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

This paper presents *Proteus*, an architecture-independent language suitable for prototyping parallel and distributed programs. *Proteus* is a high-level imperative notation based on sets and sequences with a single construct for the parallel composition of processes. Although a shared-memory model is the basis for communication between processes, this memory can be partitioned into *shared* and *private* variables. Parallel processes operate on individual copies of private variables, which are independently updated and may be *merged* into the shared state at specifiable barrier synchronization points. Several examples are given to illustrate how the various parallel programming models, such as synchronous data-parallelism and asynchronous control-parallelism, can be expressed in terms of this foundation. This common foundation allows prototypes to be tested, evolved and finally implemented through refinement techniques targeting specific architectures.

1. Introduction

Prototyping is an important technique in software development for early exploration and validation of requirements. When prototyping concurrent behavior, we face the problem of embracing a wide spectrum of models used to construct parallel programs, reflecting a variety of underlying architectures. One solution is an architecture-independent approach, in which prototypes may be experimentally implemented and then evolved into production versions through refinement techniques which can target specific classes of architectures. In this paper we describe *Proteus*, an architecture-independent language suitable for prototyping algorithms and programs for parallel and distributed systems. What we present here is very much ongoing work, one component of 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 support diverse concurrent programming styles within a single logical framework. Such a common foundation for concurrency, when combined with refinement techniques, proves valuable for prototyping.

1.1. Prototyping

In a recent assessment of the state of software development methodologies, Brooks [Bro87] concluded that one of the most successful approaches to software development was an *evolutionary model* in which software is built incrementally with limited function appearing very early on. This success was attributed in part to the feedback available from the working intermediate products, which are in essence prototypes. These prototypes serve both to rapidly explore alternatives when developing an idea of the system requirements and to validate the requirements specifications so developed. Another key aspect of the methodology is that it evolves prototypes into products. These two facets of prototyping can be supported by a flexible and expressive prototyping language and a refinement system.

It is particularly important to be able to handle concurrency within a common prototyping language. The software development 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.

1.2. Varieties of concurrent programming

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, from shared to distributed address spaces [BST89], and from local to global control. There have also been proposed a variety of abstract theoretical models of computation to express and analyze algorithms for large classes of machines

[‡]This work was supported under DARPA / ISTO contract N00014-90-K-0004 administered through ONR.

In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, (Dallas, Texas, Dec.1-5), pp. 10–19, IEEE, 1991.

(e.g., the PRAM [FW78]). Not surprisingly, 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, such as 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) [INM87, Hoa85] and Strand [FT90]. Some languages commonly used for distributed systems may assume a logical model which differs from the physical architecture. For example, Linda [CGL86] 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 typically programmed with languages that support shared variables with access-exclusion and synchronization mechanisms like monitors, such as found in Concurrent Pascal, or with semaphores such as found in Mach [BRS+85]. A theoretical model for these asynchronous shared-memory machines is found in the APRAM [CZ89].
- **Highly-parallel processors** – Applications for distributed-memory machines such as the CM-2 or the NCube are programmed using data-parallel operations and barrier synchronization. Languages used to program machine designs such as the CM and the UltraComputer include specific features that reflect the fundamental organization of the machine. The PRAM model [FW78] presents a family of abstract computational models, based on lock-step execution and synchronously updated shared memory, for this class of machines.

1.3. Towards a common foundation

The proliferation of machines and programming languages for parallel computing creates a particularly strong need for a common prototyping language in which parallel applications can initially be developed independently of the target machines, and then specialized to run on particular target machines as desired.

We have developed such a common foundation for these various machine models. Our foundation is small and spartan, yet allows for higher-level control abstractions to be built up using type abstraction and syntax extension features. The spartan set of control primitives together with higher-level extensions allows us to accommodate elements of each style. A common foundation also facilitates the prototyping of heterogeneous systems, such as a loosely coupled system containing Crays and Connection Machines linked over a high speed network, 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 [SDDS86, BDL89] and REFINE [Ref88], which employ the high-level mathematical notions of sets, tuples (or sequences), and maps (or relations). We also incorporate metaprogramming capabilities found in REFINE for syntactic language extension and transformation.

We then extend this base by allowing statements 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. 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.

In a nutshell, our language supports parallelism with 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 augment this model by providing a small set of mechanisms which can partition the initial global state into shared and private variables, where each process receives 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. To demonstrate their broad expressive power, these constructs are used to specify the general forms of totally asynchronous Gauss-Seidel relaxation, and the phased-synchronous Jacobi variant. 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 capture the CRCW model of PRAM. We conclude with a discussion of refinement strategies, related work, and directions of ongoing research. A more detailed description of *Proteus* can be found in [Nyl91].

2. Basic features of *Proteus*

The core of our language is a conventional imperative notation to the degree that it is assignment-based and block-structured; program state is main-

• Statements:	*assignments, procedure calls*	
• Guarded commands:	<expr> → <stmt>	
• Operators over statement sequences:		**Syntactic abbreviation**
Sequence:	$\mathbf{seq}\ [\boldsymbol{S}_1, \ldots, \boldsymbol{S}_n]$	$(\boldsymbol{S}_1; \ldots; \boldsymbol{S}_n)$
Choice:	$\mathbf{alt}\ [\boldsymbol{B}_1 \to \boldsymbol{S}_1, \ldots, \boldsymbol{B}_n \to \boldsymbol{S}_n]$	$(\boldsymbol{B}_1 \to \boldsymbol{S}_1 [] \ldots [] \boldsymbol{B}_n \to \boldsymbol{S}_n)$
Repetition:	$\mathbf{rep}\ [\boldsymbol{B}_1 \to \boldsymbol{S}_1, \ldots, \boldsymbol{B}_n \to \boldsymbol{S}_n]$	$(\boldsymbol{B}_1 \to \boldsymbol{S}_1 [] \ldots [] \boldsymbol{B}_n \to \boldsymbol{S}_n)*$
Parallel Composition:	$\mathbf{par}\ [\boldsymbol{P}_1, \ldots, \boldsymbol{P}_n]$	$(\boldsymbol{P}_1 \| \ldots \| \boldsymbol{P}_n)$

Figure 1: **Control primitives in *Proteus***

tained in typed, lexically-scoped variables, and assignment statements or procedure calls modify this state. However, *Proteus* includes a number of high-level predefined data types such as sets and sequences.

Sets and sequences may be constructed by enumeration or by generation based on another set or sequence. Generators are of the form:

{x **in** set | pred(x) : expr(x)} *(set)*

[x **in** sequence | pred(x) : expr(x)] *(sequence)*

Note that, like UNITY, iterators are specified first, followed by predicates and lastly expressions comprising elements in the set or sequence. For example,

{i **in** {0..5} | (i < 3) : i*i}

has value $\{0, 1, 4\}$. Standard operations on sets and sequences are present, such as concatenation and indexing on sequences, and union and arbitrary choice on sets. Also present is the APL-like **reduction** operation f/S which applies a binary function f between the elements of sequence S. For example, $+/[1, 2, 3, 4]$ is $(1+2+3+4)$ or 10. The **scan** operation $f//S$ computes the sequence of reductions over all prefixes of a sequence. We note that these examples, and those that appear later, are expressed using a provisional syntax that is likely to change.

Functions and statements are also values in *Proteus*. For example, the assignment

f := **func**(n) (**return** n+x);

yields as a value for f the closure of the function in the lexically-scoped environment. As a result, higher-order functions such as the reduction operation can be defined directly, as is the case in ISETL [BDL89].

However, unlike SETL or ISETL, statement values can also be formed. This allows the expression of familiar control constructs – such as sequential composition – as operators over sequences of statements, yielding a flexible and extensible control regime. Figure 1 summarizes a number of control operators over sequences of statements and the familiar syntax that may be used when all of the statement values are explicit rather than generated.

The power of combining sequence generators with statement values is illustrated by the following example, with performs a sort of the sequence s.

[i,j **in** [1..#s] | $i < j$:
$s(i) > s(j) \to s(i), s(j) := s(j), s(i)]*$

Given an n element sequence s, the generator produces a sequence of $n(n+1)/2$ guarded commands [Dij78], each of which can exchange a specific pair of out-of-order elements of s. This statement sequence has all guards false precisely when s is an ordered sequence. Hence execution of this statement sequence using the **rep** operator — corresponding to Dijkstra's repetitive construct [Dij78] which repeatedly executes one command selected arbitrarily from those with true guards — will nondeterministically exchange out-of-order elements of s and terminate when the sequence is sorted.

An additional consequence of including statements as values is that it permits a simple representation of a *Proteus* program as a *Proteus* value. Hence, like PCN [CT90] and LISP, this permits metaprogramming.

3. Constructs for concurrency

Having presented some basic concepts of the language we now turn to features supporting the construction of parallel programs.

3.1. Parallel composition

We postulate only one notion of concurrent composition. The statement $(P_1||P_2)$ specifies concurrent execution of the two statements P_1 and P_2 which we call processes. No assumptions about atomicity, interleaving, or relative rates of progress of P_1 and P_2 are made. That is, in our model of execution each process is viewed as a collection of events that inspect and modify a shared state, ordered by temporal constraints such as precedence or simultaneity. In the composition $(P_1||P_2)$ 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 for $(P_1||P_2)$ a *partially ordered* set of events from P_1 and P_2.

This lack of constraint is very close in spirit to the parallel composition operator of PCN [CT90], which also makes no assumption about atomicity or inter-

- Shared Memory Model: *communication through global state*
- Parallel Composition: **par** [(**private** $v_1, \ldots, v_k$ **using** f; P_1) ,..., (**private** $v_1, \ldots, v_k$ **using** f; P_n)]

  or (**private** $v_1, \ldots, v_k$ **using** f **in** $P_1 \| \ldots \| P_n$)
- Synchronization and Communication: **await** $[B_1 \rightarrow S_1, \ldots, B_n \rightarrow S_n]$

  merge $v_{i'}, \ldots, v_{k'}$

Figure 2: **Constructs for concurrency**

leaving. To control interference in our model of unconstrained parallelism, *Proteus* relies on constructs which manage isolation and sharing of state.

3.2. Shared and private state

Accessibility of variables by 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, and local declarations mask surrounding declarations. Within a parallel composition each process can reference any variable visible according to these scope rules; but now each non-local 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 parallel 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, but 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 non-local variables are shared, and hence the names of private variables must be declared in each process. 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 [BT88].

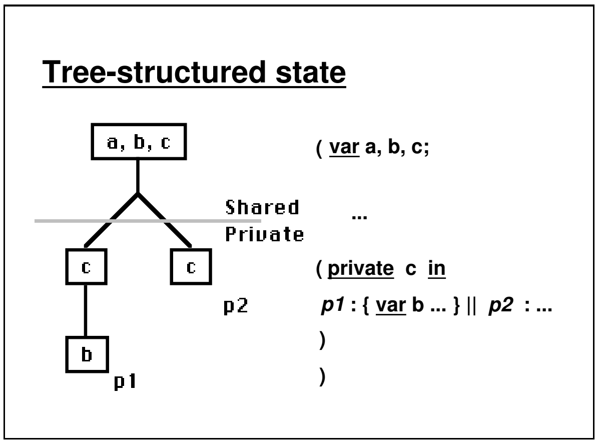

Figure 3: *Private* **and** *shared* **variables**

3.3. Barrier synchronization and merging private 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 $v_{i'}, \ldots, v_{k'}$

may be invoked within the processes P_i, and delays the process containing the operation until all other processes in the composition have reached a **merge** operation. This effects *barrier synchronization.*

At this point, each private variable has its values in all processes combined under some specifiable merging function (for example a function *arb* that arbitrarily chooses one value from among the processes), and the result updates the value of the corresponding variable in the enclosing scope. This effects *updating the global state from the private state* by combining private values using a specified merge function. We then project a portion of this merged state back down into each private state. The private variables $v_{i'}, \ldots, v_{k'}$ in each process are updated with the value of the shadowed variable in the enclosing scope. This effects *reflection* *of global into private state.* If the $v_{i'}, \ldots, v_{k'}$ specification is omitted, *all* private variables are updated from the enclosing environment. Furthermore, an implicit **merge** operation occurs at the end of every parallel composition, so that the final state of a parallel composition is determined by merging the final values of its named private variables.

The exact nature of the merge function f can be explicitly specified through the "**private**...**using** f" declaration. This specifies that every **merge** operation is to apply, for each **private** variable, the *reduction* of the binary operation f across the 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 [Sab88].

When combining a variable's value from each of the processes, a key consideration is whether the variable has changed since the last **merge** operation. Consequently we define the combining function to apply only between the *changed* values. Formally, for the combining operation we augment the value domain with a new value $\perp$ (undefined), indicating unchanged variables, which acts as an identity for every merge function. For example, the program

$s := 5;$

(**private** s **using** arb **in** $[x := 0 \parallel s := 10]$)

must yield 10 instead of 5, since s is not assigned in the first process. With this definition the *arb* combining function models the arbitrary choice write semantics of the CRCW PRAM. Since we are frequently concerned with PRAM algorithms, for the purposes of this paper *arb* is the default merge function.

It is also interesting to note that our private state and merge construct generalizes UNITY simultaneous assignment. The UNITY multiple assignment

$$\mathrm{x, y, z := p(x, y), q(x, y), r(x, y, z)}$$

indicates that the values for x, y, z are all fetched, after which the expressions p, q, r are evaluated, and the result is stored. This is just a special case of our *arb* parallelism with fully private state:

(**private** x,y,z **in**

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

In particular, we can give a meaning to "x, x := 1, 3", whereas UNITY requires that only identical values can be simultaneously assigned to the same variable. We define our multiple assignment statement using the above technique.

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 most variables private, it might be easier to name **shared** exceptions, instead of assuming that all variables are shared and naming **private** exceptions. This observation leads to a more general technique for exception naming:

allsharedexcept $v_1, \ldots, v_k$

allprivateexcept $v_1, \ldots, v_k$

which encompasses carving the state space from either end.

3.4. Point synchronization

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

await $[B_1 \rightarrow S_1, \ldots, B_n \rightarrow S_n]$

waits for a true guard B_i and then executes the guard and statement $B_i \rightarrow S_i$ atomically. In other words, it delays the process containing the operation until a state is reached in which one of the predicates B_i holds, and then executes the corresponding S_i in that state while excluding all other processes. It captures both the concepts of Hoare's conditional wait ("**wait**(B)") [Hoa74] and of selective communication [Hoa85]. Furthermore,

await $[true \rightarrow S]$

is equivalent to atomic execution of S; as a shorthand we write this as $\ll S \gg$. In combination with parallel composition, this can effect interleaving. For example:

$$\ll a := a + 1 \gg \| \ll a := a + 4 \gg$$

has the meaning $a := a + 5$. This illustrates how *Proteus* can readily capture the semantics of UNITY's statement-level atomicity.

The barrier-merge primitives can also serve as a foundation for extensions to other models of concurrency. A larger demonstration of their efficacy is their ability to be extended to express message-passing, in the simplest case the blocking communication of CSP. 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 **await**. In a similar fashion we can build Linda using shared message queues and some pattern-matching capabilities of our language.

4. Expressing synchronous and asynchronous parallelism

We now present a simple example to illustrate the diversity of parallel computations that can be accommodated in *Proteus*. The problem considered is the computation of a fixed point for a vector valued function f. What is required is to find a vector $Y \in R^n$ (for some arbitrary domain R) such that $Y = f(Y)$. This problem is characteristic of a wide range of scientific computations in the solution of linear systems and differential equations.

The technique used is fixed point iteration, i.e. the fixed point Y obtained through repetition of $Y := f(Y)$. In the parallel computation of the fixed point, we distribute the computation using n functions f_i, each of which updates Y_i using $Y_i := f_i(Y)$. The fixed point of f is reached when all f_i have reached a fixed point. The parallel computation can be structured synchronously (Jacobi iteration) in which all components of Y are simultaneously updated, or asynchronously (Gauss-Seidel iteration) in which components are updated one at a time, and the most recently computed values of other components are used.

If we let $f(i,Y)$ represent the computation $f_i(Y)$, then a totally asynchronous parallel version of this computation can be expressed in *Proteus* as follows:

par [i **in** [1..n] : (**true** $\rightarrow$ Y[i] := f(i,Y))*]

where the vector Y is a shared variable. This corresponds to n processes which continuously and independently update their respective components of Y.

Of course this computation does not terminate when the fixed point is reached. To do so it is necessary to detect the fixed point and this will involve some synchronization. A great variety of termination detection approaches have been studied for problems in this class (e.g. [DS80], [CM88]) and here we express one such solution due to Dijkstra that admits a relatively large amount of concurrency in the computation.

```
s := false;
E := [i in [1..n] : false];
par ({ i in [1..n] : Update(i)} with Detect())
```

The computation consists of n Update processes and one Detect process. The process Update(i) is given by

```
( not s →  var q,i,V;
     ≪ D[i] := true ≫;
     seq [j in [1..n] : ≪ V[j] := Y[j] ≫];
     q := f(i,V);
     q = V[i] → ≪ E[i] := D[i] ≫;
     q ≠ V[i] → (
          ≪ Y[i] := q ≫;
          seq [ j in [1..n] :
               ≪D[j] := false≫;
               ≪E[j] := false≫;
          ]
     )
)*
```

and the Detect process is simply

(**not** s $\rightarrow \ll$ s := **and** / E $\gg$)*.

In this solution q,i and vector V are local variables, and all remaining variables are shared. The statements that must be executed atomically by each Update process all involve simple references or assignments to the shared state, a condition that is usually assumed to be enforced in a shared-memory model. Hence the Update processes in this *Proteus* program model an interlock-free implementation on a shared-memory multiprocessor. The Detect process requires exclusion on the evaluation of the reduction **and** / E.

A synchronous determination of the fixed point can also be expressed conveniently in *Proteus*. In the synchronous case we may think of the evaluation of the f_i proceeding in "rounds". After each round, Y is updated at every point. To express this case in the program below we let Y and s be private variables that are merged after each iteration.

```
s := false;
par [ i in [1..n] :
    private s using and; private y using arb;
        (not s →  v := f(i,Y);
            s := ( v = Y[i]);
            Y[i] := v;
            merge;
        )*
    ]
```

The merge required of the private Y variables is the PRAM merge to provide pointwise update of the shared Y (there is no collision, so combining is not an issue), while the merge required for s is to combine all values using logical "and" reduction. The merged value of s controls termination.

The examples above illustrate that both the synchronous and asynchronous formulations of this parallel computation can readily be expressed in *Proteus*.

5. 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 [SV82] 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.

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* in Figure 4, capturing CRCW PRAM behavior by using independent-state parallelism and explicit barrier synchronization to combine the independent states at each step.

In prototyping the above algorithm by transcription into ISETL, it was discovered that although the algorithm correctly yielded connected components, it did not meet the time complexity which was established for it in [SV82]. Examination revealed an apparent program error due to subtleties in CRCW write semantics. Specifically, 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. In the algorithm in Figure 4 we correct this by merging and testing for successful update of D before updating Q in step 2.

6. Execution of prototypes

While programs in *Proteus* should be able to run on parallel platforms, it is not our intention that any single program execute well on all parallel platforms. Early prototypes that explore specifications are likely to be expressed independent of a specific class of platforms, and initially executed on sequential machines. Prototypes can then evolve to use *Proteus* in more restricted ways that are in close correspondence with a particular architecture or programming model. In common with other architecture-independent languages, refinement can help achieve this architectural specialization.

Refinement and architectural specialization: Refinement strategies whereby a program is specialized to a particular subset of the language and mechanisms for the translation of such a subset to run on a parallel platform are being developed in conjunction with our colleagues at the Kestrel Institute (the third member of our CPL team), building on their environments for transformational program development. The KIDS system (Kestrel Interactive Development System) [Smi90] has been used to develop programs from specifications, and includes a number of algorithm design tactics and data refinement transformations [BG90].

We are investigating new tactics to help make explicit the parallelism implicit in high level programs. For example, a tactic to transform the implicit data-parallelism in set and sequence operations to a more explicit form could help in the refinement of such a program to run on a highly-parallel machine. We are looking to data-refinement techniques to effect the change of notation required to yield programs suitable for execution on particular parallel platforms.

Targeting intermediate languages: Providing refinement techniques to target many specific architectures is likely to be prohibitive, hence our strategy is to refine to existing or proposed intermediate languages which permit execution on a broad class of parallel platforms. For example, we intend intially to reduce data-parallelism to the set of parallel vector operations provided by the CVL library [Ble90], developed by Guy Blelloch and colleagues at Carnegie-Mellon as a machine-independent library used in the interpretation of the data-parallel intermediate code VCODE [BC90]. Likewise, we intend to reduce process parallelism to the set of procedures provided with the threads facility of Mach [BRS+85].

7. Related work

There are a wide variety of programming languages that are cited as being useful for prototyping sequential computations. These languages include APL, SETL, Prolog, and OPS-5, to name a few. *Proteus* follows the approach typified by SETL in which high-level predefined data types supply the bulk of the expressive power. This approach is important for *Proteus* because it is the fundamental source of data-parallelism. Early forms of these ideas appear in CSP [Hoa85] and are developed further in UNITY [CM88].

For the prototyping of concurrent systems, there are a plethora of candidate parallel languages, which might be roughly divided into the following classes.

- Languages with widely translatable logical models, such as Linda's distributed data structures [CGL86], the synchronization-variable methods of Strand [FT90] and PCN [CT90], or the data-parallel abstraction of the Paralation model [Sab88].
- Languages which incorporate a large variety of parallel primitives, such as Ease [Zen90] and Alloy [MH90].
- Wide-spectrum parallel languages that rely on refinement from architecture-independent specification. Notable wide-spectrum parallel language efforts include Crystal [Che86] and variants of the Bird-Meertens functional formalism [Ski90]. UNITY, although not a wide-spectrum notation, is, as its name suggests, a particularly elegant notation for describing a large range of parallel and distributed computations.

We see *Proteus* as falling into the last category. All of these wide-spectrum languages support a methodology in which parallel specifications are refined to parallel programs for a particular class of machine. In the case of Crystal, UNITY, and the Bird-Meertens formalism the refinement steps are justified formally through inference steps or algebraic transformations.

```
Algorithm (Shiloach+Vishkin):
  Let  V = [1..n] be a set of vertex labels,  and
       E = { (v,w) | v,w in V and ∃ an edge from v to w in G }
  s,t  := 1,1;      — Iteration number
  D  := V;          — Pointer graph: every node initially points to itself
  Dp := V;          — Previous values of D in step s-1
  Q  := [i in V : 0]; — Last step D(i) updated (not stagnant node if =s).
  ( s=t →
      par [i in V :  private D;            — Shortcutting
           Dp[i],D[i] := D[i],D[D[i]];
           D[i]≠Dp[i] → Q[i] := s;
      ];
      par [[v,w] in E :  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];
      ];
      par [i in V :  private D,t;          — Shortcutting and detection of termination
          D[i] := D[D[i]];
          Q[i]=s  →  t := t+1
      ];
      s := s+1;
  )*
```

Figure 4: **Shiloach-Vishkin parallel connected-components algorithm**

In comparison with these languages *Proteus* supports fundamental parallel abstractions at a higher level (e.g. barrier merge) than UNITY and at a lower level than Crystal (where concurrency is implied by independence in the equational specification). As the prototyping process yields a procedural specification rather than an equational or predicate logic specification, *Proteus* programs can refer to shared state explicitly and must use the barrier-merge or the **await** synchronization primitives to control interference between parallel operations. Although UNITY programs also manipulate shared state, the control of interference is implicit by constraining execution to statement-level interleaving.

8. Summary and future work

In this paper we introduced *Proteus*, a prototyping language whose constructs for expressing parallelism can serve as a foundation for embracing many concurrent programming models. Synchronous and asynchronous parallel programs may be expressed with barrier and conditional synchronization, 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 such diverse concurrency models as PRAM and CSP within a single setting. *Proteus* thus provides a reasonable foundation for the construction of a wide spectrum of parallel programs, when used in conjunction with refinement techniques for architectural specialization. While we have presented here only an informal semantics for *Proteus*, we are developing a formal operational semantics based on the lambda calculus.

Ongoing work in the area of the *Proteus* language design is concentrated on two areas. First, we are are investigating the inclusion of higher-level features for distributed programming, using the notion of concurrent objects as the basis of an approach to controlling process parallelism [Agh90]. Second, we are investigating the modeling of time-constrained computation in the form of annotations for the relative execution rates of processes.

Finally, we are currently involved in the implementation of key features of the language and refinement system to assess the suitability of the approach. The long-term goal of the work is to incorporate *Proteus* into a prototyping system that links several prototyping languages, targeting different problem domains, to form an effective vehicle for the development and assessment of prototypes.

Acknowledgements

The authors would like to thank Mike Landis and Dan Palmer for their constructive advice and comments.

References

[Agh90] G. Agha, "Concurrent object-oriented programming," *Communications ACM*, vol. 33, pp. 125–141, Sept. 1990.

[BC90] G. Blelloch and S. Chatterjee, "VCODE: a data-parallel intermediate language," in *Proceedings Frontiers 90*, IEEE, 1990.

[BDL89] N. Baxter, E. Dubinsky, and G. Levin, *Learning Discrete Mathematics with ISETL.* New York: Springer-Verlag, 1989.

[BG90] L. Blaine and A. Goldberg, "Modules and types for a common prototyping language," Technical Report, Kestrel Institute, Palo Alto, California, Oct. 1990.

[Ble90] G. Blelloch, "The CVL library," Draft Technical Note, Carnegie Mellon University, 1990.

[Bro87] F. Brooks, "No silver bullet: essence and accidents of software engineering," *IEEE Computer*, vol. 20, pp. 10–19, Apr. 1987.

[BRS+85] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young, "Mach-1: An operating environment for large-scale multiprocessor applications," *IEEE Software*, July 1985.

[BST89] H. Bal, J. Steiner, and A. Tanenbaum, "Programming languages for distributed computing systems," *ACM Computing Surveys*, vol. 21, pp. 261–322, Sept. 1989.

[BT88] H. Bal and A. Tanenbaum, "Distributed programming with shared data," in *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, (Miami, Fla., Oct. 9-13), pp. 82–91, The Computer Society of the IEEE, New York, 1988.

[CGL86] N. Carriero, D. Gelernter, and J. Leichter, "Distributed data structures in Linda," in *POPL 13*, (St. Petersburg, Fla., Jan.13-15), pp. 236–242, ACM, New York, 1986.

[Che86] M. Chen, "Very-high-level parallel programming in Crystal," in *Proc. 1986 Hypercube Conference*, (Knoxville,Tn.), 1986.

[CM88] K. Chandy and J. Misra, *Parallel Program Design, A Foundation.* Addison-Wesley Publishing Company, 1988.

[CT90] K. Chandy and S. Taylor, "A primer for Program Composition Notation," Technical Report, California Institute of Technology, June 1990.

[CZ89] R. Cole and O. Zajicek, "The APRAM: Incorporating asynchrony into the PRAM model," in *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures*, pp. 169–178, ACM Press, 1989.

[Dij78] E. Dijkstra, "Guarded commands, nondeterminacy and the formal derivation of programs," *Communications ACM*, no. 18, pp. 453–457, 1978.

[DS80] E. Dijkstra and C. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, no. 11, 1980.

[FT90] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1990.

[FW78] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc.10th Ann. ACM Symp. on Theory of Computing*, pp. 114–118, 1978.

[Hoa74] C. Hoare, "Monitors: An operating system structuring concept," *Communications ACM*, vol. 17, pp. 549–557, Oct. 1974.

[Hoa85] C. Hoare, *Communicating Sequential Processes.* Reading, Mass.: Addison-Wesley, 1985.

[INM87] INMOS Ltd., "The Occam programming manual." Prentice Hall, 1987.

[MH90] T. Mitsolides and M. Harrison, "Generators and the replicator control structure in the parallel environment of ALLOY," in *ACM SIGPLAN'90*, (White Plains, N.Y.), pp. 189–196, June 1990.

[Nyl91] L. S. Nyland, *The Design of A Prototyping Programming Language for Parallel and Sequential Algorithms.* Ph.d. dissertation, Duke University, Feb. 1991.

[Ref88] Reasoning Systems, Inc., Palo Alto, California, *Refine 2.0 Language Summary*, Aug. 1988.

[Sab88] G. Sabot, *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1988.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets, An Introduction to SETL.* New York: Springer-Verlag, 1986.

[Ski90] D. Skillicorn, "Architecture-independent parallel computation," *IEEE Computer*, vol. 23, pp. 38–50, Dec. 1990.

[Smi90] D. Smith, "KIDS – a semi-automactic program development system," *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering*, vol. 16, pp. 1024–1043, Sept. 1990.

[SV82] Y. Schiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm," *Journal of Algorithms*, no. 3, pp. 57–67, 1982.

[Zen90] S. Zenith, "Programming with Ease: a semiotic definition of the language," Research Report RR809, Yale University, July 1990.