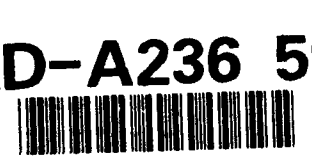
AD-A236 597



(2)

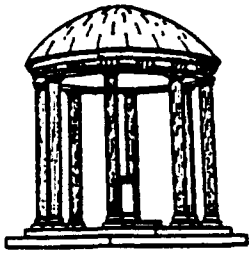
Prototyping Parallel and Distributed Programs in *Proteus*

TR90-041

October, 1990

Peter H. Mills, Lars S. Nyland, Jan F. Prins,
John H. Reif, Robert A. Wagner

DTIC
ELECTE
JUN 13 1991
S D
D



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

UNC is an Equal Opportunity/Affirmative Action Institution.

91-01694

91 6 10 033

Prototyping Parallel and Distributed Programs in *Proteus*

Peter H. Mills†, Lars S. Nyland*, Jan F. Prins†, John H. Reif*, Robert A. Wagner*

*Department of Computer Science, Duke University,
Durham, N.C. 27706

†Department of Computer Science, University of North Carolina,
Chapel Hill, N.C. 27599-3175 USA

Abstract

Prototyping is an important technique in software development for early exploration and validation of requirements. When prototyping concurrent behavior, we must be able to embrace a wide spectrum of models used to construct parallel programs, reflecting a variety of underlying system architectures. In this paper we present *Proteus*, a language suitable for prototyping parallel and distributed programs. *Proteus* starts with the high-level set-theoretic notations of SETL and REFINE. We then extend this base with the barrier-merge parallel construct, which partitions the variables used for communication in its shared-memory model into *shared* and *private* sets. Each parallel process receives an independent copy of the private variables. These private copies are independently updated, and may be *merged* into the global state at specifiable barrier synchronization points: at these points a portion of the merged state may be reflected back into each private state. We envision a layered language structure to express the various programming models, such as communicating sequential processes and data-level parallelism, in terms of this common foundation. A common foundation also facilitates the prototyping of heterogeneous systems whose concurrent parts are programmed following different models.

1. Introduction

1.1. Motivation

In this paper we describe *Proteus*, a language especially suited for prototyping algorithms and programs for parallel and distributed environments. What we present here is very much ongoing work, one part of a group effort by five research teams in a DARPA-sponsored program to develop a Common Prototyping Language (CPL) and Common Prototyping System (CPS). In its current incarnation, *Proteus* provides a high-level set-theoretic notation together with a sparse but powerful set of mechanisms for controlling parallel execution, relying fundamentally on an underlying shared-variable model of concurrency. These mechanisms serve to support diverse concurrent programming styles within a single logical framework. Such a common foundation for concurrency is an important goal, especially within the realm of prototyping. To understand why this is so, and our overall language design goals, we must understand the special needs of prototyping in general.

‡This research was supported by the Defense Advanced Research Projects Agency / Information Systems Technology Office under the Office of Naval Research contract N00014-90-K-0004.

DTIC COPY INSPECTED 5

Dist | Avail and/or Special
A-1

Prototyping can play an extremely useful role in the complex process of building large-scale software systems, serving both to rapidly *explore* alternatives when evolving an idea of the system requirements and to *validate* the requirements specifications so evolved. This experimentation and feasability demonstration helps to identify our needs and to assess and manage risk in the software engineering process. As a vehicle for such early exploration and validation, a *prototyping language* must satisfy a number of needs unique to the broad spectrum of prototyping activities.

Firstly, it should *facilitate rapid programming* by providing the user with an integrated set of language features which are flexible, powerful, and easy to use and understand. We can achieve this expressive power and semantic simplicity by using a very high-level language with a small number of simple features. Moreover, since the intent is not to produce production-quality programs, we may to some degree sacrifice efficiency for this ease of programming. On the other hand, however, we do want to *refine* the experimental prototype into the production version. This desire for graceful transition goes beyond earlier "spiral process models" of sofware development which — although they viewed prototyping as playing an important role by cyclically embodying, testing, and evolving the requirements throughout the software life cycle — still threw away the prototypes before coding the product [Boeh85]. To support refinement as well as early executable specification, the high-level dialect must not be constrained by translatability to a particular target production language or architecture. Furthermore, in keeping with the caveat that we should whenever possible assemble from existing building blocks rather than create anew (the *component-based* approach), we must allow the incorporation of possibly heterogeneous existing applications into a new prototype. This requires some provision for interfacing differing modules, and the ability of the prototype to express its algorithms at some common fundamental level. Lastly, we must provide for the expression of complex execution stategies including *time-constrained*, *distributed*, and *parallel* execution.

1.2. Varieties of concurrent programming

It is particularly important to be able to handle concurrency within a common prototyping language. The prototyping community needs to be able to prototype inherently concurrent behavior, to express algorithms and new concepts of parallel computing, and to evaluate the performance of concurrent systems. We also need to be able to write programs for new parallel machines, so as to use the prototyping language as an exploratory vehicle for new technology.

Yet devising a single framework for handling concurrency is no easy task. Over the past twenty years a great variety of parallel **machine architectures** have been proposed or developed that are of importance. These machines range the gamut from synchronous to asynchronous execution of process events, and from shared to distributed address spaces [Bal89]. There have also been proposed a variety of abstract **theoretical models** of computation to express and analyze algorithms for large classes of machines (e.g., the PRAM [Fort78]). Not surpisingly, many different **languages** are currently used to construct parallel programs, reflecting this diversity of underlying machine models. Each of the specific language features for parallelism — for specifying parallel execution and how parallel computations are mapped onto physical processors, for synchronization, for communication, and for exception handling — mirror to some extent the underlying organization of the machine. For example:

- **Distributed systems** – Applications for loosely-coupled distributed systems, like a collection of workstations connected via an ethernet, are programmed using the concepts of processes and blocking communication by message-passing. Languages for these asynchronous distributed-state systems include OCCAM (CSP) [Inmos87, Hoar85] and Strand [Fost90]. Some languages commonly used for distributed systems may assume a logical model which differs from the physical architecture. For example, Linda assumes a nondistributed state in the form of a "distributed data structure", a tuple space shared among processes.

- **Shared-memory multiprocessors** – Applications for shared-memory multiprocessors, like the BBN Butterfly or the Sequent, are programmed with languages that support shared variables with access-exclusion and synchronization mechanisms like monitors, such as found in Concurrent Pascal [Brin75]. A theoretical model for these asynchronous shared-memory machines might be found in the APRAM [Cole89, Cole90, Gibb89].

- **Highly-parallel processors** – Applications for highly-parallel machines such as the CM-2 or the iPSC are programmed using data-parallel operations and barrier synchronization. Languages used to program existing machines such as the CM and the UltraComputer include specific features that reflect the fundamental organization of the machine. The PRAM presents a family of abstract computational models, based on lock-step execution and synchronously updated shared memory, for this class of machines [Fort78].

1.3. Towards a common foundation

This proliferation of machines and programming languages for parallel computing creates a particularly strong need for a common prototyping language in which parallel applications can be developed relatively independently of the target machines, and specialized to reflect particular target machines as desired. We have developed a common foundation for these various machine models. Since the foundation is small and spartan; a layered language is envisioned in which higher-level control abstractions can be built up syntactically using syntax extension features of the notation and semantically by implementation in terms of the common foundation. The spartan set of control primitives together with the higher-level extensions allow us to *accomodate elements of each style.* A common foundation also facilitates the *prototyping of heterogeneous systems*, such as a loosely coupled system containing Butterflys and Connection Machines linked over the Darpanet, whose concurrent parts must currently be programmed following different models.

1.4. Our approach

Our language starts with rich data models and operators along the lines of SETL [Schw86, Baxt89] and REFINE [Refine88], which employ the high-level mathematical notions of sets, tuples (or sequences), and maps (or relations). REFINE also provides metaprogramming capabilities for syntactic language extension and transformation.

We then extend this base by allowing statements and expressions to be first-class objects, that is, to be themselves values in the data model. This permits us to express many notions of execution-

control in terms of operators over sequences of statements. We will see that constructs for alternative, repetitive, nondeterministic and probabilistic execution may all be expressed in this fashion.

Next we augment this framework with a foundation for parallel programming that relies on a shared-memory logical model. The starting point for our effort was the UNITY notation whose goal is also to unify several disparate notions of parallel programming by reducing them to a common foundation [Chan86]. UNITY relies on a shared-memory model in which each assignment statement is an atomic indivisible action, and treats a program as a collection of these assignment statements which execute chaotically until the state is stable. This reduces concurrent execution to fair interleavings of statements. UNITY thus addresses the problem of **interference** — that events in one processes' concurrent execution can step in and alter the expected outcome in another processes' computation — by enlarging the granularity of **atomicity** — that is, the size of effectively indivisible actions. These restrictions ensure processes interact in a disciplined way, and thus makes it easier to reason about concurrent program behavior.

Our goal –support for rapid prototyping– is, however, rather different from the goals of UNITY which address correctness of parallel programs developed from a specification. In the development of a prototype we typically do not start with a complete specification; a trial program is our best characterization. Hence we are concerned with expressability of programs, and to this end the minimal control and synchronization concepts of UNITY are too low-level for our needs. Thus, while we maintained some fundamental concepts from UNITY, the lowest level of our CPL provides more concrete control constructs that encompass different notions of concurrency more directly (an approach that UNITY correctly identifies as implementation-level detail) but which obviate the need to necessarily express concurrent execution in terms of chaotic execution. Furthermore, our parallel constructs, unlike those of UNITY, control interference of unprotected shared-variables without resorting to overly constraining statement-level atomicity.

In a nutshell, our language features for parallelism rely on one simple parallel composition operator, "||", which specifies **"cobegin/coend"**-like parallelism unconstrained by any restrictions on atomicity or temporal order of component execution. Communication between concurrent processes is through shared variables. We then augment this model by providing a small set of mechanisms which can carve the initial global state into shared and private variables, each process receiving an independent copy of the private state. These private copies are independently updated, and may be "merged" back into the global state at specifiable barrier synchronization points: at those points a subset of the merged state may be reflected back into each private copy. We call this the **barrier-merge** model.

In the rest of this paper we present the technical details of our language. First we give a brief summary of the data types and sequential control constructs. We then discuss our basic control constructs for parallelism. Our language is then used to express the Shiloach-Viskin algorithm for deriving the connected components of a graph. This example serves well to show how we can easily capture the CRCW model of PRAM. Next we examine how our construct, although embodying a logical model of nondistributed state, can be extended to express distributed message-passing. We then conclude by discussing unresolved issues and directions for future research, in particular the exploration of how *control* constructs for concurrency, which are the focus of this paper, can be integrated with the *type* system for our language being developed by Allen Goldberg and colleagues

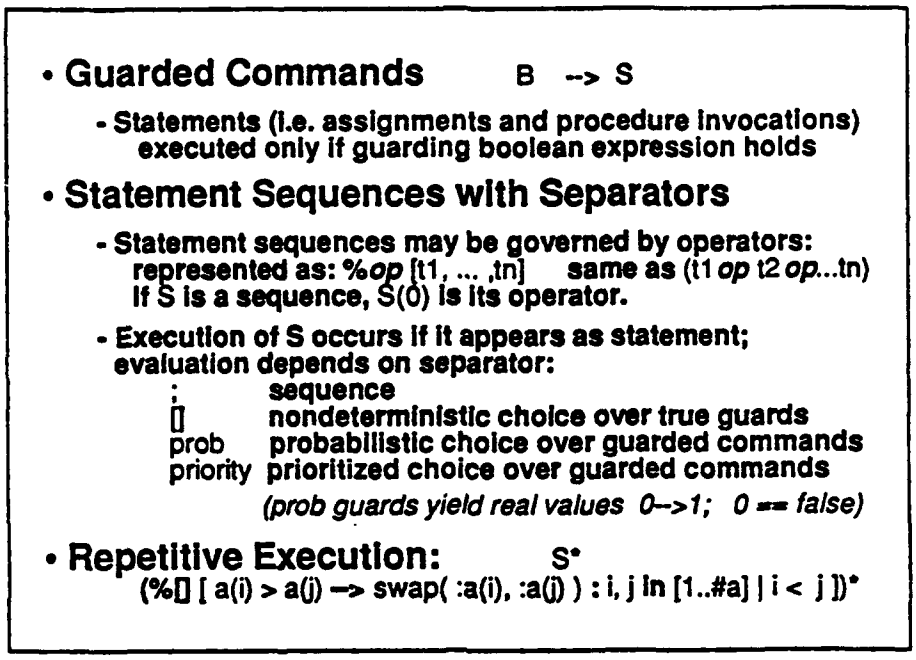


Figure 1: Control Primitives for Sequential Programs

at the Kestrel Institute of Palo Alto, the third member of our CPL team.

2. Constructs for sequential programs

We will now give a brief introduction to some of the basics of our language, just enough so that the notation and examples for concurrency given later will be understandable. The sequential core of our language is conventional to the degree that it is assignment-based and block-structured; in other words, program state is maintained in typed, scoped variables, and assignment statements modify the state. However, *Proteus* uses the high-level data types and operations from SETL [Schw86] and REFINE [Refine88]. Composite data values include sets (denoted by "{}"), tuples (finite ordered sequences of arbitrary elements, denoted by "[ ]"), and pointwise-updatable maps (sets of homogeneous 2-tuples). Sets and tuples may be generated by enumeration or by relative abstraction of the form

[expr(x): x in set | pred(x)] (called a **tuple-former**).

For example, letting S be the set 0,1,2,3,4,5:

[i*i: i in S | (i < 3)] is [0,1,4]

Expressions include composite operations such as APL-like *reduction*:

(%op S) denotes **reduce**: %+[1,2,3,4] is (1+2+3+4) which is 10

as well as *scan*, which computes a sequence of reductions over all prefixes:

(%%op S) denotes **scan**: %%+[1,2,3,4] is [1,3,6,10]

(We note that these examples, and those to later appear, are expressed with a provisional syntax that is likely to change.) Automatic extension of scalar operators to sequences is also supported:

3 + [1,2,3] is [4,5,6], and [1,3] - [1,4] is [0,-1].

Furthermore, statements can be packaged into functions which employ a call-by-value-return protocol. As in ISETL [Baxt89], high-order functions are supported.

However, unlike SETL or ISETL we allow expressions and statements to be first-class citizens. This allows the expression of familiar control primitives, such as ";", in terms of reductions over sequences of statements. Figure 1 summarizes how this technique, together with guarded commands and repetitive execution, yields a powerful control regime. The example at the bottom of Figure 1 illustrates nicely how the powerful predicative constructions of the data model can specify statements to be executed and to control the order of their execution only as necessary. This example performs a sort of elements in the sequence a. It does this by first using a tuple-former to generate guarded statements for all the possible combinations of i and j. UNITY provides a similar but weaker technique for the generation of sequences of statements: we generalize here to use the powerful tuple-formers described previously. Returning to the example, with the reduction operator we then insert the "[]" choice operator between the guarded commands, and then repetitively execute the result until all guards are false. The behavior of the guards and the "[]" and "*" operators is consistent with Dijkstra's Guarded Command Language [Dijk78] — that is, a sequence of guarded commands separated by "[]" will have one of the commands whose guard is true nondetermistically selected for execution. One notable feature is that, like PCN [Chan90] and LISP [McCa60], we represent the reduction operator as the first element in the sequence of statements to which is to be applied. *This is convenient for purposes of metaprogramming.*

3. Constructs for concurrency: the barrier-merge model

Having presented this small but expressive set of primitives, we now turn to extensions for specifying parallel execution and for regulating synchronization and communication. An overview of our approach is shown in Figure 2. As a foundation for our parallel programming, we first adopt a shared-memory model, where communication of parallel processes is through shared-variables.

3.1. Near-free parallel composition

We postulate only one notion of concurrent composition. For statements P1 and P2, the parallel composition (P1 || P2) specifies concurrent execution of the two statements which we call processes. No assumptions about the relative rates of progress in P1 and P2 are made. That is, we place no additional constraints on the temporal ordering of events that constitute the parallel execution of P1 and P2, beyond those implied by explicit synchronization commands. This yields, under our model of execution, a view of the process (P1 || P2) as a collection of events that inspect and modify a shared state *partially ordered* by constraints on their temporal precedence or simultaneity. This lack of constraint is very close in spirit to the parallel composition operator of PCN [Chan90], which also makes no assumption about atomicity or interleaving.

However, we must slightly qualify this assertion. Physical atomicity in execution occurs at

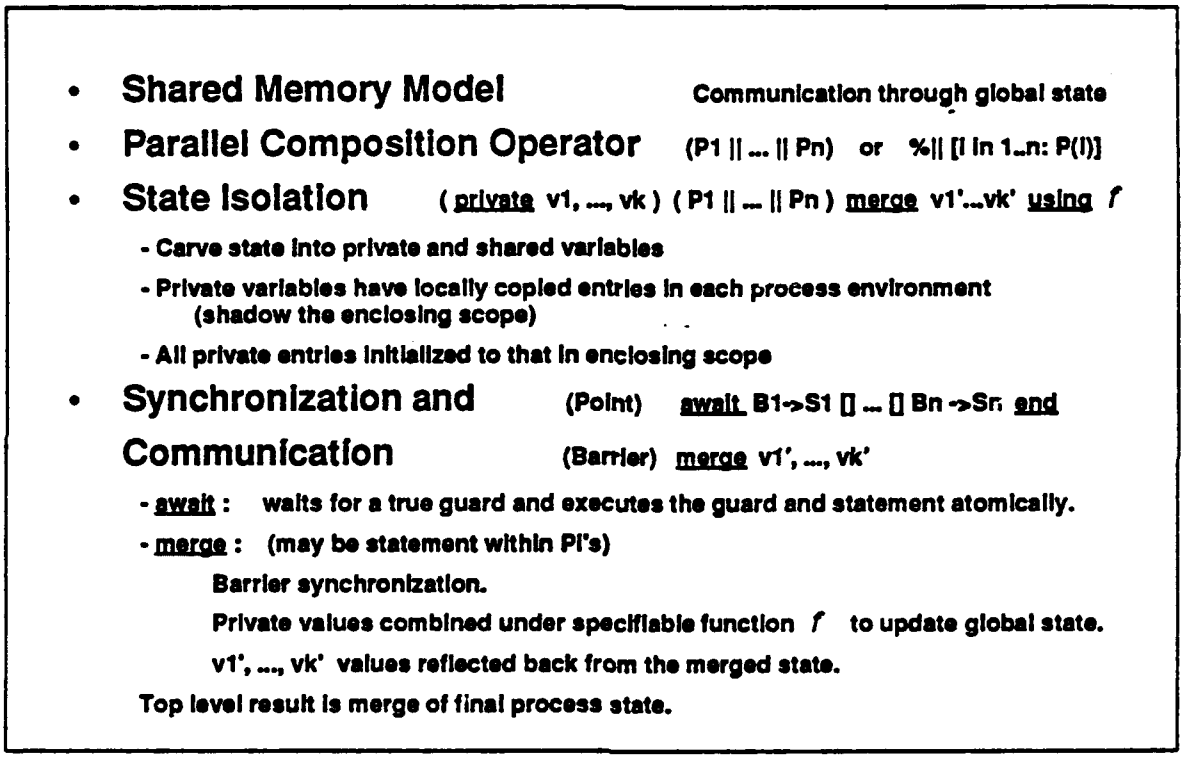


Figure 2: Approach to concurrency

some level of granularity, usually at the memory reference level. We choose to reflect this fact in our language by imposing one natural constraint — that any event within the partially-ordered event-set of (P1 || P2) which is a memory-reference is also an atomic operation. We impose this qualification for scalars and underlying scalar cells of tuples and sets, but make no such claim for atomicity when referring to whole data aggregates. To control interference in this case, techniques to be described later must be used.

3.2. State isolation

Accessibility of variables to P1 and P2, as well as synchronization, is specified through syntactic mechanisms independent of the "||" operator. We achieve controlled access by modeling a division of state into distributed and shared memory through the introduction of **private** and **shared** variables. Our technique exploits the standard scope rules for block-structured languages, where the set of variables in the state that are accessible is determined statically. Within a parallel composition (P1 || P2) each process can reference any variable visible according to these scope rules; but now each variable is specified to be in one of two disjoint sets: private and shared variables. A **shared** variable is a single entry in the state, whereas a **private** variable has an entry in each process in the construct which shadows the entry in the enclosing scope. The initial value of a private variable v is the same in all processes in the construct and is the value of v in the enclosing scope. Operations on shared variables may interfere with each other since they all refer to the same variable. Operations on private variables can never interfere. This is in contrast to PCN which prevents interference by constraining updatable shared variables to be write-once ("permanents").

Figure 3 illustrates how this concept of private variables naturally fits with standard scoping rules. We assume that by default all variables are shared, and must specify the names of private exceptions. In this example, the shared variable a is seen by both P_1 and P_2, but private copies of c are held by each. Finally, we also note that these shared and private variables have visibility analogous to the "shared" and "value" declarations in Orca [Bal88]: the difference there is that the "value" declarations are embedded in the programs P_i, and are not specified by naming private exceptions in the parallel composition construct.

3.3. Barrier synchronization and merging state

While we have discussed how we can initially distribute information down from global state to private copies for each process, we have not yet given mechanisms by which processes can communicate information from the private state into global, nor mechanisms by which they can access this new global state when it is shadowed by a private declaration. The method for doing this is a simple primitive combining two-way communication and synchronization. The **barrier-merge** operation

merge *v1',...,vk'*

may be invoked within the programs P_i, and delays the process containing the operation until all other processes in the composition have reached any **merge** operation, irrespective of mismatches in the *v1',...,vk'* annotations. This effects **barrier synchronization.** At this point, each private variable has its values in all processes combined under some specifiable merging function (typically a function that arbitarily chooses a *changed* value from an arbitrary process), and the result updates

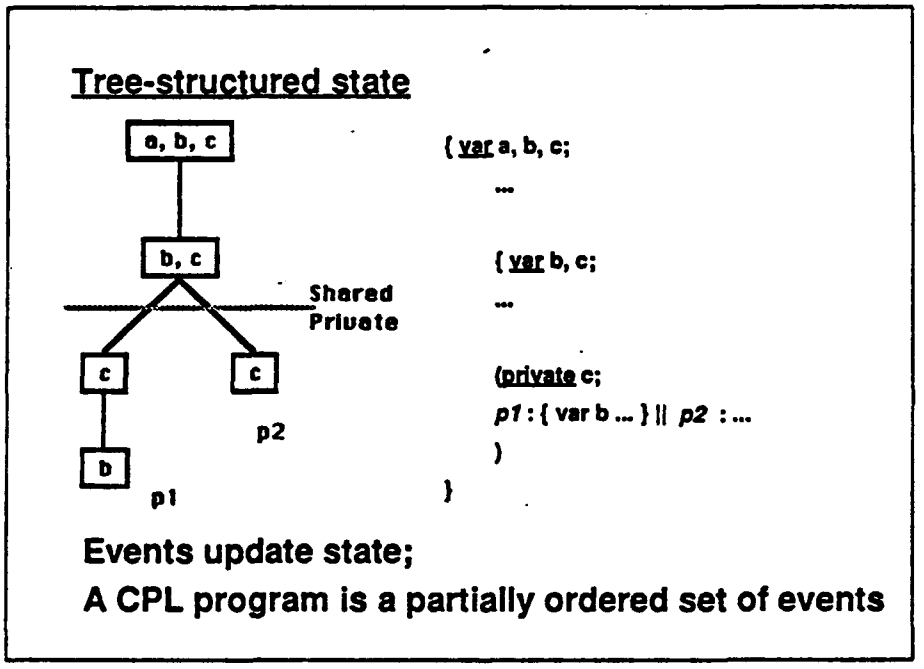


Figure 3: State Isolation through *private* and *shared* variables

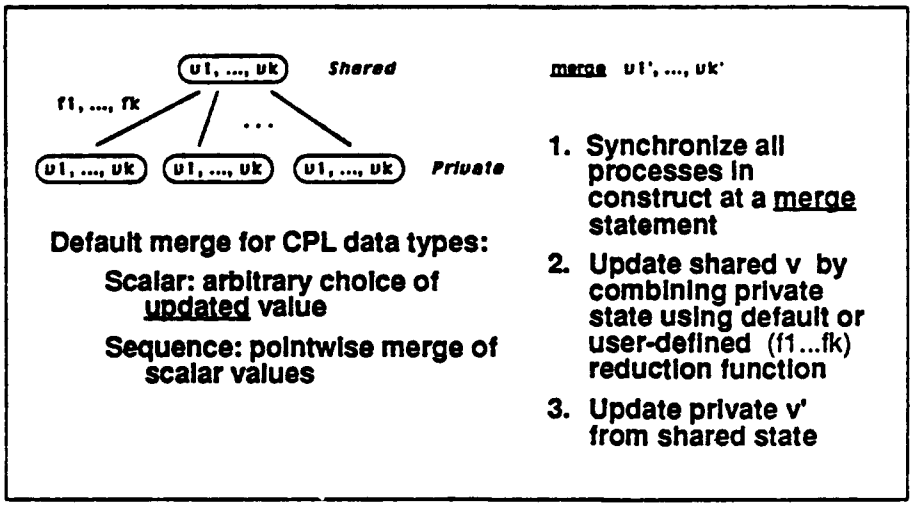


Figure 4: Merging private states into the shared state

the value of the corresponding variable in the enclosing scope. This effects updating the global state from the private states; we also wish to project a portion of this merged state back down into each private state. Now the *v1',...,vk'* annotations, each of which denotes a subset of the private variables, come into play. The private variables *v1',...,vk'* in each process are updated with the value of the shadowed variable in the enclosing scope. If the *v1',...,vk'* specification is omitted, merge assumes by default that *all* privates are to be updated from the enclosing environment. Furthermore, at the top level invocation, the parentheses surrounding a parallel composition give an implied merge operation, so that the final state of a parallel composition is determined by merging the final values of its named private variables. If the final merge or *v1',...,vk'* specification is omitted, *all* private variables are merged into the enclosing environment.

Figure 4 summarizes how barrier-merge works. Note that the exact nature of the merge function f can be explicitly specified at the top level. This causes every merge operation to apply, for each private variable, the *reduction* of the dyadic f across an ordered sequence of all processes' values, yielding the global update. This is similar in spirit to other uses of combining functions to resolve conflict in message collisions [Stee86, Sabo88]. Such user-specifiable merge functions allow handling of aggregate updates in other than default point-wise merge, for example to properly handle set union. Furthermore, we will see in a later example (Shiloach-Viskin) how this merge technique allows the simulation of CRCW PRAM, by letting the merge function be arbitrary selection from updated values. Indeed, by suitably inserting a merge after every statement we can achieve lock-step execution. It is also important to note that our construct generalizes UNITY simultaneous assignment. The UNITY multiple assignment

x, y, z := p(x,y,z), q(x,y,z), r(x,y,z)

indicates that the values for x,y,z are all fetched, after which the expressions p,q,r are evaluated,

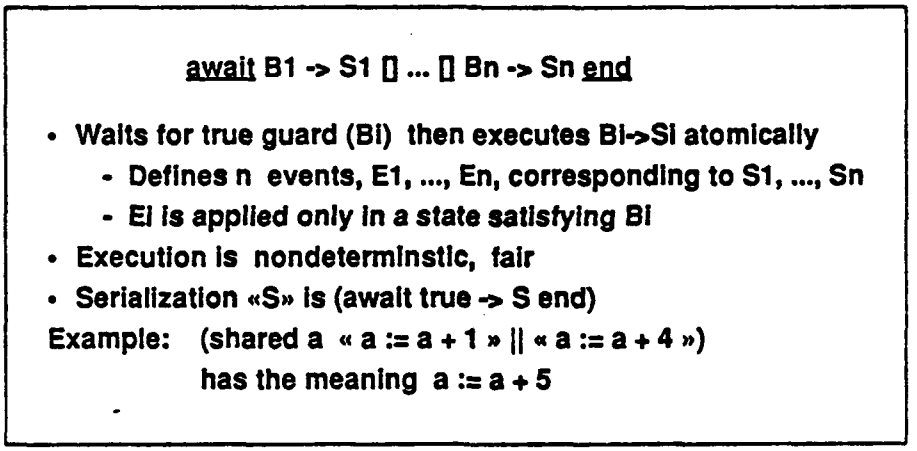


Figure 5: Point synchronization and serialization

and the result is stored. This is just a special case of our parallelism with fully private state:

(private x,y,z) (x:=p(x,y,z) || y:=q(x,y,z) || z:=r(x,y,z))

But more than that, we give a meaning to "x , x := 1 , 3", whereas UNITY requires that only identical values can be simultaneously assigned to the same variable (similar to a Common CRCW PRAM model).

Indeed, having all variables private (simultaneous assignment) or all variables shared (free parallel) represent extrema on a spectrum of what variables are shared between the state. Since sometimes we may want to carve up the space with a majority of private variables, it might be easier to assume that all variables are private and name shared exceptions, instead of dually assuming that all variables are shared and naming private exceptions. This observation leads to a more general technique for exception naming:

allsharedexcept V1,...,Vk
allprivateexcept V1,...,Vk

which encompasses carving from either end.

3.4. Point Synchronization

Our last control primitive for parallelism is a conditional await construct. The "point synchronization" operation:

await B1 − > S1 [] ... [] Bn − > Sn end

delays the process containing the operation until a state is reached in which one of the Bi holds, and then executes the corresponding Si in that state while excluding all other processes. It captures both the concepts of Hoare's conditional wait ("wait(B)") [Hoar74] and of Dijkstra's selective communication [Hoar85]. Furthermore,

await true − > S end

is equivalent to atomic execution of S; as a shorthand we write this as "$\ll s \gg$". Figure 5 summarizes the action of await and shows a simple example of serialization, which incidentally also captures the semantics of UNITY's statement-level atomicity.

4. An example: the Shiloach-Vishkin algorithm

We now give an application of our language in the specification of the Shiloach-Vishkin parallel connectivity algorithm presented in [Schi82] using the CRCW PRAM execution model. The objective of the algorithm is to identify the connected components of an undirected graph G with vertices V and edges E, specifically by assigning the same label to each vertex within a connected component while giving each component a unique label. The execution model assumed by the parallel algorithm falls into the class of *Arbitrary CRCW (Concurrent Read Concurrent Write) PRAM*, that is, all processors have access to a common memory, synchronize after each step of the computation, and resolve conflicts in simultaneous writes by arbitrary selection. In addition to these *synchronization* and *communication* disciplines, the components of the algorithm that are challenging to express within this parallel model are:

- *global control* – to determine termination
- *data parallelism* – the degree of parallelism is determined by the structure of the graph.

Informally, the algorithm as described by Shiloach and Vishkin is as follows. We are given an undirected graph G with n vertices and m edges. We represent the vertices of the graph G as numbers in the range $1..n$, and the edges as a set E of pairs of vertex numbers (both (v, w) and (w, v) are in E if (v, w) is an edge in G). To record the connected components, we use an auxiliary map D which holds, for each vertex, a pointer to another vertex or to itself. D represents a *pointer graph* G' of edges $(v \rightarrow D(v))$ which, although changing throughout execution, will always be a forest of rooted trees plus self-loops. By the end of the algorithm, for each vertex v, $D(v)$ is constructed to point to the smallest numbered vertex in its connected component in G, so that the vertices in each connected component form a rooted star in G'.

The algorithm begins by implicitly allocating a processor to each vertex and to each edge. Each $D(v)$ is initialized to v, thus making each vertex in the pointer graph a root. The algorithm then repeats the following steps until D is stable:

1. *Shortcutting*: paths in G' are halved in length by pointer doubling at each vertex v: $D(v) := D(D(V))$.

2. *Hooking trees onto neighbor's trees*: Each root v in G' (and each of its children v before shortcutting) tries to attach its tree to a neighboring smaller-numbered component reached by some edge (v, w) in G. We "hook" by redirecting the root pointer of the G'-tree to the neighbor w's smaller-numbered G'-parent.

3. *Hooking stagnant trees*: For trees in G' whose roots are *stagnant* (nothing was just shortcut to it nor attached to it), we try hooking roots and children as above, except to *any* other different component, not just smaller-numbered.

4. *Shortcutting* again.

We faithfully express this algorithm in *Proteus* below, capturing CRCW PRAM behavior by using independent-state parallelism and explicit barrier synchronization to combine the independent states at each step. Note that we use the abbreviation of "$x, y := p, q$" for "(allprivate) $(x := p \parallel y := q)$".

```
Algorithm (Shiloach+Vishkin):
  - let V = [1..n] be the vertex labels
  - let E = { (v,w) | v,w in V and ∃ an edge from v to w in G }
  s := t := 1;                  // Iteration number
  D := [i: i in [1..n]];        // Pointer graph: every node initially points to itself
  Dp := [1..n];                 // Previous values of D in step s-1
  Q := [0: i in [1..n]];        // Last step D(i) updated (not stagnant node if =s).
  repeat
   (private D) %|| [                    // Shortcutting
       Dp(i),D(i) := D(i),D(D(i));
       D(i)≠Dp(i) → Q(i) := s
     : i in V ];
   (private D,Q) %|| [                  // Hook trees
       var root,nbor := D(v),0;
       D(v)=Dp(v) and D(w)<D(v) → nbor,D(D(v)) := D(w),D(w);
       merge;
       D(root)=nbor → Q(nbor) := s;  // See if hook chosen in CRCW
       merge;                        // Hook stagnant trees next
       D(v)=D(D(v)) and Q(D(v))<s and D(v)≠D(w) → D(D(v)) := D(w)
     : (v,w) in E ];
   (private D,t) %|| [                  // Shortcutting and detection of termination
      D(i) := D(D(i));
      Q(i)=s → t += 1    : i in V]
   s += 1;
  until (s != t);                       // All stagnant
```

Some issues are brought to light by the example. The first concerns the notion of "merged state". It is semantically appealing to say that the state following the closing barrier of the parallel construct is the pointwise merge of the individual variables, with a (possibly specified) reduction function applied between values of the same variable. But this doesn't quite model the write semantics of the CRCW PRAM, because unmodified values have the same status as modified values. Using the standard combining function **arb** the final value of s in the statement

```
s:=0; [skip || s:=10]
```

could be 0, which could not be an outcome under PRAM execution. This situation comes up in the algorithm: if no process updates Q, then the algorithm can terminate. Under the simple merge rule, we could choose the final state for Q to be from a process that happened not to update Q causing premature termination. Consequently we have to define a somewhat more complex value domain which includes whether a value has been assigned. We might do this by augmenting the domain with

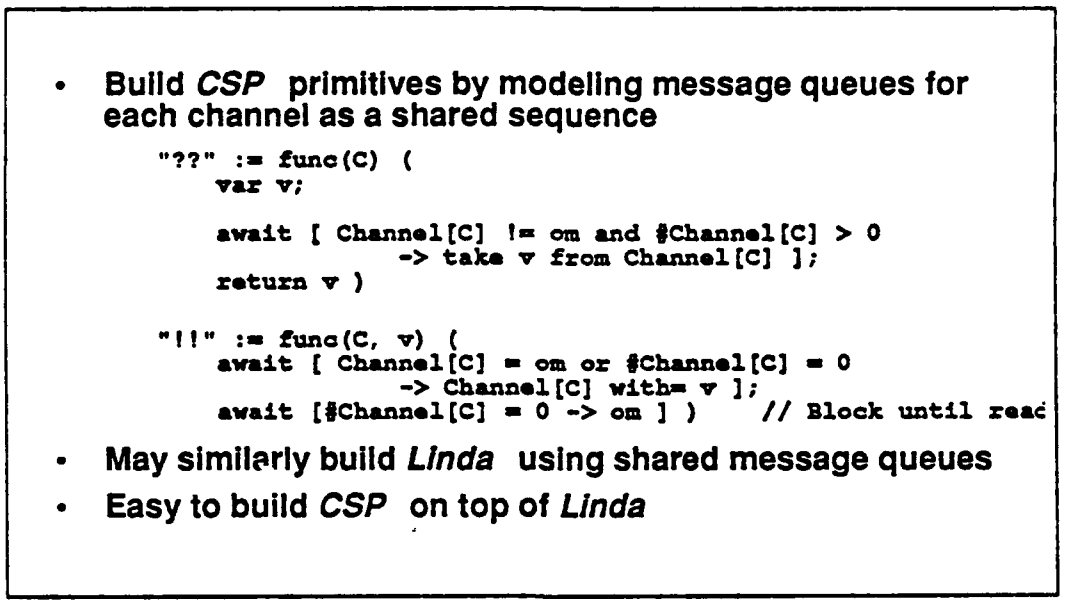


Figure 6: Extensions for distributed message-passing

a new value $\perp$ (undefined) which acts as an identity for the merge function. Alternatively, in the object-oriented style we might provide a "changed" field for each variable, which the user may set and which indicates whether participation in merge occurs. We are investigating these possibilities, as well as whether the merge function should be a function on the processor-indexed sequence of privates, rather than a binary function applied in an induced reduction tree.

A subtle aspect of the algorithm (which led to an apparent program error in [Schi82]) is that, when hooking stagnant trees, even though D(v) may be attached to many vertices, only the actual vertex attached to (as determined by the merge of D) should change its value of Q. We solve this by merging and testing for successful update of D before updating Q.

5. Extensions for distributed message-passing

We developed these barrier-merge primitives after analyzing a variety of different concurrent programs and programming languages. A larger demonstration of their efficacy is their ability to be extended to handle distributed message-passing, in the simplest case to express the blocking communication of CSP. Figure 6 shows how CSP primitives for reading and writing can be easily developed by modeling the message queues for each channel as a shared sequence, and ensuring mutual exclusion with the "await". Like OCCAM, in this example we impose the simplifying restriction that "sends" cannot appear in guards. In a similar fashion we can build Linda using shared message queues and some pattern-matching capabilities of our langauge. Indeed, once Linda is built, it is easy to build CSP on top of Linda [Carr89].

As a final aside, we note that the inclusion of statements as values permits us to express a variety of systems in *Proteus* with a minimum of extra definitions. For example, the single parallel composition operator allows us to express concurrent execution in the form of process parallelism or data parallelism. For statements P1,...,Pn:

```
( P1 || P2 )              is process parallelism
%|| [ P(i) : i in 1..N ]   is data parallelism.
```

6. Summary and future work

In this paper we have introduced some basic features of *Proteus*. We focused in particular on the constructs for expressing parallelism, and presented a simple set of primitives which can serve as the foundation for a layered language which can embrace a wide spectrum of concurrent programming models. Barrier and per-process synchronization permit asynchronous and synchronous parallel programs to be expressed, while distributed and shared memory computation are expressed with the designation of variables as private or shared across a parallel composition. With these facilities we are able to express PRAM and CSP concurrency within a single setting. While we have presented here only an informal semantics for *Proteus*, we are developing a formal operational semantics based on the lambda calculus.

Although we are convinced that we have a reasonable foundation for the construction of a wide spectrum of parallel programs, we question whether a higher-level approach to message-passing, perhaps in the form of concurrent *objects*, might be more appropriate. This concern arises in part because the barrier-merge may be too restrictive in the specification of process lifetimes. Also the issue remains of whether concurrent access to shared variables is too unmanagable; that is, whether unprotected non-private variables are too complicated to handle.

We are thus looking at evolving some form of higher-level shared-value construct based on object-oriented techniques [Agha90]. This effort dovetails nicely with the type system being developed for our language by Allen Goldberg and colleagues at the Kestrel Institute, the third member of our CPL team. Their *Data Type Refinement System* [Blai90] extends the ML type system [Miln84] to include axiomatic description of operators, and a simple form of dependent types that supports a reasonable notion of static checking. A significant challenge has been to incorporate object-oriented paradigms (structural subtyping, overloading) into an ML-style type system. In this area we follow an approach similar to Modula 3. The module construct for our type system generalizes ML's functors and modules; its theoretical basis bears strong relation to the parameterized modules in OBJ3. Techniques for composing modules via such parameterization, as well as by refinement, rely on the concept of *theory interpretations*, which translate modules over one vocabulary to modules over another.

We envision using this module system to encapsulate shared objects, specifying methods which can execute concurrently as well as constraints which enforce required exclusion. This integration of control and types is similar in concept to that employed by Reality of Stanford/TRW (another CPL team), although we as yet have not decided whether to use trigger-based asynchronous message-passing as the basis of concurrent processes interaction, as they do [Belz90].

Finally, *time* plays a very important role in concurrent systems. We are developing some techniques to associate execution time under a given execution model with the execution of a prototype. Hopefully this will enable the specification and evaluation of real-time systems, another critical area of prototyping.

References

[Agha90] Agha, G., "Concurrent Object-Oriented Programming", *CACM*, Vol.33, No.9 (Sept.1990), pp.125-141.

[Bal88] Bal, H.E. and Tanenbaum, A.S., "Distributed programming with shared data", in: *Proceedings of the IEEE CS 1988 International Conference on Computer Languages* (Miami,Fla.,Oct.9-13) (IEEE,New York,1988),pp.82-91

[Bal89] Bal, H.E., Steiner, J.G. and Tanenbaum, A.S., "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys*, Vol.21, No.3 (Sept.1989) pp.261-322

[Baxt89] Baxter, N., Dubinsky, E. and Levin, G., *Learning Discrete Mathematics with ISETL*, (Springer-Verlag) 1989.

[Belz90] Belz, Frank C. and Luckham, David C., "A New Approach to Prototyping Distributed, Time Sensitive Systems", Technical Report (Sept.1990), Stanford University, California, 17pp.

[Blai90] Blaine, L. and Goldberg, A., "Modules and Types for a Common Prototyping Language", Technical Report (Oct.1990), Kestrel Institute, Palo Alto, California. 23pp.

[Boeh85] Boehm, Barry W., "A spiral model of software development and enhancement", *IEEE Software* (May,1985) pp.61-72.

[Brin75] Brinch Hansen, P., "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering*, SE-1 (20), pp.199-207.

[Carr86] Carriero, N. Gelernter, D. and Leichter, J., "Distributed data structures in Linda", in: *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan13-15) (ACM, New York, 1986) pp.236-242.

[Carr89] Carriero, N. and Gelernter, D., "How to Write Parallel Programs: A Guide to the Perplexed", *ACM Computing Surveys*, Vol.21, No.3 (Sept.1989) pp.323-357.

[Chan86] Chandy, K.M. and Misra, J., *Parallel Program Design, A Foundation*, (Addison-Wesley Publishing Company) 1988.

[Chan90] Chandy, K.M. and Taylor, Stephen, "A Primer for Program Composition Notation", Technical Report (June 1990), California Institute of Technology, 93pp.

[Cole89] Cole, R. and Zajicek, Ofer, "The APRAM: Incorporating asynchrony into the PRAM model", in: *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures* (ACM Press, 1989) pp.169-178.

[Cole90] Cole, R. and Zajicek, Ofer, "The Expected Advantage of Asynchrony", in: *Proceedings of the Second ACM Symposium on Parallel Algorithms and Architectures* (ACM Press, 1990) pp.85-94.

[Dijk78] Dijkstra, E.W., "Guarded commands, nondeterminacy and the formal derivation of programs", *Comm. ACM* 18 (1978) pp.453-457.

[Fort78] Fortune, S. and Wyllie, J., "Parallelism in random access machines", in: *Proc.10th Ann. ACM Symp. on Theory of Computing* (1978) pp.114-18.

[Fost90] Foster, Ian and Taylor, Stephen, *Strand: New Concepts in Parallel Programming*, (Prentice-Hall, Englewood Cliffs, NJ) 1990.

[Gibb89] Gibbons, P.B., "A more practical PRAM model", in: *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures* (ACM Press, 1989) pp.158-168.

[Hoar74] Hoare, C.A.R., "Monitors: An operating system structuring concept", *Commun. ACM*, Vol.17, No.10 (Oct. 1974) pp.549-557.

[Hoar85] Hoare, C.A.R., *Communicating Sequential Processes* (Addison-Wesley, Reading, Mass., 1985).

[Inmos87] INMOS Ltd., *The occam programming manual*, Prentice Hall (1987).

[McCa60] McCarthy, J., "Recursive Functions of Symbolic Expressions", *Communications of the ACM*, Vol.3, No.4 (1960) pp.184–195.

[Miln84] Milner, Robin, "A Proposal for Standard ML", in: *ACM Symposium on Lisp and Functional Programming*, (ACM Press, Austin, TX, 1984) pp.184–197.

[Refine88] *Refine 2.0 Language Summary*, (Aug.1988), Reasoning Systems Inc, Palo Alto, California. 15pp.

[Sabo88] Sabot, G., *The Paralation Model: Architecture-Independent Parallel Programming* (MIT Press, 1988).

[Schi82] Schiloach, Y. and Vishkin, U., "An O(log n) Parallel Connectivity Algorithm", *Journal of Algorithms* (3), (1982) pp.57-67.

[Schw86] Schwartz, J.T., Dewar, R.B.K., Dubinsky, E. and Schonberg, E., *Programming with Sets, An Introduction to SETL*, (Springer-Verlag) 1986.

[Stee86] Steele, G.L.,Jr. and Hillis, D., "Connection Machine LISP: fine-grained parallel symbolic processing", in: *Proc.1986 ACM Conference on Lisp and Functional Programming* (Cambride, MA, Aug.1986) (ACM Press, 1986) pp.279-297.