On the Axiomatic Verification of Concurrent Algorithms

Christian Lengauer

Technical Report CSRG-94

August 1978

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the National Research Council of Canada.

Abstract

In recent years, axiom systems for the verification of concurrent algorithms have been developed. A set of concurrent processes is suitably represented by a concurrent statement, and variables shared between processes are represented by abstract data objects.

This thesis summarizes different formats of concurrent statements and shared abstract data types, for the representation of concurrency with varying degrees of granularity. Previously proposed axiomatic requirements on their semantics are interrelated, and new axioms, in particular for the manager data type for dynamic resource allocation, are formulated. Partial correctness as well as termination are the subject of investigation.

Strengths and weaknesses of the discussed language features and axiom systems are illustrated by comparing several implementations of solutions to a few concurrent programming problems, and their proofs. A number of basic principles facilitating the verification of concurrent algorithms are exemplified.

The study of axiomatic requirements on the semantics of concurrent programming features and of techniques for proving concurrent algorithms correct leads to more appropriate solutions and sometimes even to better specifications of programming problems with concurrency.

To my friends in the CSRG Zoo

Acknowledgements

I am indebted to Prof. E.C.R. Hehner for his critical interest, encouragement and liberal attitude in the supervision of this thesis and my graduate studies in general. It was a major factor in the successful completion.

I would also like to thank Prof. G.S. Graham for his prompt and thorough coreading.

Tom Rushworth enabled the presentation of this document by extending the TEXTURE text processing system.

Financial support for my studies towards a master degree was gratefully received from the German Academic Exchange Service.

Table of Contents

1 Introduction

With the rising number of software tools, the verification of concurrent algorithms has received a large amount of attention in recent years. Hoare was one of the first to point out the need for a formal proof technique particularly for concurrent algorithms [Hoa72a]. Time-dependent errors caused by the concurrent, (interleaving or parallel) execution of processes are an additional unreliability factor, which does not apply for sequential programs, and which is especially difficult to grasp.

Conceptually, there are two different approaches to verify concurrent systems, on the basis of [Flo67]:

a) The processes, the sequential units of the concurrent program, are first verified sequentially without consideration of their concurrent environment. Secondly, the assertions made in the sequential proofs at different points of process execution are shown to be preserved by every possible concurrent execution of the processes, a property which has been called non-interference [OwGr76a, OwGr76b].

   This approach emphasizes the system structure as a collection of sequential processes. Such a view of concurrent programs supports basic principles of software design: information hiding and modularity [Par72]. Because contemporary concurrent programming languages describe algorithms as a collection of processes, proofs can be carried out at the text level of the source program.

   Owicki and Lamport (see references in chapter 6) follow this approach. Owicki uses Hoare's method of outlining a proof by inserting assertions about the state of the program variables into the program text [Hoa69].

b) Rather than formulating assertions about the execution of sequential processes, the concurrent execution of the entire

program is described. First the program has to be transformed into a representation that suits verification, usually a flow graph model. One can view each node of the graph as some state of concurrent execution and each directed edge as a transformation of one state into another. (Keller introduces a bipartite graph instead [Kel76].) Assertions are associated with nodes; they describe the program state of every concurrent execution when it passes through that node, if it does so.

This technique does not have to contain a non-interference argument, because the execution of the entire program is described. (For multiprocessor systems interference problems may arise, though.)

In both approaches, following [Flo67], the constraints in a program specification are expressed by initial assertions and its goals by final assertions. Intermediate assertions associated with different states of execution of the program, regardless of whether they are attached to statements in the source text, to nodes in a flow graph, or to something else in a different model, will outline the transformation of the initial into the final assertion by every execution of the program. The state transformations are only valid under the assumption that the transforming actions terminate (so-called partial correctness). Termination is, in general, argued separately.

In this thesis, we shall only look at Owicki's approach, the process-by-process verification of concurrent algorithms by interleaved program assertions. The notation

"P" S "R"

expresses that, under the assumption P immediately before the execution of statement S, R will hold immediately after its termination. P is called the precondition (pre(S)), R the postcondition (post(S)) of S. The statement "P" S "R" is referred to as a proof or, if axiomatic, proof rule for S. (The

special character " may be interpreted as a comment delimiter. In the literature braces are used, {P} S {R}, but we need them for set denotation.)

Sometimes the change of a variable state by a statement is described explicitly by dummy values, such as

"x=c" S "x=f(c)"

Another notation is to express a variable's output state directly by its input state:

"pre(S)" S "x'=f(x)"

x' refers to x after the execution of S.

Abstracting completely from execution, S can be viewed as a predicate transformer mapping P onto R. Analogously, knowing S and R, one can derive the weakest precondition P=wp(S,R) such that "P" S "R" (see [Dij76]). In [Gri76], Gries points out the significance of pre- and postconditions for program specification and argues that deriving weakest preconditions from postconditions and statements aids in program development.

The tool for the solution of a programming problem is the set of language features, the programming language one decides to use. The basis for the verification of the program is a corresponding set of axiomatic proof rules for those language features. They define the semantics of the language, and the language implementor has to make sure that all axiomatic requirements are met. The user then may assume their validity without argument. An example for the axiomatic definition of a language is [Howi73], which defines Pascal. All popular sequential statements used in the following will comply to the semantics of Pascal.

The rule for the assignment statement is, for instance:

$$"P^{x}_{E}" \; x := E \; "P"$$

where P^{x}_{E} is the assertion formed by replacing every free

occurrence of x in P by E, for example

"x+1>y" x := x+1 "x>y"

In the case of the assignment statement termination is trivial: It may always be expected to terminate, presuming the expression E is legal. For other statements that perform iterations, for instance, termination is more complex. Frequently the partial correctness and termination behaviour of a language feature are axiomatized separately, but a single axiom may also comprise both aspects. We shall introduce a formalism later.

Important for the successive deduction of assertions between statements is the rule of consequence:

$$\frac{\text{"P'" S "R'"},\ P \Rightarrow P',\ R' \Rightarrow R}{\text{"P" S "R"}}$$

Both $\frac{a}{b}$ and $a \Rightarrow b$ mean a implies b. Whereas the implication denoted by the right arrow has to be proven, the horizontal bar expresses an axiomatic implication that holds by definition, if the presumption, the upper operand, is valid. (In the literature, for => often $\vdash$ or $\supset$ are used.)

The addition of concurrency to a language complicates verification. At this point, we shall not look at particular specifications of concurrency but only discuss the problem of verifying statements that, in some fashion, are specified to execute concurrently.

After the derivation of proof rules for each of the concurrent statements from assertions about their sequential contexts, non-interference must be proven additionally: For each statement, its execution may not affect any assertion made in the proofs of other statements that execute concurrently. More formally:

Let S and T be statements concurrent to each other, and let the proof for S be "P" S "R". Then T does not interfere with the proof of S iff

i) "R∧pre(T)" T "R"

ii) For any statement S' contained in S

"pre(S')∧pre(T)" T "pre(S')"

A set of statements concurrent to each other is called interference-free iff no elementary action in any of them interferes with the proof of any other statement in the set.

Elementary actions are those that can safely be treated as indivisible.

Interference must only be considered in the case of concurrency. To avoid interference problems in particular program parts, one may want to define statements as elementary. The notation is

[S];

where S is the statement to be elementary. For intermediate assertions in S, non-interference is an axiomatic requirement and need not be proven.

Elementary actions are often less appropriately called indivisible or atomic. The requirement that achieves non-interference is that, with respect to accesses to shared variables, the elementary statement acts like an indivisible statement. However, it need not be indivisible regarding accesses to other data.

To interrelate the progress of concurrent processes, one often has to enrich the algorithm by additional variables, Owicki calls them auxiliary variables. They keep track of the processes' sequential execution histories. The treatment of auxiliary variables may not affect the flow of control or data in the rest of the program. More formally:

> x is called an auxiliary variable iff x appears in the program only in assignment statements of the form x:=E, where the expression E may contain any auxiliary or program variable.

An auxiliary variable axiom states that the proof of an algorithm with auxiliary variables also holds for its implementation without auxiliary variables [OwGr76a, OwGr76b].

The following two chapters build the framework for this thesis with a discussion of recent approaches to the specification of concurrency and rules for its verification.

Chapter 2 deals with shared abstract data types that provide different grains of concurrency for the access to their data. Shared abstract data types are an important aid in the communication of their concurrent accessors. The types investigated here are monitors [Hoa74], managers [SKB77], classes synchronized by path expressions [FlHa76], and general shared classes [Owi77b]. Proof rules for partial correctness are given, if previously developed. Termination of calls to shared abstract data objects is discussed thoroughly in chapter 4.

Chapter 3 is concerned with the specification and verification of concurrent processes. A representation suitable for verification, the concurrent statement, is motivated and defined in different forms. Proof rules for partial correctness and termination are given.

Chapter 4 is the core of the thesis. The tools previously described are compared in two applications: a concurrent system synchronized by a semaphore and a system using a pool of resources. After correlating different verification techniques and motivating their necessity, new proof rules are developed: partial correctness axioms for the abstract data type manager and termination axioms for calls to shared abstract data objects, in general.

The thesis ends with some conclusions about the usefulness of the exhibited concepts and with suggestions for further research.

2 Shared Abstract Data Types

The concept of abstract data types, which first attracted attention in SIMULA [DaHo72], has in recent years received much consideration, due to its emphasis on modularity and resource protection. It is a very important structuring factor in programming, especially operating system design.

An abstract data type defines a data structure by a set of variable declarations, and all operations on them by a set of procedures. An initial statement determines the state of the data before their use. A data object of some abstract data type comprises an incarnation of the variables declared in the data type and access rights to it via the operations defined in the data type, by procedure calls of the form

```
objectname.procname(parameters);
```

The data are protected from any access of a different kind, e.g., direct assignment.

Two of the many advantages of this concept become particularly evident in this thesis:

a) Information Hiding

A programmer using an object of some abstract data type does not have to worry about implementation details concerning its access. Only the semantics of the data type operations have to be understood.

Following [Hoa72b], we shall distinguish the specifications A of some abstract data type, which state the semantic properties, and its concrete implementation C, which achieves these properties.

The specifications contain the following statements:

Req(A): states presumptions for parameters of the abstract data type,

Init(A): states the initial properties of the data,

I(A): states an invariant assertion that characterizes the data between accesses,

Operations: ...

op_i(parameters)

pre: states the presumptions for correct functioning of op_i,

post: states the result of op_i, assuming op_i.pre is fulfilled at its execution,

...

[Liz75] discusses several specification techniques for data abstractions.

To verify the correct implementation of an abstract data type, a mapping will often be needed that associates the abstract object *A*, characterized by the specifications, with the concrete object *C*, characterized by the implementation. We call this mapping the abstraction function

$$A = F(C)$$

For the correctness of the concrete object then the following have to hold:

1.) I(C) => I(A)

2.) "Req(A)" initial statement "Init(A)∧I(C)"

3.) For each operation op(var $\bar{x}$; $\bar{y}$)

"op.pre($\bar{x}$, $\bar{y}$, A)∧I(C)"
body of procedure op
"op.post($\bar{x}$, $\bar{y}$, A)∧I(C)"

where $\bar{x}$ is a vector of variable parameters,
and $\bar{y}$ is a vector of constant (value) parameters.

In the case that operations are allowed to execute concurrently on the same data object X, the call of an operation op also needs special proof rules, since the expected rule

$$\text{"P"}\ X.op(\bar{a},\bar{e})\ \text{"R"}$$

where

$$P \equiv X.op.pre^{\bar{x}\ \bar{y}\ caller}_{\bar{a}\ \bar{e}\ \ i}$$

$$R \equiv X.op.post^{\bar{x}\ \bar{y}\ caller}_{\bar{a}\ \bar{e}\ \ i}$$

may not be interference-free. To prove non-interference, one has to take the environment of the caller into account. Owicki suggests an "adaption" axiom [Owi77a, Owi77b]:

$$\frac{\text{"P"}\ X.op(\bar{a},\bar{e})\ \text{"R"}}{\text{"}\forall_{\bar{k}}(P \wedge \bigwedge_{\bar{a},\bar{z}(caller)} R \Rightarrow Q)\text{"}\ X.op(\bar{a},\bar{e})\ \text{"Q"}}$$

where $\bar{a}$ comprises the actual variable parameters,
$\bar{e}$ comprises the actual value parameters,
$\bar{k}$ comprises the variables free in P and R, but not Q, $\bar{a}$, and $\bar{e}$,
P, R as above,
and $\bar{z}(caller)$ is the list of variables changed by op.

We only state this result for the sake of completeness and shall not elaborate on it any further.

b) Stepwise Verification

The abstract data type supports a bottom-up verification of the program.

If one postulates that any two abstract data objects may not access each other in turn, directly or indirectly (forward-referencing), abstract data objects can be organized

in an access hierarchy of the following form [Len77]:

1. Level 0 is the set of data objects that do not access any other data object.

2. Level i contains all abstract data objects X with the properties:
   i) there exists a data object on level i-1 which is accessed by X,
   ii) all data objects accessed by X are on levels j, such that $j<i$.

This hierarchy can be imposed on the set of abstract data types, because all objects of the same type must belong to the same hierarchy level. The significance of hierarchies for program structuring has been recognized in [Par74].

One can prove the correctness of an access hierarchy by successive verification of its levels 0, 1, 2, etc. In particular, changes in level i can affect the correctness of only levels j such that $j \geq i$; in other words: once verified, the core of the hierarchy will be correct, in whatever environment it is implemented.

We will be looking at shared data types, i.e., data types whose objects may be required for access by several processes, possibly concurrently. The main difference from private data types is that the consistency of the shared data, expressed by the invariant assertion for the data type, has to be maintained by a proper synchronization of the accesses to them. The distinguished shared data types we shall look at differ mainly in their means of synchronizing accesses.

The verification of shared abstract data types has to contain an argument for proper synchronization as well as non-interference. Note that an operation may also interfere with itself, when several processes execute it concurrently.

Synchronizing accesses to shared data objects creates the possibility of deadlock in access hierarchies for some types. This matter will be mentioned but not covered by axioms. For proof rules, we assume that a shared data object does not access any other shared data object in its operations.

2.1 Monitors

The shared abstract data type that has first been formulated and implemented and that is today most popular is the monitor [Hoa74]. Its synchronization scheme has two levels:

i) short-term scheduling

ensures that the monitor procedures are executed mutually exclusive to each other. It is part of the language and not influenced by the programmer.

ii) medium-term scheduling

is the synchronization of processes in accordance with the state of the monitor data. It has to be defined by the designer of the monitor. The tool provided by the language is a condition data type. Variables of type condition may only be declared local to a monitor object and represent waiting queues of accessors of that object, with no further specified scheduling policy. They are initially empty and manipulated by way of two standard procedures:

If cond is a condition queue,

cond.wait; suspends the caller, adds it to the queue cond, and releases the monitor for further access,

cond.signal; if any process is waiting in cond, suspends the caller and grants access to the process in cond with highest priority. We shall only axiomatize and discuss this signalling algorithm. For different policies see [How76b].

The use of scheduling operations during a monitor access introduces the problem of deadlock in an access hierarchy of monitors. Say, monitor A accesses monitor B, and processes do not access B directly, but only through A. If then some process is being delayed inside a procedure of A because of medium-term scheduling in B, it blocks access to A, which disables others to enter B and to establish the resumption condition. The process is blocked indefinitely, and A as well as B are inaccessible. An example for such a case in a resource management system is sketched in [SKB77]; another in a message switching system is described in [Len77].

As a consequence, monitor hierarchies are not desirable. But there are other shared data types that allow at least a restricted use of access hierarchies.

To verify synchronization in monitors, we define the following partial correctness axioms for wait and signal from [Owi77a]:

"P∧I(C)" cond.wait "P∧I(C)∧B(cond)"
"P∧I(C)∧B(cond)" cond.signal "P∧I(C)"

Here, B(cond) represents the resumption condition for processes that are waiting in cond, and P contains only parameters, local variables, or constants free. C names the implementation of the monitor that contains cond. The appearance of I(C) in the rules for wait and signal indicates that the scheduling operations do not affect shared data; the appearance of P says that they do not affect local data. Only the condition queue is involved. The invariant has to be valid at times of scheduling, because it

will hold whenever the accessor of the monitor is switched.

Several different suggestions have been made for monitor proof rules. Others will be outlined and discussed in chapter 4.

2.2 Managers

The concept of a manager has recently been proposed in [SKB77]. A manager is a shared abstract data type for the dynamic allocation of objects of some other (private or shared) abstract data type to processes. It is not to be confused with the special processes that are called managers in [JaSt77].

The manager has exactly the same synchronization scheme as the monitor, but does not completely protect all of the data specified in it. One declaration in the manager is the pool of allocatable resources, e.g.,

```
var pool: array indexset of resource;
```

The data type of the allocatable objects is determined by the header of the manager definition:

```
type managername = manager of capability: resource;
```

Processes define an access right to the resource pool by declaring

```
var object: resource from managername;
```

Access rights to the manager itself are never specified.

The access right to a resource object established by the above declaration is not fixed for the entire execution time, as it would be if no manager were involved, but only temporarily. The manager distributes the temporary access rights (so-called capabilities) among processes that require them, and this may involve synchronization. For an easier implementation and

verification, the restriction is made that every process may be granted at most one access right by a manager object at a time; in other words, no process may hold more than one object from a fixed resource pool. An implicit parameter declared in the header of the manager definition, here with name capability, holds the current access right (if any) of the process that is preforming the manager call.

The binding of capabilities to processes is performed by two standard procedures which are used in manager operations:

bind(r); grants the caller a capability to resource object r,
release; withdraws the caller's current capability.

The manager resolves the deadlock problem with access hierarchies for two-level hierarchies, as described in [SKB77]: Replace the monitors in the higher level, that are in danger of blocking, by a manager. In a hierarchy that is not a tree (where the lower level means the leaves), the replacement may, however, lead to some loss of modular structure. Multi-level hierarchies remain problematic, but are probably rare in applications.

For the verification of managers, we need additional axioms for the binding operations. They have not been proposed in previous publications and will be formulated in chapter 4, together with a detailed description of the semantics of bind and release.

2.3 Path Expression Classes

[CaHa74, FlHa76] suggest defining the concurrency permitted among operations on an abstract data object by a single expression associated with the data type: its path expression. The operands in the path expression are the data type's procedures, and several operators relate their executions to

each other:

i) sequence: op1; op2

op1 and op2 are executed consecutively, op1 only before op2, op2 only after op1.

ii) selection: op1, op2

Fither op1 or op2 is executed, but not neither nor both.

iii) repetition: $(op)^n$

op is executed repeatedly, n times in sequence. If n is omitted, op is executed indefinitely.

iv) simultaneous execution: {op}

op may be executed concurrently to itself until, at some time, no process is executing op. Then no subsequent call to op will be executed.

More complex is the

v) restricted repetitive selection: $(op1 \pm op2)^n$

A sequence of calls to op1 or op2 is executed, preserving the assertion

$$0 \leq \#(op1) \pm \#(op2) \leq n$$

#(op) is the number of times op has been executed on the data object since its initialization.

The path expression is in the implementation of an abstract data type denoted by

path expression end;

The expression may be evaluated repeatedly, i.e., path...end has the same semantics as (...). As an example, the expression

path {read}, write end;

denotes the synchronization of concurrent readers and writers on

shared data: Readers may access the data concurrently as long as there is no writing access; a writer needs exclusive access.

Path expressions are a very high-level and somewhat restricted synchronization concept. Synchronization does not take place inside the operations, but rather the calls to data type procedures are synchronized, much in the spirit of the with-when statement for the specification of conditional critical sections [Hoa72a]. This may cause a loss of scheduling flexibility.

On the other hand, path expressions merge the definition of short-term and medium-term scheduling, leaving both to the programmer, and thereby introduce a more general short-term scheduling scheme than is provided by monitors and managers. Operations are not in any case mutually exclusive, but may be executed concurrently as long as the consistency of the data is preserved. However, because the calls of operations are subject to synchronization, either an entire operation op1 is mutually exclusive with an entire operation op2, or both are entirely concurrent to each other.

The feature of optional concurrency may help to avoid deadlocked access hierarchies. But cases such as the one described in section 2.1 can still be unpreventable. The major merit of path expressions is the useful combination of efficient concurrent execution and structured specification of synchronization.

In proofs, for every statement in some class procedure, non-interference with the procedures concurrent to it has to be shown. The pre- and post-conditions of any procedure need not be interference-free in the proof of the abstract data type, but the according assertions for the call of the procedure must be interference-free in the calling process (see the introduction to this chapter). As another consequence of concurrency, an additional requirement for the sequential correctness of not

completely exclusive operations is, that the class invariant must be valid between any two elementary actions of the procedure. More formally:

> Let op be a procedure of class C that is concurrent to some operation of C. Then for all elementary actions A in op,
>
> $$pre(A) \Rightarrow I(C)$$
>
> has to hold.

2.4 General Shared Classes

[Owi77b] defines the most general form of shared abstract data types, the general shared class.

Synchronization takes place inside the class operations, and any statement in some operation may be made mutually exclusive or concurrent with any other statement or itself. As a synchronization tool, condition queues are conceivable, but Owicki chooses an in terms of scheduling decisions slightly higher-level concept, the semaphore. As with path expressions, both short- and medium-term scheduling are the responsibility of the class programmer.

Proof rules are needed for the semaphore operations. [Owi77b] does not state any, we will discuss them in chapter 4.

To simplify verification, Owicki requires the body of class procedures to have the format

begin declarations; enter; operate; exit end;

where

i) enter and exit are elementary actions or null statements, and operate is composed of elementary actions,

ii) the variables accessed by enter and exit are called control variables; those accessed by operate, data variables. A variable local to a class may not be both control and data variable.

The effect of the above structure for class procedures is a separation of the variables affected by the operation into control variables that guide synchronization, and data variables that define the state of the class object as it is perceived by the caller. Consequently, control variables should not appear in pre- and postconditions for the class operations and the initial statement, but only in the class invariant. Since enter and exit can be null, this postulation is not really a restriction. It indicates, however, that a synchronization scheme that frames the operations (like the path expression) is in general easier to verify.

The proof rules for shared classes are the same as for path expression classes.

3 The Concurrent Statement

In the last decade, several representations of concurrency have been suggested [Bri73]. Restricting ourselves to a structured syntax merely leaves us with two choices:

a) Process Modules

Processes are represented as self-contained modules comprising data and actions, much in the sense of abstract data types. Execution of a process object may be caused by a special start statement.

This approach is a structured formulation of the fork statement from [Con63] and is implemented in Concurrent Pascal [Bri75]. Concurrent execution of several processes is successively initiated by forking the path of control via start statements on different process objects into several parallel paths.

The structured concurrent extension to PL/1, CSP/k [HGLS78], works in a similar manner. Here, all processes are implicitly started after program initialization.

b) The Concurrent Statement

[Con63] also introduces a join statement, dual to fork, which relates the time of termination of some process to the state of execution of other concurrent processes. With fork only, one cannot determine when a process has terminated. The structured notation is

cobegin S1//S2//...//Sn coend;

where cobegin refers to a "multi"-fork and coend to a "multi"-join of n processes Si. Concurrent execution of the Si begins at the point of cobegin and ends at coend, when all Si have terminated. Nesting of concurrent statements is legal and in particular useful for statements about the termination of a subset of m concurrent processes, m<n.

For verification, the concurrent statement is the more useful choice, because it permits the formulation of an assertion about the program state after the termination of some concurrent process, in the form of a postcondition for a concurrent statement. Without a join of parallel execution paths, we would not know at which point in the program that assertion holds.

Three formats of concurrent statements appear in the literature:

a) The general concurrent statement [OwGr76a]

   does not protect shared variables by a special data structure. Its synchronization primitive is the await statement.

b) The concurrent statement with resource [Hoa72a, OwGr76b]

   specifies shared data in a resource declaration which protects them automatically from accesses that cause inconsistency, but does not associate the data with the operations on them. Its primitive for synchronization and access of shared data is the with-when statement.

c) The concurrent statement with abstract data types [Owi77a]

   provides full protection and, moreover, hides matters of synchronization in the abstract data types.

We will not look closer at the concurrent statement with resource. It is merely a predecessor of the concurrent statement with abstract data types and has the same capabilities in expressing concurrency, if in the latter abstract data types are used that do not permit concurrent execution of their operations (monitors or managers).

The general concurrent statement is more powerful. Shared data may be accessed in non-elementary actions as in the concurrent statement with abstract data types which permit concurrent execution of their operations (path expression classes or general shared classes).

3.1 The General Concurrent Statement

As described before, the general concurrent statement has the form:

cobegin S1//S2//.../Sn coend;

Variables declared outside the statement are global to all Si, those declared inside are local to the process that contains the declaration. For simplicity, the language has no procedures. Execution of the concurrent statement terminates when all processes Si have terminated.

We need not require any assumptions about the indivisibility of statements in the Si if the following convention is obeyed:

> Each expression E may refer to at most one variable y which can be changed by another process during the evaluation of E, and E may refer to y at most once. The same restriction is required for assignment statements x:=E.

Then, only memory reference has to be indivisible. Note that assignments of the form

x := f(x);

refer to x several times as well as assignments to data structures, such as

A := B;

for arrays A and B. Such statements do not follow the previous convention and have to be specified elementary if they are desired as such:

[A := B];

The await statement provides the synchronization tool for the concurrent access of shared variables. The general format is

await B [S];

where [S] denotes an elementary statement that is executed only if B is true. If B is false, the process executing await is delayed and continues with [S] when B becomes true. It is assumed that await performs fair scheduling; more specifically, the process executing await is delayed only until B becomes true. S may not contain a concurrent statement or another await, to avoid deadlock problems caused by nested awaits (as with hierarchies of shared abstract data types).

Owicki's proof rules for the general concurrent statement are

$$\frac{\bigwedge_{i=1}^{n} \text{"Pi" Si "Ri" interference-free}}{\text{"}\bigwedge_{i=1}^{n} \text{Pi"}\ \underline{\text{cobegin}}\ \text{S1//.../Sn}\ \underline{\text{coend}}\ \text{"}\bigwedge_{i=1}^{n} \text{Ri"}}$$

$$\text{and}\quad \frac{\text{"P}\wedge\text{B" S "R"}}{\text{"P"}\ \underline{\text{await}}\ \text{B [S] "R"}}$$

where the sequential proofs for the processes, "Pi" Si "Ri", are interference-free iff for all i every statement in Si does not interfere with the assertions in the proofs of all Sj, j≠i. (For the definition of "does not interfere with" see chapter 1.) [OwGr76a] points out that one can reduce the non-interference test to await statements and assignment statements outside awaits. All other statements are elementary and therefore trivially interference-free.

3.2 The Concurrent Statement with Abstract Data Types

For many applications, especially in operating systems, it will not be necessary to manipulate data shared by concurrent processes before their execution. In this case, the effect of the concurrent statement is clarified if all shared data are defined as m abstract data objects associated with it:

shared C1:T1,...,Cm:Tm cobegin S1//.../Sn coend;

The first action of the concurrent statement is to initialize all shared objects and thereby create the initial environment for all processes. Since the synchronization is part of the abstract data type operations, no specific synchronization primitive need be defined for this form of concurrent statement.

Let us make the restriction that the concurrent statements we look at may have only the following combinations of shared abstract data types:

a) monitors or managers
b) path expression classes or general shared classes

This is only for demonstration purposes. We shall demonstrate that exclusive shared abstract data types are easier to handle than those that permit concurrency.

A more severe restriction is that we do not allow hierarchies of shared data types. With the exception of the resource type in managers, no access right may exist between different shared data objects. Therefore the proof rules introduced in chapter 2 suffice.

Owicki defines the partial correctness axiom for the concurrent statement with abstract data types, according to the previous semantic description:

$$\frac{\bigwedge_{i=1}^{n} \text{"}Pi\text{"}\ Si\ \text{"}Ri\text{"}\ \text{interference-free}}{\text{"}\left(\bigwedge_{j=1}^{m} Init(Cj)\right) \Rightarrow \left(\bigwedge_{i=1}^{n} Pi\right)\text{"}}$$

shared C1:T1,...,Cm:Tm cobegin S1//...//Sn coend;

$$\text{"}\left(\bigwedge_{j=1}^{m} I(Cj)\right) \wedge \left(\bigwedge_{i=1}^{n} Ri\right)\text{"}$$

Here, non-interference means

i) for concurrent statements with monitors or managers:

For shared abstract data types with only mutually exclusive operations, the requirement is that the Pi and Ri are safe,

i.e., do not refer to variables that are changed in several processes (in analogy to the concurrent statement with resource). Then, interference in the original sense (see chapter 1) cannot occur, because such variables are only manipulated in elementary actions.

ii) for concurrent statements with path expression classes or general shared classes:

For shared abstract data types that allow concurrent execution of their operations, the usual non-interference is required (in analogy to the general concurrent statement). However, the only statements that can interfere are calls to operations of shared abstract data types; other statements cannot access shared data.

We emphasize again that the above semantics define a quite restrictive concurrent statement, in the sense that no computation carried out before its execution has any relevance for it. More relaxed semantics as expressed by the axiom

$$\frac{\bigwedge_{i=1}^{n} \text{"Pi" Si "Ri" interference-free}}{\text{"}\left(\bigwedge_{j=1}^{m} I(Cj)\right) \wedge \left(\bigwedge_{i=1}^{n} Pi\right)\text{"}}$$

shared C1:T1,...,Cm:Tm cobegin S1//...//Sn coend;

$$\text{"}\left(\bigwedge_{j=1}^{m} I(Cj)\right) \wedge \left(\bigwedge_{i=1}^{n} Ri\right)\text{"}$$

are conceivable and will often be necessary in applications, where a concurrent computation is merely part of a larger sequential context. In this case, the shared data objects belong to the environment surrounding the concurrent statement rather than the concurrent statement itself, and they have to be initialized and may be manipulated before its execution.

This thesis will only use the former semantics. We are interested in the mere structure of concurrent computations.

3.3 Termination of Concurrent Statements

Proof rules given so far in this thesis only refer to partial correctness. For the termination argument, a set of termination axioms has to be defined, or the partial correctness axioms have to be extended to cover termination.

The termination rules described here are taken from [OwGr76a]. After the proof of sequential termination of the processes specified in the concurrent statement, it has to be shown that termination is not affected by their concurrent execution. Therefore, the definition of non-interference (see chapter 1) has to be extended by non-interference with termination.

A common practice to argue termination is to define a function on a well-founded set, a set that is ordered in such a way that no infinite decreasing sequences of elements exist [MaWa78], for instance the positive integers, and to show that the algorithm decreases this function successively. Then it must reach a lower bound in a finite time and terminate. Hence, the additional requirement for a statement T not to interfere with the proof of a concurrent statement S can be formulated:

iii) Given a proof for S that uses a termination function t, T does not interfere with the termination proof of S iff

$$\text{"}t=c \wedge pre(T)\text{"}\ T\ \text{"}t \leq c\text{"}$$

c is a dummy value as described in chapter 1.

A set of statements concurrent to each other is then called interference-free iff no elementary action in any of them interferes with the partial correctness or termination proof of any other statement in the set.

If the above non-interference is established, processes may still be held up by indefinite blocking caused through synchronization. Proof rules dealing with this problem are more

complex and, of course, depend on the synchronization scheme used. For the await in general concurrent statements, [OwGr76a] proves the following criterion:

> Let S be a statement whose partial correctness rule is "P" S "R". Let the awaits of S that are not contained in concurrent statements of S be
>
> A_j : await B_j [...]
>
> Let the concurrent statements of S which are not part of other concurrent statements of S be
>
> T_k : cobegin S_1^k //...// S_n^k coend;
>
> Define
>
> $$D(S) \equiv [\vee_j (pre(A_j) \wedge \neg B_j)] \vee [\vee_k (D_1(T_k))]$$
>
> $$D_1(T_k) \equiv [\wedge_i (post(S_i^k) \vee D(S_i^k))] \wedge [\vee_i D(S_i^k)]$$
>
> Then $\neg D(S)$ implies that no execution of S can be blocked indefinitely.

The proof uses induction on the nesting level of concurrent statements. D(S) is a recursive characterization of blocking. If the first term of D(S) is true, S is held up at some statement A_j. $D_1(T_k)$ says that some process in the concurrent statement T_k is blocked despite the fact that all others have terminated.

In the case of the concurrent statement with abstract data types, absence of indefinite blocking means that all calls to shared abstract data type operations terminate. This matter will be discussed in chapter 4.

The combined partial correctness and termination axiom for any format of the concurrent statement is

$$\frac{\bigwedge_{i=1}^{n} \text{"Pi" Si "Ri" interference-free} \;\wedge\; \bigwedge_{i=1}^{n} \text{Si cannot be blocked indefinitely}}{\text{"P" concurrent statement of S1,...,Sn "R"}}$$

where non-interference and P and R are defined for the different formats as in the sections 3.1, 3.2.

4 The Feasibility of Verification Concepts

We shall now try to evaluate the significance and suitability of implementation and verification concepts, a part of which has been introduced in the previous chapters. This is the main chapter of the thesis, and it may be understood as a collection of (hopefully) clarifying comments and new contributions concerning the verification of concurrent algorithms.

The selected applications are popular and widely-used in concurrent programming. If they appear a little arbitrary, keep in mind that their only purpose is to illustrate verification concepts. We do not attempt to cover the most important algorithms for concurrent programming.

4.1 Arguing Optimal Scheduling of Processes Synchronized by Semaphores

This section discusses the structure of arguments for efficient scheduling in a parallel programming environment. As example serves the proof of the property of a semaphore administering a section of critical code, to grant entry whenever the critical section is executed by less than a constant number, m, of processes. Whereas the former part of the requirement (to grant entry) characterizes the semaphore's scheduling task, the latter ("whenever...") is an efficiency constraint.

We shall study several semaphore implementations that have been published, and investigate the requirements that are necessary for the verification of the above scheduling property.

The first implementation and its proof is taken from [OwGr76a]. It uses the general concurrent statement.

```
s := m;
cobegin S1//S2//...//Sn coend;
```

where each Si has the form

```
while true do
  begin
     non-critical section;
     P(s);
     critical section;
     V(s);
     non-critical section;
  end;
```

Moreover, the Si handle s like a semaphore, i.e., do not access s except in calls to P and V where

$$P(s) \;\hat{=}\; \underline{await}\ s>0\ [s := s-1]$$
$$V(s) \;\hat{=}\; [s := s+1]$$

Here, $\hat{=}$ means that the left operand is a substitution for the right operand. (Remember, the language does not contain procedures.)

To prove the scheduling property, we relate the execution of the different processes to each other by auxiliary variables. For every process Si, the binary auxiliary variable INC[i] indicates whether Si is currently executing the critical section protected by the semaphore (INC[i]=1) or not (INC[i]=0). The scheduling property of the semaphore is then expressed by the invariant

$$I \;\hat{=}\; (0 \le s = m - \sum_{i=1}^{n} INC[i] \;\wedge\; \bigwedge_{i=1}^{n} INC[i] \in \{0,1\})$$

That is, the semaphore value must represent exactly the number of processes that may still enter the critical section.

Presuming that the semaphore and the auxiliary variables are only referred to in places explicitly shown, the proof outline is:

```
s := m;
INC[1],INC[2],...,INC[n] := 0,0,...,0;
"I ∧ ∧(i=1 to n) INC[i]=0"
cobegin S1//S2//...//Sn coend;
"false"
```

where for every Si

```
"I ∧ INC[i]=0"
while true do
  begin
    "I ∧ INC[i]=0"
    non-critical section;
    "I ∧ INC[i]=0"
    await s>0 [s := s-1; INC[i] := 1];
    "I ∧ INC[i]=1"
    critical section;
    "I ∧ INC[i]=1"
    [s := s+1; INC[i] := 0];
    "I ∧ INC[i]=0"
    non-critical section;
    "I ∧ INC[i]=0"
  end;
"false"
```

The postcondition is false to indicate that the process does not terminate execution.

In the proof of Si, every assertion is interference-free, since all processes keep I invariant throughout their execution, and, of the auxiliary variables, Si changes only INC[i] which is not changed by any other process.

The invariant also yields the efficiency constraint on the semaphore: The short argument that requests are granted as soon as less than m processes execute the critical section is given in [OwGr76a] (more exactly, a blocked process implies that m

processes are executing the critical section).

For the multiple assignment statement to the auxiliary variables, we use the simple axiom

$$"p^{x1,...,xn}_{c1,...,cn}"\ x1,...,xn := c1,...,cn\ "p"$$

where the c1,...,cn are constants. For a more detailed discussion of multiple assignment axioms see [Gri78].

Let us briefly review the strategy of the outlined proof: The desired property follows from a lemma based on a cleverly chosen invariant. The invariant comprises the behaviour of all processes that access the semaphore, by means of auxiliary variables that each describe the history of one process. (Some people call them history variables [Bri73, How76a].) Each auxiliary variable could have been defined locally to its process, rather than joining them all in a global array. Then the precondition for each process Si would be I, and INCi=0, where INCi is the auxiliary variable of Si, would be established before the <u>while</u> loop. The present version has been chosen for the sake of analogy with the following example.

The structure of the invariant depends on the scheduling property to be proven, and finding it may require a good deal of intuition. Owicki makes the point that more sophisticated tools are required to verify optimal scheduling in a uniform way, as opposed to termination or the absence of deadlock.

It is important to note that the optimal scheduling argument is based on the assumption that Owicki's synchronization scheme, the <u>await</u> statement, performs optimal scheduling, i.e., delays the executing process only until its resumption condition is satisfied. This presumption is a hidden requirement that is not reflected in the proof rule for <u>await</u>. The <u>await</u> statement is a very high-level synchronization scheme. In the following implementations we will use a lower-level scheme that can sufficiently be described by axiomatic pre- and postconditions.

Rather than defining the semaphore as a global variable and giving semantic restrictions for its use, one should incorporate it as an abstract data type, as monitor, for instance. Different monitor implementations of the semaphore appear in the literature (for example [Hoa74, How76a]). We define:

```
type semaphore = monitor(m: integer);
begin
   var s: integer;
       q: condition;

   proc P;
   begin
      if s=0 then q.wait;
      s := s-1;
   end P;

   proc V;
   begin
      s := s+1;
      q.signal;
   end V;

   begin s := m end;
end semaphore;

shared sem: semaphore cobegin S1//...//Sn coend;
```

where the Si have the form

```
while true do
  begin
     non-critical section;
     sem.P;
     critical section;
     sem.V;
     non-critical section;
  end;
```

Having in mind to carry through exactly the same proof as in the previous program, let us specify the properties of the semaphore monitor. Following the proof rule for the concurrent statement with monitors (chapter 3.2), we want:

$$\text{semaphore.Init} \Rightarrow (s=m \wedge \bigwedge_{i=1}^{n} INC[i]=0)$$

$$\text{semaphore.I} \Leftrightarrow (0 \le s=m-\sum_{i=1}^{n} INC[i] \wedge \bigwedge_{i=1}^{n} INC[i] \in \{0,1\})$$

The P operation will define the entry, the V operation the exit of the critical section. Thus the specifications are:

semaphore: integer s

Req: $m>0$

Init: $s=m \wedge \bigwedge_{i=1}^{n} INC[i]=0$

I: $0 \le s=m-\sum_{i=1}^{n} INC[i] \wedge \bigwedge_{i=1}^{n} INC[i] \in \{0,1\}$

Operations: P

pre: INC[caller]=0

post: INC[caller]=1

V

pre: INC[caller]=1

post: INC[caller]=0

Because the auxiliary variables have to be referenced by the semaphore operations, they must be part of the monitor. A definition local to the processes is not possible anymore. Here, a concept arises that is important for the verification of concurrent programs with shared abstract data types: The use of "private" auxiliary variables in the shared data type (recognized in [Owi77a]). All accessors of the data object need exactly the same type of auxiliary variable, in order to keep track of their history concerning the access of that data object that will be recorded inside the data type operations. Hence, only one auxiliary variable is declared, but the type prefix private indicates that each accessor has its own incarnation. To distinguish different incarnations, a caller-id is implicit

parameter of every data type operation, and a reference to the auxiliary variable x in some operation actually accesses the private auxiliary variable x[caller] of the caller.

We shall see later that the concept of private variables also has its significance for the implementation of abstract data types. Here it affects only the proof, since we only declare auxiliary variables to be private. The proof outline, using the rules described in chapter 2.1 and 3.2, is:

```
type semaphore = monitor(m: integer);
begin
   var s: integer;
       INC: auxiliary private of 0..1;
       q: condition;

   proc P;
   begin
      "I ∧ INC[caller]=0"
      if s=0 then "I ∧ INC[caller]=0 ∧ s=0"
                  q.wait;
      "I ∧ INC[caller]=0 ∧ s>0"
      s := s-1;
      "INC[caller]=0 ∧ s≥0"
      INC := 1;
      "I ∧ INC[caller]=1"
   end P;

   proc V;
   begin
      "I ∧ INC[caller]=1"
      s := s+1;
      "INC[caller]=1 ∧ s>0"
      INC := 0;
      "I ∧ INC[caller]=0 ∧ s>0"
      q.signal;
      "I ∧ INC[caller]=0"
   end V;
```

```
      begin
         "m≥0"
         s := m; INC := 0;
                                  n
         "I ∧ s=m ∧              ∧  INC[i]=0"
                                 i=1
      end;
   end semaphore;

                  n
   "(s=m ∧       ∧  INC[i]=0) => true"
                 i=1
   "true"
   shared sem: semaphore cobegin S1//...//Sn coend;
   "false"

where for each Si

   "1 ∧ INC[i]=0"
   while true do
     begin
        "I ∧ INC[i]=0"
        non-critical section;
        "I ∧ INC[i]=0"
        sem.P;
        "I ∧ INC[i]=1"
        critical section;
        "I ∧ INC[i]=1"
        sem.V;
        "I ∧ INC[i]=0"
        non-critical section;
        "I ∧ INC[i]=0"
     end;
  "false"
```

Note that, by using a lower-level synchronization scheme, no scheduling axiom need be presumed. The fact that the processes are delayed only if m other processes are executing the critical section can directly be drawn from

$$pre(q.wait) \Rightarrow \neg B(\text{cond})$$

9

That processes are resumed immediately when some leave the critical section is derived from the invariant.

It would be preferable and helpful in the search for a suitable invariant to enforce optimal scheduling of an abstract data type's synchronization scheme by its proof rules. [How76a] proposes proof rules for wait and signal that imply optimal scheduling. As a consequence, the callers scheduled by the monitor need not anymore be considered in the proof of that property. The idea is to postulate in the precondition of cond.wait and in the postcondition of cond.signal that the resumption condition for processes waiting in cond is false. The exact proof rules are:

"P∧J∧E"	cond.wait	"P∧J∧B(cond)"
"P∧J∧B(cond)∧¬cond.empty"	cond.signal	"P∧J∧E"

J∧E form the monitor invariant I, where E is a term that extends J to J∧E => ¬B(cond), for all condition queues cond in the monitor, and P is as before.

This more explicit structure of the invariant forces the verifier to produce assertions that imply optimal scheduling. The inclusion of ¬cond.empty in the precondition of cond.signal prevents a second axiom for execution on an empty queue, where signal acts like the empty statement, skip.

The semaphore implementation is very similar to the previous one:

```
type semaphore = monitor(m: integer);
begin
   var s: integer;
       q: condition;
```

```
    proc P;
    begin
       if s=0 then q.wait;
       s := s-1;
    end P;

    proc V;
    begin
       s := s+1;
       if q.empty then q.signal;
    end V;

    begin s := m end;
end semaphore;
```

The only difference is that q.signal is prefixed by an *if* to ensure its precondition.

For the proof, we need an auxiliary variable that plays the role of INC in the previous case. Private variables are of no use, since we do not want to consider the callers of the monitor individually, as we did before. The solution is to keep track of the queue lengths for condition queues in the monitor. They are the only link to the processes if one only wants to regard the monitor in the verification. For each condition cond in the monitor, we introduce an auxiliary variable condlen, the queue length of cond.

In the special case of the semaphore, the length of q, qlen, dually extends s. The expression

$$S = s - qlen$$

could be interpreted as a "hyper"-semaphore with extended range in the negative integers. $S \leq 0$ is the logical condition for synchronization to take place. If S decreases to $S<0$, a delay is necessary; if S increases to $S=0$, a resumption must be performed. A valid invariant is

$$J \equiv (\min(s, qlen) \geq 0)$$

But it does not yield optimal scheduling. We wish a waiting process to be resumed as soon as the semaphore value becomes greater than zero, i.e., the explicit resumption condition is

$$B(q) \equiv (s=1 \wedge qlen>0)$$

but, unfortunately

$$J \not\Rightarrow \neg B(q)$$

The additional requirement

$$E \equiv (\min(s,qlen) \leq 0)$$

extends J appropriately, such that the desired invariant is

$$J \wedge E = I \equiv (\min(s,qlen)=0)$$

With this invariant,

$$B(q) \Rightarrow S \leq 0$$

i.e., resumption only takes place when $S \leq 0$ and, moreover, the code of P shows that a delay occurs only if $S \leq 0$. Hence, synchronization is performed only when it is logically necessary.

The specifications of this semaphore are simple:

semaphore: integer s

Req: $m \geq 0$

Init: $s=m \wedge qlen=0$

I: $\min(s,qlen)=0$

Operations: P

pre: true

post: false

V

pre: true

post: false

The trivial rules for the semaphore operations (any assertion

may serve as pre- or postcondition), as opposed to before, illustrate that indeed no callers are involved in the proof.

Let us first give the proof outline and then discuss its problems:

```
type semaphore = monitor(m:integer);
begin
   var s: integer;
       qlen: auxiliary integer;
       q: condition;

   proc P;
   begin
      "I"
      qlen := qlen+1;
      "min(s,qlen-1)=0"
      if s=0 then "s=0 ∧ qlen≥1"
                  q.wait;
                  "s=1 ∧ qlen≥1"
      "min(s,qlen)-1=0"
      s := s-1;
      qlen := qlen-1;
      "I"
   end P;

   proc V;
   begin
      "I"
      s := s+1;
      "min(s-1,qlen)=0"
      if ¬q.empty then "s=1 ∧ qlen>0 ∧ ¬q.empty"
                       q.signal;
      "I"
   end V;
```

```
    begin
        "m≥0"
        s := m; qlen := 0;
        "I ∧ s=m ∧ qlen=0"
    end;
end semaphore;
```

The proof is acceptable as long as one presumes that

$$\neg q.empty \iff qlen>0$$

at least in the precondition of q.signal, i.e., that qlen really counts the waiting processes properly. It would be best to make this assertion part of the invariant, but unfortunately it is not invariantly true.

[How76a] performs a similar proof on the basis of Habermann's semaphore specifications [Hab72]. Substitute

$$s = nv-np$$
$$qlen = na-np$$

However, then V contains a test on qlen>0 rather than on ¬q.empty which, according to Howard, implies a non-empty condition q. This results in a change of the auxiliary variable qlen to a program variable and makes the semaphore implementation entirely artificial.

For the proposed proof rules, the use of auxiliary variables is essential. Without them, the frequent case of synchronized calls to monitor operations (in which wait and signal frame the procedure body) is not verifiable. The meaningful use of wait requires at least one assignment before its call in the monitor operation.

The cause of all the troubles is that the proper updating of qlen is the responsibility of the verifier (or even programmer) and has to be ensured by the intermediate assertions in the monitor operations. qlen should instead be an auxiliary variable manipulated only inside wait and signal, and not burden

the algorithms defining the monitor procedures.

This motivates a new concept, the concept of a hidden proof variable which only appears in assertions and not in the algorithm. Proof rules in this spirit are defined in [How76b]. The hidden variable cond.len plays the role for the former auxiliary variable condlen.

"$P \wedge J^{*} \wedge E^{*}$"	cond.wait	"$P \wedge J^{*} \wedge B^{*}$(cond)"
"$P \wedge J \wedge B$(cond)"	cond.signal	"$P \wedge J \wedge E$"
"cond.len≥0"	b := cond.empty	"b'=(cond.len=0)"

where for every condition cond in the monitor

$$B(cond) \Rightarrow cond.len > 0$$

The superscript * indicates the substitution of cond.len+1 for cond.len. The internal axiomatic requirements on cond.len are

initially: cond.len=0

invariantly: cond.len≥0 ∧ (cond.len>0 <=> ¬cond.empty)

These proof rules reflect that cond.len is incremented at the start and decremented at the end of each waiting period.

With unaltered specifications, the proof outline is:

```
type semaphore = monitor(m: integer);
begin
   var s: integer;
       q: condition;

   proc P;
   begin
      "I ∧ q.len≥0"
      if s=0 then "s=0 ∧ q.len+1≥0"
                  q.wait;
                  "s=1 ∧ q.len+1>0"
```

```
        "0≤min(s,q.len)≤1 ∧ s=1"
        s := s-1;
        "I ∧ q.len≥0"
    end P;

    proc V;
    begin
        "I ∧ q.len≥0"
        s := s+1;
        "min(s-1,q.len)=0"
        if ¬q.empty then "s=1 ∧ q.len>0"
                         q.signal;
        "I ∧ q.len≥0"
    end V;

    begin
        "m≥0"
        s := m;
        "I ∧ s=m ∧ q.len=0"
    end;
end semaphore;
```

This proof ends the investigation of the verification of optimal scheduling by semaphores.

We started with an implementation by the general concurrent statement that does not isolate the semaphore structure from its concurrent environment. The popular concept of auxiliary variables provided us with an invariant from which the desired dynamic behaviour of the system could be concluded.

Defining the semaphore as a monitor enabled a proper specification of its characteristics. A second concept, the private variable, interrelated the abstract data type and its accessors, yielding the same invariant as in the first case.

Particularly restrictive monitor proof rules and a new representation of its accessors in the proof by means of a third

concept, the hidden variable, guides the verifier in the development of assertions that imply optimal scheduling.

We do not intend to rate any of these techniques over the others. The general concurrent statement is useful for the handling of dynamically changing shared data sets, as in [Gri77]. The verification of monitors with private variables is suitable for static shared data specifications with external properties (i.e., non-trivial proof rules for the operations on them). In the case of the semaphore which only bears a purely internal synchronization property, the restrictive monitor proof rules seem most suitable.

4.2 Simplifying the Resource Pool Implementation

The purpose of this section is to investigate the suitablility of several language constructs and their proof rules for the implementation and verification of an abstract data type that allocates resources from a pool for private access. We shall start with a very general approach and discover the trade-offs in security and ease of verification by restrictions that are appropriate for this specific problem. Since resource allocation is an important operating systems concept, restrictions for its easy use seem worthy of influencing the design of a systems language.

We shall start with specifications that define the abstract private pool object R. The subsequent implementations will be related to these specifications by the abstraction function F that interprets the concrete implementation C as the abstract object R:

$$R = F(C)$$

(For more details see the introduction to chapter 2.)

The private pool will administer a set R of shared resources r_i which are either in private use or free:

$$R \equiv \bigcup_{i \in unitid} \{r_i\} = R_{inuse} \dot{\cup} R_{free}$$

($\dot{\cup}$ denotes the distinct union.) Each of the resources in use will belong to one and only one process:

$$R_{inuse} \equiv \dot{\bigcup}_{caller \in processid} R_{inuse}(caller)$$

These are the properties that characterize the abstract object and that have to be preserved by the operations. Initially, all resources will be available:

$$R_{inuse} = \{\} \wedge R_{free} = \bigcup_{i \in unitid} \{r_i\}$$

The specifications of the pre- and postconditions for the operations follow.

With some resemblance to [FlHa76], the abstract private resource pool may be summarized to:

privatepool: $R \equiv \bigcup_{i \in unitid} \{r_i\}$

Req: true

Init: $R_{inuse} = \{\} \wedge R_{free} = \bigcup_{i \in unitid} \{r_i\}$

$\wedge \wedge_{r \in R}$ r initialized

I: $R = R_{inuse} \dot{\cup} R_{free} \wedge$

$R_{inuse} = \dot{\bigcup}_{caller \in processid} R_{inuse}(caller)$

Operations: get(r)

pre: true

post: $R'_{free} = R_{free} - \{r\} \wedge$

$R'_{inuse}(caller) = R_{inuse}(caller) \cup \{r\}$

put(r)

pre: $r \in R_{inuse}$(caller)

post: $R'_{free} = R_{free} \cup [r] \wedge$

R'_{inuse}(caller) $= R_{inuse}$(caller) $- [r]$

op1(r,...)

pre: $r \in R_{inuse}$(caller) $\wedge$ ...

post: $r \in R_{inuse}$(caller) $\wedge$ ...

op2(r,...)

...

Note that the pool contains the administrative as well as the resource driving operations.

We shall now proceed with the implementations and their proofs, given in a somewhat abridged form. In every case, we first present the program text including auxiliary variables, and then give a proof outline for the two administrative operations get and put. The proof of the initial statement is trivial and left out. After that follows an argument about non-interference.

The first implementation, taken from [Owi77b], uses a general shared class. It permits complete control over the exclusion mode between any two statements within the class operations. The synchronization tool is the semaphore. Our intention is, of course, to provide mutual exclusion of get and put, and to allow any statements of driving routines to execute concurrently.

```
type unitid = 1..unitcount;

type privatepool = shared class;
begin
   var pool: array unitid of resource;
       free: powerset of unitid;
       freecount, mutex: semaphore;
       owner: array unitid of processid;
       m: auxiliary array processid of 0..1;
       a, b: auxiliary integer;

   proc get(var unit: unitid);
   begin
      [freecount.P; a := a+1];
      [mutex.P; m[caller] := 1];
      unit := oneof(free);
      owner[unit] := caller;
      [free := free-{unit}; a := a-1];
      [mutex.V; m[caller] := 0];
   end get;

   proc put(unit: unitid);
   begin
      if owner[unit]≠caller then return;
      [mutex.P; m[caller] := 1];
      [free := free∪{unit}; b := b-1];
      owner[unit] := nil;
      [mutex.V; m[caller] := 0];
      [freecount.V; b := b+1];
   end put;

   proc op1(unit: unitid;...);
   begin operate on pool[unit] end;

   proc op2(unit: unitid;...);
   ...
```

```
      begin
         free := allunits;
         freecount := unitcount; mutex := 1;
         owner := nil;
         a := b := m := 0;
         for i:=1 to unitcount do init pool[i];
      end;
   end privatepool;
```

allunits is the set constant of all one-element sets in the powerset of unitid. The odd *if* statement in put (rather than defining the entire procedure as *if* with the complementary condition and saving the *return*) is due to the required enter; operate; exit format of class operations (see section 2.4).

Owicki gives very implementation-oriented specifications for the private pool, which saves her from considering an abstraction function (we did the same for the semaphore monitor in the last section). This function is our next concern. We define the mapping

$$R = F(\text{free}, \text{owner})$$

where

$$R_{\text{free}} \;\hat{=}\; \{i \mid i \in \text{free} \wedge \text{owner}[i] = \text{nil}\}$$

$$R_{\text{inuse}} \;\hat{=}\; \{i \mid i \notin \text{free} \wedge \text{owner}[i] \neq \text{nil}\}$$

$$R_{\text{inuse}}(\text{caller}) \;\hat{=}\; \{i \mid i \notin \text{free} \wedge \text{owner}[i] = \text{caller}\}$$

The assertion

$$\bigwedge_{i \in \text{unitid}} (i \in \text{free} \Leftrightarrow \text{owner}[i] = \text{nil}) \wedge 0 \leq |\text{free}| \leq \text{unitcount}$$

implies the abstract invariant I(R), but does not suffice as class invariant, because unfortunately its invariance may be violated. The reason is that get and put do not properly exclude each other.

The shared class uses semaphores as a synchronization tool. In our case, the size of the free set determines the synchronization and has to be expressed by a semaphore, freecount. The desired relation is

$$freecount = |free|$$

But this cannot be invariantly maintained. The synchronization on freecount has to be performed outside the critical section of the mutual exclusive rest of the operations, in order to avoid deadlock. Hence, the proper maintenance of the equality may be interrupted, and callers may find freecount decreased without according reduction of free (by get), or the free set extended without proper update of freecount (by put). Two auxiliary variables, a and b, keep track of these cases. The invariant equality is

$$freecount = |free| + b - a$$

The sequencing of the auxiliary variable updates (differing from [Owi77b]) ensures that always $a \geq 0$, $b \leq 0$, and thus

$$freecount \leq |free|$$

This indicates that the semaphore synchronization may not be optimal, because not all freed resource objects may immediately be counted. It turns out that semaphores are a synchronization concept that is sometimes too constrained and not problem-oriented. Here, it leads to an unnecessarily weak invariant. (In this particular case, the with-when statement resolves all problems.)

The auxiliary array m yields the mutual exclusion invariant, quite similarly to the private variable INC for the semaphore monitor in the previous section. The entire class invariant is

$$I \equiv \bigwedge_{i \in unitid} (i \in free \iff owner[i]=nil)$$
$$\wedge\ freecount = |free|+b-a$$
$$\wedge\ 0 \le freecount \le unitcount \ \wedge\ 0 \le mutex \le 1$$
$$\wedge\ mutex = 1 - \sum_{caller \in processid} m[caller]$$

Let us proceed with the proof outline for the private pool operations. We will use the following axioms that are not stated in the references:

a) Let x, s be set variables. Then

"s≠{}" x := oneof(s) "x∈s"

In the private pool, x is a variable of type unitid. That causes no harm, because the representation of s, free, only contains subsets from unitid with cardinality 1.

b) "sem≥0" sem.P "sem'=max(0,sem-1)"

c) "sem≥0" sem.V "sem'=sem+1"

The outline for the two procedures get and put is:

```
proc get(var unit: unitid);
begin
   "I"
   [freecount.P; a := a+1];
   "I"                                                    (1)
   [mutex.P; m[caller] := 1];
   "I"
   unit := oneof(free);
   owner[unit] := caller;
   "unit∈free ∧ owner[unit]=caller"
   [free := free-{unit}; a := a-1];
   "I ∧ owner[unit]=caller"
   [mutex.V; m[caller] := 0];
   "I ∧ owner[unit]=caller"                  (implies get.post)
end get;
```

```
proc put(unit: unitid);
begin
   "I ∧ owner[unit]=caller"                (follows from put.pre)
   if owner[unit]≠caller then return;
   "I ∧ owner[unit]=caller"
   [mutex.P; m[caller] := 1];
   "I ∧ owner[unit]=caller"
   [free := free∪{unit}; b := b-1];
   "unit∉free ∧ owner[unit]=caller"
   owner[unit] := nil;
   "I ∧ unit∉free"
   [mutex.V; m[caller] := 0];
   "I ∧ owner[unit]≠caller"                                    (ii)
   [freecount.V; b := b+1];
   "I ∧ owner[unit]≠caller"          (implies part of put.post)
end put;
```

To show non-interference is always a very tiresome task. It will not be argued here. [Owi77b] states the general technique and gives some examples. Several remarks are due, though.

The non-interference of get and put is not as trivial as [Owi77b] says. If the operations were each one entire critical section, nothing were to be proven. But in the present implementation the assertions marked (i) and (ii) have to be checked. First, note that both (i) and (ii) imply the invariant I, a necessary requirement for non-interference (see section 2.3). Since (i) is only the invariant, there cannot be interference. But (ii), in fact, has to be relaxed so that the postcondition of put only implies a part of the specification put.post:

$$R'_{free} = R_{free} \cup \{r\}$$

is not guaranteed. Another process may have interrupted the execution of put and just taken the resource r. This is not harmful, though. What we really want the private pool to do is

not to lose freed resources, and the invariant together with the verifiable part of put.post state that, if the resource r is not free, it must be in use by some other process. In the specifications,

$$\text{put.post: } R'_{inuse}(caller) = R_{inuse}(caller) - \{r\}$$

really would have sufficed, and we revise it accordingly, now. Although we do not have to test the pre- and postconditions of the operations for non-interference in the proof of the abstract data type, in the verification of the callers a too strong specification may cause problems. We started with the usual private pool specifications (as in [FlHa76, Owi77b]) to become aware of the problem of overspecification. The following discussion builds on the relaxed pool specifications (even though the strong specifications might be satisfied).

[Owi77b] points out the problem of overspecification in another respect. The rules for P and V may cause interference if too strong. In fact, the axioms we use are not interference-free. Therefore we joined the call to the semaphore operation in an elementary action with an appropriate auxiliary variable assignment that relates the auxiliary variable interference-free to the semaphore state. We then only referred to this interference-free relation rather than a specific semaphore state.

The enter; operate; exit format required for class procedures is in this case:

```
get.enter  ≡  [freecount.P; a := a+1]
get.exit   ≡  skip

put.enter  ≡  skip
put.exit   ≡  [freecount.V; b := b+1]
```

The variables mutex and m are not control variables, although they are really synchronizing. The latter is, however, reflected by the fact that, in the proof outline, they only

appear in the class invariant (compare section 2.4). Identifying the set of control variables with that of the synchronizing variables was not possible because of the danger of deadlock in the procedure get. Joining mutex.P and freecount.P in one elementary action would have violated the enter; operate; exit format, but, it seems, still would have lead to a verifiable class, even with a put procedure that satisfies the stronger specifications (for the sake of coarser concurrency). The necessity of a special format requirement is at least not immediately obvious.

The occurrence of a P operation inside an elementary action requires a little explanation. In the proof, the statement

```
freecount.P;
```

for instance, is replaced by

```
[freecount.P; a := a+1];
```

where a is an auxiliary variable. (Because accesses to auxiliary variables will never be executed, they may appear unrestrictedly inside elementary actions.) However, this statement is as little an elementary action as the P operation itself. For P, there are two cases:

a) The caller is not delayed in P.
   Then P is elementary, and so are statements [S] containing a call to P.

b) The caller is delayed in P.
   Then there are two elementary actions:
   i) the delaying of the caller, and
   ii) its resumption and the update of the semaphore.

Nontheless, generally P is said to be elementary, and the delay action is interpreted as synchronizing mechanism for the call to P rather than as part of P. If one adopts this view for elementary actions whose first statement is a call to P, processes will possibly be delayed, but before the execution of

the elementary action. The semantics are the same as for the await synchronization in the general concurrent statement.

As already discussed, the difficulties with the invariant can be overcome by a higher-level synchronization scheme. To equip the shared class with with-when statements rather than semaphores might be too restrictive for a flexible specifiability of the concurrency relations, its most important property. Our application does not need a very fine concurrency grain, and we may use a more structured approach to the synchronization in shared data types, the path expression. In this scheme, the operations may be defined either mutually exclusive or entirely concurrent to each other. For our purpose, this is sufficient. We do not require partial exclusion of operations. Moreover, if the calls to operations are synchronized, the execution of mutually exclusive operations is guaranteed not to be interrupted. The implementation is:

```
type privatepool = class;
begin
   var pool: array unitid of resource;
       free: powerset of unitid;
       owner: array unitid of processid;
   path {(get-put)^unitcount, op1, op2,...} end;

   proc get(var unit: unitid);
   begin
      unit := oneof(free);
      owner[unit] := caller;
      free := free-{unit};
   end get;
```

```
      proc put(unit: unitid);
      begin
         if owner[unit]=caller then
           begin
              free := free^U(unit);
              owner[unit] := nil;
           end;
      end put;

      proc op1(unit: unitid;...);
      begin operate on pool[unit] end;

      proc op2(unit: unitid;...);
      ...

      begin
         free := allunits;
         owner := nil;
         for i:=1 to unitcount do init pool[i];
      end;
   end privatepool;
```

The abstraction function

$$R = F(free, owner)$$

stays as before, but the invariant is now as desired

$$I \;=\; \bigwedge_{i \in unitid} (i \in free \iff owner[i]=nil) \wedge 0 \leq |free| \leq unitcount$$

Additional assertions to derive proper synchronization are not necessary, since the path expression defines all that. (For the expression syntax see section 2.3.) All driving operations may execute concurrently to each other and get and put, whereas the two latter are mutually exclusive. Moreover

$$0 \leq \#(get) - \#(put) \leq unitcount$$

is enforced by synchronizing the calls to get and put. The proof outline is now easy:

```
proc get(var unit: unitid);
begin
   "I"
   unit := oneof(free);
   "unit∉free"
   owner[unit] := caller;
   "unit∉free ∧ owner[unit]=caller"
   free := free-{unit};
   "I ∧ owner[unit]=caller"                    (implies get.post)
end get;

proc put(unit: unitid);
begin
   "I ∧ owner[unit]=caller"                 (follows from put.pre)
   if owner[unit]=caller then
     begin
        "I ∧ owner[unit]=caller"
        free := free∪{unit};
        "unit∉free ∧ owner[unit]=caller"
        owner[unit] := nil;
        "I ∧ unit∉free"
     end;
"I ∧ owner[unit]≠caller"                       (implies put.post)
end put;
```

The invariant term

$$0 \leq |free| \leq unitcount$$

is preserved because

$$|free| = unitcount-(\#(get)-\#(put))$$

Non-interference of the operations is proven as before, where the mutual exclusion of get and put is now really trivial.

The merits of the path expression are apparent: All synchronizing and their associated auxiliary variables disappear from the proof. The path expression provides the control part

of the class invariant. (Note that the procedures are bound to bear the enter; operate; exit format.) The data part of the invariant has to be derived in addition from the semantics of the operations.

How serious the restrictions in synchronization with path expressions are is debatable. In our case, they are only advantageous.

The third implementation uses the recently introduced manager data type for the dynamic allocation of resources. Its purpose is to administer the access capabilities (as called in the original paper [SKB77]) to the objects in the resource pool. The granted resource, implemented as a private class object, is then directly accessible by the owner process. The advantage is that the owner process implicitly gets access to the actual resource object rather than an identification that could be misused or get lost by faulty handling. Consequently, we have to delete the driving operations from the specification of the private pool. They are part of the granted object rather than the pool manager. The implementation is:

```
type privatepool = manager of element: resource;
begin
   var pool: array unitid of resource;
       unit: unitid;
       free: powerset of unitid;
       nofree: condition;

   proc get;
   begin
      if free=nil then nofree.wait;
      unit := oneof(free);
      free := free-(unit);
      bind(pool[unit]);
   end get;
```

```
    proc put;
    begin
       for unit:=1 to unitcount
         while element≠pool[unit] do skip;
       release;
       free := free∪{unit};
       nofree.signal;
    end put;

    begin
       free := allunits;
       for unit:=1 to unitcount do init pool[unit];
    end;
 end privatepool;
```

In one respect, the semantics of the manager are too restrictive for the given pool specifications: every process may only possess at most one resource object in the pool represented by some object of type privatepool. Therefore, the precondition of get has to be strengthened:

$$\text{get.pre: } R_{\text{inuse}}(\text{caller}) = \{\}$$

Again, a few not obvious axioms will be used:

a) The following rule for the for-while loop requires that the iteration index x is known outside the loop.

$$\frac{\text{"}x \in [a..b] \wedge P([a..x-1]) \wedge B\text{"}\ S\ \text{"}P([a..x])\text{"}}{\text{"}P([])\text{"}}$$

$$\text{for } x:=a \text{ to } b \text{ while } B \text{ do } S;$$

$$\text{"}(x \in [a..b] \wedge P([a..x-1]) \wedge \neg B) \vee (x \notin [a..b] \wedge P([a..b]))\text{"}$$

b)
```
    " ∧        owner[i]≠caller ∧ owner(arg)=nil"
      i∈unitid
    bind(arg);
    "owner(arg)=caller"
```

Note that the bind procedure only corresponds to the part of the bind function in [SKB77] that assumes

entry_hold_count=0. The precondition ensures that the calling process does not already access the pool.

c) "owner(<imp par>)=caller"
release;
"owner(<imp par>)=nil"

<imp par> is the implicit parameter described in [SKB77], element, in our case.

d) Initially: owner=nil.

The last three axioms will be motivated in the next section.

The former array owner is now a function, defined on the set of process-id's, and plays the role of a hidden variable as introduced in the previous section. Hence, the abstraction function becomes:

$$R = F(\text{free}, \text{owner})$$

where

$$R_{\text{free}} \triangleq [i \mid i \in \text{free} \wedge \text{owner}(\text{pool}[i]) = \text{nil}]$$

$$R_{\text{inuse}} \triangleq [i \mid i \notin \text{free} \wedge \text{owner}(\text{pool}[i]) \neq \text{nil}]$$

$$R_{\text{inuse}}(\text{caller}) \triangleq [i \mid i \notin \text{free} \wedge \text{owner}(\text{pool}[i]) = \text{caller}]$$

The invariant is

$$I \triangleq \bigwedge_{i \in \text{unitid}} (i \in \text{free} \iff \text{owner}(\text{pool}[i]) = \text{nil}) \wedge 0 \leq |\text{free}| \leq \text{unitcount} \wedge (\text{owner}(\text{pool}[i]) = \text{caller} \Rightarrow \bigwedge_{j \neq i} \text{owner}(\text{pool}[j]) \neq \text{caller})$$

The last term in the invariant states the semantic restriction of the manager to grant at most one object at a time to a fixed process. That is, $R_{\text{inuse}}(\text{caller})$ contains for every caller at most one element.

We proceed with the proof of the manager procedures, using Owicki's axioms for condition queue handling (see section 2.1):

```
proc get;
begin
   "I ∧  ∧      owner(pool[i])≠caller" (follows from get.pre)
      i∈unitid
   if free=nil then "I ∧ free={}"
                    nofree.wait;
                    "I ∧ free≠{}"
   "I ∧ free≠{}"
   unit := oneof(free);
   "I ∧ unit∈free"
   free := free-{unit};
   "unit∉free ∧ owner(pool[unit])=nil
              ∧  ∧      owner(pool[i])≠caller"
                i∈unitid
   bind(pool[unit]);
   "I ∧ owner(pool[unit])=caller"          (implies get.post)
end get;

proc put;
begin
   "I ∧ owner(element)=caller"          (follows from put.pre)
   for unit:=1 to unitcount
     while element≠pool[unit]
        do "unit∈unitid ∧  ∧           element≠pool[i]
                          i∈[1..unit-1]
                         ∧ element≠pool[unit]"
           skip;
           "  ∧          element≠pool[i]"
            i∈[1..unit]
   "I ∧ (unit∈unitid ∧
          ∧            element≠pool[i] ∧ element=pool[unit]) ∨
         i∈[1..unit-1]
        (unit∉unitid ∧  ∧                element≠pool[i])"   (*)
                       i∈[1..unitcount]
   "I ∧  ∧            element≠pool[i] ∧ element=pool[unit]"
      i∈[1..unit-1]
   "I ∧ owner(pool[unit])=caller"
   release;
   "unit∉free ∧ owner(pool[unit])=nil"
   free := free∪{unit};
```

```
        "I ∧ unitefree"
        nofree.signal;
        "I ∧ owner(element)≠caller"                    (implies put.post)
    end put;
```

Note that the term after the ∨ in the postcondition of the for-while loop, marked (*), is false by the capability handling of managers.

This is the end of our development of private pool implementations. In our first attempt we used a very powerful concept, the general shared class. The constraints of its synchronization scheme, the semaphore, and difficulties because of too fine concurrency lead to overcomplicated assertions in the proof. We found that the common specification of a resource pool is too strong for this implementation and, in general, postulates too much.

Introducing a higher-level synchronization concept, the path expression, resolved all these problems. Still, the shared class, with whatever synchronization scheme, joins administrative and driving operations on the pool. The callers have some responsibility in not touching the resource identifications that are handed to them as "tickets" for the resource grant.

The manager data type only encompasses the pool administration and hands the granted resource to the caller. Using the manager again simplifies verification. The non-interference argument is entirely superfluous.

This section exemplifies better than the last one the proper development of an abstract data structure. One starts with specifications that comprise all desired properties. Then one tries to find the most appropriate implementation for that abstract structure. A mapping between the abstract and the concrete object has to show that really all desired properties are met.

4.3 Partial Correctness Proofs of Managers

In the last section, an example motivated the usefulness of the abstract data type manager and intuitively introduced its proof rules. Now we shall develop them in more generality and formality.

The manager acts very similarly to the monitor. Its operations are mutual exclusive, and synchronization of its accessors is provided by wait and signal on condition queues. Consequently, the axioms for the manager's invariant, initial statement, and procedure bodies are exactly the same as for the monitor.

However, additional proof rules have to define the dynamic allocation by managers: the treatment of capabilities. We require that capabilities are only updated by the standard operations bind and release, much as condition queues by wait and signal. Therefore, bind and release are defined more generally than in [SKB77] where they are only used for private but not for shared resource allocation.

We consider the abstract data type

```
type allocator = manager of capability: resource;...
```

Inside this manager, an object of type resource appears as though it were defined as

```
type Resource = record instance: resource;
                       capabilityset: powerset of processid;
                end;
```

The capabilityset contains all id's of processes that possess a capability to the resource object. For private resources it is either empty or contains one process. We define the two procedures

```
proc bind(object: resource);
begin
   if capability=nil then
     begin
        with object
          do capabilityset := capabilitysetU{caller};
        capability := object;
     end;
end bind;

proc release;
begin
   if capability≠nil then
     begin
        with capability do
          do capabilityset := capabilityset-{caller};
        capability := nil;
     end;
end release;
```

The capability is an example of a private program variable (of type resource) that is not auxiliary. Every process has its own incarnation that is only updated on its request, and all incarnations are handled similarly. Initially, all capabilities and capabilitysets are nil. We decided to implement bind as a procedure rather than a function (as done in [SKB77]), because this seems to be a more natural concept.

Now, the general proof rules for the two operations can be understood:

```
"capability[caller]≠object"
bind(object);
"capability[caller]=object"

"capability[caller]=object"
release;
"capability[caller]=nil"
```

We require that capabilities and capabilitysets are not updated outside bind and release. Then the following assertions are preserved:

$$I(\text{capability}) \equiv$$

$$\bigwedge_{i \in \text{processid}} (\text{capability}[i]=\text{nil} \iff \bigwedge_{r \in R} i \notin r.\text{capabilityset})$$

$$\wedge \bigwedge_{i \in \text{processid}} (\text{capability}[i]\neq\text{nil} \iff \bigvee_{x \in R} (\text{capability}[i]=x \wedge i \in x.\text{capabilityset} \wedge \bigwedge_{r \in R-\{x\}} i \notin r.\text{capabilityset}))$$

I(capability) expresses that, at any point of time, every process may possess at most one capability. It is part of the manager invariant, since it is true after initialization of all capabilities to nil. I(capability) enables an alternative use of the capability or capabilityset variable in assertions.

For private resource management, the rule for bind has to be stronger:

"capability[caller]=nil ∧ object.capabilityset=nil"
bind(object);
"capability[caller]=object"

It implies, together with I(capability), that capabilitysets contain at most one process. The capabilityset is a hidden proof variable, in this case, in fact, a function corresponding to the intuitively introduced owner function in the private pool example (see previous section).

Note that, by generalizing the binding scheme, bind is less powerful than in [SKB77]. There, it was introduced specifically to ensure a proper distribution of private resources. Here, it handles resource distribution in general, and only an especially strong precondition guarantees that private resources remain private.

Assertions about the resource objects (private or shared) are handled like assertions about shared abstract data types, only that for the caller, unless it is the manager, the enabling

predicate

$$\text{capability}[\text{caller}]=\text{object}$$

has to hold.

As illustration, let us modify the pool specifications to suit the allocation of shared resources and implement it by a manager.

sharedpool: $R \equiv \bigcup_{i \in unitid} \{r_i\}$

Req: true

Init: $R_{inuse} = \{\} \wedge R_{free} = \bigcup_{i \in unitid} \{r_i\}$

$\wedge \bigwedge_{r \in R}$ r initialized

I: $R = R_{inuse} \dot{\cup} R_{free} \wedge$

$R_{inuse} = \bigcup_{caller \in processid} R_{inuse}(caller)$

Operations: get(r)

pre: $(r \in R \vee r=undef) \wedge R_{inuse}(caller)=\{\}$

post: $r \in R \Rightarrow (R'_{inuse}=R_{inuse}, R'_{free}=R_{free},$

$R'_{inuse}(caller)=\{r\})$

$r=undef \Rightarrow \bigvee_{x \in R} (r'=x, R'_{free}=R_{free}-\{x\},$

$R'_{inuse}(caller)=\{x\})$

put(r)

pre: $R_{inuse}(caller)=\{r\}$

post: $R'_{inuse}(caller)=\{\}$

Note that R_{inuse} does not anymore form a distinct union of the $R_{inuse}(i)$ over i in processid. The postcondition of put is again weakly specified. An additional requirement

$$\left(\bigwedge_{i \in processid,\ i \neq caller} r \notin R_{inuse}(i) \right) \Rightarrow (R'_{inuse} = R_{inuse} - \{r\},\ R'_{free} = R_{free} \cup \{r\})$$

that the resource r is made available if the caller was its only owner could cause interference problems in the caller proof.

Since in the shared pool the resource objects are distinguishable, the precondition of put has to be more specific than in the private pool. Note that our first private pool implementations also made the objects distinguishable for the competitors (by an explicitly passed id parameter). This may be desirable, e.g., for message passing, but is not a requirement for the general specifications.

The manager implementation follows:

```
const undef = 0;

type ext_unitid = undef..unitcount;

type sharedpool = manager of element: resource;
begin
   var pool: array unitid of resource;
       unit: unitid;
       free: powerset of unitid;
       nofree: condition;

   proc get(var id: ext_unitid);
   begin
      if id=undef then
        begin
           if free=nil then nofree.wait;
           id := oneof(free);
        end;
      if id∈free then free := free-{id};
      bind(pool[id]);
   end get;
```

```
        proc put;
        begin
           for unit:=1 to unitcount
             while element≠pool[unit] do skip;
           release;
           if pool[unit].capabilityset=nil then
             begin
                free := free∪{unit};
                nofree.signal;
             end;
        end put;

        begin
           free := allunits;
           for unit:=1 to unitcount do init pool[unit];
        end;
     end sharedpool;
```

The abstraction function is

$$R = F(free, pool)$$

where

$$R_{free} \triangleq \{i \mid i \in free \wedge pool[i].capabilityset = nil\}$$

$$R_{inuse} \triangleq \{i \mid i \notin free \wedge pool[i].capabilityset \neq nil\}$$

$$R_{inuse}(caller) \triangleq \{i \mid i \notin free \wedge caller \in pool[i].capabilityset\}$$

$R_{inuse}(caller)$ contains for every caller at most one element.

The invariant is

$$I \triangleq \bigwedge_{i \in unitid} (i \in free \Leftrightarrow pool[i].capabilityset = nil) \wedge 0 \leq |free| \leq unitcount \wedge I(element)$$

The proof of get:

```
proc get(var id: ext_unitid);
begin
   "I ∧ element[caller]=nil              (follows from get.pre)
      ∧ id∈ext_unitid"
   if id=undef then
     begin
        "I ∧ id=undef"
        if free=nil then "I ∧ free={}"
                         nofree.wait;
        "I ∧ id=undef ∧ free≠{}"
        id := oneof(free);
        "I ∧ id∈free ∧ id∈unitid"
     end;
   "I ∧ id∈unitid"
   if id∈free then "I ∧ id∈free"
                   free := free-{id};
   "id∈free ∧ element[caller]=nil"
   bind(pool[id]);
   "I ∧ element[caller]=pool[id'] ∧ F(id)" (implies get.post)
end get;
```

where F(id) relates the input and output state of the parameter id as follows:

$$F(id) \equiv (id \in unitid \Rightarrow id' = id) \wedge (id = undef \Rightarrow (id' \in free \wedge free' = free - \{id'\}))$$

The proof of put:

```
proc put;
begin
   "I ∧ element[caller]≠nil"              (follows form put.pre)
   for unit:=1 to unitcount
     while element≠pool[unit]
        do "unit∈unitid
             ∧  ∧          element[caller]≠pool[i]
              i∈[1..unit-1]
             ∧ element[caller]≠pool[unit]"
           skip;
           " ∧          element[caller]≠pool[i]"
             i∈[1..unit]
   "I ∧ (unit∈unitid ∧
          ∧            element[caller]≠pool[i] ∧
        i∈[1..unit-1]
        element[caller]=pool[unit]) ∨ (unit∉unitid ∧
          ∧              element[caller]≠pool[i])"          (*)
        i∈[1..unitcount]
   "I ∧  ∧            element[caller]≠pool[i]
        i∈[1..unit-1]
     ∧ element[caller]=pool[unit]"
   release;
   "unit∉free ∧ element[caller]=nil"
   if pool[unit].capabilityset=nil then
     begin
        free := free∪{unit};
        "I ∧ unit∈free ∧ element[caller]=nil"
        nofree.signal;
     end;
   "I ∧ element[caller]=nil"                (implies put.post)
end put;
```

The for-while axiom has been introduced in the previous section. Here, more formally than in the proof there, the term after the ∨ in the postcondition of the for-while loop, marked (*), is false by the validity of I(element) in I.

4.4 Termination Proofs of Calls to Shared Abstract Data Objects

The previous proof outlines for the semaphore and the resource pool only argue the partial correctness of their implementations. Termination has to be shown separately.

This section illustrates termination arguments for operations on shared abstract data types. We shall not be looking at the termination of sequential statements. This matter receives consideration in a number of publications, e.g., [Man74, KaMa75, MaWa78].

Our concern is to derive conditions under which the synchronization of operations on some abstract data object cannot lead to non-termination of the call statement in the calling process. Again, the proof technique follows closely that of [Owi77b]. We assume that the only cause for a process delay is synchronization on some abstract data object. Effects of system scheduling are not regarded.

The notation A→B indicates that a computation is bound to eventually reach state B if it reaches state A at some point. We shall use expressions like

$$A \rightarrow B \wedge C$$

which expresses that state A is followed by the validity of B and C. Nothing is said about the length of time that B and C hold. But there is some point after A occurred at which both B and C are true.

The following predicates are of importance in termination proofs:

start(i,L):

process i is about to execute statement L,

finish(i,L):

process i just finished the execution of statement L,

blocked(i,L):

process i is delayed in the execution of statement L.

Every language statement is characterized by an axiom that defines its pre- and postcondition for termination, very similarly to its partial correctness axiom. The termination axiom has the form:

$$\text{L: statement;} \quad \frac{\text{termination condition}}{\text{termination expression}}$$

If the termination condition can be shown, the statement terminates according to the stated termination expression. Owicki's termination axiom for the concurrent statement is only suitable for the trivial case of disjoint processes (no interaction through synchronization or access of shared data [Hoa75]):

L: cobegin S1//.../Sn coend;

$$\frac{\bigwedge_{i=1}^{n} \text{start}(i,Si) \rightarrow \text{finish}(i,Si)}{(\text{start}(0,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(0,L) \wedge \text{post}(L))}$$

The termination axioms for the sequential statements in our examples are fairly straight-forward and will not be given here. We only use loops that are bound to terminate, for loops. For the axiom of the while loop see [OwGr76a].

Let us now start with the investigation of shared abstract data types. The termination of any synchronized operation $T.p(\bar{x};\bar{y})$ is expressed by its delay assertion:

$$(\text{blocked}(\text{caller},T.p(\bar{x};\bar{y})) \wedge T.p.\text{Delay}) \rightarrow \neg T.p.\text{Delay} \Rightarrow (\text{start}(\text{caller},T.p(\bar{x};\bar{y})) \rightarrow \text{finish}(\text{caller},T.p(\bar{x},\bar{y})))$$

T.p.Delay is a condition that has to be proven necessary for non-termination of procedure T.p, the delay condition of T.p. (T.p.Delay need not necessarily be the negation of the resumption condition in the proof of T.p but must imply it.) If the delay assertion holds with a proper delay condition, the procedure call in process i is axiomatized

L: T.p($\bar{a}$,$\bar{e}$);

$$\frac{(\text{blocked}(i,L) \wedge \text{pre}(L) \wedge T.p.\text{Delay}) \rightarrow \neg T.p.\text{Delay}}{(\text{start}(i,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(i,L) \wedge \text{post}(L) \wedge T.p.\text{post}^{\bar{x}\ \bar{y}\ \text{caller}}_{\bar{a}\ \bar{e}\ \ i})}$$

pre(L) and post(L) are taken from the partial correctness proof of statement L. The axiom expresses that, while i is blocked in a call to T.p, T.p's resumption condition must eventually arise. In order to prove this, it is convenient to assume that the scheduling policy of the abstract data type T is fair, i.e., only a finite number of processes may postpone the resumption of any delayed process to a point of time after their own resumption. [Owi77b] states the termination condition

$$(\text{start}(i,L) \wedge \text{pre}(L) \wedge T.p.\text{Delay}) \rightarrow \neg T.p.\text{Delay}$$

from which may be concluded only that some waiting process j is eventually resumed, i.e.,

$$(\text{start}(i,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(j,L) \wedge \text{post}(L) \wedge T.p.\text{post}^{\bar{x}\ \bar{y}\ \text{caller}}_{\bar{a}\ \bar{e}\ \ j})$$

The termination of the call statement in process i is not implied.

These tools are sufficient to prove the termination of operations synchronized by a path expression. The delay assumption and the condition for the termination of the call statement may be concluded directly from the path expression.

For shared data types with lower-level synchronization schemes, we need additional axioms. [Owi77b] defines rules for the use of semaphores, the synchronization scheme of Owicki's

general shared class:

L1: sem.P;

$$\frac{(\text{blocked}(\text{caller},L1)\wedge\text{pre}(L1)\wedge\text{sem}=0)\rightarrow \text{sem}>0}{(\text{start}(\text{caller},L1)\wedge\text{pre}(L1))\rightarrow(\text{finish}(\text{caller},L1)\wedge\text{post}(L1))}$$

L2: sem.V;

$$(\text{start}(\text{caller},L2)\wedge\text{pre}(L2))\rightarrow(\text{finish}(\text{caller},L2)\wedge\text{post}(L2))$$

Note that sem.V terminates unconditionally.

The termination postcondition is more explicit than the partial correctness postcondition, because it may include variant assertions that do not hold as precondition of the next statement. In this case,

$$\text{post}(L1) \;\hat{=}\; (\text{sem}'=\max(0,\text{sem}-1))$$

$$\text{and}\quad \text{post}(L2) \;\hat{=}\; (\text{sem}'=\text{sem}+1)$$

are not interference-free in the partial correctness proof. They are valid immediately after the termination of L1 and L2 respectively, but need not remain invariantly true until the execution of the following statement (see section 4.2).

Finally, let us define axioms for the even lower-level synchronization scheme of managers and monitors, condition queues. Here, we have the choice between different partial correctness rules, and the termination axioms look different, accordingly. Again, the superscript * indicates the substitution of cond.len+1 for cond.len, whereas the subscript $_{*}$ expresses the substitution of cond.len-1 for cond.len.

1. Following [Owi77a]

Partial correctness:

$$"P \wedge I^{*}" \quad \text{cond.wait} \quad "P \wedge I^{*} \wedge B^{*}(\text{cond})"$$

$$"P \wedge I \wedge B(\text{cond})" \quad \text{cond.signal} \quad "P \wedge I"$$

where

$$B(\text{cond}) \Rightarrow \neg \text{cond.empty}$$

Termination:

L1: cond.wait;

$$\frac{(\text{blocked}(\text{caller},L1) \wedge \text{pre}^{*}(L1) \wedge \neg B(\text{cond})) \rightarrow (\text{post}^{*}(L1) \wedge \text{start}(j,L2))}{(\text{start}(\text{caller},L1) \wedge \text{pre}(L1) \wedge \neg B(\text{cond})) \rightarrow (\text{finish}(\text{caller},L1) \wedge \text{post}(L1))}$$

L2: cond.signal;

$$(\text{start}(\text{caller},L2) \wedge \text{pre}(L2)) \rightarrow (\text{finish}(\text{caller},L2) \wedge \text{post}(L2))$$

2. Following [How76b]

Partial correctness:

$$"P \wedge J^{*} \wedge E^{*}" \quad \text{cond.wait} \quad "P \wedge J^{*} \wedge B^{*}(\text{cond})"$$

$$"P \wedge J \wedge B(\text{cond})" \quad \text{cond.signal} \quad "P \wedge J \wedge E"$$

where

$$B(\text{cond}) \Rightarrow \text{cond.len} > 0$$

and $$J \wedge E \Rightarrow \neg B(\text{cond})$$

Termination:

L1: cond.wait;

$$\frac{(\text{blocked}(\text{caller},L1) \wedge \text{pre}^{*}(L1)) \rightarrow (\text{post}^{*}(L1) \wedge \text{start}(j,L2))}{(\text{start}(\text{caller},L1) \wedge \text{pre}(L1)) \rightarrow (\text{finish}(\text{caller},L1) \wedge \text{post}(L1))}$$

L2: cond.signal;

$(\text{start}(\text{caller},L2) \wedge \text{pre}(L2)) \rightarrow (\text{finish}(\text{caller},L2) \wedge \text{post}(L2))$

In both cases, the termination axioms express the same, only in the first case they have to compensate the weaknesses of the partial correctness axioms. The influence of signal on the termination of wait has to be explicitly stated, since the synchronization scheme permits resumption conditions to remain unsignalled. It is important to realize that the consideration of queue lengths is essential for termination arguments in abstract data types with condition queue synchronization. Therefore, Owicki's partial correctness rules have been extended accordingly.

The termination of bind and release in the manager is defined by the trivial termination axiom:

L: bind(object); or L: release;

$(\text{start}(\text{caller},L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(\text{caller},L) \wedge \text{post}(L))$

We are now able to prove the termination for some of our semaphore and resource pool implementations in the previous sections.

1. semaphore monitor

a) semaphore.P;

semaphore.P.Delay: $s=0$

(If $s>0$, P trivially terminates.

$s=0 \Rightarrow \neg B(q)$, the negated resumption condition.)

The termination of P requires its delay assertion

(i) $((blocked(caller,P) \wedge s=0) \rightarrow s>0) \Rightarrow (start(caller,P) \rightarrow finish(caller,P))$

The only problem in this implication is the termination of

L: q.wait;

in P. We have to show

(ii) $(blocked(caller,L) \wedge pre_*(L)) \rightarrow (post_*(L) \wedge start(j,q.signal))$

The following holds:

$$pre_*(L) \Rightarrow (I \wedge s=0 \wedge q.len>0)$$

The only operation that can, by incrementing s, validate B(q) and hence $post_*(L)$ is V. Doing so, it is bound to execute signal, which verifies (ii).

The presumption in the delay assertion implies the execution of signal which verifies (i).

Proving that a call to P does terminate means proving the presumption in the delay assertion for P:

$$(blocked(caller,P) \wedge s=0) \rightarrow s>0$$

One has to argue that some other process in the system must eventually execute V, if s=0 and the caller is still blocked. In our case this holds because of the infinite loop structure of the processes.

b) semaphore.V;

semaphore.V.Delay: false
(V is not synchronized.)
V's delay assertion is

$((blocked(caller,V) \wedge false) \rightarrow \neg false) \Rightarrow (start(caller,V) \rightarrow finish(caller,V))$

or simply

start(caller,V) → finish(caller,V)

2. privatepool manager

a) privatepool.get(id);

privatepool.get.Delay: free=nil
(free=nil => ¬B(nofree), the negated resumption condition.)

get has the delay assertion

(i) ((blocked(caller,get)∧free=nil) → free≠nil) => (start(caller,get) → finish(caller,get))

The only problem in this implication is the termination of

L: nofree.wait;

in get. We have to show

(ii) (blocked(caller,L)∧pre_*(L)) → ($post_*$(L)∧start(j,nofree.signal))

It holds

pre_*(L) => (I ∧ free=nil ∧ nofree.len>0)

The only operation that can, by inserting in free, validate B(nofree) and hence $post_*$(L) is release. Doing so, it is bound to execute signal which verifies (ii).

The presumption in the delay assertion implies the execution of signal which verifies (i).

Proving that a call to get terminates, as before, means arguing that release must eventually be executed if the free set is empty and some process is waiting.

b) privatepool.put;

privatepool.put.Delay: false
(put is not synchronized.)

The delay assertion is, unconditionally

start(caller,put) → finish(caller,put)

3. sharedpool manager

Exactly like 2., the privatepool manager.

4. privatepool general shared class

The proof is very similar to the previous example and is outlined in [Owi77b]. Validation of the delay assertion for get is deduced from the invariant

freecount = |free|+b-a

5. privatepool path expression class

a) privatepool.get(unit);

privatepool.get.Delay:

C ≡ (#(get)-#(put) > unitcount)

(from the path expression)
The delay assertion

((blocked(caller,get)∧C) → ¬C) =>
(start(caller,get) → finish(caller,get))

is enforced by the path expression. Its presumption is verified as above.

b) privatepool.put;

privatepool.put.Delay:

C ≡ (#(get)-#(put) < 0)

(from the path expression)
The delay assertion

((blocked(caller,put)∧C) → ¬C) =>
(start(caller,put) → finish(caller,put))

is enforced by the path expression. Its presumption holds, for instance, if each process calls get and put every other time, starting with get.

Note that the path expression synchronizes put as well. Errors due to unsynchronized calls of put in the other implementations have to be (and are) taken care of in its code. The termination proofs demonstrate very well the power of path expressions.

5 Conclusions

The investigation of verification techniques for concurrent algorithms motivates several language features for the structured representation of concurrency.

We understood the suitability of the concurrent statement for the representation of a set of concurrent processes. It syntactically frames the concurrent execution in the program text with cobegin...coend and (in the more sophisticated formats) defines the data shared among processes by resource or shared. This supports the documentation of correlations and interferences between processes.

Secondly, and above all, we realized the significance of shared abstract data types for structured concurrent programming. For sequential structured programming, principles like stepwise refinement, modularity, and the development of program families have been suggested [DDH72, Par76]. Carried over to concurrent programming, the major goal comprising all these techniques is:

> Design your concurrent program as a collection of interacting processes, and keep the design problems as much as possible in a sequential context. In another step, worry about their interaction.

The process serves much the same purpose in concurrent programming as the procedure in sequential programming. The design problem is broken into parts that are easier to comprehend and solve by themselves. The call convention of the procedure corresponds to the synchronization and interference of processes. It is the interface that specifies all communication between the module (procedure or process) and its environment.

Shared abstract data types achieve the isolation of this interface from the concurrent program modules, the processes. Processes may be designed without regard to their

synchronization and interference. The proper specification of the communication device, the set of shared abstract data types for the concurrent statement, takes care of all that. The advantageous effect of this principle is documented by the verification requirements: the problem of non-interference can be (almost) entirely absorbed in abstract data types. An illustrating argument for the need for separating the interface to its environment from the process is that it becomes properly specifiable. Abstract data types are built according to a general specification pattern that comprises all external and internal requirements on the defined data.

One need not stop at this point. The manager data type indicates that abstract data types can also represent different kinds of specialized higher-level communication between processes, in this case the competition for resources. Others are conceivable.

A still not sufficiently solved problem is deadlock in access hierarchies of shared abstract data types. The manager provides a safe and useful design tool for an important subset of applications, but none of the given data types supports a practical general technique for deadlock avoidance. Recent issues of the ACM SIGOPS Journal Operating Systems Reviews contain a discussion on this subject, started by [Lis77]. It seems that deadlock avoidance imposes very strong requirements on the integrity of shared data objects.

As it appears in Owicki's proof method, the non-interference argument is the most cumbersome and problematic part of concurrent correctness proofs. It leads to a proof effort that grows non-linearly with the size of the program and possibly to a revision of the previously developed sequential proofs for the processes (if interference has been detected). Usually, interference will only become evident after the sequential proofs have been completed. The use of shared abstract data types weakens these problems somewhat and provides

two alternatives for the structured development of concurrent programs:

a) If the nature of concurrent communication is known with the program specification, the definition of the shared abstract data types is possible or has been done before the development of the processes, and it can aid in a constructive process design. This will be the approach for concurrent systems with very simple or widely-used communication patterns, e.g., competition for resources.

b) If the nature of concurrent communication is not documented by the program specification, the processes are constructed first, with calls to ficticious data type operations. Then the specifications for the abstract data types are deduced to satisfy all call assertions in the different processes, according to the rule of consequence. This will be the more frequent approach, for complex concurrent systems.

In certain cases, a complete absorption of concurrency matters in abstract data types may not be possible. This is reflected by the context sensitive "rule of adaption", which analogously also applies for procedures [Owi77b, Hoa71]. Without restrictions on concurrency, one cannot hope to reach the ultimate goal, entire isolation of interference considerations.

6 Bibliography

This chapter contains a selection of papers published on or related to the axiomatic verification of concurrent algorithms. Some contributions are briefly described, and points of discussion are stressed.

[Bri73] Brinch-Hansen, P.
Concurrent Programming Concepts
Computing Surveys 5, 4 (Dec. 1973), 223-245

Brinch-Hansen gives a history of the development of

a) concurrency concepts:
   i) fork/join
   ii) cobegin...coend

b) synchronization concepts:
   i) event statements
   ii) semaphores
   iii) critical regions
   iv) conditional critical regions

The necessity of history variables (auxiliary variables in [OwGr76a, OwGr76b]) for the verification of concurrent algorithms is recognized. History variables may be interpreted as protocol variables. Their treatment may not affect the flow of control and data in the rest of the program.

[Bri75] Brinch-Hansen, P.
The Programming Language Concurrent Pascal
IEEE Trans. on Soft. Eng. SE-1, 2 (June 1975), 199-207

[CaHa74] Campbell, R.H.; Habermann, A.N.
The Specification of Process Synchronization by Path Expressions
in "Operating Systems", Lecture Notes in Computer Science 16, Goos and Hartmanis (Eds.), Springer Verlag, 1974, 89-102

Introduction to different forms of path expressions: sequence, selection, repetition and simultaneous execution. Presentation of popular examples: readers & writers, producer-consumer, semaphore. Discussion of the implementation of the proposed synchronization scheme.

[Con63] Conway, M.E.
A Multiprocessor System Design
AFIPS Conf. Proc. 24, FJCC 1963, 139-146

[DaHo72] Dahl, O.-J.; Hoare, C.A.R.
Hierarchical Programming Structures
in [DDH72], 175-220

[DDH72] Dahl, O.-J.; Dijkstra, E.W.; Hoare, C.A.R.
Structured Programming
A.P.I.C. Studies in Data Processing 8, Academic Press, 1972, 220 p.

[Dij76] Dijkstra, E.W.
A Discipline of Programming
Prentice-Hall, 1976, 217 p.

[FlHa76] Flon, L.; Habermann, A.N.
Towards the Construction of
Verifiable Software Systems
Proc. 2nd Int. Conf. on Software Engineering,
13.-15.10.1976, San Francisco, Cal., 141-148

Habermann tries to overcome the monitor drawbacks (see entry [Hoa74]) by the following change in the concept of an abstract data type that represents shared data: rather than providing mutual exclusion and synchronization primitives for every operation, a "path expression" defines the scheduling constraints (short-term and medium-term) for the operations in respect to each other and the state of the data. It has to incorporate mutual exclusion as well as all synchronization conditions. Thereby the calls to operations are synchronized rather than their executions.

The concept is exemplified for the bounded buffer and the producer-consumer problem. It is argued that the use of path expressions not only structures the synchronization on shared abstract data types, but also facilitates the proof of synchronization properties.

Habermann's abstract data type provides a more flexible short-term scheduling scheme than the monitor but does not solve the problems with deadlocks in access hierarchies. The impacts of the restrictions on synchronization are debatable.

[Flo67] Floyd, R.W.
Assigning Meanings to Programs
Proc. of Symposia in Applied Mathematics 19,
American Mathematical Society, 1967, 19-32

[Gri76] Gries, D.
An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs
IEEE Trans. on Soft. Eng. SE-2, 4 (Dec. 1976), 238-244

[Gri77] Gries, D.
An Exercise in Proving Parallel Programs Correct
Comm. ACM 20, 12 (Dec. 1977), 921-930

As a non-trivial example for a verification with Owicki's axiom system [OwGr76a, OwGr76b], Gries proves the correctness of the on-the-fly garbage collector for a LISP implementation by Dijkstra. The concurrent environment consists of the garbage collector process and the communicating algorithm (mutator) the LISP program has to use.

Since the problem specifies a highly dynamic set of shared variables, the general concurrent statement (without resources) is used. The two processes collector and mutator are proven sequentially correct and shown to be interference-free. Termination need not be verified: both processes run forever.

Note: The problem does not require synchronization, but auxiliary variables have to be handled in elementary actions.

[Gri78] Gries, D.
The Multiple Assignment Statement
IEEE Trans. on Soft. Eng. SE-4, 2 (Mar. 1978), 89-93

Gries defines axioms for multiple assignments to simple variables, to a single subscripted variable, and, as a new contribution, to several subscripted variables.

The axiom for the multiple assignment to simple variables is

$$"p^{x1,\ldots,xn}_{e1,\ldots,en}" \; x1,\ldots,xn := e1,\ldots,en \; "p"$$

To satisfy these axiomatic semantics, all expressions ei have to be evaluated before any assignment to some xi is performed.

The axiom for the multiple assignment to a single subscripted variable in the array b is

$$"p^{b}_{(b;r:e)}" \; b[r] := e \; "p"$$

where the array (b;r:e) is defined by

$$(b;r:e)[i] \equiv \underline{\text{if}}\ i=r\ \underline{\text{then}}\ e\ \underline{\text{else}}\ b[i]$$

For several redefinitions of subscripted variables in b, Gries defines the rule

$$"p^{b}_{(b;r1:e1,..,rn:en)}" \; b[r1],..,b[rn] := e1,..,en \; "p"$$

which implicitly defines the order of assignment, because the ordering of the pairs ri:ei affects the semantics. A more general rule that allows unrestricted concurrency is mentioned but considered impractical.

Example proofs are given for an algorithm maintaining a linked list and an array sort algorithm.

Gries makes the point that multiple assignments are easier to comprehend than equivalent sequences of simple assignments. But to understand and simplify the precondition, the assertion characterizing the multiple assignment, may require considerable intuition and effort that may be exponential in the number of updates performed by the multiple assignment.

[Hab72] Habermann, A.N.
Synchronization of Communicating Processes
Comm. ACM 15, 3 (Mar. 1972), 171-176

Habermann defines the data structure semaphore through three counters:

nw(s) the number of attempted executions of wait(s),
np(s) the number of successful executions of wait(s),
ns(s) the number of executions of signal(s).

The semaphore invariant is defined as

$$np(s) = \min(nw(s), C[s]+ns(s))$$

where C[s] is the initial value of the semaphore s. Implementations of the bounded buffer and a producer-consumer system are proven.

[HGLS78] Holt, R.C.; Graham, G.S.; Lazowska, E.D.; Scott, M.A.
Structured Concurrent Programming
with Operating Systems Applications
Addison-Wesley Series in Computer Science,
1978, 262 p.

[Hoa69] Hoare, C.A.R.
An Axiomatic Basis for Computer Programming
Comm. ACM 12, 10 (Oct. 1969), 576-580, 583

[Hoa71] Hoare, C.A.R.
Procedures and Parameters: An Axiomatic Approach
Symposium on Semantics of Algorithmic Languages,
Lecture Notes in Mathematics 188, Engeler (Ed.),
Springer Verlag, 1971, 102-116

[Hoa72a] Hoare, C.A.R.
Towards a Theory of Parallel Programming
in "Operating Systems Techniques", Hoare and Perrott (Eds.), A.P.I.C. Studies in Data Processing 9, Academic Press, 1972, 61-71

Hoare lays the foundation for the formal treatment of concurrent statements. The concurrent statement and the notion of a resource as the set of shared variables associated with a concurrent statement are introduced. The with-when construct is suggested for the representation of critical sections on shared data.

Hoare imposes and formalizes semantic restrictions on his constructs for a clean handling of shared data (only resource variables may for several processes be subject to change, and then only in critical sections) and gives proof rules for the concurrent and the with-when statement. Shared data are characterized by an assertion that has to be invariant at times when the data are not accessed (compare [Hoa72b]).

[Hoa72b] Hoare, C.A.R.
Proof of Correctness of Data Representations
Acta Informatica 1 (1972), 271-281

Hoare extends his axiom system for program verification to prove abstract data types correct. Essential arguments:

a) A mapping has to provide the relationship between the specifications of the data type and its implementation.

b) A correct implementation of an abstract data type is characterized by

i) an invariant assertion that has to be valid after initialization between accesses of the shared data,

ii) the fact that each operation defined on the data type has the effect of its specification.

[Hoa74] Hoare, C.A.R.
Monitors: An Operating System Structuring Concept
Comm. ACM 17, 10 (Oct. 1974), 549-557
Corrigendum: Comm. ACM 18, 2 (Feb. 1975), 95

Hoare introduces the monitor concept for a clean access of shared variables in a concurrent environment. The monitor is an abstract data type that provides

i) mutual exclusive execution (short-term scheduling) of its operations,

ii) primitives for the synchronization (medium-term scheduling) of its accessors that may be used in any monitor operation.

The correctness of monitors is proven by Hoare's axiom system for abstract data types [Hoa72b]:

(A1)	"true"	initial statement	"I"
(A2)	"I"	each monitor operation	"I"

where I is an assertion about the monitor data that is invariant between accesses, plus the following synchronization axioms:

(A3)	"I"	cond.wait	"I∧B(cond)"
(A4)	"I∧B(cond)"	cond.signal	"I"

for each condition queue cond in the monitor, where B(cond) is the resumption condition for processes delayed in cond. (A3) and (A4) ensure that the monitor invariant holds whenever the accessor of the

monitor object changes by synchronization.

Problems: Too stringent concurrency constraints.

i) No individual short-term scheduling scheme for different types of operations (e.g., read and write).
ii) Monitor hierarchies introduce the danger of deadlock.
iii) Substructures of monitor data must be accessed mutually exclusive with respect to each other (i.e., only one substructure may be accessed at a time).
iv) The synchronization axioms do not support optimal scheduling.

[Hoa75] Hoare, C.A.R.
Parallel Programming: An Axiomatic Approach
Computer Languages 1, 2 (June 1975), 151-160

Hoare makes the first attempt to include shared abstract data types into the verification of parallel processes. Processes are classified as follows:

i) disjoint processes (no shared data)
ii) competing processes (message passing)
iii) cooperating processes (shared data, no synchronization)
iv) communicating processes (shared data, synchronization)

Proof rules for concurrent execution and restricted communication are given.

[How76a] Howard, J.H.
Proving Monitors
Comm. ACM 19, 5 (May 1976), 273-279

Exemplification of history variables in monitor proofs (see entries [Bri73, OwGr76b]). Howard argues that history variables are essential to capture the relationship between independent program modules, in this case monitor operations.

For a support of assertions that guarantee optimal scheduling, Howard strengthens Hoare's monitor invariant [Hoa74] to imply non-validity of all resumption conditions in the monitor:

$$I \equiv J \wedge E \Rightarrow \neg B(cond) \quad \text{for all queues } cond$$

The new axioms are:

(A1')	"true"	initial statement	"J∧E"
(A2')	"J∧E"	each monitor operation	"J∧E"
(A3')	"J∧E"	cond.wait	"J∧B(cond)"
(A4')	"J∧B(cond)"	cond.signal	"J∧E"

where, for convenience, B(cond) => ¬cond.empty, since cond.signal acts on the empty condition queue like a null statement. The axiom system relies heavily on the use of auxiliary variables.

[How76b] Howard, J.H.
Signaling Monitors
Proc. 2nd Int. Conf. on Software Engineering,
13.-15.10.1976, San Francisco, Cal., 47-52

Formulation of proof rules for five signalling conventions in monitors:

i) signal and wait
ii) signal and urgent wait
iii) signal and continue
iv) automatic signalling
v) signal and return

(i) to (iv) are shown to be equivalent in the sense that they can express the same class of synchronization problems. (v), the policy of Concurrent Pascal [Bri75], is strictly weaker.

Howard introduces a special type of monitor variable that plays the role of a hidden proof variable and internally keeps track of the length of a condition queue. The handling of hidden variables in proof outlines is safer and easier than that of auxiliary variables.

[HoWi73] Hoare, C.A.R.; Wirth, N.
An Axiomatic Definition of the Programming Language Pascal
Acta Informatica 2 (1973), 335-355

Although an axiomatic definition of a particular language, this paper is a good reference for correctness proofs in many algorithmic languages.

The semantics of the common language features (assignment, control structures, etc.) are concise and general. Each of them may apply for other languages, even with slightly different semantics.

The axioms provided here can also guide in the formulation of different axioms for the language of the user.

[JaSt77] Jammel, A.J.; Stiegler, H.G.
Managers vs. Monitors
Information Processing 77, Proc. of the
IFIP Congress, 8.-12.8.77, Toronto, 827-830

[KaMa75] Katz, S.M.; Manna, Z.
A Closer Look at Termination
Acta Informatica 5 (1975), 333-352

[Kel76] Keller, R.M.
Formal Verification of Parallel Programs
Comm. ACM 19, 7 (July 1976), 371-384

[Lam77] Lamport, L.
Proving the Correctness of Multiprocess Programs
IEEE Trans. on Soft. Eng. SE-3, 2 (Mar. 1977), 125-143

[Len77] Lengauer, C.
Strukturierter Betriebssystementwurf
mit Concurrent Pascal
HMI-B 236, Hahn-Meitner-Institut für
Kernforschung Berlin GmbH, July 1977, 73 p.

[Lip75] Lipton, R.J.
Reduction: A Method of Proving
Properties of Parallel Programs
Comm. ACM 18, 12 (Dec. 1975), 717-721

[Lis77] Lister, A.
The Problem of Nested Monitor Calls
ACM Operating Systems Review 11, 3 (July 1977), 5-7

[Lizi75] Liskov, B.H.; Zilles, S.N.
Specification Techniques for Data Abstractions
IEEE Trans. on Soft. Eng. SE-1, 1 (Mar. 1975), 7-19

The purpose of formal specifications for data abstractions is to describe the abstract object in a concise, complete and unambiguous manner by a set of axiomatic assertions that can be presumed in proofs of programs which use the data abstractions. That the presumptions are met by the implementation has to be shown in the correctness proof of the module representing the data abstraction.

The advantages of formal specifications are stressed: as a typical "problem-oriented" concept it is helpful in the design, implementation, verification and documentation of data abstractions and their accessors.

The characteristics of specifications of data abstractions are illustrated at a stack specification.

Different specification techniques are assessed:

i) Use of a fixed description discipline
ii) Use of an arbitrary description discipline
iii) Use of a state machine model
iv) Use of axiomatic descriptions
v) Use of algebraic descriptions

The assessment criteria are:

i) Formality (mathematically sound notation)
ii) Constructibility (support of design process)
iii) Comprehensiblity (documentary effect)
iv) Minimality (conciseness and completeness)
v) Appliciability (for a large class of concepts)

[Man74] Manna, Z.
Mathematical Theory of Computation
McGraw-Hill, 1974, 448 p.

[MaWa78] Manna, Z.; Waldinger, R.
The Logic of Computer Programming
IEEE Trans. on Soft. Eng. SE-4, 3 (May 1978), 199-229

The paper gives an overview of techniques for the development and verification of sequential algorithms. As example serve several implementations of the gcd-function. Four aspects are investigated:

i) partial correctness
ii) termination
iii) program transformation and optimization
iv) program development

The paper contains an extensive bibliography and hints for further research.

[OwGr76a] Owicki, S.S.; Gries, D.
An Axiomatic Proof Technique for Parallel Programs I
Acta Informatica 6 (1976), 319-340

Semantic restrictions for the concurrent statement are formulated that allow communication and synchronization between concurrent processes without specification of shared variables in a resource [OwGr76b]. This language is more powerful than Hoare's [Hoa72a]. Communication is controlled by way of an await statement which defines what Owicki will later call an "elementary action" [Owi77b]. Here it is conditional, i.e., may include a synchronization condition. Syntax:

await B then S;

For this construct, non-interference is harder to

express than for the concurrent statement with resource [Hoa72a, OwGr76b]. A set of processes is interference-free iff there is no statement in anyone of them that interferes with the proof of the others, i.e., whose execution changes assertions in the proof of some other process in the set. The interference argument has also to cover termination, which can be shown with Owicki's axiom system [OwGr76b].

Owicki argues that the comparison of process proofs rather than their execution greatly clarifies the verification of concurrent algorithms. Her standard proof policy is to prove the partial correctness of each concurrent process, verify non-interference, and make a termination argument for the concurrent statement.

[OwGr76b] Owicki, S.S.; Gries, D.
Verifying Properties of Parallel Programs:
An Axiomatic Approach
Comm. ACM 19, 5 (May 1976), 279-285

Hoare's axiom system for the concurrent statement with resource [Hoa72a] is relaxed concerning the use of variables in assertions and extended by an axiom that enables the use of auxiliary variables.

An auxiliary variable is defined as a variable x that appears in the program only in assignment statement of the form x:=E, where the expression E may contain any auxiliary or program variable. The axiom states that an assertion which holds for the program including auxiliary variables and does not refer to the latter, is also valid for the program without auxiliary variables.

Mutual exclusion and blocking in resp. termination of concurrent statements can be proven by

looking at specific expressions of assertions in the partial correctness proofs of the sequential actions in the concurrent statement and the resource invariant. Owicki argues that a general method of proving concurrent algorithms correct has been found, whose appliciability, however, depends on the synchronization facilities in the used programming language.

[Owi77a] Owicki, S.S.
Specifications and Proofs for Abstract Data Types in Concurrent Programs
Tech. Rep. 133, Stanford Electronics Laboratories, Apr. 1977, 21 p.

Owicki extends the verification of monitors to incorporate the processes that access them. As link between the caller and the monitor, for verification purposes the concept of a private variable is invented.

A private variable is defined in the monitor, but is represented by a private incarnation for every caller of the monitor object. Auxiliary private variables relate not only the callers to the monitor but also to each other. This enables statements about the monitor's scheduling properties if one does not want to use the restricted monitor proof rules of [How76a, How76b].

The monitor is incorporated in the concurrent statement as replacement of the resource declaration.

[Owi77b] Owicki, S.S.
Verifying Concurrent Programs with Shared Data Classes
Tech. Rep. 147, Stanford Electronics Laboratories, Aug. 1977, 20 p.

Realizing the monitor problems (see entry [Hoa74]), Owicki defines a general shared class which allows the programmer to specify the exclusion mode between any two statements in its operations. The synchronization tool is the semaphore.

To simplify verification, the format of shared class operations is restricted to an enter; operate; exit structure, where the atomic (Owicki calls them elementary) actions enter and exit only access class variables that aid synchronization (so-called control variables). The operate part only accesses variables that characterize the data object as it is viewed from the callers (so-called data variables).

Proof rules are given which include termination, interference and calls between different class objects (access hierarchies). Interference, not part of monitor proofs because of the general exclusion in monitors, turns out to be a problematic aspect. Practical hints for a safe non-interference argument and sophisticated rules for the calling environment are given. Examples include the resource allocator and readers & writers.

[Par72] Parnas, D.L.
On the Criteria Used in Decomposing Systems into Modules
Comm. ACM 15, 12 (Dec. 1972), 1053-1058

[Par74] Parnas, D.L.
On a "Buzzword": Hierarchical Structure
Information Processing 74, No. 2 (Software), Proc. of the IFIP Congress, 5.-10.8.74, Stockholm, 336-339

[Par76] Parnas, D.L.
On the Design and Development of Program Families
IEEE Trans. on Soft. Eng. SE-2, 1 (Mar. 1976), 1-9

[SKB77] Silberschatz, A.; Kieburtz, R.B.; Bernstein, A.J.
Extending Concurrent Pascal to
Allow Dynamic Resource Management
IEEE Trans. on Soft. Eng. SE-3, 3 (May 1977), 210-217

An abstract data type for the dynamic allocation of resources, the manager, is proposed. The approach is implemented in the environment of Concurrent Pascal [Bri75].

The idea is to break up the static access scheme of the abstract data type language in a safe and structured manner. Processes state an access right to the manager which incorporates the pool of allocatable resources. During program execution they may request resources from the pool, will eventually get a temporary access right (capability) granted and release it after use.

The elements in the pool are indistinguishable outside the manager, unless identifications are passed as parameter of the manager procedures. For the management of private data objects, two standard operations, bind and release, to be used inside the manager prevent multiple allocation.

The manager is a solution to the monitor hierarchy problem (see entry [Hoa74]) for two-level hierarchies. Multi-level hierarchies remain problematic.

UNIVERSITY OF TORONTO

COMPUTER SYSTEMS RESEARCH GROUP

BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
J.J. Horning and W.R. Lalonde, September 1970
[ACM SIGPLAN Notices, November 1970]

CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
W.R. Lalonde, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
J.D. Gorrie, February 1971 [M.A.Sc. Thesis, EE 1971]

* CSRG-4 DYLAN USER'S MANUAL
P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
Alan C.M. Brown and J.J. Horning, March 1971

CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
Richard C. Holt, April 1971
[Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
John Neill Thomas Potvin, August 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED ANIMATION SYSTEM
Kenneth B. Evans, January 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
William Gregg Alexander, February 1972
[M.Sc. Thesis, DCS 1971; Computer, v.8, n.11, November 1975]

CSRG-11 PROJECT SUE STATUS REPORT
J.W. Atwood (ed.), April 1972

* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL
Rupert Bramall, April 1972 [M.Sc. Thesis, DCS, 1971]

* CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD
Lewis R. James, May 1972 [M.Sc. Thesis, DCS, 1972]

+ Abbreviations:
DCS - Department of Computer Science, University of Toronto
EE - Department of Electrical Engineering, University of Toronto
* - Out of print

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING
Kenneth C. Sevcik, May 1972
[Ph.D. Thesis, Committee on Information Sciences, University of Chicago, 1971; JACM, January 1974]

CSRG-15 PROCESS STRUCTURING
J.J. Horning and B. Randell, June 1972
[ACM Computing Surveys, March 1973]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMTION OVERHEAD IS NOT NEGLIGIBLE
Kenneth C. Sevcik, June 1972
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

* CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE
K.C. Sevcik et al, September 1972
[Proceedings AFIPS Fall Joint Computer Conference, v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN
David B. Wortman, December 1972
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING
R. Kvaternik, December 1972 [M.Sc. Thesis, DCS, 1972]

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS
G.G. Kalmar, January 1973 [M.Sc. Thesis, DCS, 1972]

CSRG-23 COMPILER STRUCTURE
W.M. McKeeman, January 1973
[Proceedings of the USA-Japan Computer Conference, 1972]

* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
Eleanor A. Lester, April 1973 [M.Sc. Thesis, DCS, 1973]

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS: AN INITIAL EXPERIMENT
Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON THE REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
Marc Louis Joliat, October 1973 [Ph.D. Thesis, EE 1973]

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
Henry John Pasko, December 1973 [M.Sc. Thesis, DCS 1973]

* CSRG-31 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), Second Edition, March 1974

CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
E.D. Lazowska, May 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
F. Lochovsky and D. Tsichritzis, May 1974 [INFOR, to appear]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
P. Bernstein and D. Tsichritzis, May 1974 [Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974 [Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS
Laurence M. Weissman, August 1974
[Ph.D. Thesis, DCS, 1974]

* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
David M. Lasker, September 1974 [M.Sc. Thesis, DCS, 1974]

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
Glenn F. Stewart, September 1974 [M.Sc. Thesis, DCS, 1974]

* CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
J. Klebanoff, F. Lochovsky, A. Rozitis, and D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF RELIABLE SOFTWARE
David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
B.L. Clark and F.J.B. Ham, September 1974

CSRG-43 A DATA BASE PROCESSOR
E.A. Ozkarahan, S.A. Schuster and K.C. Smith, November 1974 [Proceedings National Computer Conference 1975, v.44, pp.379-388]

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A COMPUTING ENVIRONMENT
Eric C.R. Hehner, November 1974 [Ph.D. Thesis, DCS, 1974]

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN, DYADIC SPECIFICATION, COMPLEMENTARY SEMANTICS
J.E. Donahue, J.D. Gannon, J.V. Guttag and J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM DECISION TABLES
Helmut Schumacher, December 1974
[M.Sc. Thesis, DCS, 1974]

CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
John D. Gannon, January 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
Christopher J.M. Turnbull, January 1975
[Ph.D. Thesis, EE, 1974]

* CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
D. Tsichritzis, February 1975 [in Data Base Description, Dongue and Nijssen (eds.), North Holland Publishing Co.]

* CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
M. Brodie (ed). February 1975 [Proceedings Pacific ACM Conference, 1975]

CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS
David T. Barnard, March 1975 [M.Sc. Thesis, DCS, 1975]

* CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.V. Guttag (ed.), Third Edition, April 1975

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
Richard C. Holt and David B. Wortman, May 1975

CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

* CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL
Hans Albrecht Schmid and J. Richard Swenson, July 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975 [Ph.D. Thesis, DCS, 1975]

CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]

* CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975
[Ph.D. Thesis, DCS, 1975]

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEUPISTICS
Lazlo Sugar, December 1975 [M.Sc. Thesis, DCS, 1975]

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith, February 1976 [Proceedings National Computer Conference 1976, v.45, pp.855-862]

* CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, February 1976 [ACM Transactions on Database Systems, v.1, n:4, December 1976]

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976
[M.Sc. Thesis, DCS, 1975]

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1976 [M.Sc. Thesis, DCS, 1976]

* CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L.Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco
April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard and D. Thompson (Eds.), Fourth Edition, May 1976

* CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D. Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A DATA BASE MACHINE
E.A. Ozkarahan and K.C. Sevcik, May 1976

* CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow, R. Baker and S. Pozgaj, July 1976

CSRG-73 AN ALGORITHMIC APPROACH TO NORMALIZATION OF RELATIONAL DATA BASE SCHEMAS
P.A. Bernstein and C. Beeri, September 1976

CSRG-74 A HIGH-LEVEL MACHINE-ORIENTED ASSEMBLER LANGUAGE FOR A DATA BASE MACHINE
E.A. Ozkarahan and S.A. Schuster, October 1976

CSRG-75 DO CONSIDERED OD: A CONTRIBUTION TO THE PROGRAMMING CALCULUS
Eric C.R. Hehner, November 1976

CSRG-76 "SOFTWARE HUT": A COMPUTER PROGRAM ENGINEERING PROJECT IN THE FORM OF A GAME
J.J. Horning and D.B. Wortman, November 1976

CSRG-77 A SHORT STUDY OF PROGRAM AND MEMORY POLICY BEHAVIOUR
G. Scott Graham, January 1977

CSRG-78 A PANACHE OF DBMS IDEAS
D. Tsichritzis, February 1977

CSRG-79 THE DESIGN AND IMPLEMENTATION OF AN ADVANCED LALR PARSE TABLE CONSTRUCTOR
David H. Thompson, April 1977 [M.Sc. Thesis, DCS, 1976]

CSRG-80 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard (Ed.), Fifth Edition, May 1977

CSRG-81 PROGRAMMING METHODOLOGY: AN ANNOTATED BIBLIOGRAPHY FOR IFIP WORKING GROUP 2.3
Sol J. Greenspan and J.J. Horning (Eds.), First Edition, May 1977

CSRG-82 NOTES ON EUCLID
edited by W. David Elliot and David T. Barnard, August 1977

CSRG-83 TOPICS IN QUEUEING NETWORK MODELING
edited by G. Scott Graham, July 1977

CSRG-84 TOWARD PROGRAM ILLUSTRATION
Edward Yarwood, September 1977 [M.Sc. Thesis, DCS, 1974]

CSRG-85 CHARACTERIZING SERVICE TIME AND RESPONSE TIME DISTRIBUTIONS
IN QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS
Edward D. Lazowska, September 1977
[Ph.D. Thesis, DCS, 1977]

CSRG-86 MEASUREMENTS OF COMPUTER SYSTEMS FOR
QUEUEING NETWORK MODELS
Martin G. Kienzle, October 1977
[M.Sc. Thesis, DCS, 1977]

CSRG-87 'OLGA' LANGUAGE REFERENCE MANUAL
B. Abourbih, H. Trickey, D.M. Lewis, E.S. Lee,
P.I.P. Boulton, November 1977

CSRG-88 USING A GRAMMATICAL FORMALISM AS A PROGRAMMING LANGUAGE
Brad A. Silverberg, January 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-89 ON THE IMPLEMENTATION OF RELATIONS: A KEY TO EFFICIENCY
Joachim W. Schmidt, January 1978

CSRG-90 DATA BASE MANAGEMENT SYSTEM USER PERFORMANCE
Frederick H. Lochovsky, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-91 SPECIFICATION AND VERIFICATION OF DATA BASE
SEMANTIC INTEGRITY
Michael Lawrence Brodie, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-92 "STRUCTURED SOUND SYNTHESIS PROJECT (SSSP):
AN INTRODUCTION"
by William Buxton, Guy Ferdorkow, with
Ronald Baecker, Gustav Ciamaga, Leslie Mezei
and K.C. Smith, June 1978

CSRG-93 "A DEVICE-INDEPENDENT,GENERAL-PURPOSE GRAPHICS
SYSTEM IN A MINICOMPUTER TIME-SHARING
ENVIRONMENT"
William T. Reeves, August 1978
[M.Sc. Thesis, DCS, 1976]

CSRG-94 "ON THE AXIOMATIC VERIFICATION OF CONCURRENT ALGORITHMS "
Christian Lengauer, August 1978
[M.Sc. Thesis, DCS, 1978]