operation after it is serviced by the synchronizer. Thus the enqueueing process is delayed until the requested access is serviced.

A token is dequeued from a queue by executing

```
DEQ(<q>, <token>)
```

by which the referred token is removed from the queue and returned as result.

The following functions evaluate the status of a queue.

```
NON-EMPTY(<q>)          true if <q> is non-empty,
                           false otherwise.
```

WAITQ(<q>, 'empty)	waits until <q> becomes empty, then returns true.
WAITQ(<q>, 'nonempty)	waits until <q> becomes nonempty, then returns true.

<u>Operators for Servicing Requests</u> There are two operators for servicing requests: EX and DET_EX.

When a synchronizer evaluates

EX(<q>, <token>),

<token> is dequeued from <q> and the dequeued operation executed.

When the synchronizer evaluates

DET_EX(<synchronizer>, <q>, <token>, <counter>),

<counter> is incremented by one, <token> is dequeued from <q> and the dequeued operation executed. This execution takes place in parallel with a recursive invocation of the synchronizer. When the operation terminates, <counter> is decremented by one.

<u>Operators for Manipulating Counters</u> A counter is created by executing

GCOUNT(<initial value>)

where <initial value> is normally zero. This returns a reference to the created counter. Associated with every counter is a semaphore which provides exclusive access to the counter.

The following operators manipulate the value of a counter:

INCRC(<ctr>)	Increments the value of <ctr> by one.
DECRC(<ctr>)	Decrements the value of <ctr> by one.

The following operators evaluate the value of a counter:

CTRVALUE(<ctr>) returns the current value of <ctr>.

WAITC(<ctr>, 0) returns true when <ctr> gets a zero value.

WAITC(<ctr>, 1) returns true when <ctr> gets a non-zero value.

<u>Operators for Manipulating Synchronizer Variables</u> A synchronizer variable is used to mirror the resource state. Associated with a synchronizer variable is a semaphore which guarantees mutually exclusive access to the variable. Also, if a maximum or minimum bound is specified for a resource state variable, then the corresponding synchronizer variable also has similar

bounds.

A synchronizer variable is created by executing

GRES(<initial-value>, <min-value>, <max-value>).

This returns a reference to the created variable.

A synchronizer variable is modified by executing

CHANGERES(<variable>, <newvalue>),

when the value of <variable> is modified to <newvalue>.

The value of a synchronizer variable is returned when

RESVALUE(<variable>)

is executed, while the operator

WAITRES(<variable>, <relational-operator>, <value>)

returns true when <variable> bears the relationship given by <relational-operator> to <value>. Thus

WAITRES(a, "=", 0)

returns true only after the synchronizer variable "a" has the value 0.

Operators for selecting requests from a queue The selection functions used to dequeue tokens from queues are

FIRST(<q>)	This function returns the identity of the first request (from the front) in the referred queue.
HIGHEST-PR(<q>)	This function returns the identity of the request with the highest priority in <q>. If more than one request has the highest priority, then the first from the front of the queue is returned.
FIRST-EN(<q>)	This function returns the identity of the enabled request closest to the front of the queue.
HIGHESTPREN(<q>)	The returned request has the _highest priority_ among the _enabled_ requests in the queue.

In essence, the programmed operators provide the ability to create and manipulate all the data structures used in a synchronizer, namely,

- Queues,

- Counters, and

- Synchronizer variables.

The above operators are built using the primitives developed [13] for resource control in the demand-driven data-flow model of computation used in AMPS.

5. THE STRUCTURE OF RESULTING CODE

In this section, the structure of the synthesized code is explained. First we discuss the procedures synthesized for synchronizing access to a specific shared resource "SR". This is followed by an explication of their nesting structure.

5.1. Synthesized Procedures

The SR-DATABASE: This procedure comprises the queues, the counters and the synchronizer variables, i.e., all the data structures used for synchronization, and presumably even the shared resource. When SR-DATABASE is created, it returns a reference to the procedure (ENQUEUER) which enqueues requests for access to SR.

The ENQUEUER: This procedure possesses the knowledge required to enqueue requests into queues. Based on the conditions that prevail when a request arrives, the ENQUEUER builds a token with the information pertaining to that request and enqueues the token into an appropriate queue. To enqueue a request, a user invokes the enqueuer with the name of the access operation and the arguments to it.

The SYNCHRONIZER: This is the procedure which performs synchronization of access operations and hence is the crux of the synthesized code. It essentially waits for appropriate conditions to hold and services requests according to specifications.

In addition to the above, the "class procedures" and the "queue procedures" are also synthesized.

There are certain implications of having separate procedures for enqueuing and dequeuing. The advantages are:

1. Separation of concerns is achieved in a modular fashion.
2. It increases the parallelism; while a request is being enqueued, another request could be dequeued.

These advantages are derived at the cost of possible increased conflict in accessing the queues, counters and the synchronizer variables.

5.2. Nesting of Synthesized Procedures

The following nesting diagram portrays the structure of the synthesized code.

```
SR-DATABASE
where
  ENQUEUER
  where
   -- the class procedures --
  end
  SYNCHRONIZER
  -- the queue procedures --
end
```

The following explanation justifies this block structure.

The ENQUEUER constructs a token for each request. A token comprises the following information:

- Name of the class to which the request belongs.
- Arguments to the request.
- Value of the synchronizer variables when the request arrives.
- Reference to the code for the operation.
- Reference to the procedure which evaluates the priority of the operation.
- Reference to the procedure which evaluates the enabling condition for the operation.
- Reference to the procedure which evaluates the wait-until version of the enabling condition for the operation.
- Reference to the procedure which incorporates the changes to the synchronizer variables when the operation executes.

Since only the ENQUEUER requires the reference to the class procedures, they are nested with the ENQUEUER.

As noted earlier, to construct the queues, SR-DATABASE requires the following information:

- The name of the queue.
- The classes of operations which enqueue into the queue.
- The procedure which evaluates the minimum condition necessary for a request in the queue to be enabled.
- The procedure which evaluates the wait until version of the minimum

condition.

Hence the queue procedures are nested with the SR-DATABASE.

The rationale behind nesting the ENQUEUER and the SYNCHRONIZER within SR-DATABASE is the following: For every shared resource, there is a unique synchronizer, an associated enqueuer, and a database. Also, the only information that need be "visible" to a user are:

1. The name of the shared resource, and
2. The names of the operations on the resource.

To access a shared resource, a user process invokes the database procedure for the resource. This returns the reference to the associated enqueuer. For every access to the synchronized resource, the enqueuer is invoked with the name of the access operation and the arguments to the operation. Thus the block structure ensures that a shared resource is accessed only through the access operations. In other words, limited protection is achieved through the nesting structure.

How are the

- Shared resource, and
- Procedures for the operations on the resource

manifest in the code. The former can be part of the database, while the procedures for the operations can be conveniently nested within the database procedure, to make use of the protection offerred by the synthesized code. Obviously, since the definitions of the resources and the operations are not required for synthesis, they have to be incorporated after the code is synthesized.

6. THE SYNTHESIS SYSTEM

This section describes how the theoretical framework developed in [25] to perform synthesis is implemented to generate synchronizers in the target environment provided by AMPS and FGL. The resulting synchronizer has the structure described in the previous section. The implementation is coded in RLISP - an Algol like extension of Standard LISP [21]. To perform logical deduction, inference and simplification, the simplifier part of the Stanford Pascal Verifier [8] is used.

The six steps in automatically generating synchronization code are explained in detail in the following pages. Their functions are portrayed by block diagram in Figure 6-1.

6.1. The Parsing Phase

Specifications expressed in the language briefly described in section 3 are translated into an internal representation. This translator is written using the META/REDUCE translator writing system [20] running on top of REDUCE (a LISP extension for algebraic computation [9]).

6.2. The Translation Phase

The objective of the following translations is to derive constraints on servicing an operation. Such constraints are embedded in the specifications of exclusion, invariance, and sequencing. For each operation "c", this phase determines "enabled(c)"in the following form:

$\forall c \in C$ always{start(c) ONLYIF enabled(c)}

If enabled(c) is the conjunction of all constraints implied by various specifications, then when an operation "c" is serviced it does not violate the given specifications. The implied constraints are derived by applying certain translation rules. We will examine each of these now.

For each resource state variable, a synchronizer variable is provided to "mirror" the state of the resource. If the synchronizer variable is given the same name as the corresponding resource state variable, then every constraint can be considered to be a constraint on the value of the synchronizer variable. For instance, invariance of the value of a synchronizer variable implies the invariance of the corresponding resource state variable. Henceforth, any reference to the resource state should be considered to be a reference to the corresponding synchronizer variable. The following rule is applied to derive the constraints implied by an invariance specification.

<u>The Invariance Translation Rule</u>

From

- the invariance specification, and
- the specification of changes to the resource state,

determine

- the "precondition" [19] for operations in each <operation-class>.

```
Specifications
  V
  |
 _|_______________
|   parsing       |
|___phase_________|
  |
  |
  V Internal representation of the
  |  specifications.
  |
 _|_______________      _____________
|   translation   |-->--| simplifier |
|___phase_________|--<--|____________|
  |
  |
  V Overall constraints on servicing the operations,
  |  specified fairness.
  |
 _|_______________      _____________
| simplification  |-->--| simplifier |
|___phase_________|--<--|____________|
  |
  |
  V Simplified form of the constraints,
  |  fairness.
  |
 _|____________________
| resource allocation |
|___phase_____________|
  |
  |
  V Queues, counters, synchronizer variables,
  |  constraints, fairness.
  |
 _|_______________
| substitution    |
|___phase_________|
  |
  |
  V Queues, counters, synchronizer variables,
  |  constraints expressed in the target language,
  |  fairness.
  |
 _|________________
| code generation |
|___phase_________|
  |
  |
  V

Synthesized code.
```

Figure 6-1: Block Diagram of the Synthesis System

By starting an operation only if the derived pre-condition is true, during the execution of the operation the invariance condition will hold.

Exclusion Translation Rule

The specification "A EXCLUDE B" is translated into

$\forall a \in A, \forall b \in B,$
$Start(a) \Rightarrow \ \sim Exec\$B$
$Start(b) \Rightarrow \ \sim Exec\$A.$

The specification "A EXCLUDE" translates into

$\forall a \in A,\ Start(a) \Rightarrow \ \sim Exec\$A$

By servicing an operation only if operations that it is expected to exclude are not active, the specified exclusion is satisfied.

Sequence Translation Rule There are two cases to consider:

1. Execution of different instances of a sequence exclude each other.

2. Execution of different instances of a sequence can proceed concurrently.

Case I

"<operation-class-2> FOLLOWS <operation-class-1>"

translates to

$\exists C_{12}$, $\forall$op1$\in$<operation-class-1>, $\forall$op2$\in$<operation-class-2>,
start(op1) TRIGGERS C_{12}
start(op2) ONLYIF C_{12}
start(op2) TRIGGERS $\sim C_{12}$ (1)

where C_{12} is a predicate which is initially false.

The following translations are required to ensure that a new instance of a sequence starts only after the termination of the previous instance. For each distinct (terminator, initiator) pair, say (<operation-class-1>, <operation-class-2>), we add the following:

$\exists C_{12}$, $\forall$op1$\in$<operation-class-1>, $\forall$op2$\in$<operation-class-2>,
start(op1) TRIGGERS C_{12}
start(op2) ONLYIF C_{12}
start(op2) TRIGGERS $\sim C_{12}$ (2)

where C_{12} is a predicate which is initially true. Statements in (2) are similar to those in (1) above.

Case II

"<operation-class-1> FOLLOWS <operation-class-2>"

translates to

$\exists C_{12}$, $\forall p$,
$\forall op1 \in$ <operation-class-1>, $\forall op2 \in$ <operation-class-2>,
start(op2) & (C_{12} = p) TRIGGERS (C_{12} = p+1)
start(op1) ONLYIF $C_{12} > 0$
start(op1) & (C_{12} = p) TRIGGERS (C_{12} = p-1)

where C_{12} is a counter which is initially 0.

The motivations behind these translations is to constrain the servicing of operations in a sequence through predicates which are enabled only after a previous operation in the sequence is serviced.

Priority Translation Rule Suppose

- "Higher-priority-classes(OPC)" is the set of operation classes whose operations have higher inter-class priority than operations in OPC.

- "Highest-priority-op(OPC)" gives the set of operations in class OPC which have the highest priority in class OPC.

- "Highest-priority-enabled-op(OPC)" is similar to highest-priority-op(OPC) except that it applies only to enabled operations in class OPC.

If inter-class priority applies to *requested* operations,

$\forall$OPC1, $\forall op1 \in$OPC1,
$\forall$OPC2$\in$higher-priority-classes(OPC1),
always {start(op1) ONLYIF ~req$OPC2}

If inter-class priority applies to *enabled* operations,

$\forall$OPC1, $\forall op1 \in$OPC1,
$\forall$OPC2$\in$higher-priority-classes(OPC1),
$\forall op2 \in$OPC2,
always{start(op1) ONLYIF ~enabled(op2)}

If intra-class priority applies to all *requested* operations,

$\forall$OPC $\forall op \in$OPC,
always{start(op) ONLYIF op$\in$highest-priority-op(OPC)}

If intra-class priority applies only to *enabled* operations,

$\forall$OPC $\forall op \in$OPC,
always{start(op) ONLYIF
        op$\in$highest-priority-enabled-op(OPC)}

Refer to the appendix (page 42) to see the effect of the translation phase for the resource manager problem.

6.3. The Simplification Phase

In this phase, the derived conditions are simplified. They are

1. Enabling conditions of operations.

2. Conditions that trigger changes to the synchronizer variables.

3. Constraints implied by priority specifications.

(1) and (2) require straightforward simplification. Let us consider (3) in some detail.

In the translation of inter-class priority specifications if priority is applicable to all requests, the absence of requests in the higher priority classes when an operation starts, satisfies the priority specification.

On the other hand, if priority is applicable only to enabled operations (as in the Resource Manager problem), then priority specification translates to the absence of enabled operations in the higher priority classes. Suppose enabled operations in "OPC2" have higher priority than enabled operations in "OPC1". Let us say that a condition "C" is true if there are no enabled operations in OPC2. One naive method to determine if C is true is to examine each request in OPC2 and test if it is enabled. This blind search can be avoided by recognizing the following facts: If all operations in OPC2 have the same enabling condition "E", then either they are all enabled or disabled. Thus,

~E => C.

Also, when an operation is serviced, all constraints including the priority constraint should be satisfied. Thus it is necessary to evaluate ~E only if other constraints are satisfied. So the constraint on starting operations in OPC1 implied by higher priority to operations in OPC2 is derived by simplifying ~E assuming the enabling condition for OPC1.

It is worth pointing out that the implementation so far has not utilized any information about the target environment. Subsequent phases will be target language dependent. Refer to the appendix (page 42) to see the effect of the simplification phase for the resource manager problem.

6.4. The Resource Allocation Phase

The resources necessary for synchronization, namely queues, counters and synchronizer variables, are allocated in this phase.

To increase the flexibility of the use of queues, 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. This use departs from the special case of first in first out queues. We still prefer to call them queues, since in a majority of situations, the first element <u>in</u> happens to be the first element <u>out</u> of the queue.

<u>Enqueuing a request</u> For the purposes of enqueuing, a queue can be

- a simple queue, or

- a priority queue.

In a simple queue, a new request is enqueued at the end of the queue. In a priority queue, all requests are ordered according to some priority criterion and hence a new request takes its position in accordance with its priority relationship with respect to those already in the queue.

<u>Dequeuing a request</u> For the purposes of removing a request from a queue, there are various factors which determine which request is to be dequeued next for service. These are based upon

- the class of the requested operation,

- the priority of the operation, and

- whether the operation is enabled or not.

The "selector functions" defined in section 4.2 are based on these factors. Given that there are a number of ways in which a request for an operation can be enqueued into a queue and dequeued from a queue, how does the synthesis algorithm choose the appropriate one? The choice of enqueuing and dequeuing procedures are determined by the "queue allocator". The queue allocator chooses a particular type of queue and the associated selector function on the basis of priority and fairness specifications. The rationale for the queue allocator's actions will confirm the fact that priority and fairness criteria are preserved by the queues.

<u>The Queue Allocation Rule</u>

<u>Case</u> 0:
It is required to retain the ordering among all the requests that access the shared resource:

<u>Case</u> 0.1:
The scheduling discipline requires that operations be serviced as per their order of arrival.

Allocate a single <u>simple</u> queue for all requests
Always service the <u>first</u> request.

Rationale:
By the definition of first come first served.

<u>Case</u> 0.2:
The scheduling discipline requires that all requests which satisfy the constraints infinitely often, be eventually serviced:

Allocate a <u>simple</u> queue for all requests
Always service the <u>first</u>-<u>enabled</u> operation

Rationale:
Since the first enabled operation is always served, every request, if not already serviced, will eventually reach the front of the queue. Due to its being enabled infinitely often, the first operation will eventually be enabled, at which time it will be serviced.

If a total ordering of all requests is not required, then requests in each class will be enqueued into individual queues. The following discussion pertains to a single class of operations.

<u>Case</u> 1:
There is no intra-class priority applicable to that class:

<u>Case</u> 1.1:
All requests for operations in the class have the same enabling condition:

Allocate a <u>simple</u> queue for that class
Always service the <u>first</u> operation from that queue

Rationale:
Since all operations have the same enabling condition, the operations should be serviced in the order of arrival of their requests.

<u>Case</u> 1.2:
All requests for operations in the class do not have the same enabling condition:

Allocate a <u>simple</u> queue
Always service the <u>first-enabled</u> operation from that queue

Rationale:
By servicing the first-enabled operation, all operations that satisfy the constraints at a given instance will be serviced.

<u>Case</u> 2:
There is an intra-class priority specification for that class:

<u>Case</u> 2.1:
There is a unique intra-class priority rule

applicable to that class:

Case 2.1.1:
Priority rule applies to all operations in that class:

Case 2.1.1.1:
Priority for an operation does not change with the resource state:
Allocate a priority queue for that class
Always service the first request from that queue
Rationale:
Since priority applies to all operations in that class, and the priority for an operation is static, by enqueuing the requests according to their priority, it suffices to examine only at the first request in the queue.

Case 2.1.1.2:
Priority for an operation does change with the resource state:
Allocate a simple queue for that class
Always service the highest-priority operation from that queue
Rationale:
Since priority changes with resource state, the request with the highest priority needs to be determined dynamically and serviced.

Case 2.1.2:
Priority rule applies only to enabled operations in that class:

Case 2.1.2.1:
Priority for an operation does not change with resource state:
Allocate a priority queue for that class
Always service the first-enabled request from that queue
Rationale:
Since only enabled operations need to be considered, by enqueuing operations according to their priority (which does not change), always the first enabled operation needs to be serviced.

Case 2.1.2.2:
Priority for an operation does change with the resource state:
Allocate a simple queue for that class
Always service the highest-priority-enabled request from that queue
Rationale:
Since priority changes with resource state, the operation with the highest priority among the enabled operations needs to be determined dynamically.

Case 2.2:
Multiple priority rules are applicable to that class:
(depending on the resource state)

```
Case 2.2.1:
 Priority applies to all operations in the class:
   Allocate a simple queue for that class
   Always service the highest-priority request
                                   from that queue
 Rationale:
   Since priority changes with resource state, this is similar to case
2.1.1.2 above.

Case 2.2.2
 Priority applies only  to enabled  operations in that class:
   Allocate a simple queue for that class
   Always service the highest-priority-enabled request
                                   from that queue
 Rationale:
   This is similar to case 2.1.2.2 above.
```

Thus the queue allocator determines the number of queues required for a particular synchronization problem. For each queue it determines whether it is a simple or priority queue and decides the appropriate selection function for that queue. Notice that the translation rule for intra-class priority specifications is embedded in the queue allocator.

Knowing the enabling condition for the operations that enqueue into a queue, the queue allocator also derives the minimum condition necessary for an operation in the queue to be enabled. This is derived as follows:

1. If operations in a single class enqueue into a queue, then the conjunction of constraints common to every operation in that class is the required minimum condition.

2. If more than one class of operations enqueue into a queue, then the conjunction of constraints common to every operation in those classes is the required minimum condition.

Here again, the simplified form of the derived minimum condition is used. One feature which merits mentioning is that the queue allocator incorporates certain efficiency improvement strategies, by which a selection function with a higher efficiency is chosen in preference to a more obvious but less efficient selection function. (Ref. cases 2.1.1.1 and 2.1.2.1 of the queue allocator.)

Returning to the functions of the resource allocation phase, the counter allocator allocates one counter per class and is fairly straightforward. The synchronizer variable allocator allocates one variable per resource state variable. Information on bounds of resource state variables is extracted from the specifications. Also allocated in this phase are the counters and predicates introduced by the translation of sequence specifications. Refer to the appendix (page 43) to see the effect of the resource allocation phase for the resource manager problem.

6.5. The Substitution Phase

In this phase, conditions that trigger any synchronizer action, namely, the enabling conditions, the queue conditions and the triggering conditions are expressed in terms of FGL operators.

The following mapping is used to determine the truth of the above conditions.

```
req$C     -> NON-EMPTY(C_q)

~req$C    -> EMPTY(C_q)

exec$C    -> CTRVALUE(C_ctr) > 0

~exec$C   -> CTRVALUE(C_ctr) = 0

sr_var=p -> RESVALUE(sr_var)=p
```

In the above mappings, "C" is an operation class, "C-q" is the queue allocated for this class, "C_ctr" is the counter allocated for the class, "sr_var" is a synchronizer variable and "p" is a constant.

The following mapping is used to determine the wait-until version of the above conditions.

```
req$C     -> WAITQ(C_q,'nonempty)

~req$C    -> WAITQ(C_q,'empty)

Exec$C    -> WAITC(C_ctr,1)

~Exec$C   -> WAITC(C_ctr,0)

sr_var=p -> WAITRES(sr_var,"=",p)
```

Using these mappings from the primitives in the specification language into primitives in FGL, the enabling conditions and queue conditions are expressible in terms of FGL primitives. For example,

```
1) req$C & ~exec$C
   ->
   NON-EMPTY(C-q) & C_ctr=0

2) wait until(~req$C & synchronizer_variable=p)
```

```
    ->
    SPAR(WAITQ(C_q,'empty),WAITRES(synchronizer_variable,"=",p)

 3) wait until(req$C V exec$C)
    ->
    ARBIT(WAITQ(C_q,'nonempty),WAITC(C_ctr,1))
```

These mappings essentially involve substitutions and hence the name of this phase. Refer to the appendix (page 43) to see the effect of the substitution phase for the resource manager problem.

6.6. The Code Generation Phase

The code generation phase is designed to generate code for the various procedures which constitute the synchronization code and is implemented as a set of modules, one for each procedure in the synthesized code. Each module outputs code in conformity with the syntax of FGL using the information resulting from the earlier phases. In addition, the resulting procedures are nested to conform to the structure defined in section. Code generation for all procedures but the SYNCHRONIZER is facile.

In simple terms, a synchronizer takes certain actions when certain conditions are true. In other words, synchronization code is primarily a set of guarded commands [5]. Basically a synchronizer takes two types of actions. They are:

1. Starting an operation.

2. Cause appropriate changes to the state of the resource.

For (1), one of the FGL procedures EX or DET_EX is used. Information required for (2) is obtained from the specifications of changes to the resource state by the operations. In the substitution phase we provided rules for translating primitives in the specification language into primitives in the target language. Hence constraints for starting an operation, and conditions which trigger state changes, are expressible in terms of the primitives in the target language. Thus, information required to code the guarded commands is available at the end of the mapping phase. What remains is to order these guards suitably so as to maintain the fairness.

As was explained in section 3, our specification language permits a user to specify the fairness desired by him. In the synchronizer that we envisage, the guards will be examined in an order which preserves the fairness specified. This is unlike the usual guarded commands where if more than one guard is true, one of the corresponding commands is chosen nondeterministically.

If the scheduling discipline is "first come first served", a single simple queue exists for all requests. The synchronizer waits until the first request is enabled and then services it.

If the scheduling discipline requires that those operations which are enabled infinitely often be eventually serviced, then again a single queue is used for all requests. The synchronizer examines every request from the front of the queue until it finds one which is enabled. That request is then serviced. So in these two cases, request arrival order is utilized for servicing.

Suppose that the scheduling discipline specified is: if an operation is enabled, then it should be eventually serviced. If only one class of operations requires this type of service, then operations in this class are considered for service before operations in other classes. If more than one class of operations require this type of service, then all requests in such classes are enqueued into a single queue and always the first enabled operation in that queue is serviced.

Suppose it is required that an operation be serviced only if it remains enabled until the synchronizer services it. This is the easiest service to guarantee, for it suffices to test the enabling conditions of such operations only after testing enabling conditions of operations which require better forms of service.

Other significant aspects of generating the SYNCHRONIZER are detailed below.

- All conditions that trigger changes to the synchronizer variables are first tested and if they hold, the specified changes are effected.

- Enabling conditions are evaluated and operations serviced in a sequence which preserves the specified fairness.

- An EX or a DET_EX procedure is invoked when an operation becomes serviceable.

- Changes triggered by the "start" of an operation are effected before an EX or DET_EX is executed.

- When an operation is executed using EX, the synchronizer continues with its actions after the termination of the operation.

- When an operation is executed using DET_EX, a new invocation of the SYNCHRONIZER starts execution in parallel with the execution of the serviced operation.

Also, to avoid unnecessary computations, the following optimizing steps are employed:

- Control within the SYNCHRONIZER does not progress until one of the classes is enabled.

- If operations in a queue have different enabling conditions, then they are not examined unless the queue condition (the minimum condition for an operation in the queue to be enabled) is satisfied.

- If the sequence specifications imply a totally ordered execution of all operations, then operations are executed in sequence, using the FGL primitive SEQ.

Thus at the end of the code generation phase, code is automatically produced from a given set of specifications.

Before this section is concluded, some remarks are in order. This implementation is for the most general synchronization scenario. For instance,

- Operations in a class can have different enabling conditions.

- Operations in a queue can have different enabling conditions.

- Enqueuing into a queue and dequeuing from a queue can occur in arbitrary order.

- Priority for an operation can vary dynamically.

Hence we need to have procedures which evaluate

- The enabling conditions of each operation.

- The queue conditions.

- The priority of each operation.

Evidently, such generality is unnecessary for a number of problems and leads to inefficient solutions. For instance, if all operations in a class have the same enabling condition, then the cost of evaluating an operation's enabling condition through a procedure call can be avoided by evaluating it "in place". Such optimization situations can be enumerated and are fairly straightforward to implement. (The code produced by the code generator for the resource manager problem is shown in the appendix (page 44). This code incorporates such optimizations.) The significant aspect of the implementation described here is that by providing for the most general synchronization situation, no synchronization problem is a priori eliminated as being unsynthesizable.

6.7. Efficiency of the Synthesized Synchronizer

A synchronizer accepts requests for operations on a shared resource and services them when it is satisfied that requisite constraints are met. The function of the synchronizer then is to evaluate the constraints and determine which request to service next. Efficiency of a synchronizer relates to this function.

Suppose two synchronizers A and B are constructed for a given problem. Given similar synchronization situations, i.e., for a particular state of the resource and a set of waiting requests, synchronizer A is said to be <u>more efficient</u> than synchronizer B if the time taken to determine the next request to be serviced is smaller for A than for B. This qualitative definition will suffice for the purposes of our discussion here.

Factors affecting efficiency are

1. Efficient evaluation of constraints

2. Efficient selection of operations.

Each of these factors is considered in greater detail below.

<u>Efficient Evaluation of Enabling Conditions</u> 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. Suppose

Enabling-condition(op) = cond1 & cond2 & ... & condn.

Then, only if cond1 or cond2 or.. condn is true need the enabling-condition be evaluated. Also, if one of cond1, cond2, ... condn is known to be false, then the enabling-condition need not be evaluated.

Following is a list of strategies which can be used in this regard.

<u>Strategy 1</u>: The enabling condition for a class of operations should be evaluated only if some minimum constraint is satisfied. A necessary servicing constraint for servicing an operation is that there be a request for it. Thus it is unnecessary to evaluate enabling conditions for a particular class of operations unless there are requests in that class.

Strategy 2: Information on the current state of the resource should be used to avoid unnecessary evaluations.

Strategy 3: Order the evaluation of the enabling conditions in such a way that those with less constraints are examined first.

Strategy 4: If sequence specifications imply a totally ordered execution of all operations, evaluation of the enabling conditions and the servicing of the operations can be ordered according to the sequence specification.

Strategy 5: Simplify all enabling conditions. Since individual constraints are derived from various specifications, it is possible for the same constraint to be derived from two different specifications. Hence a simplification may prove useful in reducing the number of terms in an enabling condition.

We can also use the "influence relationships" [7, 27] among the predicates to improve the efficiency of evaluation of the enabling conditions.

Efficient Selection of Operations Another strategy to improve the efficiency of resulting synchronizers is to construct them so that they select the next operation to be serviced with minimal computation. Since requests manifest themselves as elements in queues, this stipulation translates to an efficient choice of elements from a queue. The four selector functions

1. First-req(q)
2. Highest-priority-req(q)
3. First-enabled-req(q)
4. Highest-priority-enabled-req(q)

are listed in the order of increasing complexity (in the worst case), i.e., a selector function lower in the list takes more time to return the identity of the appropriate request than the one higher in the list. The synthesis algorithm should utilize all the information possible to choose a selector function which has higher efficiency. One such information concerns priority and is used by the queue allocator (ref. cases 2.1.1.1 and 2.1.2.1) In some cases, further information can be inferred from other specifications [25].

A note of caution is germane before we conclude our discussion on efficiency. Since evaluation of enabling conditions and subsequent servicing of requests affect the fairness of the synchronizer, decisions made for improving the efficiency should be consistent with the order of evaluation mandated by the fairness specification.

6.8. Correctness of the Synthesized Synchronizer

One of the attractions of the synthesis approach to program development stems from the claim made that an automatic programming system always produces a correct program. This is not an inherent feature of the synthesis approach. Correct code will result only if every decision and action of the synthesis algorithm is taken with the aim of producing code that conforms to a given set of specifications.

The synthesis algorithm proposed in this paper has been developed with a view to produce correct synchronizers, i.e., the code generated will preserve the specified properties of the shared resource and the operations on the resource. This is a result of the correctness of

1. the rules applied during the transformation of a given set of specifications into code for a synchronizer,
2. the substitution rules, and
3. the rule applied to order the servicing of operations according to specified fairness.

Formal proofs of correctness appear in [25].

7. SYNTHESIZING CODE FOR OTHER TARGETS

So far we have discussed an implementation of the synthesis algorithm for a specific target environment, one composed of AMPS and FGL. To achieve this implementation, we needed the information on

1. The operators available in FGL and their semantics,
2. Structure of synthesized code, and
3. The syntax requirements of FGL.

This is the information required for synthesizing code in any target.

In this implementation, the first three phases are target language independent. The mapping between primitives in the specification language and the primitives in the target language is embedded in the substitution phase. Since this phase is essentially a set of rewrite rules, recoding this phase for a different target is trivial once the synchronization primitives in the target language are known. Since efficiency considerations and syntax requirements of the AMPS-FGL environment are "hard-wired" into the code generator, this phase of the implementation has to be rewritten for other targets.

Now we focus attention on three different target synchronization mechanisms and discuss the salient features of synthesizing code for these mechanisms.

7.1. ADA

In ADA, tasks are the program units for concurrent programming. An entry within a task can be thought of as an operation on the resource that is encapsulated by the task. A call on an entry within a task can be executed only when there is a ready accept statement. This acceptance takes place when a rendezvous occurs. A rendezvous consists of executing statements between a DO and an END following the accept statement. (For details of the ADA tasking facility, see [11, 12].)

Because of the built-in mutual exclusion for performing a rendezvous, parallel execution of operations on a shared resource have to be engineered using "bracketing operations". Thus if operations in a class do not exclude each other, then the code for the class should be surrounded by calls to entries "start-operation" and "term-operation", and placed outside the task. Constraints on starting an operation will become constraints on the "start-operation" entry.

In ADA, every entry "E" has an FCFS queue associated with it and E'COUNT denotes the number of tasks waiting in the queue for "E". Therefore

E'COUNT=0 is equivalent to ~req$E.

Since all queues are FIFO, intra-class priority has to be handled in a cumbersome manner.

- If the priority of operations in a class is static and if the number of priority "levels" is finite and small, then separate entries for each level can be used in conjunction with the COUNT facility to realize intra-class priority.

- If the priority of the operations is dynamic, one plausible technique seems to be the following: Accept all requests, determine the priority of each request and "reenter" the request. Once this is done for all requests in the queue, the highest priority will be known. Again, the queue is scanned by accepting all requests. Every request except the one with the highest priority is reentered. In order to ensure termination, the queue has to be "frozen" before this procedure is attempted.

Section 11.2.10 of [12] has different strategies to process entry calls out of arrival order.

One major shortcoming of ADA from the point of view of fairness, is the possibility of SELECT making an unfair choice due to the nondeterminism assumed therein. However, one of the suggested implementations of the SELECT statement, namely the "order of arrival method" can guarantee fairness for entries which are enabled infinitely often.

7.2. Serializers

Serializers are similar to monitors with improvements intended to overcome some of the deficiencies observed in the latter. Two specific improvements are, 1) access operations on a shared resource are executed outside the synchronizer thereby permitting concurrent execution of the operations, and 2) the unstructured signalling mechanism of monitors is replaced by an automatic signalling mechanism. Details regarding serializers can be found in [1].

A serializer code consists of a database where the data required for synchronization is declared and one procedure for each type of access operation. In its most general form, a procedure enqueues a request into an appropriate simple or priority queue based on the conditions that exist when the request arrives. The request waits until a specified condition is satisfied and when that happens, the request <u>joins</u> a designated <u>crowd</u>, whence the operation is executed by another process. When the operation terminates, the requesting process acquires possession of the synchronizer, takes necessary actions and cedes control of the serializer.

Our synthesis algorithm is capable of deriving all the information required to construct each procedure in a serializer. Even in serializers, there is a possibility of unfair selection of requests; if more than one request (in different queues) satisfies the enabling conditions, the choice made by the serializer is not defined by the authors of the serializer. Thus a specified fairness may not be guaranteed by a synthesized serializer.

7.3. Monitors

Monitors [10, 3] are perhaps the first structured mechanisms developed for synchronization. As was mentioned earlier, they are plagued by numerous shortcomings [2, 18]. Like a serializer, a monitor consists of a set of procedures, one for each operation on a shared resource, and local data needed by those procedures. The problem of unfair scheduling pervades monitors too.

Means to synthesize monitors in spite of their two major limitations, are:

1) Since monitors assume mutual exclusion of the synchronized operations, as was done for ADA, bracketing operations have to be introduced.

2) In monitors, when conditions required to execute an operation do not hold, a request <u>waits</u> in a queue until the holding of the condition is <u>signalled</u> by another operation. Since signalling in monitors is done explicitly, the issue of consequence is: Where should a condition be signalled? Suppose for every action of the monitor, the set of conditions that may become true by that action, is compiled. For example, since a request ceases to exist once it is serviced,

```
    start(op) TRIGGERS ~req(op).
```

Thus, starting an operation in a class OPC may enable ~req$OPC. If after every action, such a condition is tested and found to be true, then its truth can be signalled, provided there are requests waiting for it to become true.

Once code is generated for a monitor, using the algorithm given in [10] it is possible to produce code in terms of lower-level synchronization primitives such as semaphores.

8. SUMMARY

Statements in a temporal logic based specification language are used to express the synchronization of concurrent processes. Such specifications are supplied to an automatic synthesis system which constructs code for the process which synchronizes the interactions of the concurrent processes.

The theory behind synthesis of synchronizers is based on the recognition of the following facts:

- Synchronization code is essentially a set of condition-action pairs; when certain conditions hold, certain actions are taken.

- Conditions are evaluated and actions taken according to a certain order; this order guarantees the fairness specified.

There are basically two types of actions a synchronizer is involved in:

1. Servicing a request for access to a shared resource.

2. Making changes to local variables.

These are indicative of the types of information that should be extracted from a set of specifications. Our synthesis algorithm derives the information needed to determine the condition-action pairs and orders them to satisfy the fairness.

- From the specifications of mutual exclusion, priority, resource state invariance, and sequencing, the constraints for starting an operation are determined.

- From the specifications of resource state changes and triggered conditions, the modifications to internal variables are derived.

- From the specifications of sequencing, priority and fairness, the basis for ordering the condition-action pairs is extracted.

This provides the infrastructure to generate synchronization code. In addition, heuristics is employed to improve the "efficiency" of the resulting synchronizer.

- From the knowledge of the primitives available in, and the syntax of the target language, synchronization code is constructed for a specific target.

We described the details of an implementation of the algorithm to automatically generate synchronization code. One of the motivations for this experiment has been to demonstrate the feasibility of the synthesis approach to constructing synchronizers and examine the practical issues involved. Listed below are the salient features of this work.

- The theory behind the algorithm is general enough to be applicable to synchronization mechanism that conform to the model assumed: monitors, serializers, and Ada tasks.

- The synchronizer produced by the algorithm is guaranteed to be correct.

- The basic algorithm is linear. Incorporation of efficiency improvement techniques increases its complexity.

- The algorithm is indeed practical, as evidenced by the implementation of the algorithm to generate synchronizers for AMPS in FGL.

It was stated in the concluding paragraphs of section 6.7, that since efficiency improvement techniques are essentially based on heuristics, any practical synthesis system ought to be able to "learn" in the process of synthesizing code. This would be possible if the synthesizer interacts with the user and let its strategies be known. Thus if the user determines that a decision of the synthesizer could lead to inefficient code, he could "direct" the synthesizer towards a better solution. For this to be feasible, the synthesis system should be interactive.

The fact that we are utilizing a theorem prover should not come as a surprise since programming as a task involves considerable logical inferencing and this is what a theorem prover is used for in this implementation. Since for our purposes, a logical simplifier will suffice, a general purpose theorem prover is not necessary. This implementation uses the simplifier part of the Stanford Pascal verifier for its purposes.

We have used this system to automatically generate synchronization code for a variety of synchronization problems found in literature. These include the disk scheduler problems [10] involving the elevator algorithm as well as the one that minimizes disk arm movement. The resulting code is executable on the AMPS machine. This has helped us evaluate the performance of the resulting code. For instance, it can be said that, in most cases, the synchronizer resulting from using a simplifier to perform logical inferencing and simplifications is more efficient than without it. Further implementations details are provided in [26].

Since our interest is in synthesizing synchronizers from the specifications, if the specification language is extended to allow other types of specifications, it should be brought about 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 synchronization code, needs to be further investigated. In practice, synchronizer 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 synthesis of exception handling code, are extensions that need to be incorporated in a fullfledged synthesis system.

In [25] we discuss certain types of "erroneous" specifications (for instance, inconsistent specifications) and techniques to detect their presence in a set of specifications. Such techniques could be implemented as a specifications preprocessor, so that synthesis is not undertaken unless the specifications are synthesizable.

I. An Example

Here we take the reader through all the phases of synthesizing code for the Resource Manager problem. The specification for this problem is given in section 3.2. To keep this presentation brief, only necessary details are presented.

The Translation Phase

The translation phase produces the following for each operation:

- The servicing constraints for the operations.
- The priority constraints for the operations.
- The resource state changes effected by the synchronizer when it services operations.

For Allocate operations:

```
RESOURCE STATE CHANGES:
       avail <- (avail - 1)

SA1) SERVICING CONSTRAINTS:
       (req$allocate &
        ~exec$free & ~exec$allocate &
        ((avail - 1) <= max) & ((avail - 1) => 0))

PA1) PRIORITY CONSTRAINTS:
       ∀f∈free ~enabled(f)
```

For Free operations:

```
RESOURCE STATE CHANGES:
        avail <- (avail + 1)

SF1) SERVICING CONSTRAINTS:
       (req$free &
        ~exec$free & ~exec$allocate &
        ((avail - 1) <= max) & ((avail - 1) => 0))
```

The Simplification Phase

```
SA2) SERVICING CONSTRAINTS FOR ALLOCATE OPERATIONS:
       (req$allocate & ~exec$free &
        ~exec$allocate &  (0 < avail))

PA2) PRIORITY CONSTRAINTS FOR ALLOCATE OPERATIONS:
       (~req$free V avail=max)

SF2) SERVICING CONSTRAINTS FOR FREE OPERATIONS:
       (req$free & ~exec$free &
        ~exec$allocate &  (avail < max))
```

-- notice that the preconditions are derived in a simplified
form (assuming that the invariance holds at the beginning of
an operation).

The Resource Allocation Phase

In this phase, the synchronizer variables, the counters and the queues are allocated. For each queue, the procedure that should be invoked to select the next request to be dequeued from that queue is also determined. Also, the minimum condition necessary for requests in a queue to be enabled is derived.

```
CONSTANTS: max = 2

SYNCHRONIZER VARIABLE:
  avail  initial value max
         minimum value 0
         maximum value max

COUNTERS:
        allocatecount for  allocate operations
        freecount for  free operations

QUEUES:
  A simple queue "allocateq" for  allocate requests
  Selector function: FIRST
  Minimum queue condition:
        (~exec$free) & (~exec$allocate) & req$allocate


  A simple queue "freeq" for  free requests
  Selector function: FIRST
  Minimum queue condition:
         (~exec$allocate) & (~exec$free) & req$free
```

The Substitution Phase

```
SA3) SERVICING CONSTRAINTS FOR ALLOCATE OPERATIONS:
     Status determining version:
       ((NONEMPTY(allocateq) AND
        (NOT (CTRVALUE(freecount) > 0)) AND
        (NOT (CTRVALUE(allocatecount) > 0)) AND
        (0 < RESVALUE(avail)))

     Wait until version:
       (SPAR
        (WAITQ (allocateq , 'NONEMPTY)),
        (WAITC (freecount , 0)),
        (WAITC (allocatecount , 0)),
        (WAITRES (avail , "GP" , 0)))

PA3) PRIORITY CONSTRAINTS FOR ALLOCATE OPERATIONS:
     Status determining version:
       (EMPTY(freeq) V avail=max)
     Wait until version:
       ARBIT(WAITQ(freeq, 'EMPTY),
             WAITRES(avail,"=",max)

SF3) SERVICING CONSTRAINTS FOR FREE OPERATIONS:
     Status determining version:
       ((NONEMPTY(freeq) AND
        (NOT (CTRVALUE(freecount) > 0)) AND
        (NOT (CTRVALUE(allocatecount) > 0)) AND
        (RESVALUE(avail) < max))

     Wait until version:
       (SPAR
        (WAITQ (freeq , 'NONEMPTY)),
        (WAITC (freecount , 0)),
        (WAITC (allocatecount , 0)),
        (WAITRES (avail , "LP" , 0)))
```

The Code Generation Phase

Except for some minor editorial changes, the following code was generated by the implemented synthesis system. To conserve space, only the code generated for the DATABASE, the ENQUEUER and the SYNCHRONIZER are shown; the names of the imported functions are not shown.

```
PROCEDURE resman-database
 COMMENT -- this  creates the queues, counters and the shared
  resource(s). Returns the funarg of the procedure that
  enqueues requests, and the  synchronizer of these requests;
IMPORTS(...)
LET
COMMENT -- each queue has a unique name and the following
                associated with it:
   1) name(s) of operation(s) that enqueue onto that queue
   2) wait until version of the minimum condition required
        for an operation in that queue to be serviceable
   3) non wait until version of (2);
allocateq BE   GQUEUE('allocateq,
                      CONS('allocate),
                      wuc_allocateq,
                      c_allocateq),
freeq     BE   GQUEUE('freeq,
                      CONS('free),
                      wuc_freeq,
                      c_freeq),
max       BE   2,
allocatecount BE GCOUNT(0),
freecount BE GCOUNT(0),
avail     BE GRES(max, 0, max),
dbm       BE synchronizer()
RESULT
COMMENT -- return a reference to the enqueuer.
PAR(CONS(enqueuer, dbm), dbm)

WHERE
```

```
PROCEDURE enqueuer(database, op, args)
COMMENT -- this procedure enqueues requests for access to the
        resman database ;
IMPORTS(...)
LET
COMMENT -- In this case, a token consists of the following:
        1) name of the operation
        2) apply(operation,args)
        3) parameters of the request
enq_allocate BE enq(allocateq,
                    CONS('allocate, APPLY( allocate, 'nil), nil)),
enq_free     BE enq(freeq,
                    CONS('free, APPLY(free, 'nil), nil))
RESULT
COMMENT -- enqueue the request onto
                        the appropriate queue;
IF op = 'allocate
 THEN <<enq_allocate>>
 ELSE  IF op = 'free
        THEN <<enq_free>>
        ELSE PRINT 'error
```

```
PROCEDURE synchronizer
COMMENT -- the synchronizer;
IMPORTS(...)
LET
max                BE 2,
a                  BE FIRST(allocateq),
pr_condition_f     BE ARBIT (WAITQ (freeq, 'EMPTY),
                            WAITRES (avail, ">=", max)),
wu_a               BE (SPAR
                       (WAITQ (allocateq, 'NONEMPTY)),
                       (WAITC (freecount, 0)),
                       (WAITC (allocatecount, 0)),
                       (WAITRES (avail, "GP", 0)))
                       AND  pr_condition_f,
newavail_a         BE RESVALUE(avail)-1,
f                  BE FIRST(freeq),
wu_f               BE (SPAR
                       (WAITQ (freeq, 'NONEMPTY)),
                       (WAITC (freecount, 0)),
                       (WAITC (allocatecount, 0)),
                       (WAITRES (avail, "LP", 0))),
newavail_f         BE RESVALUE(avail)+1,
choice             BE ARBIT(wu_f, wu_a)
RESULT
COMMENT -- wait for a request to be serviceable.
        modify the resource state as specified.
        service the request in detached mode;
IF choice
 THEN <<CHANGERES(avail, newavail_f),
        DET_EX(synchronizer, freeq, f, freecount)>>
 ELSE <<CHANGERES(avail, newavail_a),
        DET_EX(synchronizer, allocateq, a, allocatecount)>>
END
```

The following are the notable aspects of this code:

1. The constraints for free and allocate are evaluated "in place".

2. In the synchronizer, pr_condition_f is the simplified form of the constraint derived from priority specification.

3. The synchronizer avoids all busy waiting by using the "wait until" version of the constraints.

4. Changes to the resource state are effected before DET_EX is invoked.

5. Since the queue procedures are not invoked by the code, they are not shown here.

REFERENCES

1. Atkinson, R.R. and Hewitt, C.E. Specification and Proof Techniques for Serializers. IEEE Transactions on Software Engineering SE-5 (Jan 1979), 10-23.

2. Bloom, T. Evaluating Synchronization Mechanisms. Proc. Seventh Annual Symposium on Operating Systems Principles, Dec, 1979, pp. 24-32.

3. Brinch Hansen, P. Concurrent Programming Concepts. Computing Surveys 5 (Dec 1973), 223-245.

4. Dahl, O.J., Dijkstra, E.W. and Hoare C.A.R. Structured Programming. Academic Press, 1972.

5. Dijkstra, E.W. Guarded commands, non-determinacy, and a calculus for the derivation of programs. Communications of the ACM 18, 8 (August 1975), 453-457.

6. Elschlager, R. and Phillips. J. Automatic Programming. Tech. Rept. STAN-CS-79-758, Stanford University, 1979.

7. Ford, W.S. Implementation of a Generalized Critical Region Construct. IEEE Transactions on Software Engineering SE-4 (Nov 1978), 449-455.

8. German, S.M., Luckham, D.C., et al. Stanford Pascal Verifier User Manual. Stanford University, 1979.

9. Hearn, A.C. REDUCE 2 Users Manual. Tech. Rept. Utah Symbolic Computation Group UCP-19, University of Utah, 1973.

10. Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Communications of the ACM 17 (Oct 1974), 540-557.

11. Ichbiah J.D. et al. Preliminary ADA Reference Manual. SIGPLAN Notices 14 (June 1979).

12. Ichbiah J.D. et al. Rationale for the Design of the ADA Programming Language. SIGPLAN Notices 14 (June 1979).

13. Jayaraman, B. and Keller, R.M. Resource control in a demand-driven data-flow model. Proc. 1980 International Conference on Parallel Processing, August, 1980, pp. 118-127.

14. Keller, R.M. Sentinels: A Concept for Multiprocess Coordination. Tech. Rept. UUCS-78-104, University of Utah, June, 1978.

15. Keller, R.M. Semantics and applications of function graphs. Tech. Rept. UUCS-80-112, University of Utah, October, 1980.

16. Keller, R.M., Jayaraman,B., Rose, D., and Lindstrom, G. FGL (function graph language) programmers' guide. AMPS Technical Memorandum No. 1, University of Utah, July, 1980.

17. Keller, R.M., Lindstrom, G. and Patil, S. A Loosely-coupled Applicative Multi-processing System. AFIPS proc., June, 1979.

18. Lister, A. The Problem of Nested Monitor Calls. *Operating Systems Review* *13*, 2 (July 1977).

19. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

20. Marti, J.B. The META/REDUCE Translator Writing System. *SIGPLAN Notices* *13*, 10 (1978), 42-49.

21. Marti, J.B., et al. Standard LISP Report. *SIGPLAN Notices* *14*, 10 (1979), 48-68.

22. Pnueli, A. The Temporal Semantics of Concurrent Programs. In Khan, Ed., *Semantics of Concurrent Computation*, *Springer Lecture Notes in Computer Science*, Springer-verlag, 1979, pp. 1-20.

23. Ramamritham, K. and Keller, R.M. Specification and Synthesis of Synchronizers. Proc. 1980 International Conference on Parallel Processing, Aug, 1980, pp. 311-321.

24. Ramamritham, K. and Keller, R.M. On Synchronization and its Specification. Proc. CONPAR, June, 1981.

25. Ramamritham. K. *Specification and Synthesis of Synchronizers*. Ph.D. Th., The University of Utah, 1981.

26. Ramamritham, K. Automatic Synchronizer Synthesis System User's Manual. University of Utah, April, 1981.

27. Schmid, H.A. On the efficient Implementation of Conditional Critical Regions and the Construction of Monitors. *Acta Informatica* *6* (1976), 227-249.