MULTITASKING, SCHEDULING: APPROACHES FOR ADA

Fred Maymir-Ducharme
IIT Research Institute
4600 Forbes Blvd
Lanham, MD 20706

Mike Kamrad
Unisys CSD
M/S Court
P.O.Box 64525
St. Paul, MN 55164

INTRODUCTION

The subject of modifying Ada's runtime environment (RTE) in order to meet stringent timing and scheduling requirements is being addressed by various special interest groups, including: Joint Integrated Avionics Working Group (JIAWG), the Common Ada Run Time Working Group (CARTWG), and the Ada Run Time Environment Working Group (ARTEWG). This paper discusses the proposals reviewed by some ARTEWG members, using the proposed Catalog of Interface Features and Options (CIFO) [15] entry for dynamic priorities as the basic concept to meet scheduling needs. The basic premise is to build on the proposed implementation of pragma DYNAMIC_PRIORITIES and expand the definitions as necessary.

This position paper discusses the extension of dynamic priorities from just being used to schedule ready queues, to also serve as a mechanism for scheduling entry queues and the select alternative. Other scheduling mechanisms (e.g., Preference Control and explicit FIFO) are also discussed. To minimize the effect on Ada, this position paper takes the approach that these modifications should be implemented as pragmas instead of as new language constructs. Pragmas are implementation-dependent features, as defined by the Ada Language Reference Manual. Programmers must be made aware that a compiler can ignore a pragma without warning if it the compiler does not support that specific pragma. Recent efforts have compiled various lists of existing pragmas for the various validated Ada compilers. Formal definitions of pragmas and standardized pragmas are needed in the future to allow the efficient and effective use of pragmas.

The suggested syntax and semantics are defined and illustrated in the following text. These proposals have been submitted to the Ada 9X Project and to the CIFO for the Ada RTE.

SCHEDULING MECHANISMS FOR READY QUEUES

Presently, the only support for Ada priorities is the pragma PRIORITY, which is static. A CIFO proposal for a pragma to support dynamic priorities will be assumed as the base in this section - pragma DYNAMIC_PRIORITIES. The DYNAMIC_PRIORITIES pragma and the PRIORITY pragma will have to be mutually exclusive; the CIFO proposal suggests that if DYNAMIC_PRIORITIES are needed, then only DYNAMIC_PRIORITIES should be used. The proposal includes the following procedures for the viewing and setting of priorities dynamically.

```
procedure SET_PRIORITY(T:in TASK_ID; P:in PRIORITY);

procedure GET_PRIORITY(T:in TASK_ID; P:out PRIORITY);
```

Procedure SET_PRIORITY sets the priority of the task associated with the TASK_ID "T" with PRIORITY "P". SET_PRIORITY is a scheduling point in same sense that Ada defines synchronization. Procedure GET_PRIORITY gets the priority of the task associated with the TASK_ID "T" and returns this value in "P." The need for dynamic priorities is illustrated by several papers in the 1988 & 1989 International Workshops on Realtime Ada Issues and by several entries to the Ada 9X Project Revision Request Report [1].

The following assumptions are made by this proposal: Scheduling is the state transition operation that makes tasks runable. Dispatching selects a runable task for execution. Dispatching is done on priority-basis, at the instance of dispatching. The dispatching of tasks with the same priority is arbitrary; that is, left to the compiler-specific implementation (e.g., time-slice, round robin, run-until-blocked, etc.).

Presently, the scheduling of a task in a ready queue is done in First In, First Out (FIFO) order, unless pragma PRIORITY (if implemented by the specific compiler) has been specified. The dispatching of ready tasks is actually done on a priority basis - although some implementations' range of priorities only consists of one unique value. If pragma PRIORITY is not specified for a task, then its priority is undefined. The ordering of entry queues is FIFO per the Ada semantics. Pragma PRIORITY is statically defined and therefore very limiting in many applications. Many scheduling algorithms (e.g., Rate Monotonic) that require the RTE to be modified, are under discussion as possible solutions for today's real-time, embedded systems scheduling requirements. Other solutions entail extending the task related data to include additional variables that can be read, modified, and utilized for scheduling purposes. This paper supports the extension of the priority concept from being static to include dynamic capabilities as defined above. The implementation of dynamic priorities can make use of existing language constructs and code used by pragma PRIORITY, and thus minimize the need to modify the language and the RTE.

SCHEDULING MECHANISMS FOR ENTRY QUEUES

Entry queues are ordered in FIFO order. If several calls to a specific entry are made, they are queued and accepted in the order they were received. The need to allow priorities to override the FIFO order is supported by several entries in the Ada 9X Project Revision Request Report [1], several papers [4,6,9,11], the implementation of dynamic priorities, and the "by" and "suchthat" constructs in Concurrent C. A new pragma - pragma PRIORITY_FIRST (NAME_OF_ENTRY) - is needed to specify this new ordering mechanism for entry queues.

Pragma PRIORITY_FIRST is defined per entry, not per each individual accept statement, nor by the select statement. The pragma should be declared in the task specification, not in the task body.

```
task FUZZ is
entry BUTTON;
pragma PRIORITY_FIRST(BUTTON);
...
end FUZZ;

OR

task type FUZZY is
        entry BUTTON;
pragma PRIORITY_FIRST(BUTTON);
        ...
        end FUZZY;
        WUZZY:FUZZY;
```

Figure 1. Use of pragma PRIORITY_FIRST

All of the entry queues for BUTTON of all FUZZY objects are ordered by priority; that is, the call from the task with the highest priority is serviced from the queue first. The default scheduling mechanism will continue to be FIFO when pragma PRIORITY_FIRST is not specified for that specific entry. This pragma must be specified for each individual entry declared in the task specification. One could not specify the pragma randomly because the same queue serves callers of the same entry and conflicts could arise; and conversely, it would be too limiting to specify it only once per program and have it affect every entry queue.

The specification of pragma PRIORITY_FIRST indicates to the compiler that the entry queue should always have the call with the highest priority at the head of the queue. Whether the queue is designated as FIFO or PRIORITY_FIRST, the accept statement will always take the task at the head of the queue. Every entry queue operation is defined to be atomic. (e.g., dequeue, enqueue, read_queue, write_queue, ...) The default scheduling mechanism for entry queues will (implicitly) be the existing FIFO ordering, unless pragma PRIORITY_FIRST has explicitly been declared. The option exists to define another pragma - pragma FIFO_FIRST(NAME_OF_ENTRY) - which allows the explicit specification of FIFO ordering for specific entry queues. As stated earlier, invoking PRIORITY_FIRST is for a specific entry queue, regardless of whether the accept statement is within or outside of a select statement. The declaration of PRIORITY_FIRST and FIFO_FIRST belong in the task specification.

SCHEDULING MECHANISMS FOR SELECT ALTERNATIVES

Race and Availability Control Definitions

Nondeterministic selection: an unconstrained choice from a finite number of alternatives. Existing nondeterministic constructs are not sufficiently structured for today's concurrent programming language needs. The classification of controls on nondeterminism facilitates the learning and understanding of nondeterministic constructs. A better understanding of these controls allows the programmer to become more efficient in controlling nondeterminism and thereby exploiting the available parallelism to a larger degree. The classifications of controls on nondeterminism also aids in testing and debugging by allowing the tester to explicitly specify the different execution paths desired by using the different controls.

Availability Controls on the Selection Process

Hoare [14] uses the notion of guards to denote the availability of an alternative for selection. If all of the guards for each alternative hold true, then one alternative is selected nondeterministically; otherwise, only the alternatives with true guards are considered for selection. We will classify this selection criteria among alternatives, controlled by the state of the alternative's guard, as Availability Control. Availability Control is the mechanism used to enable or disable each alternative's guard, thereby controlling the domain of alternatives available for selection. Availability Control can be further subdivided into the following categories:

PRIVATE CONTROL: nondeterminism restricted by conditions over variables local to the task. Private control is considered open if the condition is true; otherwise it is

closed. Private control was not intended to be allowed in Ada (but it can be implemented in Ada using a combination of the "when" and the "delay" primitives within the select statement). When private control is specified for an alternative, it can only be chosen if it is open. Private control restricts the nondeterministic choice by not considering alternatives with closed private control.

CONSENSUS CONTROL: nondeterminism restricted by environmental/communication constraints.
The alternatives are restricted to only those for which the communication is available from the rest of the system. Consensus control is considered established if the rendezvous can be established. This control can be attained in Ada by the use of the "accept" primitive within the select statement. An alternative is ready if all of the controls have been satisfied, i.e., if the private control is open and/or if the consensus control is established. The nondeterministic construct will check all of the controls in each alternative and then choose one of the ready alternatives. In many cases using the nondeterministic construct includes using a combination of these controls.

MUTUAL CONTROL: the capacity to enable/disable a nondeterministic choice based on an expression over both the caller's state and the server's state.
The major difference between mutual control and private control is that the conditional guard contains variables local to the server and from the callers parameters. Ada does not include a mutual control construct.

HYBRID CONTROL: the combination of any of the three latter controls described above.

Race Control on the Selection Process in Ada

Beyond Availability Control, there exists other races between different groups of entities and at different levels. Priorities are managed and supported at the operating system level. Other controls are exercised at the programming language level. We will classify the following controls at different levels as follows:

PRIORITY CONTROL: controls the race among different tasks at the system level.
In Ada, if two of more tasks with different priorities are in the ready state, the task with the highest priority will be selected for running.

FORERUNNER CONTROL: controls the race among different entries in an entry queue.
This control can be used to prioritize pending calls within an entry queue.

PREFERENCE CONTROL: controls the race among different alternatives within the select statement.
We classified preference control as nondeterminism restricted by a preferential order. Preference control gives the programmer the power to assign preferences to the alternatives within the nondeterministic construct. Each alternative inside a nondeterministic select construct may (but need not) have a preference value. A lower value indicates a lower degree of urgency. The range of preferences is implementation defined (e.g., a ready entry with a preference = 2 is chosen before a ready entry of preference = 1). Preference control gives nondeterminism a defined relational order between alternatives. The original suggested syntax for Ada was:

pref <expression> when <condition> => accept <entry>

This implementation would entail changing the language, adding a new language construct and modifying the select statement.

```
select
        pref 3 when B1 => accept entry1(...)
                do S1; end entry1
  or
        pref 2 when B2 => accept entry2(...)
                do S2; end entry2
  or
        pref 2 when B3 => accept entry3(...)
                do S3; end entry3
  or
    pref  1 when B4 => accept entry4(...)
                do S4; end entry4
end select;
```

Figure 2. Use of the Preference Control Construct

All of the nondeterminism constraints are listed before the entry call: first, the preference control (pref constant); next, the private control (when < expression >); and finally, the consensus control (accept <entry>). If a preference is not specified, it should default to the lowest value (e.g., if negative values for preferences are not allowed, then default to 0). The same value preference can be assigned to different entries within the same select statement to allow a greater amount of nondeterminism.

Ada includes various language constructs for the control of nondeterminism within a select statement, such as Private Control (e.g., when statement), Consensus Control (e.g., accept statement) and Hybrid Control (combining when and accept statements). Various forms of Race Controls (e.g., Preference Control, priority_select and dynamic priorities) have been suggested to meet the needs of the Ada community, which recognizes the expressive power of the select statement, but also demands the ability to better control it explicitly and dynamically.

Select Statement Scheduling Pragmas

Because we have discussed the use of dynamic priorities for the scheduling of ready queues and entry queues, it logically follows that the use of dynamic priorities for the scheduling of select alternatives should be investigated. This paper proposes a new pragma -pragma PRIORITY_SELECT - which explicitly tells the RTE to consider the priorities of the entry calls queued within the select statement. As defined in

[6,9], RACE CONTROL considerations come into place after the evaluation of the AVAILABILITY CONTROLS Pragma PRIORITY_SELECT specifies to the RTE to select the open alternative whose queue's head task has the highest priority. Pragma PRIORITY_SELECT is independent of pragma PRIORITY_FIRST, which specifies the ordering of each entry queue. Hence, if PRIORITY_FIRST is specified for each of the entries within the select statement, and PRIORITY_SELECT is also specified, the open alternative with the highest priority caller will always be selected. The combination of PRIORITY_FIRST queues with FIFO queues within a select statement with PRIORITY_SELECT will give the user more expressive power to easily meet varying requirements. PRIORITY_FIRST and PRIORITY_SELECT implementations must consider their interaction with DYNAMIC_PRIORITIES. PRIORITY_FIRST and PRIORITY_SELECT affect the ordering of the queue each time a new entry call is enqueued if pragma PRIORITY is used. Combining them with DYNAMIC_PRIORITIES adds the complexity of re-ordering dynamically any time a task's priority changes.

Pragma PRIORITY_SELECT can only be declared within a select statement, as illustrated in Figure 3. The user should be cautioned that mixing the ordering of entries' service by select may present an awkward protocol. The default ordering of entry queues will be FIFO (the existing mechanism). This will be explicitly specified and defined by the pragma.

```
task FUZZ is
        entry A();
        entry B();
        pragma FIFO_FIRST(A,B);
        entry C();
        pragma PRIORITY_FIRST(C);
end FUZZ;

                        ******

task body FUZZ is
        select
        pragma PRIORITY_SELECT;
                accept A();
        or
        accept B();
        or
                                accept C();
        end select;
end FUZZ;
```

Figure 3. Use of PRIORITY_SELECT

Note that a call to the entry B cannot be denied by a call to the entry A in the case where accept B is open and accept A is not AND A's caller has a higher priority than B's caller. The determination of open alternatives through the evaluation of guards is performed before any subsequent open alternative selection, per existing Ada language semantics.

Another very useful pragma for the scheduling of select alternatives is pragma PREFERENCE_SELECT. Pragma PREFERENCE_SELECT is an alternative implementation of the Preference Control construct [3] for Ada. PREFERENCE_SELECT explicitly tells the RTE to consider the preference of the entries within the select statement before making the selection. As previously defined, RACE CONTROL considerations come into place after the evaluation of the AVAILABILITY CONTROLS. Hence, pragma PREFERENCE_SELECT specifies to the RTE to select the available alternative with the highest preference. Pragma PREFERENCE_SELECT and pragma PRIORITY_SELECT are mutually exclusive. Pragma PREFERENCE_SELECT is independent of pragma PRIORITY_FIRST, which specifies the ordering of the entry queue. An associated pragma - pragma PREFERENCE_VALUE (ENTRY_NAME,PREFERENCE_EXPRESSION) - is necessary for each alternative to specify a preference value. Pragma PREFERENCE_VALUE has two parameters. The first is the entry name, and the second is the preference value or expression that will be associated with that entry during the selection process.

```
task FUZZ is
        entry A();
        pragma PREFERENCE_VALUE(A,X)
        entry B();
        pragma PREFERENCE_VALUE(B,Y);
        pragma FIFO_FIRST(A,B);
        entry C();
        pragma PREFERENCE_VALUE(C,X+Y)
        pragma PRIORITY_FIRST(C);
end FUZZ;

                        ******

task body FUZZ is
        X := 5;
        Y := -1;
        select
        pragma PREFERENCE_SELECT;
                accept A();
                        ... X:= - 1'
                        end A;

        or
                accept B();
                        ... Y := Y + 2;
                        end B;

        or

                accept C();
                        end select;
end FUZZ;
```

Figure 4. Use of PREFERENCE_SELECT and PREFERENCE_VALUE

Preference Control is dynamic. This is possible because expressions are legal parameters for pragmas. The amount of nondeterminism may be increased by giving more than one alternative the same preference. See [5,7,10] for various applications for Preference Control. Entries without a specified preference value will be assigned the lowest

preference value. Declaring PREFERENCE_SELECT in the task specification limits the use of Preference Control to alternatives with entries. Preference Control can be extended to include other non-accept alternatives by allowing the declaration of PREFERENCE_VALUE within the select statement. A preference value cannot be assigned to an "else" statement.

```
task FUZZ is                    task body FUZZ is
        entry A();              X := 5;
        entry B();              Y := -1;
        entry C();
end FUZZ;

select

        pragma PREFERENCE_SELECT;
                accept A();
                pragma PREFERENCE_VALUE(X);
                        ... X := X - 1;
                        end A;
or
                accept B();
                pragma PREFERENCE_VALUE(Y);
                        ... Y := Y + 2;
                        end B;
or
                when X > Y:
                pragma PREFERENCE_VALUE(Z);
                        ... Z := 2*Z;
or
                accept C();
                pragma PREFERENCE_VALUE(X+Y);

        end select;
end FUZZ;
```

Figure 5. Extending PREFERENCE_SELECT and PREFERENCE_VALUE

In this case, it is not necessary to pass the entry's name as a parameter to PREFERENCE_VALUE for every alternative (nor is it possible, because non-accept alternatives are not named, nor declared in the task specification), so pragma PREFERENCE_VALUE had be modified as follows and moved down to the task body, within the select statement for proper binding: pragma PREFERENCE_VALUE (PREFERENCE_EXPRESSION). The first implementation of PREFERENCE_VALUE is easier to implement because a preference value can easily be associated with the accept entry. It is also simpler and easier to understand, since the entry name and associated preference value are both on the same pragma call and all of this information can be included in the task specification. But the second implementation is much more powerful: it allows the user the ability to also control non-accept alternatives in the selection process and override Ada's implicit preference of accept entries over non-accept entries when required.

In all fairness, we may also want to include another pragma, FIFO_SELECT (very similar functionally to Burns' priority_select extension. Burns [2,8] suggests an enhanced select statement called priority select, which specifies to the RTE to select the first open alternative in the sequence ordered within the select statement. This construct is by definition, static. The only way to change the selection order is to re-write the select statement and re-order the alternatives. The assignment of the same preference to several alternatives is possible by nesting select statements within priority_select statements. Obviously, PRIORITY_SELECT, PREFERENCE_SELECT and FIFO_SELECT must be mutually exclusive, to avoid conflicting scheduling constraints.

If the decision is made to support all three of the select statement scheduling mechanisms discussed above, the implementor may consider combining the three pragmas into one - pragma SELECT_MECHANISM. Pragma SELECT_MECHANISM can be passed the desired scheduling mechanism as a parameter.

```
type SCHEDULING_MECHANISM is
(PRIORITIES,PREFERENCES,FIFO);

pragma
SELECT_MECHANISM(SCHEDULING_MECHANISM);
```

As in the previous discussions, pragma SELECT_MECHANISM must be declared within the select statement.

CONCLUSION

The International Workshop on Real-time Ada Issues has discussed dynamic priorities, priority inversion[13] and race controls the last couple of years. The extension of dynamic priorities for scheduling entry queues and select alternatives, as well as the implementation of the preference control and FIFO pragmas, will interact with the previously proposed constructs and have major effects requiring additional discussion and further analysis.

These proposals have been submitted to the Ada 9X Project and to the CIFO for the Ada RTE. These, and other proposed changes, are currently being reviewed and considered as changes for Ada in the 90s. Their effect on the Ada language as well as their impacts on priority inversion and other related Ada issues must be further scrutinized. Additionally, the interactions between these pragmas and CIFO entries such as dynamic priorities, trivial entries and asynchronous entry calls must be better defined and understood.

6. ACKNOWLEDGEMENT

The authors wish to thank Offer Pazy who was instrumental in the definition and specification of the proposed pragmas. Mr. Pazy's suggestions that additional research is needed regarding the justification or need for the explicit declaration of default mechanisms (e.g., FIFO_FIRST), the possible introduction of illegal or erroneous constructs, and the interactions and complexities introduced by applying PREFERENCE_VALUE to non-accept statements within a select statement will be investigated and reported on at some later time.

REFERENCES

[1] Ada 9X Project Revision Request Report, Office of the Under Secretary of Defense for Acquisition, Washington DC, August 1989.

[2] A. Burns, Using Large Families for Handling Priority Requests, Ada LETTERS January, February 1987 Vol. VII, No. 1.

[3] T. Elrad, F. Maymir-Ducharme Introducing the Preference Control Primitive: Experience with Controlling Nondeterminism in Ada, Proceeding of the 1986 Washington Ada Symposium in Laurel, Maryland, March 24-26, 1986, pp. 265-270.

[4] T. Elrad, F. Maymir-Ducharme Distributed Language Design: Constructs for Controlling Preferences, Proceedings of the 1986 International Conference on Parallel Processing in St. Charles, Illinois, August 19-22, 1986, pp. 176-183.

[5] T. Elrad, F. Maymir-Ducharme Efficiently Controlling Communication in Ada Using Preference Control, Proceedings of the IEEE 1986 Military Communications Conference in Monterey, California, October 5-9, 1986, pp. 22.3.1-22.3.8.

[6] T. Elrad, F. Maymir-Ducharme Race Control for the Validation and Verification of Ada Mulitasking Programs, Proceedings of the Sixth Annual National Conference on Ada Technology, March 14-17, 1988.

[7] T. Elrad, F. Maymir-Ducharme Satisfying Emergency Communication Requirements With Dynamic Preference Control, Proceedings of the Sixth Annual National Conference on Ada Technology, March 14-17, 1988.

[8] T. Elrad, F. Maymir-Ducharme Letter to the Editor in response to the A. Burns Jan./Feb. 1987 article "Using Large Families for Handling Priority Requests." Ada LETTERS May, June 1987 Vol. VII, No. 3, vii. 3-14 - vii.3-16.

[9] T. Elrad, Comprehensive Race Controls: a Versatile Scheduling Mechanism for Real-time Applications, Proceedings of the Ada-Europe International Conference, Madrid, June 13 -15, 1989.

[10] T. Elrad, F. Maymir-Ducharme Preference Control: A Language Feature for AIDA Applications, Proceedings of the 1987 Third Annual Conference on Artificial Intelligence & Ada, George Mason University, VA, October 14 - 15, 1987.

[11] N. Gehani, W. Roome Concurrent C* AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1985.

[12] N. Gehani, W. Roome The Concurrent C Programming Language Silicon Press, New Jersey, 1989.

[13] J. Goodenough, L. Sha, Real-Time Scheduling Theory and Ada, Technical Report CMU/SEI-89-TR-14 ESD-TR-89-22, April 1989.

[14] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[15] A Catalog of Interface Features and Options for the Ada Runtime Environment, Ada Runtime Environment Working Group Interfaces Subgroup, ACM SIGAda, December 1987.