Pascal-F Verifier Internal Design Document

Language Issues

A Safe Approach to Verifiable Multiprogramming

John Nagle
FACC / Palo Alto

1. Multiprogramming

The defining memorandum for Pascal-F describes a form of multiprogramming that appears to be strictly run-to-completion. Some form of preemption is clearly necessary for dealing with some of the more time-critical functions encountered in engine control. The expression of interrupts as signals is a good, consistent approach. However, not distinguishing between interrupts of differing importance will probably prevent the system from responding to time-critical interrupts rapidly enough. Some form of pre-emptive scheduling, in which external events can cause low priority tasks to be suspended in favor of high priority processing required to handle the external event, appears to be required.

In the Pascal-F manual, the concept of monitor priorities is introduced, which provides such a means for pre-emption.

1.1 A Safe Approach to Multiprogramming

Introducing the concept of pre-emptive scheduling in a run-to-completion multiprogramming system introduces the problem of *clashing.* In a run-to-completion system, each process can assume that shared variables will not be changed by another process unless the first process explicitly gives up control by making a scheduling request (such as a ‘send’ or ‘wait’). In a multiprogramming system with pre-emptive scheduling, this assumption does not hold. When one process changes a shared variable while another process is in the midst of code that accesses that variable, we have a clash. There are two kinds of sharing that can lead to clashes. These situations are:

A. when a high priority process writes into a variable that can be read by a low priority process, and

B. when a low priority process writes into a variable that can be read by a high priority process.

As shown by the following two examples, either of these types of sharing can introduce clashes.

It is easy to see how type A sharing can get one into trouble. The following sequence of statements ought to exchange the values of the variables X and Y.

```
1: T := X;
2: X := Y;
3: Y := T;
```

But suppose X and Y are variables accessible to more than one monitor, with the above statements being in a low priority process. Next suppose an interrupt occurs after statement labeled 2, so that the higher priority process gets control and changes the values of X and Y. When control returns to the lower priority process, statement 3 will be executed and the values of X and Y will be inconsistent. The variable X will contain the value stored by the high priority process, and Y will contain the value stored by the low priority process. Thus, type A sharing should not be allowed.

On first inspection, there does not appear to be any harm in type B sharing, in which a high priority process can interrupt a low priority process and simply inspect its variables. Yet consider the following example.

```
4: STP := STP + 1;
5: STACK[STP] := NEWVAL;
```

Suppose STACK and STP are variables that are accessible to more than one monitor, and that a high priority procedure can interrupt the above statements and inspect the top of the stack. If the interrupt takes place after the statement labeled 4, the high priority procedure will consider whatever garbage is in STACK[STP] to be the top of the stack. Thus, type B sharing should not be allowed, either.

In both these examples, a low priority process P1 is being interrupted by a high priority process P2 while P1 is updating a pair of variables in what ought to be an indivisible operation. In the first case, the solution is to provide access procedures UPDATE(NEWX, NEWY) and SWAP to the monitor in which X and Y are declared. The exclusion implicit in monitor procedures will prevent the UPDATE operation from taking place while the SWAP is in progress. In a similar fashion, the problem in the second example is solved by providing PUSH and TOP procedures, so that TOP cannot be executed while a PUSH is in progress.

These solutions are valid only if the implementation has the property that at each priority level, there can be only one process that has control of a monitor. And the notion of priority is meaningless unless of all the processes that are eligible to run, the one with the highest priority is running. The approach defined by the following rules has these two properties.

1. Every module has a scheduling priority. The scheduling priority may be specified in the source program. The absence of a level specification implies that code in the module executes at the lowest scheduling priority. For this discussion, scheduling priorities are positive integer values and the largest number is the highest priority.

2. All executable statements execute at the priority of the module in which they are enclosed.

3. The priority of a process is determined by the module in which it is currently executing code.
4. The states defined for a process are RUNNING, PRE-EMPTED, READY TO RUN, and WAITING.
5. The process currently in control of the CPU is in the state RUNNING. No more than one process may be in this state at any time.
6. When a running process requests a ‘wait’ on a signal, the process’s state changes to WAITING.
7. When a signal is activated by a ‘send’ operation or an interrupt, the highest priority process WAITING on the signal becomes READY TO RUN.
8. When a change in priority or state of any process occurs, it may be necessary to select a new process to run. It is necessary to select a new process only when

    - the level of the RUNNING process drops below that of some READY TO RUN or PRE-EMPTED process.
    - the level of the RUNNING process becomes equal to that of some PRE-EMPTED process.
    - the RUNNING process performs a ‘wait’ (which changes its state to WAITING)
    - the RUNNING process performs a ‘send’ on a signal upon which a process is waiting (which changes the state of the WAITING process to READY TO RUN) and the priority of the WAITING process is higher than that of the RUNNING process

    When any of these conditions occur, the state of the RUNNING process becomes READY TO RUN or WAITING, as appropriate, and one of the highest priority PRE-EMPTED or READY TO RUN processes in the system becomes the RUNNING process. When it is necessary to select one of several processes at the same priority, PRE-EMPTED processes are selected before READY TO RUN processes.

The above rules have the consequence that for a given priority level, only the following conditions can exist:

- There is one RUNNING process and no PRE-EMPTED process.
- There is no RUNNING process and one PRE-EMPTED process.
- There is no RUNNING process and no PRE-EMPTED process.

This property is very valuable. Processes change to WAITING or READY TO RUN states only at points of their own choosing. We refer to these as ‘singular points’. Clashes can thus occur only between processes in the RUNNING or PRE-EMPTED state. The above property implies that only one process per level can be in such a state. Therefore, two processes at the same level cannot become involved in a clash.

If two processes at the same level cannot become involved in a clash, restricting access to shared variables to a single level associated with the variable will guarantee freedom from clashes. We can easily enforce this syntactically in Pascal-F by forbidding imports of global variables into a module of different level than the enclosing module.

It is also necessary to restrict the circumstances under which a monitor variable can be passed as a VAR argument to a routine not in the monitor. There are two pitfalls.

The first problem is one of aliasing. If a monitor variable is passed to another level, the same location can be known by different names at different priority levels. It must be made impossible for the higher priority process to interrupt the lower priority process and change the variable through the alias.

There is a second, more fundamental, problem. The principle reason for using monitors is the monitor invariant. If the monitor routines are written to make the monitor invariant TRUE whenever the monitor is left, it can be assumed that the invariant will also be TRUE whenever the monitor is entered. If a monitor variable is passed as a VAR argument to a routine outside the monitor, the code outside the monitor can change the monitor variable in such a way as to invalidate the monitor invariant. Therefore, whenever a variable is passed as a VAR argument to a routine R outside the monitor, the Verifier must check that the EXIT assertion for R is strong enough to ensure that the monitor invariant is still TRUE when the routine again enters the monitor by returning from R.

This test is not, however, sufficient to prevent problems. When routine that is passed a monitor variable is allowed to invalidate and then re-establish the monitor invariant, the monitor must not be entered during the time in which the invariant has been broken. Enforcing this second restriction also prevents problems with aliasing, because the code that accesses the monitor variable using the name by which it was declared will not cannot be executed while there is another name for the variable.

Suppose a routine R is passed a static variable declared in a monitor M. We will call the routine "unsafe" if it is possible for the monitor M to be reentered before R returns. The routine is unsafe whenever one of the following conditions hold.

- R has a different priority than M.
- R calls a routine exported from M.
- R performs a ‘wait’ operation.
- R calls an unsafe routine.

A routine is "safe" whenever it is not unsafe. When a process P leaves monitor M by calling a safe routine, the process must stay at the same priority as M until the call to R returns. Because the process cannot execute a ‘wait’ during that time, it cannot encounter a singular point. A process with a priority higher than P can interrupt P by pre-empting it. If the intervening process attempts to lower its priority and enter monitor M then process P, which will be in the PRE-EMPTED state, will get control instead.

The definition of "safe" given above is more restrictive than necessary. For example, a routine could be considered safe if it performs a ‘wait’, as long is it could be demonstrated that during that ‘wait’ no other process could get control of monitor M. However, the definition given for "safe" is a static property of the program that can be checked without the use of a theorem prover.

1.2 Sending up, down, and sideways

We see no fundamental problem with sending signals to processes at a higher level, an equal level, or a lower level. Given a suitable implementation following the rules above, sends are not a singular points, as shown by the following analysis.

1. Sends to lower priority processes obviously must not give up control, since this would violate the priority rules. Thus sends down are not singular points.
2. Sends to higher priority processes must give up control, because the process being sent to preempts the sending process. This must be treated as a preemption, with the sending process changing from RUNNING to PRE-EMPTED. Since pre-emption does not destroy the validity of monitor invariants, sends up are not singular points. The normal isolation between priority levels makes this work.
3. Sends to processes of the same level should not give up control. This is merely to assure uniform semantics. Sends sideways, implemented in this way, are thus not singular points.

We recommend that send be implemented in such a way as to insure the integrity of the above statements. The send mechanism will then be simpler, both to verify and understand.

1.3 ‘Safe’ functions

The draft manual refers to a special kind of priority-independent routine that is intended to function at the level of its caller. This useful facility is intended for general-purpose routines, such as standard mathematical functions. One should think of such a routine as behaving as if there was a local copy of the routine in each monitor using it. This view leads naturally to the restriction that such routines may not reference global variables, or call non priority-independent routines. Access to global variables would imply data sharing across priorities, an unsafe practice.

While it is quite possible for the compiler to recognize routines that could be safely treated as priority-independent automatically, we see this as undesirable. There is a danger that the programmer will be unaware that one of his routines has been transmuted into priority-independent mode. Consider, for example, a call to a lower-priority routine performed specifically to permit another process of a lower priority to run. If the compiler recognized priority-independent routines automatically, the programmer’s intended priority drop would be ommitted and the programmer’s intent not followed correctly. We recommend that priority-independent routines be identified by a suitable keyword, and that the compiler enforce the rules above.

1.4 Initialization of monitors

Since variables have no defined initial value, it is not possible to detect from within a monitor whether the monitor has been initialized. This fact makes it impossible to write a monitor whose exported procedures protect themselves against calls made before the monitor is properly initialized. The monitor invariants need such protection; clearly the monitor invariants do not hold at startup, but must hold when exported procedures are called. It seems reasonable to require that the monitor invariants be made valid by code in the body of the montitor before the first WAIT or exit from the monitor body. Thus, an INIT of the monitor will force the monitor variables into a consistent state.

There is a problem with calls to procedures exported from a monitor. Although calls to such procedures can be made before the INIT of the monitor, a procedure called in such a way has no reliable knowledge about the state of the monitor, which prohibits operations dependent on the values of monitor variables. The only use which seems reasonable for such procedures is allowing the export of access procedures for DEVICE variables.

When a process terminates by reaching the end of a monitor body, it should maintain the validity of the monitor invariant. This insures that once valid, the monitor invariant remains valid.

We currently see re-INIT of a monitor as being associated with exception handling, and in programs in which exceptions are not raised, monitors should be INITed only once.

1.5 Implementation Issues

Implementation of this approach is straightforward. A scheduling nucleus is required that, when requested to do so, will find the active process with the highest priority and activate it. The compiler must generate code to call this nucleus as required, which is not difficult. The ‘send’ and ‘wait’ primitives generate explicit calls to the nucleus. Implicit calls must be generated when the priority of a process changes.

The priority of every line of code is known at compile time. When control passes from one module to another at a different level, the compiler must generate appropriate code. Going to a higher level merely requires setting a global flag to the current level; going to a lower level requires that a check be made for other processes waiting for execution at a higher level. This is best done by maintaining a global flag indicating the priority level of the highest priority process waiting to run. The compiler should generate code to compare the level being dropped to with that of the highest process waiting, and call the dispatcher only if the waiting process is of higher priority. This avoids going into the nucleus every time a module boundary is crossed.

This model of multiprogramming is safe from clashes, but it may be unwieldy. It will force a programming style in which access to shared variables is always performed through access procedures. This style of programming may be inefficient unless optimization algorithms are provided in the compiler to handle small procedures efficiently.

The most important optimization, and one almost essential in any systems programming or real-time language, is the ability to develop procedures in-line when required rather than generating out-of-line code. Procedures should be generated in-line in any of the following situations.

- The procedure is only called from one place
- The procedure is smaller than its calling sequence, including setup and fetching of arguments

This optimization eliminates the inefficiency associated with calling tiny procedures that access single variables, and its use counters the argument that highly modular programming is inefficient.

An additional machine-dependent optimization is possible to improve the handling of the implicit level changes generated when control crosses a module boundary. A peephole optimizer should look at the output code, either in assembly or in some intermediate form, and when sequences of the form

```
RAISE priority level from level 9 to level 10
any non-interruptable hardware operation
DROP priority from level 10 to level 9
```

are noted, the RAISE and DROP operations can be omitted. With both of the optimizations shown implemented in the compiler, there is no performance penalty associated with using the safe approach to multiprogramming described here.