pSather monitors: Design, Tutorial, Rationale and Implementation

Jerome A. Feldman* Chu-Cheow Lim†

Franco Mazzanti‡

TR-91-031

September 1991

Abstract

pSather is a parallel extension of Sather aimed at shared memory parallel architectures. A prototype of the language is currently being implemented on a Sequent Symmetry and on SUN Sparc-Stations. pSather *monitors* are one of the basic new features introduced in the language to deal with parallelism. The current design is presented and discussed in detail.

*ICSI and Computer Science Division, U.C. Berkeley. E-mail `jfeldman@icsi.berkeley.edu`.

†ICSI and Computer Science Division, U.C. Berkeley. E-mail `clim@icsi.berkeley.edu`.

‡ICSI and Istituto di Elaborazione dell'Informazione, CNR Pisa Italy. E-mail `mazz@icsi.berkeley.edu`.

Contents

1 Introduction

General purpose parallel programming has been an elusive goal, but continues to be essential for the future of the field. The pSather project is an attempt to provide simple but powerful support for the development of imperative parallel programs. A central assumption of our work is that the abstraction and re-usability features of object-oriented languages will be even more important for parallel codes. This report provides more than most people want to know about the pSather monitor construct, the cornerstone of our design.

Part of this introduction is extracted from "sather_summary.txt" by Stephen M. Omohundro (which is part of the Sather distribution). Sather ([41], [35]) is a new programming language under development at the International Computer Science Institute. Sather has clean and simple syntax, parameterized classes, object-oriented dispatch, multiple inheritance, strong typing, and garbage collection. The compiler generates efficient and portable C code which is easily integrated with existing code.

The initial beta test release of the language was in June, 1991. The compiler, debugger, Emacs development environment, documentation, and library classes are available by anonymous ftp from "icsi-ftp.berkeley.edu". "sather@icsi.berkeley.edu" is a mailing list for discussing aspects of Sather and "sather-admin@icsi.berkeley.edu" should be used for bug reports and requests to be added or deleted from the mailing list.

Sather is based on Eiffel [37] but is more concerned with efficiency and less with some of the formal and theoretical issues addressed by Eiffel. The language is much smaller than the current Eiffel, it eliminates over 40 keywords and simplifies the syntax and inheritance rules. Several features were added to increase efficiency and to simplify programming. Efficient arrays are built into the language itself (objects may have a dynamically allocated array portion after their static features). The typing scheme allows the programmer to distinguish between dispatched and non-dispatched declarations. As in C++, local variables may be declared at the point of use. Sather classes may have shared variables which are accessible from every instance of that class. For efficiency reasons, the exception handling mechanism based on setjump and long-jump has been eliminated and the garbage collector is not based on the Dijkstra algorithm (but on the algorithms described by Boehm and Weiser in [13]). Many cosmetic issues have been changed (eg. more than one class may be defined in a file).

Like Eiffel, Sather code is compiled into portable C and efficiently links with existing C code. The Sather compiler is written in Sather and has been operational for almost a year, though it is still being improved. Preliminary benchmarks show a performance improvement over Eiffel of between a factor of 4 and 50 on basic dispatching and function calls. On the benchmarks used at Stanford to test Self (including 8 queens, towers of hanoi, bubblesort, etc), Sather is even slightly faster than C++.

The Sather compiler and libraries are publicly available under a very unrestrictive license aimed at encouraging contribution to the public library without precluding the use of Sather for proprietary projects. The goal is to establish a repository for efficient, reusable, well written, publicly available, classes for most of the important algorithms in computer science. There are currently several hundred classes in the library. The libraries are growing quickly and will collect together classes from many authors under the same unrestrictive license.

A GNU emacs development environment for Sather is available. In addition to automatically indenting Sather code, it automatically generates documentation files, runs the compiler, parses compiler error messages and puts you at the error, runs the debugger and graphically points the current line, keeps track of the inheritance hierarchy and provides search facilities across all classes in a program.

A debugger based on gdb from the Free Software Foundation is also available. This allows you to set breakpoints and step through Sather code, to set variables, and browse through Sather objects. In conjunction with the Emacs environment, it graphically displays the source code corresponding

the the currently executing code. It switches into debugging C code when any user written C code is encountered.

From the outset, one goal of the Sather project has been the incorporation of constructs to support parallel programming. Parallelism is still not well understood and we explicitly allowed the parallel constructs to lag the base language by about a year. There is now an experimental parallel Sather (called pSather) running on the SUN Sparcstation and the Sequent Symmetry. The Sequent implementation is the more serious effort; the current Sparcstation supports no parallelism and its pSather is mainly useful for working out programs in the comfort of your own workstation. Section 6 discusses some of the basic implementation issues and how they affected the current design.

The references ([7], [8], [6], [21], [10], [29], [11], [38], [28], [17], [19], [22], [27], [30], [1], [18], [40], [43], [39], [3], [4], [5], [50], [2], [53], [52], and [16].) constitute a reasonably comprehensive literature of various designs and implementations of parallel object-oriented languages.

The design of pSather proceeded from several basic considerations. The parallel constructs had to be a natural extension of existing Sather and had to maintain the basic design goals of supporting the development of safe, efficient and reusable library classes. In fact, this philosophy seems even more important in parallel programming where it is much more difficult to code effectively. We plan to eventually merge pSather into a standard Sather release and drop the 'p' forever.

For a variety of reasons, pSather has been targeted for shared-memory architectures. One major reason is that Sather (like other object-oriented languages) is committed to pointer manipulation. We aim to support parallelism on the full range of data-structures and algorithms and these are much more naturally expressed in a shared-memory model. Currently there is a great deal of effort on realizing large shared-memory architectures, usually in non-uniform memory access (NUMA) fashion. Even in distributed memory systems, one can expect to see multiple active threads of control per processor and the language should support this. An additional goal of the pSather project, beyond the scope of this paper, is to help support the development of large NUMA architectures. We are not convinced that efforts to provide efficient shared-memory abstraction uniformly in hardware will scale adequately. Our hope is that appropriate high-level constructs, like those in pSather, will enable the user, compiler, and run-time routines to provide significant help in supporting shared-memory abstractions.

These (perhaps overly ambitious) goals led us to consider a wide range of old and new mechanisms for the pSather primitives. Many of the constructs in current parallel languages have their origins in multi-programming of sequential machines, in distributed computing, or in distributed memory architectures. We have found that the requirement of efficiently supporting high degrees of shared-memory parallelism suggests somewhat different parallel constructs. Fortunately, the strong typing of Sather permits solutions not possible in looser languages such as C++[49]. It turns out that the pSather mechanisms also support loosely coupled programming quite well, but we have not seriously studied pSather for machines without shared-memory support.

The current pSather design adds relatively few constructs to Sather. By far the most important of these is the *monitor* construct[1] and its associated mechanisms which are the focus of this report. The three features not discussed here are mutex classes, safe variables, and placement pragmas. We will certainly add some statement-level parallel construct, but its form remains undecided.

There is one special kind of class, *mutex*, which is declared by "mutex class DUSTY". The system assures that only one thread at a time can be active in an object of mutex type. If a thread attempts to enter a busy mutex object it will be suspended. Mutex classes support a crude form of parallel programming. Mutex is useful for incorporating serial Sather code in pSather programs, but is not sufficient. Care must be taken with user or library classes that use class access, shareds or constants when moving serial Sather code to pSather.

Attributes, local variables, formal parameters, and return results can be specified as *safe*. Declaring a MONITOR-type variable safe means that the expressions returned by read and take are safe.

[1] Our *monitor* construct differs from earlier monitor concepts for which [15] gives an interesting account.

The main use of safe variables is to support attributes that are read-only except in their defining context. Safeness of expressions is maintained over dotting, class prefixing, subscripting and function calling.

A safe expression can be the LHS (target) of only direct, local assignments. Syntactically, this means that the LHS must have no suffixes and no prefix except "self". This implies that the assignment must be in the defining context. A safe expression of a pointer ($OB) type can not be used in the RHS of an assignment unless this is a direct assignment to another safe variable. Similarly, a safe pointer-type actual parameter can not be substituted for an "unsafe" formal parameter. Safe expressions of $ARRAY type can not be assigned outside the defining context of the array and can not be passed as actual parameters. This is because array indexing is a direct assignment primitive and assigning an array outside its defining class violates its safety. Similarly safe $MONITOR expressions can not be assigned outside of the defining context or passed as actual parameters. It is illegal to have a safe MONITOR variable as the target of a deferred assignment except in its defining context. We expect the 'safe' construct to significantly help in NUMA memory placement.

The placement pragmas in pSather are quite simple, but seem to be adequate. Any call of 'new' or 'copy' can be affixed with an integer expression that specifies a logical processor number on which the new object should be located if feasible. A typical call might be:

```
his.x := x.copy@his_proc
```

Placement pragmas can also be appended to thread calls, which are discussed in detail in this report. The following call will (pragmatically) start a new thread executing 'fun' of object 'ob' on logical processor 'lightest' and pass it a reference to a local copy of attribute 'x', which might, e.g., be an array:

```
ans :- ob.fun( x.copy@lightest )@lightest
```

Similar constructs can be used to support message-based and other loosely coupled programming styles, but this will not be elaborated here. We also envision the placement pragmas being used by library classes, together with system inquiries, to do data and situation dependent optimizations.

Section 2 presents the design of the *monitor* functionality, while in Section 3 it is shown with some examples how they can advantageously be used to write parallel programs. In Section 4 the reasons behind the current design are explained in more detail. Section 5 explains why several alternative choices have not been made. In Section 6 more insights on the ongoing Sequent and Sparc implementations are given. Finally in Section 7 are described some possible directions for further research and our plans for the continuation of the project.

2 Monitor Design

pSather is a parallel extension of Sather for programming on shared memory architectures. In pSather, *thread-calls* provide the means to start new threads, executing in parallel a function call or a routine call. *monitors* serve to synchronize these threads.

In particular,monitors are objects of some special built-in types. These types (called monitor types) are the MONITOR0 type, and any instantiation of the parameterized version MONITOR{T}. The user can define other monitor types via class inheritance. In the next Sections 2.1, 2.2, 2.3 is presented the functionality of the MONITOR{T} type. The typeless version of monitors is illustrated in Section 2.4.

2.1 Locking

Monitors have a built-in locked/unlocked status.

Two new statements (called locking statements) can be used to lock a monitor. These statements are the *lock-statement* and the *try-statement*. They have the form:

```
lock <monitor-list> then
   < statement-list-1 >
end;
```

```
try <monitor-list> then
   <statement-list-1>
[else
   <statement-list-2>]
end;
```

Where `<monitor-list>` is a sequence of monitor expressions (each possibly suffixed by a predefined test operation, as explained later in this section), separated by commas.

In the execution of a *lock-statement* or of a *try-statement*, all the expressions of the `monitor-list` are pre-evaluated. It is an error if any of these expressions evaluates to "void".

If all the denoted monitor objects are available for locking (i.e. they are not already locked by another thread) then they are all atomically locked and the `<statement-list-1>` is executed.

If any of the denoted monitors is not available for locking the execution of the two statements proceeds differently. In the case of a *lock-statement* the current thread is suspended until all the monitors become available for locking, after which they are locked and the `<statement-list-1>` is executed. In the case of a *try-statement* the alternative `<statement-list-2>` is executed (if provided) without locking any monitor.

When a *lock-statement* or a *try-statement* is completed, all the monitors which have been locked by the statement (and which have not been already unlocked, see below) are unlocked. Unlocking restores the locking status existing before the execution of the locking statement. That is, if a monitor was already locked by the same executing thread, after the nested locking statement it is still in the original locked status.

During the execution of a locking statement the unlocking of a monitor can be anticipated by using the *unlock-statement* which has the form:

```
unlock <monitor-expr>;
```

An *unlock-statement* can only appear inside the scope of a *lock-statement* or the "then-branch" of a *try-statement*. It is a run-time error to try to unlock a monitor which is not one of those monitors locked by a syntactically enclosing locking statement. As before, unlocking has the meaning of restoring the previous locking status.

Any attempt by a first thread to lock a monitor already locked by a second thread, will result in the first thread being suspended until the monitor is unlocked.

A *break-statement* appearing in the `<statement-list-1>` of a locking statement forces the completion of the locking statement.

A *return-statement* appearing in the `<statement-list-1>` of a locking statement forces the completion of all the locking statements of the current routine, before returning.

Beyond having a locked/unlocked status, monitors also have a built-in bound/unbound status, and they can be used to start parallel execution of new threads. These additional functionalities interact with the basic locking scheme, allowing users to exploit more powerful synchronization patterns.

In particular the monitor types predefine some built-in BOOL functions which can be used to check the status of a monitor. This functions are: *is_bound*, *is_unbound*, *has_threads*, *no_threads*. All these operations can be performed independently of the locked/unlocked status of the monitor and do not suspend the executing thread. Moreover their names can be used as suffixes for the monitor expressions in the `<monitor-list>` of locking statements.

If the expression `<monitor-exp>.<monitor-predicate>` appears inside the `<monitor-list>` of a locking statement, the monitor denoted by the expression `<monitor-exp>` is locked if, beyond being available for locking, it is also in a status in which the `<monitor-predicate>` is true (e.g. bound or without attached threads). If the monitor status does not satisfy this condition, in the case of a *lock-statement* the executing thread is suspended until the condition becomes true, after which it is normally locked. In the case of a *try-statement* the alternative sequence of statements (if any) is executed.

The monitor types predefine additional operations related to the other aspects of the monitor functionality. These operations are *read*, *take*, *set*, *enqueue*, *clear*, *copy* and "`:-`". Any attempt to execute these operations on a monitor which is currently locked by another thread suspends the executing thread until the monitor is unlocked.

2.2 Signals

Beyond having a locked/unlocked status, monitors also have a built-in bound/unbound status which is associated with a value of type T, and a queue of values generated by the completion of "attached threads" (presented in detail in the next section) or provided by *enqueue* operations.

Upon creation a monitor is in the unbound status, and the queue of values is empty.

binding by *set* or *enqueue* If the monitor is unbound, the execution of a *set* or *enqueue* operation causes the monitor to become bound. These operations also take a parameter of type T which becomes the current value associated with the monitor bound status.

If the monitor is already bound, the execution of a *set* operation simply overrides the previous value associated with the bound status with the new value provided by the operation (i.e. it is a normal assignment). The execution of an em enqueue operation simply enqueues the new value. In both cases the bound status remains unchanged.

binding by thread completion In the case of the completion of an "attached thread" (see next section), if the monitor is unbound the completion makes it bound and the value returned by the thread becomes the current value associated with the monitor bound status. If the monitor was already bound, the completion of an attached thread simply enqueues the returned value. The binding of a monitor (or the queuing of the return value) is performed independently of the monitor locking status.

unbinding When the monitor is in a bound status, if the queue of values (provided by thread completions or *enqueue* operations) is empty, the *take* operation puts the monitor again into the unbound status. If the queue of values is not empty, the monitor remains bound but the first value of the queue is removed and becomes the value associated with the current bound status.

waiting for the bound status The *take* and *read* operations of monitors of type MONITOR{T} are functions and return the value (of type T) which was associated with the bound status. If a *read* or *take* operation is attempted when a monitor is unbound, the executing thread is suspended until the monitor becomes bound.

If there is more then one *read* and *take* suspended operations when the monitor becomes bound, they are resumed and executed in FIFO order, as far as the monitor status allows it. E.g if two *take* operations are suspended, only the first one will be resumed. Similarly, if three *take* operations are suspended on a locked monitor, and the monitor is finally released, if the monitor is bound and an additional value is present in the queue, only the first two of the three *take* operations will be resumed (leaving the monitor unbound).

As already mentioned, any attempt to *set*, *enqueue*, *read*, *clear*, *take* or *copy* a monitor which is currently locked by another thread suspends the executing thread until the monitor is unlocked.

The call of a *copy* operation of a monitor returns a new (unlocked) monitor in the same bound status as the copied monitor and, if the monitor is bound, with the same value associated with the bound status. The queue of returned values from the already completed threads is not copied.

2.3 Forking

Monitors can appear in the left-hand-side of thread-calls. A thread-call has the form:

```
<monitor-expr> :- <routine-call>;
```

In the execution of a thread-call (also referred to as "deferred-assignment"), the `<monitor-expr>` in the left-hand-side, and the object and actual parameters of the routine call are pre-evaluated. It is an error if the monitor expression evaluates to "void".

If the monitor denoted by the `<monitor-expr>` is currently locked by another thread, the executing thread is suspended until the monitor becomes unlocked.

Then the routine given in the right-hand side is called with the given parameters, executing it as a new parallel thread of control (it becomes a so-called "attached thread" of the monitor), while the thread executing the deferred-assignment is allowed to proceed in parallel.

If the left-hand side of a thread-call is a monitor of type MONITOR{T} then the forked routine must be a function whose result type conforms to T. On the other hand, if the monitor is of type MONITOR0, then the forked routine may or may not return any result.

When the called thread completes its execution, returning a value, if the monitor was unbound it is made bound, otherwise the returned value is queued (see previous section). This binding (or queuing) is performed independently of the monitor locking status.

The monitor operation *clear* will detach all the "attached threads" of the monitor (if any), disallowing them from binding the monitor or queuing their return values. Moreover the queue of values from the already completed threads is emptied, and the monitor status is made unbound. Finally a flag is set for each detached thread, which can be checked by a thread with a call to the BOOL function `CONFIG::clear_request`. In this way threads prepared to safely abandon their execution might take advantage of a 'clear' event and release all the used resources performing some kind of "early termination". A *terminate* operation, which beyond clearing a monitor, will also attempt to cause an asynchronous termination of the attached threads, may be introduced as part of a forthcoming exception package.

The "parallel" status of a monitor (i.e. if there is still some executing attached thread possibly going to bind the monitor or queue a signal) can be tested by the monitor predicates *has_threads* or *no_threads*.

The *copy* operation on a monitor does not copy any thread-related information from the original monitor, like the queue of return values, or the information on the attached threads.

2.4 Typeless monitors

Beyond the parameterized MONITOR{T} class, pSather defines a non-parameterized version of monitor, called MONITOR0, which has the same functionality of the parameterized version, but without the aspects related to its type parameter.

In particular, the bound status of the monitor has no associated value of type T. This also implies that the *set* and *enqueue* operations do not need any parameter, and that *read* and *take* are no longer functions (they simply wait/reset the unbound status).

Finally, with respect to thread-calls, routines which are not functions are also allowed to be forked as threads using this typeless kind of monitors. Normal functions can still be forked using these monitors (possibly starting an heterogeneous set of parallel threads) but any return values are discarded. In any case (whether functions or procedures) the completion of a thread or the execution of an *enqueue* operation either binds the monitor or enqueues a signal marking a binding event.

2.5 MONITOR within Sather Class System

Each class in Sather falls into one of the following categories:

Basic Class There are five disjoint basic class hierarchies, for `CHAR`, `INT`, `BOOL`, `REAL`, and `DOUBLE`.

Non-Basic Class In general, any non-basic class can be a descendent of any other non-basic class, except for arrays. A descendent of an array class can only inherit from another non-array class or another array class of the same dimension. This category includes a set of built-in classes of which the compiler has some particular knowledge. Examples of these built-in classes are the IN, OUT, FILE, ARRAY, STR, SYS, C, classes. In some cases, the redefinition of some of the features of these classes are forbidden (e.g. the 'extend' feature of ARRAY).

Foreign Class This class hierarchy presents a Sather interface to foreign objects. The top-most ancestor class is `F_OB`.

With respect to the above categories, `MONITOR0` and `MONITOR{T}` classes are built-in, non-basic classes with the following properties.

- A descendent of `MONITOR0` (`MONITOR{T}`) class can inherit from any non-basic class including `MONITOR{T}` (`MONITOR0`). An inherited monitor class that appears later will overshadow the earlier monitor class.
- A monitor class can inherit from any array class, with the restriction that it cannot then inherit from an array class of a different dimension.

If a class inherits from a monitor type, the predefined operations and tests cannot be redefined (or undefined).

3 Monitor Tutorial: Supported programming styles and paradigms

The pSather monitor construct combines functionalities that have been kept separate in previous parallel programming languages. Clearly, even if monitors provide a much more complex functionality than a simple lock or signal, this does not mean that they cannot be used for directly supporting in a still efficient way these very simple kinds of synchronizations and, beyond these, many other common concurrent programming paradigms. However, apart from their "crude" use for direct thread synchronization, one of the major roles of monitors is that of constituting the hidden building block of higher level abstractions providing more sophisticated tools (e.g. see the examples at the end of this section).

3.1 Asynchronous signal/message exchange

One of the obvious "crude" uses of monitors, is simply to act as an asynchronous signal or message among threads. In the following we will refer to the parameterized version of monitors, i.e. to objects of type MONITOR{T}. There is a typeless variant MONITOR0, but we will usually not explicitly discuss it in this section.

An object of monitor type, e.g.:

```
mi: MONITOR{INT} := mi.new
```

can be seen as containing a private attribute of its parameterized type, some state, and system queues of threads waiting to set or use the monitor. A monitor object can be used almost like a simple variable, accessed by 'mi.read' and set by 'mi.set'. The state associated with a monitor specifies whether it is bound or unbound and whether it is locked or unlocked, and whether it has attached threads. Here we are mainly concerned with the bound state.

A monitor is unbound if its private attribute has no value; monitors are created unbound and are bound by a 'mi.set' call or by the completion of a deferred assignment (Section 2.3). A monitor that is bound can be made unbound by the call 'mi.take', which also returns the current value of the private attribute. Any thread that attempts to access an unbound monitor will be suspended and placed on a queue waiting for the monitor to be bound. There are non-blocking predicates 'mi.is_bound' and 'mi.is_unbound' with the obvious meaning. The original motivation for 'unbound' came from the BBN Monarch design that used the equivalent 'stolen' as the sole synchronization primitive. The pSather unbound state is the only asynchronous signal in the language, but it has several additional roles.

If only *set* and *take* operations are used on a monitor, what we obtain is a simple one-to-one synchronization, as illustrated in the following example:

```
m1:MONITOR{INT}:= m1.new;  -- initially unbound
m2:MONITOR{INT}:= m2.new;  -- initially unbound

thread1 is                 thread2  is
   ...                        ...
   m1.set(111); ---------->   v:INT := m1.take;  -- wait message
   v:= m2.take; <---------    m2.set(v+v);       -- send back a message
   ...                        ...
end;                       end;
```

This one-to-one synchronization can be extended to a richer set of threads possibly generating much more complex patterns of synchronization.

The *read* operation, is more naturally aimed at some kind of broadcast or one-to-many communications as illustrated by the next example:

```
mv:MONITOR{ARRAY{INT}}:= mv.new;   -- initially unbound
 ...
writer is
   v:ARRAY{INT}:= v.new(n);  -- create and initialize a new vector
   v[0] := ...;
   v[n-1] := ...;
   ...
   mv.set(v);         -- make the vector available to all the readers
   ...
end;

reader_1 is                                        reader_n  is
   v:= mv.read[0]; <-- wait data from writer -->  v:= mv.read[n-1];
   ...                                                ...
end;                                               end;
```

Another example of *read* used to model a broadcast signal, is shown by the following implementation of a "barrier":

```
class BARRIER is
   private counter:INT;           -- how many hits in current iteration
   private all_done:MONITOR0;     -- used as a broadcast signal
   private barrier_lock:MONITOR0; -- used to get mutual exclusion among 'hit'
   private level:INT;             -- the number of hits for each iteration

   hit is
      broadcast:MONITOR0:= all_done;  -- reference to shared signal
      wait_needed:BOOL := true;
      lock barrier_lock then
         counter := counter + 1;
         if counter = level then
            broadcast.set;             -- resume all previous hits
            all_done:= MONITOR0::new;  -- create new signal for next iter.
            counter := 0;              -- reset counter for next iteration
            wait_needed := false;      -- this call does not need to wait
         end;
      end; -- lock
      if wait_needed then
         broadcast.read;               -- wait until a 'set' is done
      end;
   end; -- hit

   create(level:INT):SELF_TYPE is
      res:=res.new;
      res.all_done    := MONITOR0::new; -- create initial broadcast signal
      res.barrier_lock:= MONITOR0::new; -- create barrier lock
      res.level := level;
   end; -- create
end; -- BARRIER
```

But the *set* operation is not the only one which generates an asynchronous signal. The completion of a thread can supply the bound value to a monitor, as illustrated below:

```
mv:MONITOR{ARRAY{INT}}:= mv.new;   -- initially unbound
 ...
writer(...): ARRAY{INT} is     -- now 'writer' is a function
   res := ARRAY{INT}::new(n);  -- create and initialize a new vector
   res[0]   := ...;
   ...
   res[n-1] := ...;
end;

mv :-  writer(...);          -- the function is executed in parallel

reader_1 is                                        reader_n  is
   v:= mv.read[0]; <-- wait data from writer -->  v:= mv.read[n-1];
   ...                                             ...
end;                                               end;
```

Moreover, since monitors are standard objects of the language, they can be assigned or passed as parameters, capturing very well the essence of "future" values.

E.g. we could also have written:

```
reader(future_vect:MONITOR{ARRAY{INT}}, index:INT) is
   ...
   v:= future_vect.read[index];
   ...
end;
```

creating many parallel 'reader' threads using the same future in the following way:

```
   readers: MONITOR0:= readers.new;
   i:INT;
   ...
   until i = n loop
      readers:- reader(mv,i);
   end;
```

There are several other ways in which a monitor can control a set of parallel threads, handling their completion signals, but they will be discussed in Section 3.3

3.2 Mutual exclusion and more

The other basic state information on each monitor is whether it is locked or unlocked. As we will see, there are several advantages to having lock status directly associated with data rather than assuming all uses will obey some lock protocol. This seems to be particularly important for object-oriented programming with reusable classes.

The locking discipline of pSather fits nicely into the overall design. Because we wanted a safe and efficient locking scheme for application programmers, we adopted a critical section mechanism that automatically releases locks on termination. There is, as usual, a blocking and a non-blocking form of the construct:

```
lock <mon list> then
   <statement 1>
end;
```

```
try <mon list> then
   <statement 1>
[else
   <statement 2>]
end;
```

Simple mutual exclusion

The most basic functionality provided by locking statements is clearly the possibility of easily modeling mutual exclusion. For example, in order to guarantee the serialization of the calls of the routines of an object we can easily define:

```
class SERIALIZED is
   private my_lock:MONITOR0;
   ...
   operation1 is
      lock my_lock then
         ...
      end;
   end;
   ...
   operation2 is
      lock my_lock then
         ...
      end;
   end;
   ...
   create:SELF_TYPE is
      res := res.new;
      res.my_lock := MONIOTR0::new;
   end;
end; -- SERIALIZED
```

Simple mutual exclusion can also be expressed by *mutex* classes (Section 1).

But the mutual exclusion of the operations of an object can follow much more complex schemes than a simple full serialization. Notice that already in the previous example the "create" operation is allowed to proceed more freely than the others. Typically we could use more than one internal lock, serializing the operations in subgroups, depending on the kind of side effects they generate.

Moreover, we might even make public some of the serializing locks, allowing the clients to explicitly get access privileges for a subgroup of operations. For example, in the case of parallel queue, we might be interested to give to a client thread the possibility of becoming the only "reader" of the queue, yet not preventing others from enqueuing elements. (A similar pattern is shown the CHANNEL example of Section 3.7). This can be easily achieved in the following way:

Given the definition of the following queue:

```
class PARALLEL_QUEUE{T} is
   pushers:MONITOR0; -- used for simple locking
   poppers:MONITOR0; -- used for simple locking

   create:SELF_TYPE is ...end;
   ...
```

```
    ...
    push(v:T) is
       lock pushers then
          ...    -- more synchronization is probably needed inside
       end;
    end;

    pop:T is
        lock poppers then
          ...    -- more synchronization is probably needed inside
        end;
    end;
 end; -- PARALLEL_QUEUE
```

A thread desiring to acquire exclusive "popping" privileges for some time can simply write:

```
queue: PARALLEL_QUEUE{INT};
...
its_only_mine is
   lock queue.poppers then
       queue.pop;
         ...
       queue.pop;
         ...
       queue.pop;
   end;
end;
```

Multiple locking

pSather, moreover, allows one to specify more than one monitor inside a locking statement, guaranteeing that all or none of the locks in the `<monitor-list>` are acquired. This simplifies many common synchronization problems and also adds power to the monitor construct as we will see.

A quite common programming pattern captured by this functionality, is for example the splitting of object operations into different groups according to the set of resources they need, sequentializing all the calls of the same operation, but allowing different operations to proceed in parallel if they require a disjoint set of resources.

```
class RESOURCES is
   private resource1, resource2 : MONITOR0;

   create:SELF_TYPE is .. end;

   operation_1 is lock resource1 then ...  end; end;
   operation_2 is lock resource2 then ...  end; end;
   operation_3 is lock resource1, resource2 then ...  end; end;
end;
```

Using multiple locking has the advantage that it removes the danger of deadlocks, which might otherwise easily result if a nested locking schema were followed instead (this issue is discussed in more detail in Section 4.2)

As another example of multiple locking, we can see how the required synchronizations for the "dining philosophers" problem might be achieved.

```
class SUPPER is
   private chopstick:ARRAY{MONITOR0}; -- One for each chopstick.
   done:BOOL;                         -- Set true from outside.

   diner(i:INT) is
      until done loop
         lock chopstick[left(i)], chopstick[right(i)] then
            eat(i);
         end;
      end; -- until
   end; -- diner

   eat(i:INT) is
      OUT::s("number ").i(i).s(" is eating").nl; -- print the number
   end; -- eat

   start (count:INT) is
      chopstick:= ARRAY{MONITOR0}::new(count);
      done := false;
      dummy:MONITOR0 := dummy.new;   -- For thread starting.
      j,k:INT;
      until j = count loop
         chopstick[j]:= MONITOR0::new;
         j:=j+1;
      end; -- until j
      until k = count loop
         dummy :- diner(k);
         k:=k+1;
      end; -- until k
   end; --start

   left(i:INT):INT  is res:= (i-1).mod(chopstick.asize); end;
   right(i:INT):INT is res:= (i+1).mod(chopstick.asize); end;
end;  -- class SUPPER
```

Early unlocking

pSather allows one to explicitly unlock a monitor while still executing the body of the lock statement which locked it. The usefulness of this feature is quite general. We show here a small example of how to take advantage of this feature.

For example, suppose we have a sequence of monitors giving access to a sequence of critical sections:

```
 mon_vect: ARRAY{MONITOR0} := ...;

step (n:INT) is
   switch n
     when 0 then ...
     when 1 then ...
     when 2 then ...
   end;
end;
```

Users of this pipeline of critical sections are supposed to get the first lock, perform the first step, get the second lock and atomically release the first one, perform the second step, and so on.

Notice that we cannot first unlock the first monitor and then lock the second one, because this might allow some other thread to lock the first monitor, do the first step, unlock the first monitor and lock the second monitor, all in the small window between the time the first monitor is released and the second monitor is acquired, losing the original order in the pipeline.

The above pipelining can be modeled in the following way:

```
lock mon_vect[0] then
   step(0);
   lock mon_vect[1] then
      unlock mon_vect[0];
      step(1);
      lock mon_vect[2];
         unlock mon_vect[1];
         step(2);
      end;
   end;
end;
```

Since one lock is released only AFTER the next one has been acquired, there is no danger of losing the initial sequencing of operations.

Other paradigms

When "locking" and "signals" are put together as "monitors", new functionalities become possible.

One of these is that the flow of signals passing through a monitor by means of explicit *set* and *take* operations (e.g. as illustrated in the first example of Section 3.1) can be frozen by a third thread by simply locking the shared monitor. This gives additional power and flexibility in controlling synchronizations.

Similarly, by locking a shared monitor acting as a future, we can easily prevent other threads from taking the incoming value, or from attaching new additional threads.

Finally, and probably most important, new powerful atomic operations can be provided easily. In the design, an element of the lock-list of a locking statement can include a binding predicate and the lock waits until the predicate holds.

This enables us to model some kinds of event-driven or condition-driven locking, as illustrated by the example of parallel stack of Section 3.6. In that example, the bound status of the lock will be used to ensure mutual exclusion among the operations. It also carries the information about whether or not the stack is empty, allowing a client to acquire the lock only if there actually is some element to be read. This is modelled by the following:

```
pop:T is
   lock stack_lock.is_bound then
       ...           -- Remove the element
   end;
end;
```

The 'push' operation is allowed independently of the bound status of the 'stack_lock', and has the additional effect of setting to 'stack_lock' status to bound.

```
push(v:T) is
   lock stack_lock then
       ...            -- Add element to the stack
      stack_lock.set; -- Enable 'pop' operations
   end;
end;
```

As another example of the usefulness of this functionality, we show how an integer monitor can be used to model a counting semaphore.

```
class COUNTING_SEMAPHORE is

    private m: MONITOR{INT};

    create:SELF_TYPE is ... end;  -- initialize m

    get:INT is
       lock m.is_bound then    -- wait for a set operation if unbound
          res := m.read;
          m.set(res - 1);
          if res = 0 then
             m.take;  -- will not wait, simply makes the monitor unbound
          end;
       end;
    end;

    set(new_value:INT) is
       m.set(new_value);
    end;

end; -- COUNTING_SEMAPHORE
```

Many other useful cases of interactions between locking and signals are related to the completion of some parallel threads. Some of these cases will be discussed in the next subsection.

3.3 Paradigms for thread-calls

The third basic use of pSather monitors is as the target of a thread call (also known as deferred assignments). If `fun(x:INT)` is an INT function, the statement:

```
mi :- fun(some_value);
```

will start a new thread to evaluate 'fun(some_value)' and place the thread in the set of possible binders for 'mi'. This construct can be used in a variety of ways, both alone and in combination with other constructs.

An important point is that a monitor also acts as a handle on the threads which have been forked on it. For example, by using *is_bound* and *has_threads* predefined predicates, it is possible to check whether or not at least one of the forked threads, or all of them, have terminated. Moreover, using the *clear* operation on a monitor it is possible to set a flag for the forked threads notifying them they have been detached (i.e. that their returned values are no longer wanted).

In the following, some of the more useful programming paradigms are illustrated in more detail.

Simple "futures"

The simplest way to use a thread-call is just to fork a new thread on an unbound monitor. The returned value can be read or consumed, when available, by performing the monitor operation "read" or "take". In the meanwhile, the monitor itself can be passed around to other routines or threads, acting as a reference to "a value to come".

This way of using monitors has been already presented in Section 3.1. Here we only point out that a single monitor can easily be "recycled" as a future, once its value has been taken.

E.g. we can write:

```
lock f then
   f.clear;                      -- reset the future status
   until done loop
      f:- int_fun(some_value);   -- start new thread
      ...
      v:INT:= f.take;            -- wait and take result
   end; -- loop
end;  -- lock
```

Notice also that, since the binding of the monitor by a completing thread occurs even if the monitor is currently locked by a different thread, we can perform the whole cycle of forking and taking inside a lock statement (hence guaranteeing the absence of external interference) without the danger of deadlocks.

Searching for the "first" result

Another example of parallel programming paradigm is the following: Suppose we want to start several activities in parallel (e.g. searching for a server for a client "xwebster" application on several connected subnetworks), with the intention that as soon as one result is available, we would like to have it, and this "first" one is in some sense the "most interesting" for us (e.g. because the server is the nearest).

Also this case can be easily modeled using an unbound monitor as a future, with the only difference that more than one thread is actually forked using this monitor.

E.g. we could write:

```
my_server: MONITOR{STR}:= my_server.new;  -- Initially unbound

my_server :- search(subnet1, ...);   -- Fork three searching threads
my_server :- search(subnet2, ...);
my_server :- search(subnet3, ...);

result:STR := my_server.take;     -- Wait until at least one is completed
my_server.clear;                  -- Reset the monitor (detach the threads)
```

It is quite easy to model some kind of "OR parallelism" in which we are interested in the disjunction of the results of several parallel threads.

Modeling co-begin/co-end

Suppose we need to fork several parallel routines which are not required to return any specific result. More generally, we might not be particularly interested in the value returned by them (if any) but just in the fact that they have all done their job.

Monitors can be easily used to model also this kind of functionality. The first part (i.e. the co-begin) is immediate to model, being simply a loop in which we fork all the threads we want.

Then, we want be able to wait until all the started concurrent activities are completed. Actually, we have two predefined monitor predicates which allow us to check whether or not our condition is met without becoming suspended. They are the predicates: *has_threads* and *no_threads*. However, if we want to wait until our condition is satisfied we must use these predicates as monitor suffixes inside a locking statment, to get the equivalent blocking semantics.

The result is something like:

```
m:MONITOR0 :=m.new;
...
i:INT;
until i=n loop
   m:- foo(...);          -- Start several concurrent activities
   i:=i+1;
end;

lock m.no_threads then    -- Wait until they are all completed
   m.clear;               -- Discard their binding signals
end;
```

Again, if 'm' is a shared monitor, we might execute the cycle of thread-calls and the locking statement inside an outer locking statement ensuring the absence of external interference. Notice, that the pSather design allows the forked threads to return values the queue of a monitor (when they complete) even if the monitor is locked, and that the re-locking of an already locked monitor (by the same thread) is allowed and does not cause a deadlock.

Many threads producing many results

Yet other reasonable programming paradigm might consist in creating several threads, each one returning some result, and then collecting all the responses for performing some kind of final computation. Of the various styles seen so far, this is probably the most complex, and uses almost all the monitor functionalities.

As an example, we might consider the case of the comparison of two large vectors, returning some kind of measure of how they differ. This task could be achieved by checking in parallel the two vectors using "n" tasks, each one working on its own section of them and then merging the results.

```
v1:ARRAY{INT}; -- size 100000
v2:ARRAY{INT}; -- size 100000

differences(from,to:INT): INT  is
   res:=0;
   until from = to loop
      if v1[from]/= v2[from] then res := res+1; end;
      from := from+1;
   end;
end;

m:MONITOR{INT}:= m.new;
i:INT;
until i=10 loop                              -- fork 10 threads
  m:- differences (10000*i, 10000*(i+1));
end;
...
```

```
...
total:INT;
until not m.has_threads loop
   total:= total+ m.take;    -- possibly wait until next result arrives
end;
until m.is_unbound loop
   total:= total+ m.take;    -- read all the possibly queued results
end;
```

Notice how, even without knowing the number of forked threads, it is quite easy to get all the results, by performing a loop of 'take' operations until all the forked threads are completed, and until no more results are available.

Early results from parallel threads

We do not have to assume that a forked thread produces a meaningful result only when it completes. If this is not the case, however, it is likely that the result is made available to the rest of the program not using the same monitor on which the thread has been attached, but using some other monitor accessible by the routine (e.g. an actual parameter, an attribute, a shared).

An example of this way of programming could be the following:

```
some_parallel_computation (early_res: MONITOR{INT}) is
   ...                          -- some initial computation
   early_res.set(some_value);   -- make available the result
   ...                          -- some other computation
end;

main_activity is
   my_result: MONITOR{INT}:= MONITOR{INT}::new; -- a container for the result
   ...
   just_for_forking: MONITOR0:= MONITOR0::new;
   just_for_forking :- some_parallel_computation(my_result);
   ...
   my_result.read;        -- wait the early result from the parallel activity
   ...
end;
```

The same pattern could be used if the forked parallel activity does not produce a single early result, but a sequence of them. The only change is to use *enqueue* operations instead of *set*.

```
some_parallel_computation (early_res: MONITOR{INT}) is
   ...                               -- some initial computation
   early_res.enqueue(first_value);   -- make available the first result
   ...                               -- some other computation
   early_res.enqueue(other_value);   -- make available some other result
   ...                               -- some other computation
   early_res.enqueue(other_value);   -- make available some other result
   ...                               -- some other computation
   ...
end;
...

...
```

```
main_activity is
   my_result: MONITOR{INT}:= MONITOR{INT}::new; -- a container for the result
   ...
   just_for_forking: MONITOR0:= MONITOR0::new;
   just_for_forking :- some_parallel_computation(my_result);
   ...
   v:INT;
   v := my_result.take; -- wait first result from the parallel activity
   ...
   v := my_result.take; -- wait second result from the parallel activity
   ...
end;
```

Active objects

A programming style which is particularly easy to support is the case in which at most one thread is executing inside an object, representing in some sense the object "internal agent", communicating with other agents by message passing or entry calls. This can be modelled in pSather defining a private routine representing the internal activity and forking a thread executing it when the object is created. The only public interface of the object might consist in "communication channels" (possibly with different synchronization flavours) as those illustrated in Section 3.7.

```
E.g. class ACTIVE_AGENT is
        interface1 (v:INT):INT is
           data_in.send(v);
           res := data_out.receive;
        end;
        interface2: CHANNEL{SOME_TYPE};

        private data_in:  CHANNEL{INT};
        private data_out: CHANNEL{INT};
        private m: MONITOR0;

        create is
           res := res.new;
           res.m := res.m.new;
           res.data_in := CHANNEL{INT}::create(...);
           res.data_out:= CHANNEL{INT}::create(...);
           res.interface2:= CHANNEL{SOME_TYPE}::create(...);
           res.m :- internal_activity;
         end;

        private internal_activity is
           ...
           data_in.receive;
           ...
           data_out.send(..);
           ...
        end;
     end; -- ACTIVE_AGENT
```

Using a similar style, we can define classes simulating quite closely the behavior of Ada task

types, even though the semantics of conditional entry-calls and selective wait are more difficult to reproduce.

In Section 3.8 it is illustrated another example of this programming style, applied to the task of finding the maximun value inside an array.

3.4 Fairness Issues

If several threads are suspended on a single condition (i.e. involving no more than one monitor), then the waiting threads are resumed in the FIFO order (i.e. the thread which has become suspended first will be resumed first). This means that a queue is associated with any monitor, with the purpose of ordering the threads waiting for some event on the monitor.

```
E.g.
     s:MONITOR0 := s.new;

     --- thread1--     -- thread2--     -- thread3--

      lock s then         ...
        s.set;            ...           ...
        ...               s.take;       ...
        ...               ...           s.read;
      end;                ...           ...
```

In the above case thread1 will first gain the access to s, thread2 and thread3 will become suspended (both waiting for 's' to become unlocked and bound). Thread2 will be resumed before thread3 and will restore the monitor in the unbound status. This will prevent the resumption of thread3 until a new 'set' operation (or a thread completion) will bind 's' again.

The FIFO resuming order is not guaranteed during the execution of locking statements for threads which become suspended on more than one monitor. This issue is descussed in more detail in Section 6.2.3. Moreover, since pSather does not specify or require any particular scheduling strategy (e.g. FIFO, time-slicing), it is not guaranteed that all the created threads will surely be executed. In the current implementation this might really happen if a number of threads equal to the number of processors are concurrently executed and they never become suspended (other threads might never be executed).

Even if partial, this degree of fairness and FIFO behaviour is quite useful because it makes it easier to understand the actual program execution. For example, suppose that all output operations are protected by a lock which serializes the program output. In this case the FIFO fairness of the locking allows us to rely on the program output to have some (approximate) idea on how the program execution has evolved. This is useful, until additional powerful tools are designed and implemented, to at least allow some kind of simple debugging.

3.5 Parallel classes as higher level abstractions

We have seen how powerful pSather monitors are in their more or less "crude" version. However we must remember that the basic philosophy of object oriented programming is based on the development of new useful abstractions under the form of well-designed classes.

The constructs described in the preceding sections contribute significantly to the ease of writing parallel programs, but it is still a difficult and error-prone task. Our current best hope is that much of the complexity can be hidden in general classes and that most programming will not involve the explicit use of complex parallel constructs. This follows the philosophy of object- oriented languages in general and Sather in particular. For this to work out in practice, library classes must be efficient,

easy to understand at a functional level and sufficiently general. We believe that pSather provides a substrate for constructing such libraries, but have only begun to test this belief.

The following three sections contain simple examples of the kinds of classes we envision. These are also the most complete examples of pSather programming in the paper. The first example class is for a general STACKT that can be used by multiple threads without explicit synchronization. It also shows how serial Sather classes can be extended to parallel use. The second example illustrates how the abstraction capabilities of Sather can be combined with monitors to produce a set of CHANNEL constructs that use varying buffer disciplines invisibly to users.

The third example is less complete, but is the most indicative of what we expect to be the general programming style. Here the class VECTORT is envisioned as containing a variety of operations, of which max is illustrated. Operations on vectors are assumed to be done by a number of parallel tasks, decided at execution time. The two code fragments show how this might be done with threads sharing one structure or with separate objects having local data. A user should only deal with the unitary vector abstraction. Our current thinking on these parallel data abstractions is to try to keep the data distributed in NUMA architectures with the algorithms doing as much work as possible locally. This is fairly straightforward for arrays, but is a basic research question for sets, trees, graphs, etc.

3.6 Example: Parallel Stack

The following example is a simple version of a concurrently usable stack.

Inheritance The original 'pop' and 'top' operations, which simply return a null value if the stack is empty, are provided for compatibility with the original stacks in the Sather library.

Two other operations: 's_pop' and 's_top' are introduced to support potentially suspensive pop and top operations (if the stack is empty).

Just to make more evident the concurrent structure of the class, we prefer not to duplicate all the code implementing the sequential stack, but we make use of a private attribute ('s') which is a standard sequential stack.

Synchronization A stack status (free/busy, empty/not-empty) can be directly modelled upon a monitor status (busy => locked, empty => unbound). A 'stack_mon' monitor can be used for this purpose, i.e. to allow access to the stack operations only in the corresponding well-defined status.

The usefulness of the *is_bound* suffix in lock statements is evident here. If waiting for a condition, and locking a descriptor when the condition becomes true were not provided as an atomic language primitive, modeling the synchronizations between the operations would be much more complex.

For example we cannot simply write:

```
s_pop:T is
   can_read.take;        -- allows only one reader when stack is not empty
   lock internal_lock then -- mutual exclusion with other operations
   ...
end;
```

Because, if the check on the readability of the stack and the acquisition of the internal lock are not atomic, we are going to have problems. In particular, if a 'clear' operation (of the PSTACK{T} class) is executed after the 'take' operation (inside the above definition of 's_pop'), it may succeed in gaining the internal lock, clearing the stack itself and thus making false the precondition of the 's_pop' operation.

We cannot force the stack 'clear' operation to take the 'can_read' signal as well, because 'clear' operations are allowed even the stack is empty.

And we clearly cannot write:

```
s_pop:T is
   lock internal_lock then -- mutual exclusion with other operations
   can_read.take;        -- allows only one reader when stack is not empty
   ...
end;
```

Because if the stack is empty the executing thread becomes suspended leaving the stack locked. Instead, the "locking when bound" facility of pSather allows us to write:

```
s_pop:T is
   lock internal_lock.is_bound then
      ...
   end;
end;
```

easily expressing the required synchronizations.

```
class PSTACK{T} is ---- A concurrently usable stack --------------
   STACK{T};       ---- Quite compatible with the standard one
   ----------------------- Public features -------------------------
   -- create():SELF_TYPE;     Creates a new empty stack            |
   -- size:INT; (inherited)   The size of the stack                |
   -- push(e:T);              Inserts an element in the stack      |
   -- is_empty():BOOL;        Checks whether the stack is empty    |
   -- pop():T;                Removes last element (null if empty) |
   -- s_pop():T;              Removes last element (wait if empty) |
   -- top():T;                Returns last element (null if empty) |
   -- s_top():T;              Returns last element (wait if empty) |
   -- clear();                Empties the stack                    |
   ------------------------------------------------------------------

   private s: STACK{T};            -- a sequential stack
   private stack_mon:MONITOR0;     -- for synchronizing stack operations

   create:SELF_TYPE is
      -- Creates an empty stack.
      res := new;
      res.s := res.s.create;
      res.stack_mon := MONITOR0::new;
   end; -- create

   push(e:T) is
      -- Adds an element into the stack
      lock stack_mon then                      -- get exclusive access
         s.push(e);                   -- underlying stack fixes the size
         size:= s.size;
         if size = 1 then stack_mon.set; end; -- enable reading if needed
      end;
   end; -- push

   is_empty:BOOL is
      -- Returns 'true' if stack is empty.
      res := (size=0);
   end; -- is_empty

   pop:T is
      -- Returns the top element and remove it.
      -- Returns 'null' if empty
      lock stack_mon then                       -- get exclusive access
         res:= s.pop;
         if size=1 then stack_mon.take; end;
         -- If size=1, the last element has now been removed and the
         --  'stack_mon' signal has to be made unbound.
         size:= s.size;
      end;
   end; -- pop
   ...
```

```
   ...
   s_pop:T is
      -- Returns the top element and removes it.
      -- Waits if empty
      lock stack_mon.is_bound then             -- get access when not empty
         res:= s.pop;
         -- if size=1, stack_mon is surely bound
         size:= s.size;
         if size=0 then stack_mon.take; end;   -- if needed disable reading
      end;
   end; -- pop

   top:T is
      -- Returns the value of the top of the stack.
      -- Returns 'null' if empty.
      lock stack_mon then                      -- get exclusive access
         res:= s.top;
      end;
   end; -- top

   s_top:T is
      -- Returns the value of the top of the stack.
      -- Waits if empty.
      lock stack_mon.is_bound then             -- get access when not empty
         res:= s.top;
      end;
   end; -- top

   clear is
      -- Empties the stack.
      lock stack_mon then                      -- get exclusive access
        s.clear;
        if stack_mon.is_bound then
           stack_mon.take;                     -- unbind if necesary
        end;
        size:= 0;
      end;
   end; -- clear

end; -- class PSTACK{T}
```

3.7 Example: Communication channels

The following is an example of a data channel. This example is interesting because it shows several kinds of synchronization schemes.

The channel can be created with a null data buffer, a bounded data buffer, or an unbounded data buffer. If the channel has no buffer, 'send' and 'receive' operations are strictly synchronized (a 'send' waits until the corresponding 'receive' is performed, and vice-versa). If the channel has a bounded buffer, a limited number of 'send' operations are allowed to proceed even if no 'receive' has been executed (until the buffer is full). If the buffer is unbounded, 'send' operations are asynchronous with respect to 'receive' operations.

This example also illustrates a way to use monitors to specify selective access rights for particular class operations. Any user of this channel class is allowed to acquire exclusive "sending" rights, or exclusive "receiving" rights, or both. This is easily modelled by defining a monitor attribute (used as a lock) for each of the locking subconditions we want to associate with the channel object.

This example also shows how the use of monitors inside a class allows us to define parallel data structures which do not fully sequentialize all calls to the public routines.

In this case, if the channel is not full or empty, 'send' and 'receive' operations are allowed to proceed completely in parallel without any synchronizing bottleneck.

Finally, it is interesting to observe how, in order to avoid the need for explicit synchronization between concurrent send and receive operations in accessing the same "list of free elements", the use of two private lists and of one "interface" variable is helpful.

```
class CHANNEL{T} is -------------------------------------------------------
-- CHANNEL{T} exports two operations 'send' and 'receive' which can be
-- used to pass streams of data between threads.
--
-- Depending on the 'size' used to create a channel, the communications are more
-- or less synchronized. If a positive number is given as the channel size,
-- the value is used as the size of a bounded data buffer for the channel.
-- This allows a certain number of send operations to proceed asynchronously
-- with respect to the receiving operations.
-- If the given channel size is zero, then there is no data buffering, and
-- communications are fully synchronous (i.e. a send must wait until a
-- receiver is ready to read the data, and vice-versa).
-- If the channel size is negative, then the channel data buffer is unbounded.
-- (i.e. send operations can be freely executed also in absence of 'receive').
--
-- A channel can be shared among several threads (senders and receivers).
-- However, if one thread desires to gain exclusive access to the channel,
-- it can obtain it by acquiring one of the two channel locks:
-- 'exclusive_send_rights' or 'exclusive_receive_rights'
-----------------------------------------------------------------------------
   ...
```

```
   ...
   exclusive_send_rights:MONITOR0;
   exclusive_receive_rights:MONITOR0;
   private my_channel: $VIRTUAL_CHANNEL{T};  -- the actual channel structure
   ...
   create(size:INT):SELF_TYPE is
      res:=res.new;
      if size = 0 then
         res.my_channel := SYNCHR_CHANNEL{T}::create;
      elsif size >0 then
         res.my_channel := BOUNDED_CHANNEL{T}::create(size);
      else
         res.my_channel := UNBOUNDED_CHANNEL{T}::create;
      end;
      res.exclusive_send_rights:= MONITOR0::new;
      res.exclusive_receive_rights:= MONITOR0::new;
   end; -- create

   send(v:T) is
      lock exclusive_send_rights then
         my_channel.send(v);             -- proceed unless channel is locked
      end;
   end;

   receive:T is
      lock exclusive_receive_rights then
         res:= my_channel.receive;       -- proceed unless channel is locked
      end;
   end;
end; -- CHANNEL{T}

class VIRTUAL_CHANNEL{T} is
    send(v:T) is end;
    receive:T is end;
end; -- VIRTUAL_CHANNEL{T}
```

```
class SYNCHR_CHANNEL{T} is ------------------------------------------------
--                                                                        |
-- create:SELF_TYPE;                                                     |
-- send(v:T);                                                            |
-- receive:T;                                                            |
--                                                                        |
---------------------------------------------------------------------------
-- This class is a specialization of a generic VIRTUAL_CHANNEL{T}.        |
-- A 'send' operation waits until a 'receive' operation is executed       |
-- (and a 'receive' operations waits until a 'send' is executed), so that |
-- the data exchange occurs only when both parts have requested it.       |
---------------------------------------------------------------------------
   VIRTUAL_CHANNEL{T};

   private can_send, can_receive: MONITOR0;

   private tmp:T;      -- Variable acting as a single position buffer

   create:SELF_TYPE is
      res := res.new;
      res.can_send := MONITOR0::new;
      res.can_receive := MONITOR0::new;
      res.can_send.set;
   end;

   send(v:T) is
      -- All 'send' operations are already sequentialized by the outer lock.
      -- There is at most one concurrent 'receive' at any time.
      tmp :=v;          -- Store the value temporarily
      can_receive.set;  -- Allow the receiver to continue
      can_send.take;    -- Wait until the value has been received
   end;

   receive:T is
      -- All 'receive' operations are already sequentialized by the outer lock.
      -- There is at most one concurrent 'send' at any time.
      can_receive.take; -- Wait until the value has been sent
      res:=tmp;         -- Read the value
      can_send.set;     -- Allow the sender to continue
   end;

end; -- SYNCHR_CHANNEL;
```

```
class BOUNDED_CHANNEL{T} is ------------------------------------------------
--                                                                          |
-- create(size:INT):SELF_TYPE;    -- size must be positive                  |
-- send(v:T);                                                               |
-- receive:T;                                                               |
--                                                                          |
------------------------------------------------------------------------------
--      [0]  [top]                              [bottom]  [max_size]        |
--            ---   ---   ---   ---   ---   ---   ---                       |
--      ...   | |   | |   | |   | |   | |   | |   | |    ...                |
--            ---   ---   ---   ---   ---   ---   ---                       |
--             | ->                                | ->                    |
--             |_top_                              |_bottom_               |
--                                                                          |
-- This class is a specialization of a more generic VIRTUAL_CHANNEL{T}.     |
-- Moreover, in this case a channel is modelled as a specialization of array.|
-- The data exchange between sender and receiver is partially buffered.     |
-- The sender must wait for the receiver when the buffer is full, and       |
-- the receiver must wait for the sender when the buffer is empty.          |
--                                                                          |
-- In order to allow 'send' and 'receive' operation to proceede independently|
-- when the buffer neither empty nor full, any synchronization bottleneck   |
-- between these two operations must be avoided.                            |
-- A possible way to achieve that is  implementing the channel as a vector  |
-- of more complex elements (ELEMENT{T}) each one of which contains all the |
-- needed information for enabling/disabling read or write operations.      |
--                                                                          |
-- Objects of type ELEMENT{T} contain also a feature ('next') which is not  |
-- used by this class (being all the elements stored as array elements),    |
-- but which is used by the next example.                                   |
------------------------------------------------------------------------------
  VIRTUAL_CHANNEL{T};                -- this is a specialization of channel
  ARRAY{ELEM{T}};                    -- and also a particular kind of vector
  top,bottom:INT;                    -- pointers into the vectors

  private max_size:INT;

  create(size:INT):SELF_TYPE is
     res := SELF_TYPE::new(size);
     res.max_size:=size;
     n:INT;
     until n=size loop
        res[n]:= res[n].create;      -- create buffer elements
        n:=n+1;
     end;
  end;
  ...
```

```
  ...
  send(v:T) is
     -- All 'send' operations are already sequentialized by the outer lock.
     -- There is at most one concurrent 'receive' at any time.

     -- Store the data in the vector (when possible),
     -- and update the pointer to the bottom
     self[bottom].can_write.take;
     self[bottom].value := v;
     self[bottom].can_read.set;
     bottom := (bottom+1).mod(max_size);
  end; -- send

  receive:T is
     -- All 'receive' operations are already sequentialized by the outer lock.
     -- There is at most one concurrent 'send' at any time.

     -- Retrieve the data from the vector (when possible)
     --  and update the pointer to the top.
     self[top].can_read.take;
     res:= self[top].value;
     self[top].can_write.set;
     top := (top+1).mod(max_size);
  end; -- receive

end; -- BOUNDED_CHANNEL


-------------------------------------------------------------------------------
-- Objects of type ELEMENT{T} are used as components for the buffers in the  |
-- examples of BOUNDED_CHANNEL{T} and UNBOUNDED_CHANNEL{T}.                  |
-- The feature ('next') which is only used by UNBOUNDED_CHANNEL{T}           |
-- Each element has its own readable/writeable status.                       |
-------------------------------------------------------------------------------
class ELEM{T} is
   can_read:MONITOR0;
   can_write:MONITOR0;
   value:T;
   create: SELF_TYPE is
      res:= res.new;
      res.can_read:= res.can_read.new;
      res.can_write:= res.can_write.new;
      res.can_write.set;
   end;
   next:SELF_TYPE;  -- for easy constructions of lists (see UNBOUNDED_CHANNEL)
end;
```

```
class UNBOUNDED_CHANNEL{T} is ---------------------------------------------
--                                                                          |
-- create;                                                                  |
-- send(v:T);                                                               |
-- receive:T;                                                               |
--                                                                          |
------------------------------------------------------------------------------
--                                                                          |
--          ---     ---     ---     ---     ---     ---                     |
--     ...  | | --> | | --> | | --> | | --> | | --> | | ...                 |
--          ---     ---     ---     ---     ---     ---                     |
--           | ->                                      | ->                 |
--           |_top_                                    |_bottom_            |
--                                                                          |
-- This class is a specialization of a more generic VIRTUAL_CHANNEL{T}.     |
-- The data exchange between sender and receiver is totally buffered.       |
-- The sender must NEVER wait for the receiver in order to proceede, but    |
-- the receiver must wait for the sender when the buffer is empty.          |
--                                                                          |
-- In order to allow 'send' and 'receive' operation to proceede independently|
-- when the buffer not empty, any possible synchronization bottleneck between|
-- these two operations must be avoided.                                    |
-- In this case this is achieved implementing the channel buffer as a list  |
-- of independent elements (the same used in the BOUNDED_CHANNEL{T} case),  |
-- each one of which contains all the needed information for                |
-- enabling/disabling read or write operations.                             |
--                                                                          |
-- Even of object of type ELEMENT{T} cointain a 'next' feature usable to    |
-- build list of elements (and used to build the list actually modelling the |
-- channel buffer) the lists of free elements produced by the completion of |
-- 'send' and 'receive' operations are explicitly modelled as explicit      |
-- instances of LIST{T}. This is done simply for readability reasons.       |
-- (the code would become too complex if all the list were manually handled) |
------------------------------------------------------------------------------
  VIRTUAL_CHANNEL{T};

  private       -- these lists are used for recycling the created queue items;
    send_free_list,
    receive_free_list,
    interface_free_list: LIST{ELEM{T}};

  private top, bottom:ELEM{T};  -- head and tail of the buffer list

  create:SELF_TYPE is
     res := res.new;
     res.top := ELEM{T}::create;
     res.bottom:= res.top;
     res.send_free_list:= LIST{ELEM{T}}::create;
     res.receive_free_list:= LIST{ELEM{T}}::create;
  end;
  ...
```

```
  ...
  -- All the 'send' are already sequentialized by the outer lock
  -- There is at most one concurrent 'receive' at any time
  send(v:T) is
     new_item:ELEM{T};
     -- If the 'interface_free_list' is not empty, empty it and attach it
     -- to the 'send_free_list'.
     if interface_free_list /= void then
        send_free_list := send_free_list.append(interface_free_list);
        interface_free_list := void;
     end;
     -- Store the data in the vector (when possible),
     -- and update the pointer to the bottom
     bottom.can_write.take;
     bottom.value := v;
     -- prepare new bottom element
     if send_free_list.is_empty then
        new_item:= new_item.create;
     else   -- RECYCLE!
        new_item:= send_free_list.pop;
     end;
     bottom.next := new_item; -- Append the new element to the end of the buffer
     bottom.can_read.set;     -- Allow reading of this element
     bottom:= new_item;       -- Make the new element the new bottom
  end; -- send

  -- All the 'receive' are already sequentialized by the outer lock
  -- There is at most one concurrent 'send' at any time
  receive:T is
     new_item:ELEM{T};
     -- Retrieve the data from the vector (when possible)
     -- and update the pointer to the top
     new_item:= top;
     top.can_read.take;
     res:= top.value;
     top.can_write.set;
     top:= top.next;
     -- RECYCLE the element
     new_item.next:= void;
     receive_free_list := receive_free_list.push(new_item);
     -- If there is no danger of interfering with concurrent 'send'
     -- move the current list of free items to the shared interface
     if interface_free_list = void then
           interface_free_list := receive_free_list;
           receive_free_list:= receive_free_list.create;
     end;
  end; -- receive
end; -- UNBOUNDED_CHANNEL;
```

3.8 Example: Finding a max

The following two examples present contrasting ways to break up a task (max) into subtasks.

The first one does it using separate objects. The worker objects are of class MAX_MODULE and each has a 'main'. In this case, each worker copies a portion of the original array. In our simple case such a copying is not actually needed, but this example gives an idea of how in general a more complex task could be split in parallel subtasks. Once the workers have been created, their 'main' routine is called in parallel, and the generated results are further elaborated by "max".

The second example does much the same thing in one object, using threads to set up many copies of the procedure 'pmax' for each call of 'max' ('pmax' and 'max' are two routines of the same object). The threads all share the monitor 'rmax', which holds the running value of 'max'. They also can directly access the array of their object (using 'self[i]'). The threads could even update this array directly.

The big difference between the two styles is that the second example is more efficient, but less flexible. In the first example, the separate worker objects could communicate with one another, call each other's functions, etc. In the second case, threads can only communicate by shared variables.

```
-------------------------------------------------------------------------------
-- Example of max using objects to encapsulate parallel processes.
-------------------------------------------------------------------------------

 class MAX_MODULE{T} is ARRAY{T};
    private my_base:VECTOR{T};
    private my_start, my_grain:INT;

    crt(base:VECTOR{T}; start,grain :INT):SELF_TYPE is
       res:= res.new(grain);
       res.my_base:=base;                  -- a reference to the global vector
       res.my_start:=start;
       res.my_grain;=grain;
    end; -- crt

    main:T is
       i:INT;
       until i = my_grain loop             -- make local copy of data
          self[i] := my_base[my_start +i]; -- (in this case not quite necessary)
          i:=i+1;
       end;
       res := self[0];
       i:INT := 1;
       until i = my_grain loop             -- elaborate local data
          if self[i] > res then
             res := self[i];               -- update local max
          end;
          i := i+1;
       end;
    end; -- main
 end; -- class MAX_MODULE
```

```
...

class VECTOR{T} is ARRAY{T};
   max: T is
      -- adjust parameters
      n:INT := 10;              -- the maximum number of workers to use
      grain:INT := 10000;       -- the minimum number of operations per thread
      if asize < n * grain  then
         n := asize / grain;      -- reduce number of workers
      else
         grain := asize / n;      -- increase granularity
      end;

      -- create and start workers
      ans:MONITOR{T}:= ans.new;   -- holds the queue of partial results
      worker: MAX_MODULE{T};      -- each worker is a module object
      i,start:INT;
      until i >= (n - 1) loop          -- fork first n-1 workers (if n>1)
         worker := worker.crt(self,start,grain);
         ans :- worker.main;           --  start worker activity in parallel
         start := start + grain;
         i:= i+1;
      end;
      -- last thread gets remainder
      worker:= worker.crt(self, start, asize - start);
      ans :- worker.main;              -- start activity of last worker

      -- collect results
      lock ans.no_threads then end;   -- wait for all workers
      res := ans.take;                -- read the first result
      until ans.is_unbound loop       -- check the other queued results
         next_result:T := ans.take;
         if next_result > res then
            res := next_result;       -- update the final max
         end;
      end;
   end; -- max
end; -- VECTOR{T}
```

```
-------------------------------------------------------------------------
-- Example of max done with threads in one object.
-------------------------------------------------------------------------

 class VECTOR{T} is ARRAY{T};
    max: T is
       n:INT := 10;             -- the maximum number of workers to use
       grain:INT := 1000;       -- the minimum number of operations per thread
       rmax:MONITOR{T} := rmax.new;       -- running maximum, directly updated!
       workers:MONITOR0 := workers.new;   -- holds all the threads
       rmax.set([0]);
       if asize < n * grain  then
          n := asize / grain;                -- reduce number of workers
       else
          grain := asize / n;                -- increase granularity
       end;

       i,start:INT;
       until i >= (n - 1) loop               -- fork first n-1 workers (if n>1)
          workers :- pmax(start, grain, rmax);
          start := start + grain;
          i:= i+1;
       end;
       workers :- pmax(start,asize-start,rmax);-- last thread gets remainder

       lock workers.no_threads then            -- wait until all done
          res := rmax.read;                     -- return the result
       end;  -- join
    end; -- max

    private pmax(lower_bound, grain:INT;
                 rmax:MONITOR{T}) is         -- workers, each update rmax
       temp_max:T := self[lower_bound];       -- workers all share the big array
       i:INT := lower_bound;
       until i = (lower_bound + grain) loop
          if self[i] > temp_max then
             temp_max := self[i];             -- find local maximum
          end;
          i := i+1;
       end; -- loop
       lock rmax then
         if temp_max > rmax.read then
            rmax.set(temp_max);               -- adjust global maximum
         end;
       end;
    end; -- pmax
 end; -- class VECTOR{T}
```

4 Monitor Rationale

4.1 Why a unique "monitor" construct

From Section 3, the advantages of having a unique powerful "monitor" construct instead of some separate lower-level mechanisms should, at least in part, have become evident. Some of the fine-points related to this choice will be discussed in Section 4.2 while the overall issue is summarized here.

Monitors are the language means by which a program can support:

- Mutual exclusion
- Futures
- Signalling between threads
- Forking of new threads

Often these four basic mechanisms are kept separate in the language definition. E.g. it is quite common to have "locks" used to support mutual exclusion of critical regions, "messages" or "signals" used to suspend/resume a thread on some event, and "fork" operations executed to start new parallel threads. At least at the operating system level, or at the run-time-system level, these three mechanisms are often provided separately. Although there has been a great deal of effort along these lines, it has not been so successful as to discourage other designs.

A first problem is that there is no general agreement of which should be some standard low-level synchronization features. Many languages and systems define their own "basic primitive mechanisms" and it is quite difficult to find two languages which have adopted the same choice. Just to mention some of most common terms, we might consider the case of "spinlocks", "locks", "signals", "barriers", "semaphores", "lock-conditions", "monitors". Also when two languages or systems are said to be inspired by the same generic concept (e.g. a "monitor") the actual differences of the provided functionalities might be very great (i.e. "MESA monitor", " Hoare monitor", and now "pSather monitor").

Even when the functionality (or at least the syntax) could be more or less the same, the same mechanism is supported with different "flavours" in different systems, actually resulting in completely different semantics. This is true also for the most simple abstractions (e.g. a semaphore, or a spinlock) which can, for example, differ in the way in which the suspended threads are handled (e.g. resuming them in FIFO order, LIFO order, an unspecified order, ...)

The absence of any "standard" language construct or system primitive, makes easier the decision to define a new language concept, at least because it removes the issue of the loss of a potential portability to and from other systems and languages.

A second major problem in using separate low-level mechanisms, is that the basic locking/signal/fork functionalities alone are not always at the desired level of abstraction for supporting most high level programming paradigms. For example, "locks" and "signals" often need to be joined together into another basic primitive modeling the concept of "lock-condition", supporting an atomic operation which suspends a thread on a certain condition contemporaneously releasing a held lock. Another symptom of this is the continuous attempts to provide more sophisticated mechanism in the language or in the system (e.g. "barrier" primitives are provided directly at the low levels of the system or run-time). Higher level abstractions can always be built upon some lower level ones, but if we leave this task to the user we would get a much lower efficiency (the user not being able to perform the same degree of optimizations as the system) and we would lose the possibility of exploiting with some syntactic sugar some desirable high level synchronization mechanisms.

One of the goals in pSather is exploring this aspect of parallel programming and this is done also by experimenting the multiform concept of "pSather monitor": a mechanism which can be used in

a simple way for achieving the same functionality of a basic "spinlock", or a true suspending "lock", or a basic "signal", but which allows also much more complex synchronization schemas. The idea of having a unique built-in powerful, but flexible synchronization construct in the language seems to us quite attractive, even though more experience is probably needed for a complete evaluation of the usefulness of this choice.

A potential disadvantage of the choice of having a unique integrated mechanism (i.e. monitors) instead of three separate ones (i.e. locks, signals, and threads) is that each one of the separate mechanisms could be implemented in the most efficient way. Indeed, if having a unique "monitor" concept results in a much more inefficient realization of critical sections and basic thread signalling, probably the choice of having a unique multi-purpose construct should not be considered a good choice. However, we believe that the choice of having a unique "monitor" construct is not in conflict with the goal of being able to support in a very efficient way the other kinds of basic synchronization mechanisms. This simply means that some care must be taken in fine-tuning the monitor implementation in such a way as not to penalize its use in the simplest cases, and have the compiler detect and optimize simpler cases.

Another potential disadvantage of the choice of a unique mechanism is related to the issue of program readability. For example, if we see inside a class definition an attribute of the kind:

```
private my_lock:LOCK;
```

it is immediately clear that 'my_lock' is going to be used to ensure mutual exclusion in the execution of some code, or in the access of some data. This is particularly true when lock objects are exported as part of some package interface. Let us consider, for example, the case study of Section 3.7.

```
class CHANNEL{T} is
   send(v:T) is ... end;
   receive:T is ...  end;
   exclusive_send_rights: LOCK;
   exclusive_receive_rights:LOCK;
end;
```

In the above case, an explicit declaration of a LOCK object (instead of a more generic MONITOR0 object) makes it clear that the purpose of the object is exactly that one of allowing one thread to gain exclusive read or write access access to the channel, e.g. by executing:

```
   lock exclusive_send_rights then
        send(..);
        send(..);
        send(..);
   end;
```

Actually, this clarity is partially hidden anyway by the use of monitors, and we have to rely more on informal comments and meaningful names for making explicit the purpose of some object:

```
class CHANNEL{T} is
   send(v:T) is ... end;
   receive:T is ...  end;
   -- The following monitors are used only for locking purposes.
   -- Their bound/unbound/parallel status is irrelevant with respect to
   -- the class functionality.
   exclusive_send_rights: MONITOR0;
   exclusive_receive_rights:MONITOR0;
end;
```

But after all this is what we always do in programming, the many uses of integers also need to be spelled out in nouns and comments.

The choice of introducing the monitor concept under the form of pSather class instead of as a new language construct (another possibility was a special kind of class attribute), has been made after a long discussion on its relative advantages and disadvantages. This discussion is reported in detail inside Section 5.1 to which we refer.

Here we only mention that the differences between the two approaches were not big, and most of what could be done in one approach can be done also with the other.

The current choice has been made because it is more in the philosophy of Sather, and of object oriented languages in general, to include as much language functionalities as possible inside classes, instead of building them using special purpose constructs.

Apart from the syntactic difference of some operations the major difference between the two approaches is that in the case of monitors as special constructs we have a really built-in functionality, probably easier to optimize, and which has fixed and unchangeable semantics. The fixed and unchangeable semantics can be a property also of the current approach (monitor classes) if restrictions are imposed on what inheriting classes are allowed to do (and so it is). This approach, however, leaves surely more freedom in experimenting with the monitor functionality and in investigating the best kind of interface between runtime system and program code.

4.2 Details of the monitor design

The underlying idea on the pSather concept of "monitor", is that it should exploit the advantages of the fusion of the concepts of "lock", "signal", and "thread_call". Actually, there are several ways in which these basic concepts could be glued together and presented to the programmer. Most of the new functionalities resulting from combination of locks, signals, and thread calls could also be obtained from the original constructs themselves, often following appropriate programming paradigms (in the following sections several examples are illustrated). We have followed the following general criteria in the current design of monitors.

One of these criteria is the "usefulness" of the functionality (i.e. how likely is it to be used). If a given functionality is supposed to be used quite often, it is probably better to make it predefined either in the language definition or in some predefined library class. The "MONITOR" class is obviously a candidate for packaging these useful functionalities, even if it is not necessarily the unique class to serve for this purpose.

Another criteria to be taken into account is that one of "efficiency". In particular this criteria is important for deciding whether a certain functionality could be included inside some predefined, but "normal" library class (i.e. a library class fully written in pSather, maybe with some C interface, but which can be freely inherited and redefined) or whether a certain functionality should be included in some "built-in" "system" class (as ARRAY classes are), for which the compiler is able to perform specific optimizations, and maybe forbid the user from redefining certain features.

Another important criteria is that one of "rigor". Sometimes we want the language definition to precisely define the semantics of some operation, making it implementation independent and "user independent". This is often (but not necessarily) achieved by associating the functionality with a specific new language construct, and/or defining special built-in system classes or operations, of which the user ability to inherit and redefine has been constrained.

Sometimes it is just the "readability" or "ease of use" criteria which suggests the implementation of certain functionality as a special language construct.

The effects of the above principles on the MONITOR design are described in the following subsections.

Explicit "locking" statement

The usefulness of a new block-oriented construct for specifying a critical region controlled by a lock is quite obvious.

With a construct:

```
lock m then  ... end;
```

the language can guarantee that all the appropriate locks are released when the critical region is completed, and that only the thread actually holding a lock is allowed to release the lock itself. Moreover, all this can be done by the compiler, improving both run-time efficiency and program readability.

The alternative solution, i.e. that one of providing two explicit "lock_acquire" and "lock_release" operations seems much more error prone, and also resulting in less readable code.

Clearly this choice requires a "rigorous" semantics for locking/unlocking, but also this aspect does not seem undesirable.

Multiple locking

If more then one resource is needed before entering into a critical section, a possibility is that one of acquiring them in a known, deadlock-free order.

```
E.g     lock mon1 then
          lock mon2 then
              ...
          end;
        end;
```

One of the disadvantages of this style, is that the user must explicitly define and handle some kind of ordering of the locks used by the program. It is considered desirable if the language could simplify this kind of activity, requiring less effort by the user and reducing the risk of errors.

Another disadvantage of this style, is that while waiting for 'mon2', 'mon1' is held by the thread, and this might prevent other threads requiring only 'mon1', to continue their execution. This problem might become really serious if for some reason 'mon2' becomes never available, because in this way, a single thread deadlock might quickly propagate to many others.

The solution experimented in pSather is that one of providing a built-in construct in the language to acquire a set of locks atomically.

With the statement:

```
lock mon1, mon2 then
   ...
end;
```

the body of the statement is executed only when all the locks have been acquired, and that they are acquired only when they are all available together.

In this way the risk of deadlock is greatly reduced, even if the price is paid in terms of fairness. In particular, let us consider the following situation (already presented in Section 3.4, and further analized in Section 6.2.3):

```
lock mon1 then      lock mon2 then       lock mon1, mon2 then
   ...                 ...                  ...
end;                end;                 end;
```

If we use the construct:

```
lock mon1, mon2 then ... end;
```

instead of a pair of nested lock statements, it is true that we reduce the possibility of program deadlocks, and that the overall parallelism of the program is increased. However, now it is much more difficult for the thread needing both 'mon1' and 'mon2' to be executed, because it is more difficult for its two resources to be simultaneously available. In an extreme scenario, it is still possible that the thread needing both locks never gets executed (if the others are executing unbounded loops).

This loss of fairness can be considered acceptable, since it might be considered as part of the semantics of the multiple locking statements, (i.e. if it is never true that all the locks are available, it is intended that the lock-statement should never be executed). In fact, if a stronger fairness is needed, nothing prevents the programmer from explicitly using nested single locking statements.

Non-blocking locking

A functionality which has been considered quite useful, is the possibility to lock a monitor only in the case in which it is actually available. It seems that this functionality should be part of the language definition, since implementing it at the user lever might be quite complex and expensive.

This is why pSather defines the so-called try-statements:

```
try mon1 then
   -- critical-sect1
else
   -- something else not needing mon1
end;
```

A possible user-defined mechanism for implementing this functionality might make use of a specific class as in:

```
class NON_BLOCKING_LOCK is
   MONITOR0;
   actual_lock: MONITOR0;
end;
```

The conditional locking could be achieved by adopting the following locking paradigm:

```
mon1: NON_BLOCKING_LOCK;

lock mon1 then
   if mon1.actual_lock.is_unlocked then
      lock mon1.actual_lock then   -- no wait
         unlock mon1;
         -- critical-sect1
      end;
   else
      -- something else not needing mon1
   end;
end;
```

With a non-blocking version of locking statement (whose usefulness is also enriched by the possibility of associating predicates to monitor expressions) we get also the advantages of:

- language efficiency: try-statement requires at most one locking, and is easy to implement very efficiently.

- program readability: "try" is much more readable then the equivalent programming pattern.
- program safety: A programmer does not have to follow strange programming paradigms for obtaining similar functionalities.

It seems that the advantages of a built-in implementation of non-blocking locking overcome the disadvantages of a slightly more complex language definition and implementation.

Explicit unlocking

Currently pSather allows one to explicitly unlock a monitor while still executing the body of the lock-statement which locked it.

```
E.g.        lock m, n then
               ...
               unlock n;
               ...
            end;
```

This is an example of primitive functionality which otherwise would be impossible to support if it were not provided in the language definition. The usefulness of this statement has already been discussed in Section 3.2.

The unlock operation is lexically constrained to appear inside the body of a locking statement. Moreover the monitor to be unlocked must not only be actually locked by the executing thread (clearly an attempt to unlock a monitor not locked by the executing thread is an error), but must have been locked by a lexically enclosing locking statement.

The intention, here, is to make as far as possible explicit the underlying synchronization pattern. If unlock operations were allowed to appear anywhere (e.g. inside a routine), it can become very difficult to understand the concurrent behavior of a pSather program. For this reason the following piece of code would raise a run-time error.

```
m: MONITOR0 := m.new;
n: MONITOR0 := n.new;

foo is
   lock m then
      unlock n;  -- Even if 'n' is currently locked by the executing
   end;          --   thread, its unlocking is a detected run-time error.
end;

main is
   lock m then
      foo;     -- An attempt to unlock inside 'foo' is a run-time error.
      ...      -- At this point 'm' is still locked.
   end;
end;
```

The unlock operation introduces some complexity in the implementation of locking statements, because the implementation must keep track of which monitors should be released when a locking statement is completed. We will describe an efficient implementation of this feature in Section 6.

The "unlock" functionality has been introduced as a new statement, instead of as a monitor operation, for its similarity with the break-statement: In both case there are some lexical restrictions on the places where they can appear.

"void" monitors in locking statements

During the execution of a locking statement it is considered a run-time error if any of the monitor expressions evaluates to "void". At a first glance this restriction might appear unnecessary, since "void" values could be easily discarded.

The reason for this rule is related to the safety of locking statement, in a way partly similar to the constraints on the use of early unlocking.

Let us consider the following code:

```
lock m then
   -- critical section
end;
```

If for some previous error "m" evaluates to void, but this event is not considered an error, the critical section would be executed without ensuring mutual exclusion. This can be terribly hard to find.

Having a monitor expression which evaluates to void in the left-hand-side of `:-` does not have these nasty consequences (since any attempt to take read or lock results into an error) but for uniformity we have decided to also treat this case as an error.

Monitor predicates in locking statements

The usefulness of this monitor feature has already been presented with some examples in Section 3.2. In Section 5 this will be further analyzed from a different point of view (why isn't this feature even more powerful).

One of the first questions raised by the possibility of locking a monitor only when a certain test condition is true, is whether this functionality could be simulated easily by simple locking and simple signal exchange. Even if in some particular cases this functionality can be obtained by not so complex alternative programming styles, clearly in general this is not true.

The easiest predicate to simulate is surely the *is_bound* test. In fact we already have two similar suspensive operations (*read* and *take*) which could be used to get part of the original functionality.

Simulating the other conditions (*has_threads*, *no_threads*, *is_unbound*) would require some probably drastic rewriting of the program, resulting in much less readable and efficient code.

Locking the "signal" operations

As already presented in Section 3.1, one of the properties of "monitors" is that we can use their "lock" aspect to protect and control their other "signal" aspect. If fact, when a monitor is locked by one thread, no other threads are allowed to perform read/set/take/thread-call operations on it.

This functionality can be almost completely modelled using separate lower-level lock and signal abstractions, as the following example tries to illustrate:

```
class CONTROLLED_SIGNAL is
    obj_lock:LOCK;                 -- only simple locking supported
    private is_bound:BOOL;
    private waiting:QUEUE{SIGNAL}; -- only 'set' and 'take' supported

    create:SELF_TYPE is
      obj_lock:=obj_lock.create;
      waiting:= waiting.create;
    end;
    ...
```

```
...
set is
  lock obj_lock then        -- get the lock
    if not waiting.is_empty then
       waiting.pop_top.set;  -- resume first waiting thread;
    else
       is_bound:=true;
    end;
  end;                       -- release the lock
end; --set;

take is
   my_signal:SIGNAL;
   lock obj_lock then       -- get the lock
      if not is_bound then   -- store my_signal
         my_signal:= my_signal.create;
         waiting.insert_bottom(my_signal);
      else
         is_bound:=false;
      end;
   end;                      -- release the lock
   if my_signal/=void then
       my_signal.take;       -- wait if necessary
   end;
  end; -- take
end;-- CONTROLLED_SIGNAL
```

An aspect which is very difficult to re-introduce and control in this way is the "parallel" status of the monitor. Even if it were possible to execute a thread-call using this CONTROLLED_SIGNAL instead of a MONITOR (e.g. making CONTROLLED_SIGNAL inherit from MONIOTR0, and allowing the redefinition of 'set' and 'take'), achieving a consistent handling of `:-` and thread completions would be extremely difficult.

Clearly this kind of user-defined functionality would also result in much lower efficiency than a built-in definition.

Handling of thread completions

The current design, in which a signal (or return value) is associated to each thread upon completion, possibly used to bind the monitor (or add an element in the monitor queue), has been reached after many revisions, some of which are presented in Section 5.5.

With the current design, we wanted to easily support the four programming styles presented in Section 3.3, in the subsections:

- Simple "futures"
- Searching for the "first" result
- Modeling co-begin/co-end
- Many threads producing many results

The current design is supposed to be the simplest which allows us to achieve all the above goals.

One decision which might at first glance not appear fully justified, is that the signal caused by a thread completion is not subject to the same locking constraints as the signal generated by

a *set* operation. Actually, there is another major difference between *set* operations and thread completions, which is that a *set* operation on an already bound monitor does not "enqueue" the signal or value, but directly overwrites the current value associated with the monitor bound status.

The fact is that, when monitors are manipulated by means of *set*, *read* or *take* operations, we would like to see them as some kind of "protected" and "synchronized" variable (or attribute).

When monitors are used inside thread-calls they get the additional role of acting as a "controller" for a set of threads (the attached threads), extending the idea of a "protected variable" to the idea of a "protected queue of values". The choice of making thread completions insensitive to locking is justified mainly by the desire to preserve the same programming style while using a local monitor or a shared one. In the second case, all we have to do is to use it inside a locking statement for preventing external interferences. E.g.

```
m:MONITOR0;     -- a local monitor;
m.clear;
m :- foo(..);
m.take;

shared m: MONITOR0;     -- a shared monitor;
lock m then
   m.clear;      -- (1)
   m:-foo(..);   -- (2)
   m.take;
end;
```

Notice that, if in (2) we would disable the final binding performed by the thread 'foo', this second case would result into a deadlock. This actually means that, once a monitor is locked, the completion of its attached threads is not considered an "external" interference, since the "attached threads" are already considered part of the monitor status (if we do not want this part of the monitor status to possibly affect the subsequent execution we can still clear it as done in `(1)`).

Why a parameterized type

Whenever we execute thread-call:

```
int_mon :-  int_function(...);
```

We rely on a predefined system operation which will get the integer result, using it for binding the monitor, or storing it inside a queue. This is the basic reason for which we need a predefined built-in parameterized version of monitors.

If monitors were only used for locking, setting, reading and taking, and NOT for forking, we might define MONITOR{T} as a non-predefined type, simply constructed from MONITOR0 in the following way:

```
class MONITOR{T} is
    MONITOR0;
    private v:T;

    read_value:T is
       lock self.is_bound then
          res:= v;
       end;
    end;
    ...
```

```
      ...
      set_value(new_v:T) is
         lock self then
            self.set;
            v := new_v;
         end;
      end;

      take_value:T is
         lock self.is_bound then
            res:= v;
            self.take;
         end;
      end;
   end; -- MONITOR{T}
```

We have also an efficiency gain, since predefined classes are more optimizable by the compiler and runtime system than user-defined classes.

Why a non-parameterized type

Since MONITOR{T} is needed as a primitive type, then why don't we get rid of MONITOR0? After all, we might suppose that if a procedure, and not a function, is forked from a monitor 'm' of type MONITOR{T}, a "void" value is simply used to bind the monitor (or enqueued).

A drawback of this possible approach, is that currently we can write:

```
m : MONITOR0;

m :- int_function(11);        -- return value discarded
m :- bool_function(22);       -- return value discarded
m :- real_function(33);       -- return value discarded
m :- some_procedure(44);

lock m.no_threads then end;   -- wait for all
```

Meaning that, in the case in which we fork some function, the return value is discarded (but not the completion signal). But now, if we use a typed monitor instead of MONITOR0, as in:

```
m : MONITOR{SOME_TYPE};

m :- int_function(11);
m :- bool_function(22);
```

This would be now illegal, since INT is probably incompatible with SOME_TYPE, and in any case incompatible with BOOL, etc. We now have stricter type rules to follow.

Another problem is that now we can write:

```
m : MONITOR0;
m.set;        -- 'set' has no parameters
```

while if we use any instantiation of the parameterized version, we have to use a dummy value as a parameter of the 'set' operation. I.e. the above would become:

```
m : MONITOR{SOME_TYPE};
m.set(dummy_value);       -- at best it could be 'void'
```

If we simply need to set the bound status, having to specify some dummy value in any case is not nice.

4.3 Inheritance

One of the effects of introducing in pSather the monitor functionality under the form of classes, is that the user is now allowed to define new classes which inherit from the predefined ones.

We do not want to give the user the possibility of changing the semantics of lock-statement or thread-call. This means that classes inheriting from MONITOR{T} do not have visibility of the internal operations and structures used by the runtime system.

Not having the visibility of the internal structures, means also that the user is unable to give any reasonable redefinition of operations like *set*, *read* or *take*. This is why also these operations are made not redefinable inside the descendent classes.

Allowing the user to add new functionalities

What a user is allowed to do inside a class which inherits from a MONITOR class, is to add new features completely unrelated to the original monitor functionality, or to extend the primitives with extra functionality (using a new name for the extended version).

For example, a `next: MONITOR0` feature could be added in order to simplify the handling of lists of monitors without having to explicitly embed them inside other structures. Similarly a "fetch_and_add" operation could be added to an INT instantiation of MONITOR{T}.

```
E.g.   class INT_MON is
          MONITOR{INT};
          fetch_and_add(incr:INT):INT is
             lock self.is_bound then
                v:INT := self.take;
                self.set(v+incr);
                res:=v;
             end;
          end;
       end; -- INT_MON
```

The following is an extension of the predefined *take* allowing to collect some kind of profiling information on the monitor.

```
my_take is
   lock self.is_bound then
      self.take_counter:=  self.take_counter + 1;
      self.take;
   end;
end;
```

For easiness of writing, finally, it could be useful to define:

```
join is
    lock self.no_threads then end;
end;
```

5 Discussion of alternative choices

In this section we summarize some aspects of the monitor design for which alternative solutions had been considered at some time, even though they were discarded eventually. Some of these points have required long discussions and analysis before being dismissed, others have had a much shorter life.

The reading of this section is not essential for a comprehensive understanding of the current design and rationale (which are also structured in different sections), but might still give some insights to the interested reader.

5.1 Predefined classes vs. special entities

Two main approaches have been discussed at length about how to introduce in pSather the "monitor" functionality. They are first briefly introduced, and their advantages and disadvantages compared.

5.1.1 Monitors as special declarations

The first approach (referred to as "monitors as special declarations") is to introduce in pSather a new declarator which allows one to specify when a class feature or variable should be considered a monitor.

i.e. if we declare:

```
class A is
  monitor m :INT;
  ...
end;
```

This means that the 'm' attribute, beyond being an integer attribute, has also all the properties of monitors. Upon creation it is unbound, hence any attempt to read its value before it has become bound causes the thread to become suspended. (Similarly if 'm' is a local variable or a shared)

```
foo is
   monitor m :INT;
   n:INT := m; -- the thread becomes suspended
end;
```

The monitor entity 'm' becomes bound when some value is assigned to it:

```
foo is
   a1:A := A::new;
   a1.m := 111;
   n:INT := a1.m;
end;
```

or when a thread forked from the monitor finally terminates:

```
foo is
   monitor m: INT;
   m :- new_thread(..);
   n:INT:= m;   -- wait until 'new_thread' is terminated
end;
```

In addition to the operations defined by its type, a monitor attribute supports the *take* operation (a language keyword) and can appear inside locking statements.

```
foo is
   a1:A := A::new;
   a1.m := 111;              -- now 'a1.m' becomes bound
   n:INT := a1.m.take;       -- now 'a1.m' becomes unbound again

   lock a1.m.is_bound then   -- wait until 'a1.m' is bound, then lock it
      a1.m.take;             -- at this point 'a1.m' was surely bound,
   end;
end;
```

5.1.2 Monitors as objects

The second approach (referred to as "monitors as objects") consists of packaging the monitor functionalities inside a parameterized class MONITOR{T};

In this case, there is no special kind of attribute, but any attribute, local variable or routine parameter can hold a reference to a monitor object, by simply being declared of type MONITOR{T}. This is the current design described in this report.

Since the type of a monitor object is MONITOR{T}, and not T, we have the immediate consequence that `m1 := m2` (if 'm1' and 'm2' are two monitor objects) now means that the variable 'm1' now holds a reference to the same object as 'm2', and not that the T value of 'm2' is read (supposing 'm2' is bound) and used to bind 'm1'. If we want to get the T value associated with a monitor object 'm1', now we have to use an explicit reader operation, and if we want to bind a monitor object with a new T value we must use an explicit writer operation:

```
class A is
   m:MONITOR{INT};
   ...
end;

foo is
   a1:A;
   ...
   a1.m.set(111);     -- the object referred by 'a1.m' becomes bound
   n:INT:= a1.m.read; -- the value of the monitor referred by 'a1.m' is read
end;
```

5.1.3 Advantages of the "monitors as special declarations" approach

Implicit binding and reading operations A bound monitor can be accessed as simply as any other attribute or local variable.

For example, we can write:

```
class A is
   private monitor m: INT:
   ...
   incr is
      m:= m+1;
   end;
end;
```

On the contrary, in the "monitors as objects" case we have to use the less pleasant syntax:

```
class A is
   private m: MONITOR{INT};
   ...
   incr is
     m.set(m.read +1);
   end;
```

However, we could also argue that the simpler syntax for monitor entities is even too simple because it hides behind an apparently normal assignment and entity name evaluation a much more complex interaction (i.e. the fact that the thread calling 'incr' might become suspended when reading the integer value of 'm', if 'm' is unbound or locked).

Actually, writing `m:=m+1` is much easier than writing `m.set(m.read +1)`, and the writer is likely to be aware of the fact that 'm' is a monitor.

Monitor entities are not variables When a class declares a monitor attribute, the program syntax makes very clear the fact that the monitor is some kind of stable (constant) property of the objects of the class.

I.e. in the following example:

```
class A is
   monitor m:INT;                    -- (1)

   foo is
      lock m then                    -- (2)
         if not m.is_unbound then    -- (3)
            m.take;                  -- (4)
         end;
      end;
   end;
end; -- A
```

It is perfectly clear from the class definition that 'm' denotes exactly the same monitor "entity' in all the points `(1)`,`(2)`,`(3)`, `(4)`. This increases the program readability and allows the compiler to perform some kind of optimizations (e.g. not to check for the locked status in `(3)`).

The above example, in the case of "monitors as objects" would become:

```
class A is
   m: MONITOR{INT};                  -- (1)

   foo is
       lock m then                   -- (2)
         if not m.is_unbound then    -- (3)
            m.take;                  -- (4)
         end;
       end;
   end;
   ...
end; -- A
```

Even if the structure of 'foo' is exactly the same, now 'm' denotes a necessarily non-constant reference to a monitor. In particular, when an `A` object is created by a 'A::new' operation, 'm' is "void". In this case probably a 'create' operation is needed to properly initialize the value of 'm',

and only a global checking of the class code allows us to deduce that, once initialized, the monitor denoted by 'm' is always the same. In this case, in fact, attributes of type MONITOR{T} behave as variables holding some monitor object reference, and they no longer directly denote a constant monitor entity.

We can probably argue that this loss of expressibility in the "monitor as objects" case is due to the Sather weakness of not allowing object-based constant attributes (i.e. attributes which get their initial value from their initialization expression, and which cannot be further assigned during the lifetime of the object to which they belong).

Another annoying consequence, in the case of "monitors as objects", is related to the fact that they normally behave as variables. It is that directly exporting (as a public feature) an attribute of type MONITOR{T} might give the clients of the class the wrong impression that they should have some reason for assigning a value to the attribute itself from the outside.

Actually the declaration of a public attribute of type T is something which explicitly says: "Hi, I am a directly READABLE and WRITABLE attribute of type T".

If this is not what intended by the class designer, in order to have its code deliver the correct message to the reader, we have to write:

```
class A is
   m:MONITOR{INT} is res:= my_m; end;    -- a reader function
   private my_m: MONITOR{T};             -- a private object
   ...
end;
```

that is, to declare the attribute as private and export a reader function which returns its value.

Alternatively we could write something like:

```
class A is
   m:MONITOR{INT};
   -- BEWARE! THE ABOVE ATTRIBUTE SHOULD NOT BE TARGET OF ASSIGNMENTS!
   ...
end;
```

In the previous case the first situation is formally more correct, and some language tool could easily understand that the monitor is not updated from the outside of the class, without studying all the program, but just noticing that it is declared as private. However the class looks more complex for an human reader, and the code is less efficient. In the last case, the comment makes clear for other readers the intentions of the designer of the class, but unfortunately they are usually not correctly interpreted by most of the language tools (typically no compiler will give any warning if some client wrongly updates the attribute value).

In any case, unless object-based constants and local attribute initializers are introduced in the language, the clarity and efficiency achievable in the case of "monitors as special declarations" can hardly be matched by the other approach.

No need of explicit monitor creation Since the monitor aspect of monitor attributes is directly part of the attribute structure, we do not need any explicit creation operation to be called to set up this kind of object information.

I.e. in the case:

```
class A is
   monitor m:INT;
   ...
end;
```

Each time an 'a' object is created (of type 'A'), automatically a monitor entity is created and associated with the 'a.m' attribute.

On the converse, in the "monitors as objects" case, there is no special kind of information associated with any attribute, since all the needed monitor structures are part of the monitor objects eventually pointed by the class attributes. This implies that these monitor objects have to be explicitly created.

E.g. the above example would become:

```
class A is
   m: MONITOR{INT};

   create: SELF_TYPE is
      res:= res.new;
      res.m:= res.m.new;
   end;
end;
```

For very simple classes, this might imply the definition of an additional 'create' routine, which might otherwise often be avoided. Apart from the greater complexity of the resulting code, the additional function call might have an impact also in terms of efficiency.

The situation in this second case improves if we are allowed to specify initializers at the point of declaration of class attributes (supposing that any call to 'A::new' properly initializes all the class attributes).

e.g. if we are allowed to write:

```
class A is
   m: MONITOR{INT} := m.new;
end;
```

The current design allows this kind of attribute initializations, which should however be used with care in order to avoid unbounded recursions during creations.

Stack allocation for monitor local variables Another advantage of having the monitor status directly associated with a language construct, is that in the case of monitor local variables the information relative to the monitor status could be stored into the current stack frame, instead of in the heap. The advantage of this choice is essentially less memory garbage, thus reducing the need of garbage collecting, and probably also a greater efficiency not having to ask for memory from the program heap and achieving a greater locality of the program data.

Notice in fact that no other references to a monitor local variable can exist when the routine declaring the monitor is terminated, because monitor entities cannot be passed as routine parameters and cannot be assigned (only their value can be accessed).

5.1.4 Advantages of the "monitors as objects" approach

Explicit binding/ reading operations We have discussed in the previous Section the advantages of a simpler syntax for monitor reading and binding, from the point of view of the programmer who can simply write 'm := m+1' instead of 'm.set(m.read +1)'.

However, classes are often written once (by one person) and read many times, by many persons.

And when reading a class routine:

```
foo (x:INT) is
   a.m:= 111+x;
   n := a.m;
   b.bar(a.m);
   ..
end;
```

it might actually be helpful to be able to understand, at a glance, that 'a.m' is a monitor and not simply an integer attribute, because this fact gives some immediate insight on the concurrent structure and execution costs of the program.

So the fact of having explicit *read/set* operations might be considered an advantage from the point of view of program readability, because it makes more evident the concurrency aspects of the program.

Finally, if the binding operation is explicit, we can easily suppose the existence of a parameterless 'set' operation used for binding a typeless monitor (i.e. a monitor not associated with any T value).

E.g.
```
class A
   m: MONITOR0;

   foo is
      m.set;
      m.take;
   end;
end;
```

In the case of "monitors as special declarations" we would still be able to define a typeless monitor entity, but we would probably need an additional explicit operation to be able to bind it.

E.g.
```
class A
   monitor m;      -- without any specific type

   foo is
      m.set        -- instead of 'm := ...'
      m.take;
   end;
```

Monitor objects and monitor types are sometimes necessary One of the intended uses of monitors, is that of constituting a reference to some ongoing concurrent computation, whose result will be used to bind the monitor itself. When this functionality is packaged inside an object, what we obtain is a so-called "future", that is, an object whose reference can be freely assigned or used as actual parameter, but which initially does not yet hold a defined value. This "future" functionality is directly modeled by the monitor abstraction in the case of "monitors as objects", while in the case of "monitors as special declarations" can be supported by a library class of the kind:

```
class MONITOR{T} is
   monitor m:T;
end;
```

Objects of this class are very similar to the monitor objects in the "monitors as objects" case, the main difference being the fact that the monitor properties of these objects are explicitly associated with their 'm' attribute (and the syntax for binding and reading is different) and not with the object itself.

I.e. in the "monitors as special declarations" case we have to write:

```
my_mon: MONITOR{INT}:= my_mon.new;

lock  my_mon.m then                -- Lock the monitor
   my_mon.m:- forked_thread(..);   -- Fork a thread
   n:INT := my_mon.m.take;         -- Wait the thread termination
end;
```

In the "monitors as declarations" case, the class MONITOR{T} plays a very important role, because the explicit use of this type name is needed if we want, for example, an array of monitors, a stack of monitors, or simply to pass a monitor reference to a routine.

In all these cases, in fact, we cannot avoid to use an explicit monitor type name as in:

```
my_vect: ARRAY{MONITOR{INT}};

my_stack: STACK{MONITOR{INT}};

foo (monitor_arg: MONITOR{INT}) is ... end;
```

From this point of view, the "monitors as objects" case has the advantage of generating a more uniform language, being monitor objects. Above all, monitor types are probably unavoidable in many cases.

Memory allocation In the previous Section we have said that an advantage of the "monitors as special declarations" approach was the possibility to allocate the monitor data on the stack instead of from the heap (producing less garbage, and improving data locality).

One of the disadvantages of that approach, however, is that the lifetime of a monitor is necessarily tied to the lifetime of the declaring routine. The threads forked from a monitor declared as a monitor local might have a much longer lifetime, and when terminating, they should be able to detect that the original monitor no longer exists. This makes the overall semantics of forking threads a little strange, and in any case makes the implementation of threads more complex.

If monitors are heap-allocated instead, their memory space is not released until all the forked threads are terminated, and a forked thread can rely on the fact that its monitor still exists for the final binding operation.

Clearly, nothing prevents an implementation from allocating monitors on the heap also in the case of "monitors as special declarations". However, if an implementation adopts this choice (allocating them in the heap and not releasing their space upon the routine completion) a lot of memory garbage is produced in an hidden way, which the programmer cannot neither see nor control.

At this point it is much better to make the creation explicit and under user control, so that, if memory is a concern, the user might rely on its own recycling of monitors (e.g. handling a pool of objects) instead of continuously creating them and putting them into the garbage.

Finally we could argue that, if data locality and non-heap allocation were a language goal, probably this kind of issues should be handled in a more general way and not restrict them to the monitor case.

One less declarator If monitor are just a particular kind of objects, of type MONITOR{T} or MONITOR0, there is no need of the special "monitor" declarator. This makes the language a little simpler, and maybe easier to maintain in the long term. E.g. if in the future a new kind of synchronization type is found useful, it is much easier from the compiler point of view to introduce a new class other than a new declarator, since this does not affect all the compiler front-end.

Sather philosophy of classes Probably it is more in the philosophy of object oriented languages in general, and in Sather in particular, to include as much language functionality as possible inside classes, instead of building them inside special purpose constructs. E.g. often also arrays are introduced in the language just like a normal parameterized class, even if in Sather, for efficiency reasons, this choice has not been adopted.

Possibility of extending the definition (inheritance) One of the major impacts of the "monitor as objects" approach, is that, at least in principle, it becomes now possible to define new classes which inherit from the predefined ones, e.g. adding new attributes and features.

If adding new independent features (which do not interfere with the original monitor semantics) is a clean and safe practice, the possibility or redefining the main synchronization features (i.e. the *set* or *take* operations) might be the source of major program inconsistencies.

How to constrain what can be done by the classes which inherit from a monitor class is still an issue not completely investigated, and which poses interesting problems. From one side, in fact, we do not want to give the user the possibility of interfering with the basic runtime system functionalities (i.e. the semantics of locking and `:-` should not be affected by any user defined code). From the other side it is really interesting to investigate the possibility of allowing the user to define new kinds of monitors. E.g. "instrumented" monitors which collect statistics on their usage, or which log into a file all the operations performed on them for a subsequent program execution analysis.

For now we can suppose that monitor classes behave in a way very similar to the array classes. I.e. the user can extend them with new independent features, but not redefine predefined operations (e.g. the meaning of indexing).

5.1.5 Conclusions

The differences between the two approaches are not big, and most of what can be done in one approach can be done also with the other. Apart from the syntactic difference of some operation (i.e. direct reading/ assigning vs. read/set operations), the major difference is that in the "monitors as special declarations" case we have a really built-in functionality, probably easier to optimize, and which has fixed and unchangeable semantics. The fixed and unchangeable semantics CAN be a property also of the other approach ("monitors as objects"), if restrictions are imposed on what inheriting classes are allowed to do. This second approach, however, also leaves more freedom in experimenting with the monitor functionality and in investigating the best kind of interface between runtime system and program code.

5.2 Disjunctive locking

Disjunctive locking is one of the features which has not been introduced in the pSather definition. A possible syntactic presentation of "exclusive locking" might be following:

```
lock mon1 then
   -- critic-sect1
or_lock mon2 then
   -- critic-sect2
end;
```

With the above we might express the intention that, if 'mon1' is available then it is acquired and the first critical region is executed, otherwise if 'mon2' is available then 'mon2' is acquired and the second critical region is executed, else the thread is suspended until one of the two locks becomes available, after which it is acquired and the corresponding critical section is executed.

While approximating the required behavior by means of busy-waiting is quite easy, achieving the same result by thread suspension and resumption is quite harder. For example we might make use

of signals and conditional locking in the following way. Suppose that whenever one of the two locks 'mon1' or 'mon2' is released, a signal 'retry' is set:

```
mon1, mon2, retry: MONITOR0;

until done loop
   done:=true;
   try mon1 then
     -- critic-sect1
     retry.set;
   else
      try mon2 then
        -- critic-sect2
        retry.set;
      else
        retry.take;
        done:= false;
      end;
   end;
end;
```

Clearly the above implementation is less fair than what the language itself could guarantee with an explicit `lock ... or_lock ... end;` statement. Moreover we must rely on the fact that a consistent programming style is used by the programmer(s) (i.e. 'retry' is always set when a lock is released). However, since this kind of functionality does not seem very common, the disadvantages of the user-based approach seem preferable to the additional complexities which would otherwise need to be introduced in the language.

5.3 Atomically unlocking and waiting on another condition

Suppose that, while holding the lock of a first monitor, we want to atomically release the lock and wait on some other condition (represented by the taking of a second monitor).

Currently, if we write the following code, we simply wait on the second monitor leaving our first monitor locked.

```
lock my_mon then
   ...
   other_mon.take;
   ...
end;
```

On the other hand, if we write:

```
lock my_mon then
   ...
   unlock my_mon;
   other_mon.take;
   ...
end;
```

it might happen that between the time we release the first monitor and the time we take the second one, some other thread locks our first monitor and takes the second one, thus violating our atomicity assumption. (suppose the first monitor was supposed to protect the second one)

This particular kind of synchronization (releasing one monitor and taking another) could be obtained by an extension of the monitor operations.

Suppose that we introduce a new "unlock_taking" statement which has the effect of atomically releasing the lock of a monitor and waiting until it becomes bound. In this case we could solve the original synchronization problem in the following way:

```
lock my_mon then
     ...
     lock other_mon then
        unlock my_mon;
        unlock_taking other_mon;
     end;
     ...
end;
```

Even more generality would be achieved by allowing the "unlock_taking" statement to work on two different monitors (i.e. unlocking the first one and taking the second one).

But this risks to become an open door for all the possible combinations of two-monitor synchronization primitives.

An example of the usefulness of a "unlock&wait" operation is the common paradigm associated with standard (HOARE [26]/ CONCURRENT-PASCAL [14]) monitors.

The following is a typical example of HOARE-monitor construct (in Sather syntax):

```
class HOARE_BOUNDED_BUFFER{T} is

   private non_empty:CONDITION;
   private non_full:CONDITION;

   append (x:T) is
      if <buffer full> then
         non_full.wait;
      end;
      <store element>
      non_empty.signal;
   end;

   remove:T is
     if <buffer empty> then
        non_empty.wait;
     end;
     <delete element>
     non_full.signal;
   end;
   ...
end; -- HOARE_BOUNDED_BUFFER{T}
```

The underlying programming pattern is that, when any of the 'append' or 'remove' operation is called, the buffer is locked by the executing thread (so that all the class operations are mutually exclusive). If the operation can be completed the buffer lock is released at the end of the operation. If the operation cannot be completed immediately, the buffer lock is released, while the thread is contemporaneously suspended on its completing condition. When a thread is resumed, it gets again the buffer lock and completes its previously suspended operation.

The above programming pattern can be easily supported in **pSather**, introducing a new attribute in the class, explicitly modeling the buffer lock and adding two others attributes modeling the queue of events.

E.g.

```
class BOUNDED_BUFFER{T} is
   private buffer_lock: MONITOR0;
   private not_full: CONDITIONS;
   private not_empty: CONDITIONS;

   create(n:INT):SELF_TYPE is ...  end;

   append(x:T) is
      buffer_lock.take;
      if <buffer-full> then
         not_full.wait(buffer_lock);
      end;
      <store the element>
      non_empty.signal(buffer_lock);
   end;

   remove:T is
      buffer_lock.take;
      if <buffer empty> then
         not_empty.wait(buffer_lock);
      end;
      <remove the element>
      not_full.signal(buffer_lock);
   end;
end; -- BOUNDED_BUFFER{T}


class CONDITIONS is
   QUEUE{MONITOR0};

   wait (my_lock:MONITOR0) is
      my_event:MONITOR0:= MONITOR0::new;
      insert_bottom(my_event);
      my_lock.set;            -- release buffer lock
      my_event.take;          -- wait for my event
   end;

   signal (my_lock:MONITOR0) is
      if is_empty then
         my_lock.set;      -- release the buffer lock;
      else
         pop_top.set;      -- resume suspended operation
      end;                 -- leaving the buffer locked for it
   end;
end; -- CONDITIONS
```

Instead of using an explicit queue of events for modeling the wait conditions, in certain cases

(i.e. when the wait conditions are pre-conditions) we might make use of the implicit queue of the monitor definition. For example, the bounded buffer (which is one of these simpler cases) might be re-written in pSather as:

```
class BOUNDED_BUFFER{T} is

   private buffer_lock:MONITOR0;          --  sequentialize append/remove
   private not_full, not_empty:MONITOR0;  --  preconditions  for append/remove

   create(size:INT):SELF_TYPE is
     ...
   end;

   append(v:T) is
     not_full.take;             -- wait until allowed to store data
     lock buffer_lock then      -- ensure mutual exclusion with remove
        <store element>
        not_empty.set;
        if <not buffer full> then
           not_full.set;
        end;
     end;  -- lock
   end; -- send

   remove:T is
     not_empty.take;            -- wait until allowed to read
     lock buffer_lock then      -- ensure mutual exclusion with append
        <remove element>
        not_full.set;
        if <not buffer empty> then
           not_empty.set;
        end;
     end; --lock
   end;
end;  -- BOUNDED_BUFFER{T}
```

In conclusion since it does not seem difficult to write the appropriate abstractions explicitly using monitors and signals, it is probably better not to introduce additional primitives to the already available ones. After all, the HOARE-MONITOR abstraction is not the best concurrent style which is encouraged by pSather. In fact there is in general no reason for forcing all the operation of a class to be fully sequentialized. For example, the definition of CHANNEL (Section 3.7) provides a better implementation of a bounded buffer, which allows append and remove operations to proceed in parallel.

5.4 Predefined monitor test operations

We note the absence of the obvious monitor predicate *is_locked*. Actually, in earlier versions of the language this predicate was defined. It has been removed for essentially three reasons:

- It can be easily defined, if needed, by explicitly introducing in a subtype of MONITOR the definition:

```
is_locked:BOOL is
   try self then
      res:= false;
   else
      res:= true;
   end;
end;
```

- The pure "is_unlocked" functionality is not so useful, in practice, because the monitor could immediately be locked. The 'try' statement is much safer.
- If "is_locked" were a predefined predicate, either we had to introduce a non-uniformity in the use of predefined predicates inside locking statements, or we had to allow the following coding:

```
lock m.is_locked then
    ...
end;
```

  whose semantics is clearly inconsistent.

Another observation is that, since the bound status of a monitor is associated with a value, we could have introduced a predefined test of the kind:

```
monitor.has_value(v);
```

Clearly the only usefulness of this predicate would be as a locking predicate, to suspend a thread until a monitor becomes not only bound, but bound with a specific value. Notice that this predicate would require a check of the predicate at every "set" operation, while currently a "set" operation might affect a thread suspended on a locking statement only if the monitor was unbound.
E.g. if 'int_mon' is of type MONITOR{INT}

```
lock int_mon.has_value(0) then end;
```

This additional predicate has not been considered worth the additional (mainly conceptual) MONITOR class complexity.

5.5 Alternatives for forking

The current design for controlling concurrent threads has the goals of supporting at least four major paradigms (Section 3.3):

- Simple "futures"
- Searching for the "first" result
- Modeling co-begin/co-end
- Many threads producing many results

Actually, if a less ambitious objective is taken, it is possible to reach alternative and simpler designs.
The simplest alternative, for example, might be aimed to support only the "simple future" paradigm, raising a run-time error if an attempt is made to fork a thread on an already active

monitor. In this case we no longer need a queue of values inside the monitor, and its overall functionality is quite simplified. In this case, if we really want to fork several activities in parallel, we have to use a different monitor for each of them. New classes should probably be defined for supporting more complex operations on more than one thread (such as waiting until one of them, or all of them, are terminated). This alternative is attractive for its simplicity, but unfortunately the development of the needed library classes risks to be quite difficult. Moreover, in order to get the task done with a reasonable efficiency, we have probably to make them predefined built-in classes, thus reintroducing the complexity in another part of the language.

Another alternative, which for some time was a good candidate for the final design, is to allow multiple forking on the same monitor, but without queuing all the returned results. In this case, whenever a thread completes it would perform a "set" operation possibly overwriting a previous result. In this case we are able to support three of the four desired programming paradigms, losing the case in which all the results are of interest. Moreover the possibility of a continuously changing value of the monitor would introduce a certain degree of non-determinism in the program execution, which is not at all pleasant.

6 Implementation

We have prototype implementations of pSather on a Sequent running DYNIX(R) V3.0.12, and on Sun Sparcstations running SunOS 4.1.1. The pSather compiler is derived from the Sather compiler without the debugger support[42]. The runtime support has been extended from that of Sather.

6.1 Compilation

The extension to Sather compiler involves the expansion of the statement and expression class hierarchy ([42]) to handle the new syntactic constructs. These classes are used in the abstract syntax trees to represent the new constructs:

DEFER_ASSIGN_STMTOB_S. Represents deferred assignment statement.

LOCK_STMTOB_S. Represents lock-statement.

TRY_STMTOB_S. Represents try-statement.

UNLOCK_STMTOB_S. Represents unlock-statement.

LOCK_EXPROB_S. Represents expression found in the locking-statements.

The extension of the compiler has followed the design principles:

- We wanted the same compiler for both the Sparc and Sequent implementations. Thus, the pSather compiler has to generate C code which is acceptable by both machines and gets compiled correctly. The main differences between the versions are due to the following:
  - There is a distinction between shared and private data on the Sequent.
  - Shared and constant features in classes are implemented as global variables.

  As a result, a minor change is needed so that shared variables are declared correctly for shared and constant features. Using C macros, the generated code:

  ```
  extern /*shared*/ SHARED_DATA_ ptr SYS13_class_table_;
  ```

  is acceptable on both machines. Our compiler is currently portable for both machines.

- We avoid any major modification of existing compiler classes. To handle new language constructs, it is unavoidable to construct new classes (eg `LOCK_STMTOB_S`). However, we have created a new class for monitor expression (`LOCK_EXPROB_S`), even though it seems unnecessary at first. This allows us to avoid modifying the other expression classes to check for the special case when the expression occurs in a locking-statement. In this way, the object-oriented design of the compiler has allowed us to handle extensions without major modification to the existing compiler classes and overall compiler structure.

- We generate specialized code where possible. This allows C runtime routines to be written in an efficient manner for the common cases. For example, a locking statement:

  ```
  lock <expr1>, <expr2>....<exprn> then
     <statement list>
  end;
  ```

  is translated into:

```
temp1 <- <expr1>
temp2 <- <expr2>
...
tempn <- <exprn>
grab_locks(temp1, temp2, ..., tempn);
<statement list>
release_locks(temp1, temp2, ..., tempn);
```

Although a general routine can handle any number of monitors to be locked, the compiler generates calls to different runtime routines when the number of monitors is 1 and when the number of monitors is more than 1. (With more experience at programming in pSather, we might decide that it would be worthwhile to distinguish among 1, 2 or more than 2 monitors.)

- As both the language design and runtime support has evolved continuously, we avoid building fixed constants into the compiler. Examples are the names of predefined monitor operations and predicates, and the layout of monitor objects (as described in Sections 6.1.1 and 6.2.2).
- We try to retain Sather's concept of optional runtime error checking. For example, in a deferred assignment:

  ```
  m :- f(...)
  ```

  it is a runtime error if 'm' is void. Since this is similar to the case in which we try to access an attribute of a void object, the strategy to handle this runtime error is similar to that described in [42]. Runtime checking in pSather is further discussed in Section 6.2.1.

Next, we describe the extensions in the compilation process to handle:

- Predefined `MONITOR0` and `MONITOR{T}` classes
- Deferred assignment
- Locking- and unlock-statements

6.1.1 Monitor Classes

`MONITOR0` and `MONITOR{T}` are predefined classes in pSather. From previous sections, we see that a monitor object needs extra slots to keep track of runtime queues, locking status etc. A simple solution would have been to define private attributes in the definitions of `MONITOR0` and `MONITOR{T}` classes. This, however, would restrict a descendent of a monitor class from using any names used for the private attributes. Two implementations were considered.

- One alternative is to tell the `pSather` compiler about extra allocated space for monitor classes. The information would be in the form:

  ```
  (<class name>, <extra number of bytes>)
  ```

  However, this method will not work well because `MONITOR{T}` is a parametrized class and one of the extra attributes (containing the value in a monitor) has type `T`. Hence the extra number of bytes needed for a `MONITOR` class differs from one instantiation to another.

- The current pSather compiler dynamically decides the extra amount of space needed for a monitor. On certain command switches, the compiler will handle differently the class definitions it reads in. Normally, class definitions are installed in a global table, and instantiated and expanded later. In this special case, the routines are marked non-redefinable, and the attributes are marked invisible.

```
Read a class definition.
If <Class is not previously defined>
   Install class definition.
Elsif <Compiler is handling predefined class+features>
   Add the new features to previous definition:
    -- Routines are marked non-redefinable.
    -- Attributes are marked invisible.
    (There is no use for shared/constant features.)
Endif
```

This implementation has several advantages:

Variable Preallocated Space The total sizes required by the invisible attributes are computed and added to the base size of the object. The extra space for a monitor object can be easily varied by adding or removing attribute definitions in a file containing Sather class definition. This allows us to easily experiment with different implementations of the monitor object, without installing a new version of the compiler.

Avoid Name-Space Cluttering These invisible attributes are not considered in other phases of computation, and a user may reuse any name of the invisible attributes without conflict. That is, even though we may use the name 'active_queue_size' to specify a slot for storing the size of active queue associated with a monitor, the user can write the following:

```
class M_DESCENDENT is
     MONITOR0;
     active_queue_size:INT;
end;
```

and still access the user-defined attribute correctly.

Efficient Runtime Access By maintaining the following properties:

- The ordering of the invisible attributes must be preserved according to what is given in the class definition.
- The space for these invisible attributes is always allocated before the user-defined attributes.

The location of, say 'active_queue_size', in any monitor object is fixed and the runtime code can access these fields in the monitor directly without consulting any runtime table. (We will describe the layout of monitor objects in Section 6.2.)

We have seen how the compiler handles the allocation of space for monitor objects. Next we describe how the compiler checks for monitor operations and predicates. We have the following requirements:

- We want the predefined monitor operations and predicates to have a fixed semantics, so that a user cannot redefine routine such as `'is_unbound'` in a monitor class.
- On the other hand, disallowing a user to define a feature named `'is_unbound'` in a non-monitor class is too restrictive. A user should be able to reuse the same identifier in a non-monitor class to denote any feature.
- Any call to a predefined monitor operation, eg `m.take` should call a runtime C routine directly. This eliminates the extra costs of calls via a Sather interface.

The current implementation meets these requirements by treating monitor operations and predicates like any other class features. However, as described in the treatment of `MONITOR` class, they are marked as non-redefinable, so that the compiler is able to check that no descendent class of `MONITOR0` or `MONITOR{T}` redefines routines such as `'is_unbound'`. Since these features are predefined only with respect to monitor classes, a user can reuse such identifiers as `'is_unbound'` in other classes.

In order to call the runtime C routine directly, we adopt the following name convention for non-redefinable Sather routines. During C code generation, a predefined operation say 'is_unbound' will be associated the C name formed by:

```
<Class-definition-name>_<Routine-name>_
```

Hence the code "m.is_unbound" where 'm' is of type `MONITOR0` will be translated as "MONITOR0_is_unbound_(m)". This C name convention departs from that followed by user-defined routines which is described in [42]. In the case of operations such as 'take' which involve type parameters, the C name is formed by:

```
<Class-definition-name>_<Routine-name>_<C-type-of-type-parameter>_
```

so that "m.take" where 'm' is of type `MONITOR{INT}` is translated into "MONITOR_take_int_(m)".

The current compilation strategy offers the following advantages:

Flexibility for Experimentation It is easy to re-name the predefined monitor operations and predicates. We can do this by simply editing a Sather class definition (and altering the names of C runtime routines). Any changes in semantics involve only changes in the runtime and not the compiler.

Reduced Code Size The executable code size is reduced. If the predefined operations and predicates were written in pSather, the same routine would be generated for each monitor class, so that for:

```
class M1 is                    class M2 is
   MONITOR0;                      MONITOR0;
   ...m1:SELF_TYPE...             ...m2:SELF_TYPE...
   ...m1.is_unbound...            ...m2.is_unbound...
end;                           end;
```

each C file generated for classes 'M1' and 'M2' would get its own C routine 'is_unbound'. In the current strategy, all monitor classes share the same C routines (except for those operations such as 'take' whose C implementation differ between `MONITOR{INT}` and `MONITOR{DOUBLE}`).

Efficiency Since predefined operations are direct C routine calls, we avoid any extra costs of procedure calls that would be incurred if these operations or predicates are implemented in pSather itself. Also, using macro-expansion and other C programming techniques, we can improve the efficiency of runtime routines more easily.

6.1.2 Deferred Assignment

This is the basic mechanism in pSather to create new threads of execution. In general, it is treated almost like the ordinary assignment statement in most of the compiler phases. The major difference is the semantic check that is required. We give an outline of the semantic check in Figure 1.

Note that unlike an ordinary assignment statement, we do not need to check that the LHS expression evaluates to an assignable location. In the generated C code, the thread creation routine gets the following parameters:

- Pointer to monitor object

```
Perform the semantic check on LHS expression.
Perform the semantic check on RHS expression.
Check that the RHS is a valid function call.
Check that LHS expression yields a monitor object.
If (LHS has MONITOR0 type) then
   RHS expression may or may not return any value.
elsif (LHS has MONITOR{X}) then
   Type of RHS expression must conform to X.
else
   Error.
EndIf
```

Figure 1: Semantic check for deferred assignment.

- C type of result value
- Sather type index (if runtime type-checking is required) to be checked according to Sather conformance rules
- Number of arguments
- Function called
- List of C types of arguments
- List of argument values

We observe that this thread creation routine is like most other thread creation routines in general threads packages, except that we need to customize it to keep track of the monitor associated with the thread, handle return result (of various C types) from the forked routine, and keep information to do runtime type-checking if necessary.

6.1.3 Locking- and Unlock-Statements

As mentioned in the beginning of this section, these new statements are implemented with the introduction of new compiler classes. These compiler objects are handled like most others in the earlier phases of the compiler. Each construct has additional properties:

Lock expression These are the expressions evaluated in the lock- and try-statements. Each lock expression must check whether it is of the form:

```
<monitor expression>
```

or

```
<monitor expression>.<monitor predicate>
```

or

```
<monitor predicate>
```

The last form is a simplification of `self.<predicate>` which may occur in the following situation.

```
class SPECIAL_MONITOR is
    MONITOR0;

    new_operation is
       ...
       lock is_bound then ... end;
       ...
    end;
end;
```

During C code generation, extra temporary variables must be pre-allocated to hold pointers to the evaluated monitor objects. This ensures that each monitor expression is evaluated exactly once, and avoids any re-evaluation if locking does not succeed on the first try.

Locking statement This description covers both the lock- and try-statements. These two constructs are quite similar except that the lock-statement may block the executing thread, but not the try-statement.

We have to take special care of return- and break-statements which occur within any locking statement.

- **Return statement**. Before exiting from a routine via the return-statement, all the locks acquired in the enclosing lock-statements (or then-branch of try-statements) have to be released.
- **Break statement**. A break-statement transfers control to the end of the enclosing lock- or try-statement. Hence all the locks acquired in this enclosing locking-statement must be released before control is transferred to the end of the locking-statement. (Note that if a break-statement occurs in the else-branch of the try-statement, there is no need to release any lock.)

To implement these constructs, a particular occurrence of break- or return-statement keeps a list of the monitor expressions which are to be released before control is transferred. To keep track of these expressions, we simply implement a stack which keeps track of the current lock-expressions seen so far, as we traverse down the abstract syntax tree. During C code generation, the lock-statement ensures that all monitors which are locked will be released at the end without re-evaluating the lock-expressions.

Unlock statement One of the requirements of an unlock-statement is that it occurs lexically within a lock-statement (or then-branch of try-statement). This check is done just like the requirement for break-statement to occur lexically within the loop-statement. However, there is an added complexity:

- The monitor object evaluated in the unlock-statement must have been locked in an enclosing locking-statement in the current routine. That is, if we have:

```
lock <expr1>, <expr2>....<expr-n> then
   ...
   unlock <expr>;
   ...
end;
```

The monitor object evaluated from `<expr>` must be equal to one of those evaluated by `<expr1>`, `<expr2>`,...`<expr-n>`.

This is a runtime requirement and cannot in general be checked by the compiler. If we try to implement this by a stack of monitors locked by the thread so far, this stack has to be delimited by the routine call boundaries. Otherwise, we might have:

```
-- Assume 'm' and 'n' refer to distinct monitor objects.
f is
   lock m then
      ...g;...
   end;
end;

g is
   lock n then
      ...
      unlock m;
      ...
   end;
end;
```

The monitor object 'm' is unsuspectingly unlocked by 'g' and this violates our requirement.

We handle this using a combination of code-generation and runtime strategies. The key is that monitors are objects and pointers to these monitor objects are kept in temporary variables. It is simplest to illustrate this by the top-level description of the C code generated. Suppose we have the code in Figure 2.

From Figure 2, the monitor specified in the unlock-statement is pre-evaluated and the pointer to the monitor object stored in a temporary variable. This is then compared with the other pointers to monitor objects that have been locked so far. (Note that in our implementation, the pointers to monitor objects in the closest enclosing locking-statement are compared first, and as will be illustrated later, the ordering is relevant.) If an equality is found, the corresponding temporary variable is set to 0. This prevents the same monitor from being unlocked again in another unlock-statement. When the final 'release_locks' routine is called, it ignores any void pointer. (If the original lock expression evaluates to void, we would have encountered an error in 'grab_locks' routine. Hence a void pointer in 'release_locks' is definitely not an error.)

This implementation ensures that the locked status of any monitor at the end of a routine call is the same as at the beginning, or viewed in another way, every lock operation is matched by a corresponding unlock operation in the same routine. An unlock-statement will not cause a mismatch between the number of lock and unlock operations performed on the monitor.

A simple observation shows why the previous claim is true. Each temporary variable is used exactly once in a locking statement. Once the monitor object is unlocked, the temporary variable no longer contains a reference to the monitor object. Hence each temporary variable ensures that the monitor object is locked and unlocked exactly once. Theoretically, at the point (*) (in Figure 2b) we should set the temporary variables to void. But since no further references to these temporary variables are possible (this is enforced by the compiler), we can avoid nullifying the temporary variables in this case.

Another point to note is that the order of comparison for the temporary variables in the enclosing locking-statements is important. Consider the following code:

```
lock m then      -- 'T1' is the temporary variable here.
   ...
   lock m then   -- 'T2' is the temporary variable here.
```

```
lock <expr1>, <expr2>....<exprn> then
   ...
   try <expr(n+1)> then
      <statement list 1>
      unlock <expr'1>;
      <statement list 2>
   end;
   ...
end;
```

a. pSather code.

```
temp1 <- <expr1>
temp2 <- <expr2>
...
tempn <- <exprn>
grab_locks(temp1, temp2, ..., tempn);
   ...
   temp(n+1) <- <expr(n+1)>
   if (grab_locks(temp(n+1)) == SUCCESS) then
      <statement list 1>
      temp'1 <- <expr'1>
      if (temp'1 == temp(n+1)) then temp(n+1) <- 0;
      elsif (temp'1 == temp1) then temp1 <- 0;
      ...
      elsif (temp'1 == tempn) then tempn <- 0;
      else ERROR
      release_locks(temp'1)
      <statement list 2>
      release_locks(temp(n+1))
   end;
   ...
release_locks(temp1, temp2, ..., tempn);
               <---- (*)
```

b. Corresponding generated C code.

Figure 2: (a) and (b) illustrates code generation for unlock-statement.

```
        ...
                <--- (1)
        unlock m;
        ...     <--- (2)
    end;        <--- (2.a)
    ...         <--- (3)
end;
```

The semantics that we want is that at (1), the monitor object is locked at level 2, at (2), it is locked at level 1, while at (3), it remains locked at level 1 because the unlock-statement only undoes the closest enclosing locking-operation.

In this example, there are two temporary variables (say 'T1' and 'T2') containing references to the same monitor object at (*). Whether 'T1' or 'T2' is nullified at the unlock-statement, at execution point (2), the lock-level of the monitor object has been reduced to 1. The difference is that if 'T2' has been nullified, the monitor object will not be unlocked again at the end of the inner lock-statement. Hence, the monitor object remains locked at level 1 at (3). This will not be the case if 'T1' has been nullified first. (In any case, ignoring any possible parallel access, the lock status of the monitor object at the end of all locking statements is restored to its state at the start of that statement).

Intuitively, the order of comparison enforces the semantics that the unlock-statement forces an early unlocking of the monitor locked in the closest enclosing locking-statement.

6.2 Runtime Support

One over-riding concern in the runtime implementation is efficiency. We try to achieve this by several strategies:

- Where possible, we provide specialized routines that can be used in the compiler-generated C code.
- As described in the design principles of the compiler, we eliminate unnecessary checks for runtime errors. This also allows us to avoid passing unnecessary parameters. For example, if we decide to check for void monitor object in the 'grab_locks' routine, it will be useful if the message can indicate exactly where (file name and line number) the error occurs. In such a case, extra code has to be executed to pass the file name and line number via parameters or assign them to global variables to be read. This will definitely slow down the program execution.
- Often we trade memory for time. An example is the additional fields in thread object to hold pointers when the thread is in a waiting queue of the monitor object.
- The runtime support routines for monitors are carefully written to release the monitor internal lock as early as possible. The layout of monitor objects has been carefully considered for efficient execution, though further experimentation is definitely possible to look for more improvements. Later in the description, we will point out further possible improvements where relevant.
- We have retained the thread management schemes in FastThreads package which have been carefully designed for efficiency. An example is to keep a pool of threads and stacks so that a completed thread and its stack are returned to the common pool.

  However, sometimes we have to alter certain implementation aspects to get the desired language semantics. For example, to ensure a FIFO semantics for the execution of forked threads, the queue discipline in FastThreads was modified from LIFO to FIFO.

```
Initialize global data structures, eg ready queues,
  stack pool, thread pool etc.
Create a thread which will execute the Sather program.
Create N processes (real or virtual).
foreach process
   Initialize local attributes of process eg id, local
     queue etc.
   loop
      Look for an executable thread in ready queues.
      if (thread available) then
         Start executing thread
      elsif (all other processors are idle) then
         Quit
      endif
   endloop
endfor
```

Figure 3: Top-level driver code for pSather programs.

Another concern is the portability of the runtime. As a result, when the runtime was ported from Sequent to Sparc, we modified the existing Sequent code instead of rewriting the runtime support from scratch. From the experience so far, we have identified certain components that, if replaced, should allow the runtime to be ported across various architectures relatively easily.[2]

Locking Mechanism. The basic locking primitives vary in different architectures. An example is the 'exchange-byte' operation on the Sequent. Otherwise, the interface to the spinlock package is machine independent.

Threads Package. Stack manipulation and register handling vary on different machines. The most relevant routine is the 'startup_thread(t)' routine which starts the execution of a new thread. Otherwise, the structure of thread objects has remain largely unchanged.

Real/Virtual Process Package. On shared memory (uniform or NUMA) multiprocessors (eg Sequent, Butterfly), these are provided as C library routines. On the single processor machine (eg Sparc), we have to implement a virtual process package that allows some data to be shared among different virtual processes and other data to be private for each virtual processor. Since this depends on threads package, the virtual process package should be machine-independent for single-processor machines.

The runtime support for pSather has evolved from incorporating the FastThreads[9] into the original runtime support for Sather. Figure 3 shows the top-level design of the driver program that creates multiple processes, and schedules the threads.

The initial implementation of the runtime was done on the Sequent. When the runtime was ported to the Sparc, we tried to modify the runtime package as little as possible. The major hurdles to portability are:

- The thread suspension and resumption routines are machine-dependent.

[2] The efficiency would of course vary widely. The current design does not take into account the efficiency issues on NUMA machines.

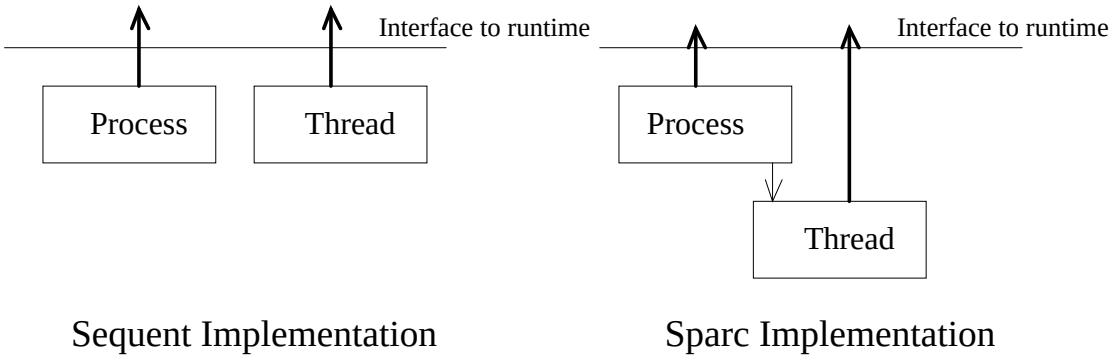

Figure 4: Process and thread packages on Sequent and Sparc.

- Since the system 'fork' routine on the Sparc does not support shared data across address spaces, we have to implement a virtual processor package. A threads package is already needed to support thread management for pSather programs. We re-used that package to simulate the virtual processors (Figure 4).
- In the Sequent implementation, a program can specify whether a declaration is per-processor or shared by all processors. On the Sparc, together with the virtual processors package, we have implemented additional code to swap private data when switching virtual processors.

Otherwise, the routines that handle spinlocks, monitors, thread pools, stack pools etc and top-level structure (Figure 3) have remain largely unchanged.

In the next sub-sections, we describe the runtime support to handle:

- Predefined `MONITOR0` and `MONITOR{T}` classes (including deferred assignment)
- Locking

6.2.1 Runtime Checks

As mentioned in Section 6.2, both in the Sather generated C code and runtime routines, we avoid extra checks for errors to improve efficiency. However, we still want to be to detect runtime errors, such as during debugging. Following the design of the Sather compiler and runtime support, a compiler option can be specified if runtime checks are desired. These runtime checks include those found in the Sather compiler:

- Accessing feature of a void object.
- Accessing an array beyond its bound.
- Non-conformance of object type.

In addition, in the pSather compiler, we can detect the following:

Void monitor object The deferred assignment, lock- and try-statements all require the monitor objects to be non-void. It is a runtime error if, for example, we have:

```
m:MONITOR0;       -- 'm' is void.
m :- foo;         -- Runtime error.
```

To avoid unnecessary overhead, we generate extra code to check for these kinds of error only if the compiler option is on.

Type conformance Another type conformance problem may arise in the deferred assignment statement. Suppose we have:

```
class A is ... end;
class B is A; end;
class MAIN is
    foo:$A is res := A::new end;
    main is
       a:MONITOR{B} := MONITOR{B}::new;
       a :- foo;          -- 'A' does not conform to 'B'
    end;
end;
```

If the monitor object is of type MONITOR{T}, we have to make sure that the type of the returned value conforms to that required by the monitor object. This non-conformance is detected if runtime check is specified to the compiler.

Stack overflow This problem does not exist for sequential Sather where there is only one stack, and as long as the recursion level does not exceed the system stack size, there is no problem. However, in the pSather runtime support, stacks for different threads have to be explicitly allocated. And in most cases, we cannot know a priori the stack size required. In particular, a pSather program's stack requirements may vary widely from few, large stacks to many, small stacks. If the stack size is fixed, a perfectly reasonable program can get an error simply for lack of stack space. To remedy this situation, the following strategy is adopted.

If the runtime check option is specified to the compiler, the stack space will be protected, so that a signal handler is invoked when the stack overflows. If the default stack size is insufficient, the compiler provides an option for the user to specify the required stack size (either in number of pages or bytes). This involves a slight overhead, because the stack size is no longer a built-in constant in the runtime. However, we feel that this is justified because we cannot expect the user to re-compile the runtime support for different programs.

6.2.2 Monitor classes

Each `MONITOR` object has to keep track of the following:

Bound status. This indicates if the monitor is bound or not.

Internal lock. This internal lock keeps a monitor's internal state consistent before and after a predefined monitor operation.

Lock level. It is not sufficient to keep a toggle status about the lock status of a monitor, because a thread may lock a monitor more than once.

Locking thread. If the monitor is currently being locked, this is a pointer to the thread which currently locks the monitor.

Active thread queue. The initial implementation explicitly kept a list of the active threads associated with the monitor. When a thread completes, it has to be deleted from this list. In order to keep deletion time constant, a doubly linked list was used. However, when it became clear that this list of active threads is only used in the 'clear' operation, we decided on an alternative implementation.

In the current implementation, we keep a 'clear_level' field in the monitor which is incremented each time the 'clear' operation is called. A thread knows that it has been dissociated from the monitor if the thread's 'clear_level' field is less than that in the monitor. In this case, a 'active_queue_size' counter has to be kept in the monitor in order for the monitor predicates 'has_thread' and 'no_threads' to work correctly. In this implementation, when a thread completes, the time taken to execute the house-keeping routine (which may set the monitor, etc) is independent of the number of threads. Furthermore, the clear operation now takes constant time, since we only need to increment 'clear_level' and set the 'active_queue_size' to 0. (In the doubly linked list implementation, the 'clear' operation will take time proportional to the number of active threads.)

Waiting queue. This contains all the threads that have been suspended because the monitor is not in the correct state for the action to be executed. For example, if we have:

```
lock m.no_thread then ... end;
```

the monitor 'm' is suspended if active threads are still associated with the monitor.

Currently each monitor object has exactly one waiting queue associated with it. When the monitor status changes (eg when it becomes bound), the runtime code needs to check the waiting threads to see if any can be put back on the ready queue (of active threads). If there is a large number of waiting threads, this may lead to significant search time for finding a thread to be resumed. An alternative implementation is to have multiple queues, each having threads waiting on a different condition.

In addition, a `MONITOR{T}` object has 'value' and 'value_queue' fields associated with it. A `MONITOR0` object has a counter to count the number of times it has been bound. For example if we have:

```
m:MONITOR0 := MONITOR0::new;
...
m :- f1;
m :- f2;
...
```

when both 'f1' and 'f2' complete, the 'take' operation can be performed twice on 'm' without blocking.

A common characteristic of the monitor operations is that when a monitor operation cannot be performed because the monitor state does not satisfy certain conditions, the current thread is suspended and put on the waiting queue of the monitor. For example, a thread 't1' is suspended if it tries to perform a 'take' on an unbound monitor 'm'. At a later point in time, when another monitor operation (eg 'set') is performed on 'm' which alters the monitor state, the waiting queue is searched to see if any thread can be resumed. In our example, 't1' would be put back on the ready queue, and the monitor's internal lock would be held by 't1'. This prevents another thread 't2' from performing a 'take' (or any monitor operation) on 'm' in the time interval between 't1' being put back on the active queue, and 't1' resuming execution.

In this way we enforce the FIFO semantics for threads performing operations on the same monitor. (In this case, 't1' is guaranteed to get the value before 't2'.)

One of the operations on the monitor object is the deferred assignment. All deferred assignments are translated into C routines which get a thread and insert it into the active queue.

6.2.3 Locking

The implementation of the lock-statement involves trying to trade off among the following criteria:

```
[Thread 1]                  [Thread 2]

lock m1 then                lock m2 then
   ...                         ...
   lock m2 then                lock m1 then
      ...                         ...
   end;                        end;
   ...                         ...
end;                        end;
```

a. Nested locking-statements.

```
[Thread 1]                  [Thread 2]

lock m1 then                lock m2 then
   ...                         ...
   f1;                         f2;
   ...                         ...
end;                        end;

f1 is                       f2 is
   lock m2 then                lock m1 then
      ...                         ...
   end;                        end;
end;                        end;
```

b. Nested locking-statements.

```
[Thread 1]                  [Thread 2]

lock m1.is_bound then       lock m2.is_bound then
   ...                         ...
   m2.set;                     m1.set;
   ...                         ...
end;                        end;
```

c. Locking with condition on monitor state.

Figure 5: (a), (b) and (c) illustrate possible reasons for deadlock.

```
Sort m1,... m<n>
for i = 1,..,n
   if (lock(m<i>) != SUCCESS)
      Suspend current thread on m<i>
   endif
endfor
```

Figure 6: An implementation of lock-statement.

- Deadlock-free.
- Maximal concurrency.
- Efficiency.
- Fairness.

We shall discuss each of the four points next.

Deadlock-free. In pSather, there is no way to prevent the user from writing code which will result in deadlock. Consider the examples in Figure 5.

In Figure 5(a), a deadlock may occur because pSather does not restrict nested locking-statements. Even restricting nested locking-statements statically is not sufficient, as illustrated by Figure 5(b). In a sense, (b) is an example of dynamic nested locking-statements. Furthermore, as illustrated in Figure 5(c), the ability to specify locking a monitor based on its state may also result in deadlock even if there is no nested (static or dynamic) locking-statements.

From such considerations, the runtime currently does not attempt to resolve deadlocks. However, we do envision the following possibilities with respect to the deadlock problem:

Compilation warning. The compiler may be able to analyze the pSather program and give warning of possible deadlocks.

Execution time deadlock detection. A program may be compiled with a given compiler option to activate additional runtime routines that will detect deadlock during program execution.

Despite the possibility of deadlock due to user programming, we do require that the runtime support must not cause any deadlock. If two separate threads are each executing the code:

```
lock m1, m2, m3 then ... end; -- Thread 1

lock m3, m2, m1 then ... end; -- Thread 2
```

the runtime support must not result in any deadlock. One way of implementing the semantics is to simply release the monitors locked so far when not all the monitors can be locked, before the current thread is suspended on the unavailable monitor. However, this may result in two threads both releasing all the monitors and getting suspended, when in principle, exactly one of them should proceed. One solution is to sort the list of monitors and acquire them according to the sorted order (Figure 6). This, however, does not give the desired semantics for the locking statement, and is related to the next point: maximal concurrency.

```
   Sort m1,... m<n>
try_again:
   for i = 1,..,n
      if (lock(m<i>) != SUCCESS)
         for j = 1,...,i-1
            Release m<j>
         endfor
         Suspend current thread on m<i>
         Goto try_again
         -- Try again when the thread is woken up.
      endif
   endfor
```

Figure 7: One possible implementation of lock-statement.

```
   Sort m1,... m<n>
try_again:
   for i = 1,..,n
      if (lock(m<i>) != SUCCESS)
         Suspend current thread on m<i>
         -- Check thread status when it is woken up.
         if (thread notified of preemption) then
            Goto try_again
         else
            Continue
            -- Continue with trying to lock the other
            -- monitors.
         end;
      endif
   endfor
```

Figure 8: Another implementation of lock-statement.

Maximal concurrency. Next we describe a reason for elaborating the previous code (Figure 6) to the form shown in Figure 7.

Suppose we have:

```
lock m1 then ... end; -- Thread 1

lock m2 then ... end; -- Thread 2

lock m1, m2 then ... end; -- Thread 3
```

respectively and the ordering is 'm1 < m2', and thread 2 has acquired 'm2'. When thread 3 tries to lock both 'm1' and 'm2', it cannot succeed. Next thread 1 starts to execute. According to the semantics of the pSather lock-statement, since thread 3 did not succeed, 'm1' should not have been locked and thread 1 will be able lock 'm1' and proceed. If we simply acquire the locks in sorted order, thread 3 would have locked 'm1' and be blocked on trying to lock 'm2'. This prevents thread 1 from locking 'm1', and violates our "maximal concurrency" requirement.

Figure 7 works correctly, except for its apparent inefficiency which we will discuss next.

Efficiency. The locking operations must be done efficiently – on the first try if the thread can successfully grab all the monitors. Even if a thread cannot successfully grab all the monitors, the releasing of all previously locked monitors as shown in Figure 7 may not be needed. If we have:

```
lock m3 then ... end; -- Thread 1

lock m1, m2, m3 then ... end; -- Thread 2
```

Suppose thread 1 locks 'm3' successfully. When thread 2 tries to lock the monitors (assumed in the correct order), it will lock 'm1' and 'm2', finds that 'm3' is locked and releases the two monitors. This release is not absolutely necessary, because in the time interval between thread 2 getting suspended and thread 2 being resumed (when thread 1 unlocks 'm3'), no other thread is trying to acquire 'm1' or 'm2'. It would be more efficient to leave both 'm1' and 'm2' locked by thread 2.

Therefore the code is further elaborated as shown in Figure 8.

In the locking code, when a thread (T1) finds that a monitor is locked by another thread (T2) which has been suspended in a locking statement, it alters the status of the thread (T2), and surreptiously "steals" the monitor. In effect, T1 forces T2 to release the monitor (and all others that T2 is currently locking). As a result, when T2 is finally woken up, it has to retry locking all the monitors it needs.

On the other hand, if no other thread needs any monitor locked by the currently suspend thread (T2), none of T2's monitors is released. When T2 is woken up, it can continue to try to grab the other monitors. Figure 8 only shows the code for the thread whose locks might be stolen. Figure 9 describes the 'lock-stealing' mechanism in more detail. Here we give an argument outline to show that it achieves the desired semantics.

First, we give proofs of some properties of the mechanism.

Claim 1 *If a thread t1 steals a lock m from locking_thread successfully, (in the semantics of lock-statement) m should have been released by locking_thread before locking_thread suspends.*

```
-- Monitor to be locked is denoted by `m'.
if (m is locked and m is not locked by current thread) then
   -- `locking_thread' is the thread that currently
   -- locks `m'.  Try to steal `m' from `locking_thread'.
   steal_lock <- 0
   -- Whether `m' is actually stolen depends on whether the
   -- value of `steal_lock' is greater than 0.
  if (locking_thread is suspended) then
      -- `m1' is the monitor on which the locking thread
      -- is suspended.
      Grab internal lock of m1.
      if (locking_thread is still suspended on m1) (*1*) and
         (locking_thread is waiting for a lock) (*2*)then

         -- The check at (*1*) is necessary because in the
         -- time interval between finding out about `m1',
         -- and getting `m1''s internal lock,
         -- `locking_thread' may have been resumed.
         -- At this point, we know that `locking_thread' is
         -- suspended at a lock-statement; hence, current thread
         -- can proceed to steal `m'.
         for each ``stealable'' monitor m<i> locked by
         `locking_thread' in the current lock-statement
            if m<i> = m then
               steal_lock <- 1
               Notify locking_thread of pre-emption.
               Mark m<i> as non-stealable.
               Add 1 to counter.
               -- `counter' keeps track of number of times
               -- `m' has been locked in current lock-statement.
            endif
         endfor
         if (counter > 0) then
            if (number of times m has been locked != counter) then
               steal_lock <- 0
               Restore locking_thread pre-emption status.
               Re-mark m<i> as ``stealable'' for each m<i> = m.
            else
               Undo all other locks currently held by locking_thread,
                 and mark them non-stealable.
            endif
         endif
      endif
      Release internal lock of m1.
   endif
endif
-- Current thread either suspends or locks `m'.
```

Figure 9: Lock pre-emption mechanism.

Proof: Suppose $t1$ succeeds in stealing the lock m. From the implementation, we see that m must be one of the locks held by *locking_thread* in the current lock-statement. In addition, *locking_thread* is suspended at a lock-statement by the checks at (*1*) and (*2*) (in Figure 9). Hence m should have been released by *locking_thread* by the semantics of the lock-statement. □

Claim 2 *If a thread $t1$ steals a lock m from locking_thread successfully, m has been locked by locking_thread only in the current lock-statement (on which locking_thread is suspended).*

Proof: Suppose $t1$ succeeds in stealing the lock m. From Figure 9, we must have *counter* equal to the number of times m has been locked. But *counter* keeps track of the number of times m has been locked in current lock-statement. Hence, m has been locked by *locking_thread* only in the current lock-statement. □

Claim 3 *Once a thread $t1$ succeeds in stealing a lock m from locking_thread, all other locks held by locking_thread in the current suspended lock-statement are released.*

Proof: This is obvious from the code in Figure 9. We also note that if a monitor $m1 \neq m$ is released, and becomes unlocked as a result, then another thread wanting $m1$ can lock $m1$ without executing the piece of code in Figure 9. If $m1$ remains locked, then because it would have been marked as non-stealable, another thread will not be able to steal it.

Claim 4 *If two threads $t1$ and $t2$ try to steal the same lock m from locking_thread in parallel, exactly one of the threads will succeed, or none of them will succeed.*

Proof: Suppose both $t1$ and $t2$ succeeds in stealing the lock m. Then both $t1$ and $t2$ must have found $m < i1 >$, $m < i2 >$ respectively, where $i1 \neq i2$ and $m < i1 >= m < i2 >= m$. Without loss of generality, suppose $t1$ grabs the internal lock of $m1$ before $t2$. Since $m < i1 >= m < i2 >= m$, both $m < i1 >$ and $m < i2 >$ would have been marked non-stealable by $t1$, and it is not possible for $t2$ to steal m again. □

It does not really matter if neither $t1$ nor $t2$ succeeds in locking m. If m has been locked in a successful lock-statement, then it is obvious that both $t1$ and $t2$ must fail to steal lock m. Otherwise, since the speed of execution is unknown, we can always assume that $t1$ and $t2$ are executing at such a speed that they both fail to steal m because m has not been released.

Suppose we have the following situation:

```
lock m1, m2 then ... end; -- Thread 1

lock m1 then ... end; -- Threads 2, 3
```

Thread 1 could have just locked 'm1'. Just before it gets suspended on 'm2', threads 2 and 3 try to lock 'm1'. According to the lock-statement semantics, 'm1' is not yet released by thread 1 and both threads 2 and 3 will fail to lock 'm1'.

From claims 1 and 2, we guarantee that a thread can only steal a lock m if m was not locked in a successful lock-statement, and that m should have been released by *locking_thread* which is suspended on the lock-statement. Furthermore, claim 4 guarantees that exactly one or no thread will succeed in stealing m. Claim 3 ensures that a successful steal attempt will make *locking_thread* release all its locks in the current lock-statement (which is exactly what it should have done before it is suspended).

Intuitively, it seems that not all locks held by *locking_thread* in the current lock-statement need to be released. However, this would give rise to the following inconsistent situation:

```
lock m1, m2, m3 then ... end; -- Thread 1

lock m1 then                  -- Thread 2
   try m2 then ...
   else
      -- Error, since 'm2' should not be locked
      -- when thread 1 is suspended, and thread 2
      -- can succeed in grabbing 'm1'.
   end;
   ...
end;
```

Suppose thread 1 locks 'm1' and 'm2' and is suspended on 'm3'. Thread 2 then steals the lock 'm1'. If 'm2' is not released, then we have an inconsistent program state, because the lock-statement is supposed to either grab all the locks or none.

Fairness. The following code illustrates that fairness and maximal concurrency are conflicting objectives.

```
lock m1 then ... end; -- Thread 1

lock m2 then ... end; -- Thread 2

lock m1, m2 then ... end; -- Thread 3
```

If we want maximal concurrency, thread 3 may be prevented from entering its critical section by thread 1 and 2. On the other hand, if we want fairness, then concurrency must necessarily be reduced. We can achieve a strong fairness property if the implementation follows what outlined in Figure 6. That is, any thread trying to lock any set of monitors will eventually succeed, provided that there is progress towards satisfying the locking condition.

The current implementation (Figure 8) does not ensure strong fairness. Intuitively, a thread requiring fewer monitors is more likely to succeed entering the critical section, than a thread requiring more monitors (since the latter are more likely to be 'preempted'). We discuss two properties of the current implementation.

- It does not ensure that locking statements are always executed in a FIFO order. Consider Figure 10.

  Even if the locking statement of 'foo1' has been started before the locking statement of 'foo2', and they become both successfully executable when 'm2' is unlocked, it is 'foo2' which is executed first because 'foo1' has in some sense lost its FIFO position when swapping from the queue of 'm1' to the queue of 'm2'. This could be repeated several times, delaying 'foo1' arbitrarily.

- It allows a 'weak' fairness property to hold. If we have the following:

  ```
  lock m3 then ... end; -- Thread 1

  lock m1, m2, m3 then ... end; -- Threads 2, 3
  ```

  Thread 1 succeeds in grabbing 'm3'. Thread 2 will be blocked on 'm3'. When thread 3 starts executing, thread 2 will be forced to unlock 'm1' and 'm2'. When 'm3' is unlocked, however, thread 2 will be resumed before thread 3; this time, thread 3 will be forced

```
m,m1,m2:MONITOR0;
...
foo1 is   lock m1, m2 then end; end;
foo2 is   lock m1, m2 then end; end;  -- as foo1
...
main is
   lock m2 then
      m :- foo1;
      -- 'foo1' tries to lock first 'm1' and then 'm2', failing
      -- the second time and becoming suspended
      ...
      lock m1 then
         m :- foo2;
         -- 'foo2' tries to lock 'm1' (before 'm2') and fails
         -- immediately, becoming suspended
         ...
         unlock m2;
         -- 'foo1' is resumed, and tries to re-lock 'm1', failing
         -- for the second time and becoming suspended, but now
         -- it is enqueued waiting for 'm1' after 'foo2'.
         ...
         unlock m1;
         -- at this point 'foo2' is resumed and executed, because
         -- 'foo2' is now the first thread of the queue of 'm1',
         -- even if 'm2' is now available, and 'foo1' could have
         -- been executed too.
      end;
   end;
end;
...
```

Figure 10: Non-FIFO locking

to unlock 'm1' and 'm2' to be grabbed by thread 2, which will succeed, provided in the meanwhile, no other thread has grabbed 'm3'.[3]

This weak fairness property does not prevent lock-out, but any thread trying to lock monitors will eventually get a chance to execute again, provided that the locking condition is eventually satisfied.

6.3 Possible Further Improvements

In addition to the compiler and runtime implementation described above, we are beginning to take the first steps to improve pSather's overall environment.

Programmer Aids? In the process of experimenting with the language design, we find what all other parallel programmers also encounter: parallel debugging support is inadequate to effectively track down bugs, especially those related to race-conditions. As a first step to helping the parallel programmer understand the runtime behaviour of his/her program, we incorporated an additional compiling option. When this option is specified, the compiler will generate additional information that is used by the runtime to trace the execution of various threads. At program termination, a trace of the threads is printed. Undoubtedly, this additional tracing mechanism skews the relative speeds of various threads. However, we believe that this has less impact than if the programmer has to explicitly insert print statements.

Optimization? Besides improving the runtime efficiency and generating specialized code, one way to improve the efficiency of a parallel program is for the compiler to analyze the user code further, and generate extra information to be used by the runtime. For example, we note that the deferred assignment statement is almost like a function call with an assignment statement. Furthermore, if it is impossible for the forked thread to be suspended, then, intuitively, the semantics of the program does not change if in fact, a function call is made instead of a thread being forked. This brings in the possibility of having the compiler generate code that avoids forking if the system load is high. Suppose we have the code:

```
m :- f(...)        -- 'f' never suspends.
```

the following code can be generated:

```
if (low system load + other criteria hold) then
   Create a new thread
else
   Pre-process monitor object
   Perform a function call
   Post-process monitor object with function result (if any)
end;
```

In the pre-processing phase, the current thread has to assume a new identity (in terms of thread id, associated monitor etc) so that the program retains the original semantics of concurrent execution. This information is restored in the thread object in the post-processing phase, and any returned value is stored in the monitor.

We have added the capability in the compiler to analyze whether a routine is blockable or not, and instrumented the compiler to generate the above code when requested. This feature, however, has not been fully elaborated since it is not well-understood in several aspects:

[3] Even if some other thread has grabbed 'm3', thread 2 has been given its chance.

- From the point of view of this feature acting as dynamic load-balancing, we understand that dynamic load-balancing does not always improve the program performance. In particular we do not know what is a good 'low system load + other criteria'.
- Another point of view is that this feature is a form of high-level code improvement. There are other possible code improvements which we have not explored.

Characteristics	Time (seconds)
(i) 1-processor, Sequential	0.12
(ii) 1-processor, Parallel, Normal thread creation	0.45
(iii) 1-processor, Parallel, Optional thread creation	0.16

Table 1: Times to quicksort 10,000 integers on a Sparc 2.

An example is replacing monitors by simpler objects as shown by the following example:

```
shared m:MONITOR0 := MONITOR0::new;
....
lock m then
   -- 'm' is not used anywhere else.
end;
```

In this example, since 'm' is used only to protect a critical section, we may perhaps simply use a waiting lock. (In discussing the fairness aspect of the locking statement in Section 6.2.3, we have pointed out that the locking statement does not ensure FIFO semantics.)

- That the transformation preserves the semantics of the program is intuitive. We need further proof and/or analysis to show that this is always the case.

With some initial experimentation, we have a simple scheme in which a thread is forked if the following criteria holds:

```
(number-of-processors > 1) and (idle processor exists)
```

This simple scheme allows a 1-processor execution of a parallel quicksort algorithm to be only a little slower than the sequential quicksort algorithm. Tables 1 and 2 show some initial timings we obtained on the Sparc 2 and Sequent from a simple fine-grained implementation of quicksort algorithm (outlined below).

```
quicksort_range(ls:LIST{T}; l,u:INT) is
   -- Use quicksort to sort the range '[l,u]' of 'ls'.
   .....
   -- Divide into two ranges of values and either create
   -- two threads to work on them, or perform recursive
   -- calls.
      if u-l>grain then
         coord :- quicksort_range(ls, l, m-1);
         coord :- quicksort_range(ls, m+1, u);
      else
         quicksort_range(ls, l, m-1);
         quicksort_range(ls, m+1, u);
      end; -- if
   ....
end; -- quicksort_range
```

Note that the sequential version does not create any threads during the sorting process. The following points should be noted from tables 1 and 2.

Characteristics	Time (seconds)
(i) 1-processor, Sequential	2.10
(ii) 1-processor, Parallel, Normal thread creation	2.58
(iii) 1-processor, Parallel, Optional thread creation	2.25
(iv) 2-processor, Parallel, Normal thread creation	1.48
(v) 2-processor, Parallel, Optional thread creation	1.38
(vi) 4-processor, Parallel, Normal thread creation	1.71
(vii) 4-processor, Parallel, Optional thread creation	1.73

Table 2: Times to quicksort 10,000 integers on a Sequent (Grain-size $\leq$ 30).

Characteristics	Time (seconds)
(i) 1-processor, Parallel, Normal thread creation	2.33
(ii) 2-processor, Parallel, Normal thread creation	1.29
(iii) 4-processor, Parallel, Normal thread creation	0.81

Table 3: Times to quicksort 10,000 integers on a Sequent (Grain-size $\leq$ 100).

- Although the improvement is not substantial with 2 or 4 processors, it does allow a parallel program to be executed on a single processor, with a lower performance penalty than would normally be incurred.
- This example is simply to illustrate the usefulness of code improvement. Hence not much attention was paid to the appropriate grain-size that would achieve better speedup. In Table 2, each thread sorts 30 or fewer items. When we changed the grain size to 100 or fewer items per thread, more substantial time improvements were obtained without the code-improvement feature (Table 3).

This simple example gives a glimpse of the importance of careful algorithm design and implementation. As will be pointed out in the conclusion, we plan to look further in the issues of implementing algorithms and data structures in pSather. There appears to be no inherent problem in applying expression-level parallelism [20] to pSather programs and we will explore the possibility that some of the same thread mechanisms can be used in such cases.

7 Future Directions

This paper has described the design and implementation of the monitor concept which is the basic parallel construct in pSather. We need to study more carefully "safe" declarations, "mutex" classes and data placement pragmas (Section 1). At the same time, our next step is to study parallel programming using pSather on an actual MIMD uniform shared memory machine (Sequent) and uniprocessor (Sparc) in the following respects.

- We plan to study how some of the serial library classes can be converted to work correctly in a parallel program. The aim of this exercise is to learn more about the usefulness of the parallel constructs and any limitations.
- Substantial theoretical work has been done in designing parallel algorithms using PRAM as the machine model. However, the efficiency of an implementation often falls short of expectations (eg[45]). We therefore plan to implement a number of selected graph algorithms. There are a number of reasons for picking graph algorithms over other kinds of algorithms.

  Symbolic vs Numerical The parallel implementation of symbolic algorithms as a whole are not as well understood as numerical algorithms.

  Data structures The data structures are unlike the regular arrays used in numerical code for dense matrix computations. As a result, we need to understand how to program, so that more than one thread can work simultaneously on the same data structure.

  Availability of Theoretical Results A number of algorithms (eg connected component[48], maximum flow[23]) have well-understood theoretical complexity. The question is whether these algorithms can be implemented efficiently in an actual parallel environment.

  Using the performance statistics of these implementations, we can further fine-tune the runtime and compiler in the following respects.

  Garbage collection The current Sather compiler uses a standard off-the-shelf garbage collector and the pSather compiler does not provide any garbage collection. There is a large body of literature on garbage collection, and we would like to study techniques which are suitable for the pSather (parallel) environment.

  Performance Improvements There are two aspects of performance improvements in the current implementations.

  The first is dynamic optimizations. We plan to study what kind of information can be gathered and generated by the compiler for the runtime support to do some optimizations An example of this can be found in a quicksort program shown in Section 6.3. From some performance measurements on the Sequent, sorting 10,000 integers (with grain-size ≤ 30 items) on 1 processor using the parallel implementation took 2.53 seconds as compared to 2.10 seconds for the sequential implementation. As discussed, an obvious optimization might be to avoid creating new threads in the 1-processor case.[4]

  The alternative is static optimizations. The compiler may explicitly re-structure the code to reduce thread-switching and synchronization. For example, an implementation of the connected component algorithm[48] is such that for each vertex of the graph, we create a new thread. However, each iteration of the algorithm has several points where all the threads have to synchronize. This requires a large amount of thread-switching, and the performance deteriorates rapidly with large number of vertices. There are two possible approaches to this problem. The first is to redesign the algorithm to reduce the amount of synchronization. The second is to implement the algorithm such that

[4] This can only be done if certain criteria, eg the thread being non-blocking, are satisfied.

only as many threads as processors are created, and each thread handles more than one vertex. This exposes program pragmatics such as the number of processors, which however could be made available in a rather machine independent way by defining a standard class (customized by each implementation) providing general ways to get this kind of information.

We also plan to port pSather to other machines, in particular, NUMA multiprocessors.[5] This will enable us to determine the portability of the runtime support and the effectiveness of C as a high level intermediate language. As described in Section 6.1, we currently have a pSather compiler that generates identical C code for both Sparc and Sequent. We shall examine the portability of this compiler when we try porting pSather to other (NUMA) machines.

The whole set of problems mentioned above require further work in a NUMA architecture. Our strategy is to first try to solve the compiler portability issue in a NUMA machine, and then re-examine the implemented algorithms in that environment. In addition, we plan to examine a sufficiently large application, such as the N-body problem[24], to study data placement problems which will arise. There has been some work done on the data allocation problem on distributed memory machines([31], [36], [12], [25], [47], [32]), but the studies have been limited to the allocation of arrays and/or SIMD machines (such as the Connection Machine). We are also investigating general algorithms for parallel objects spread across a NUMA architecture, such as parallel sets and graphs.

There are other possible directions in further research with pSather which were not mentioned above.

- A major complaint of most parallel programmers is insufficient programming/debugging support. Parallel debuggers is still an active area of research,[6] and we do not yet have a good idea of useful debugging mechanisms for parallel programs.

- Additional language constructs may be needed to do parallel programming effectively. We give two examples here.

  The first glaring language construct that is missing in the current Sather/pSather implementation is the exception mechanism. [46] gives an initial proposal for an exception mechanism in Sather. However, since our eventual goal is to integrate both compilers, and eliminate the distinction between Sather and pSather, we would have to review the exception mechanism carefully before extending the language.

  A second construct that we are considering adding is a statement-level parallel iterator (which would be "paraloop" in Sather) which is similar to the parallel *DO* statement of numerical languages like Fortran D.

We expect pSather to be a practical language for exploring parallel program implementation because of the following advantages:

- As pointed out in [51], abstraction mechanisms are useful in building and debugging large parallel programs. This is where the object-oriented aspects of pSather should come in handy.

- Unlike object-oriented languages (such as Smalltalk and Eiffel) which incur high runtime costs, a performance evaluation of Sather[35] shows that the performance of a Sather program is close to a comparable C program.

- It is a relatively clean language, offering certain constructs such as strong typing, storage management and class parameterization which are not available in efficient object-oriented languages such as C++.

[5] [34] and [33] are examples of work in building NUMA multiprocessors.
[6] [44] gives a non-exhaustive list of references of work on parallel debuggers.

Acknowledgements

Thanks to Krste Asanovic, Joachim Beer, Jeff Bilmes, Steve Omohundro, Abhiram Ranade and Heinz Schmidt, who have contributed much to the language design and implementation.

References

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[2] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, Cambridge, Massachusetts, 1987.

[3] Pierre America. *Issues in the Design of a Parallel Object-Oriented Language*. Philips Research Laboratories, Eindhoven and University of Amsterdam, March 1 1989. Part of POOL2/PTC Distribution Package.

[4] Pierre America. *Programmer's Guide for POOL2*. Philips Research Laboratories, Eindhoven and University of Amsterdam, January 10 1991. Part of POOL2/PTC Distribution Package.

[5] Pierre America and Ben Hulshof. *Definition of POOL2/PTC, a Parallel Object-Oriented Language*. Philips Research Laboratories, Eindhoven and University of Amsterdam, March 15 1991. Part of POOL2/PTC Distribution Package.

[6] Birger Andersen. Ellie - a general, fine-grained first class object based language. Technical report, University of Copenhagen, July 1991.

[7] Birger Andersen. *Ellie Language Definition Report*. PhD thesis, University of Copenhagen, Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copen-Hagen, Denmark, June 1991. Second edition of paper in ACM SIGPLAN Notices, 25(11):45-64, November 1990.

[8] Birger Andersen. *Fine-grained Parallelism in Ellie*. PhD thesis, University of Copenhagen, Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copen-Hagen, Denmark, June 1991.

[9] Thomas E. Anderson. Fastthreads user's manual. FastThreads software package manual, January 1990.

[10] Colin Atkinson, Stephen Goldsack, Andrea Di Maio, and Rami Bayan. Object-oriented concurrency and distribution in dragoon. Technical Report Research Report DoC 89/3, Imperial College, June 1989.

[11] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A language for distributed programming. *SIGPLAN Notices*, 25(5):17–24, 1990.

[12] Vasanth Balasundarm, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, pages 213–223, Williamsburg, Virginia, April 1991.

[13] H. Boehm and Weiser M. Garbage collection in an uncooperative environment. Software Software Practice & Experience pp. 807-820, September 1988.

[14] P. Brinch Hansen. The programming language concurrent pascal. IEEE Transactions on Software Engineering 1: pp. 199-207, June 1975.

[15] P. Brinch Hansen. Monitors and concurrent pascal: A personal history, June 1991. Private communication.

[16] D. Caromel. A general model for concurrent and distributed object-oriented programming. *SIGPLAN Notices*, 24(4), April 1989.

[17] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Cool: A language for parallel programming. Technical Report CSL-TR-89-396, Computer Systems Laboratory, Stanford University, October 1989.

[18] Andrew A. Chien and William J. Dally. Concurrent aggregates (ca). In *Proceedings of the ACM SIGPLAN Conference on the Principles and Practice of Parallel Programming*, 1990.

[19] Michael Coffin. Par: A language for architecture-independent parallel programming. Technical Report TR 89-18, Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, September 28 1989.

[20] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8-11 1991.

[21] Flavio De Paoli and Mehdi Jazayeri. Flame: A language for distributed programming. Hewlett-Packard Laboratories, Palo Alto, CA 94304.

[22] N. H. Gehani and W. D. Roome. Concurrent c. *Software - Practice and Experience*, 16(9):821–844, September 1986.

[23] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[24] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. ACM Distinguished Dissertations. The MIT Press, Cambridge, Massachusetts, 1988.

[25] Seema Hiranandani, Joel Saltz, Harry Berryman, and Piyush Mehrotra. A scheme for supporting distributed data structures on multicomputers. Technical Report NASA Contractor Report 181987, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665.

[26] C. A. R. Hoare. Monitors: an operating system structuring concept. Communications of the ACM 17: pp. 156-164, April 1974.

[27] Waldemar Horwat, Andrew A. Chien, and William J. Dally. Experience with cst: Programming and implementation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.

[28] Jin H. Hur and Kilnam Chon. Overview of a parallel object-oriented language clix. Technical Report CS-TR-87-25, Computer Science Department, Korea Advanced Institute of Science and Technology, Seoul, Republic of Korea, 1987.

[29] Thomas W. Doeppner Jr. and Alan J. Gebele. C++ on a parallel machine. Technical Report CS-87-26, Brown University, Department of Computer Science, Brown University, Providence, RI 02912, November 17 1987.

[30] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[31] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[32] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. Technical Report NASA Contractor Report 181981, ICASE Report No. 90-7, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665., Jan 1990.

[33] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. Technical Report CSL-TR-89-404, Computer Systems Laboratory, Stanford University, December 1989.

[34] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. Design of the stanford dash multiprocessor. Technical Report CSL-TR-89-403, Computer Systems Laboratory, Stanford University, December 1989.

[35] Chu-Cheow Lim and Andreas Stolcke. Sather Language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, Berkeley, Ca., May 1991.

[36] Richard J. Littlefield. Efficient iteration in data-parallel programs with irregular and dynamically distributed data structures. Technical Report 90-02-06, Department of Computer Science and Engineering, FR-35 University of Washington, Seattle, Washington 98195, USA., Feb 1990.

[37] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[38] Betrand Meyer. Sequential and concurrent object-oriented programming. In *TOOLS*, 1990.

[39] Thanasis Mitsolides. *The Design and Implementation of ALLOY, a Higher Level Parallel Programming Language*. PhD thesis, Department of Computer Science, New York University, June 1991.

[40] Greg Nelson, editor. *Systems Programming in Modula-3*. Digital Equipment Corp., October 17 1990.

[41] Stephen M. Omohundro. The Sather Language. Technical report, International Computer Science Institute, Berkeley, Ca., 1991.

[42] Stephen M. Omohundro, Chu-Cheow Lim, and Jeff Bilmes. The Sather Language compiler/debugger implementation. Technical report, International Computer Science Institute, Berkeley, Ca., 1991 (in preparation).

[43] Joseph Ira Pallas. *Multiprocessor Smalltalk: Implementation, Performance and Analysis*. PhD thesis, Stanford University, June 1990. Also available as technical report CSL-TR-90-429.

[44] Cherri M. Pancake and Sue Utter. A bibliography of parallel debuggers, 1990 edition. *SIGPLAN Notices*, 26(1):21–37, Jan 1991.

[45] Edward Rothberg and Anoop Gupta. Parallel iccg on a hierarchical memory multiprocessor – addressing the triangular solve bottleneck. Technical report, Department of Computer Science, Stanford University, Stanford, Ca., September 1990.

[46] Heinz W. Schmidt and Jeff Bilmes. Exception handling in psather, 1991. Extended Abstract.

[47] L.R. Scott, J.M. Boyle, and B. Bagheri. Distributed data structures for scientific computation. Technical Report IMA Preprint Series #291, January 1987, Institute for Mathematics and Its Applications, University of Minnesota, 514 Vincent Hall, 206 Church Street S.E., Minneapolis, Minnesota 55455.

[48] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[49] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[50] Jan van den Bos and Chris Laffra. Procol: A concurrent object-oriented language with protocols delegation and constraints. Technical report, Department of Computer Science, University of Leiden, December 6 1990.

[51] Katherine Anne Yelick. *Using Abstraction in Explicitly Parallel Programs.* PhD thesis, MIT, MIT Laboratory for Computer Science, Cambridge, MA 02139, December 1990.

[52] Y. Yokote and M. Tokoro. Experience and evolution of concurrentsmalltalk. In *Proceedings of OOPSLA*, pages 406–415, Orlando, Florida, December 1987. ACM.

[53] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* The MIT Press, Cambridge, Massachusetts, 1987.