1988

Execution of a High Level Real-Time Language

Luqi; Berzins, Valdis

IEEE

http://hdl.handle.net/10945/43626

Execution of a High Level Real-Time Language

Luqi

Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Prototype System Description Language (PSDL) is a high level real-time language with special features for hard real-time system specification and design. It can be used to firm up requirements through execution of its software prototypes. The language is designed based on a real-time model merging data and control flow and its implementation is beyond conventional compiler technology because of the need to meet real-time constraints. In this paper we describe and illustrate our research results: a special scheme used to meet the hard real-time constraints and the guidelines used to implement such a language in the target language Ada. The required software tools for automated translation and scheduling are also discussed. These tools have been designed and an prototype version has been partially implemented.

1. Introduction

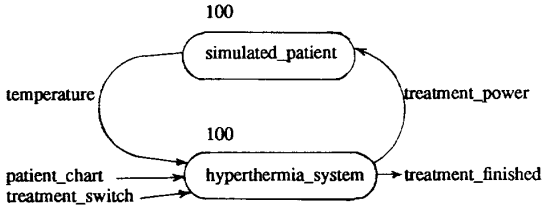
Achieving cost effective production of software systems and increasing the quality of software products with respect to meeting user requirements are especially important for the design and implementation of hard real-time systems. These types of problems are often too complex for unaided human understanding. The real-time requirements for responses of an embedded system needed to achieve the behavioral requirements of the enclosing system are often difficult to determine, and the need for meeting real-time deadlines often results in designs where code for conceptually unrelated tasks must be interleaved, complicating the design of such systems [6].

Rapid prototyping is one of the most promising approaches proposed for solving this problem. A prototype is an executable pilot version of the intended system, which is used as an aid for analysis and design rather than for delivery to the user. Rapid prototyping is especially effective for ensuring that the requirements accurately reflect the real needs of the user, increasing reliability, and reducing costly requirements changes. Rapid prototyping is more promising than other approaches because customers can rarely describe what they need, but can recognize it when they see it demonstrated. Prototypes are also useful for evaluating proposed designs with respect to performance goals. Computer aid is essential for the rapid construction and evaluation of prototypes for complex, real-time systems due to the difficulties outlined above [11].

PSDL (Prototype System Description Language) is a language designed for clarifying the requirements of complex real-time systems, and for determining properties of proposed designs for such systems by means of prototype execution [14]. PSDL is the basis for a proposed automated prototyping system that speeds up the prototyping process by exploiting reusable software components and providing execution support for high level constructs appropriate for describing large real-time systems in terms of an appropriate set of abstractions [1, 3, 11]. The language was designed to simplify the description of such systems and to support a prototyping method that relies on a novel decomposition criterion [15]. Design complexity is reduced by presenting a high level view of the system that provides separation of concerns, and the interleaving required for timely execution is achieved with the aid of automated tools for scheduling and translation. PSDL views a software system as a network of independent operators communicating via data streams. The operators are subject to non-procedural timing and control constraints. Each PSDL operator consists of two parts: operator specification and operator implementation. The operator specification is used as a basis for retrieving reusable components and for specifying system components during the prototyping process. The operator implementation part provides dataflow decomposition of the operator and timing and control constraints associated with the operators or an Ada reusable component if it is available. An example of a PSDL operator definition is shown in Fig. 1 [11].

Timing constraints for systems modeled as finite state machines have been classified as maximum, minimum, and durational [4]. PSDL has facilities for expressing these types of constraints. PSDL describes software systems as networks of operators connected by data streams, and subject to timing and control constraints. Operators can be either periodic or triggered

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart: medical_history,
   treatment_switch: boolean
 OUTPUT treatment_finished: boolean
 STATES temperature: real
  INITIALLY 37.0
 DESCRIPTION
 { The brain tumor treatment system kills tumor cells
   by means of hyperthermia induced by microwaves.
 }
END
IMPLEMENTATION
 GRAPH
```




```
 DATA STREAM treatment_power: real
 CONTROL CONSTRAINTS
  OPERATOR hyperthermia_system
   PERIOD 200 BY REQUIREMENTS shutdown
  OPERATOR simulated_patient
   PERIOD 200
 DESCRIPTION { paraphrased output }
END
```

Fig. 1 PSDL Operator Specification and Implementation

[1]This research was supported in part by the National Science Foundation under grant number CER-8710737.

by the sporadic arrival of input data. A dataflow model of real-time systems where input data arrive only at fixed rates has been studied in [19]. Timing requirements affecting the safety of software systems must be specified before they can be verified. This proposal addresses the problem of specifying timing requirements in a way that allows the generation of an executable prototype. The verification of safety assertions involving timing constraints by means of RTL (real-time logic) is discussed in [8].

Currently software in high level languages such as Ada is developed mostly by manual techniques. Significant gains in software quality and reliability can be obtained by automating the more labor-intensive parts of the process and using a computer-aided prototyping system (CAPS) based on PSDL [11, 16]. The CAPS architecture is shown in Fig. 2.

The user interface makes it convenient for the system designer to express the behavior of the intended system in PSDL. The purpose of the rewrite subsystem [12], the software design management system, and the software base is to support the retrieve of reusable software components based on their PSDL specifications. This part of the system does a limited amount of bottom-up design, and allows the designer to work at a high level without the need for full knowledge of the set of available software components [11].

The execution support system is the heart of CAPS, since it enables the execution of PSDL prototype descriptions. The implementation of PSDL is beyond conventional compiler technology because of the difficulty of implementing the aspects of PSDL associated with hard real-time constraints. Conventional compilers take procedural statements and transform them into code for sequential execution, without regard for timing or performance constraints. PSDL merges data flow and control flow in a computational model with inherent parallelism, and couples this model with timing constraints and non-procedural control constraints. None of the conventional translation technologies possiblely handle all of these features simultaneously, and few target languages support strict guarantees for meeting real-time constraints.

It is also difficult to have direct implementation of real-time constraints in Ada. The Ada implementation of real-time constraints as the PERIOD, MET (Maximum Execution Time), MCP (Minimum Calling Period), and MRT (Maximum Response Time) of PSDL is not trivial. The Ada DELAY and SELECT constructs cannot be used to implement these performance constraints directly for a system of operators. Ada DELAY by itself gives just a lower bound on the delay implied, without any guaranteed upper bound. The use of the type DURATION allows the approximation of an interval in a loop construct but it does not provide a guarantee of service. The use of TASKS in Ada provides more capability through the use of conditional entry calls. The problem with these constructs is that they require a good deal of effort on the part of the programmer, and the program is operating at the mercy of the Ada run-time system.

The degree of effort required to implement these constructs directly in Ada is out of proportion with the aims of the rapid prototyping methodology. A more abstract and direct syntax is required to specify hard, real-time constraints which will make construction and demonstration of prototypes possible. If the designer is required to invest nearly as much effort into the creation of the prototype as the development of the system itself, there is no advantage to prototyping. Furthermore, the Ada run-time system will not guarantee that the prototype design behaves exactly as specified.

The PSDL execution support system contains a translator, static scheduler, and a dynamic scheduler. The parts of the execution support system are shown in Fig. 3.

The purpose of the static and dynamic schedulers is to ensure that the prototype functions within the real-time constraints applied to the design. Barring errors in design, the feasibility of such aspects of the system as control flow, order of firing of program modules, time behavior, and I/O formats can be demonstrated with CAPS. The execution support system frees the designer from the implementation effort required in Ada by automatically generating executable code in Ada, and by automatically generating control code in the form of Static and Dynamic schedules which enforce control and timing behavior. Therefore, PSDL supports development of large and embedded Ada programs directly and easily.

The translator generates code binding together the reusable components extracted from the software base. Its main functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints. The dynamic scheduler together with the debugging system (see Fig. 2) also allows the designer to control and examine the execution of the prototype.

This paper describes a single processor implementation of these components. The translator, static scheduler, and dynamic scheduler are described in sections 2, 3, and 4 respectively, while section 5 presents our conclusions.

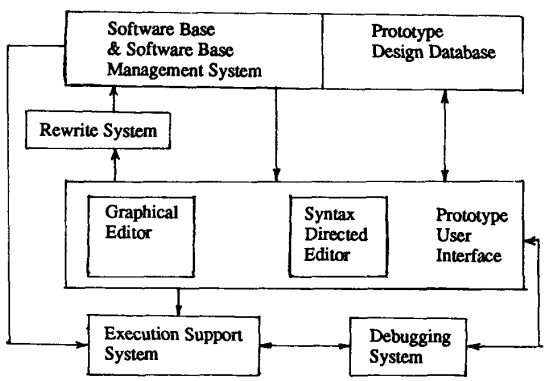


Fig. 2 CAPS Architecture

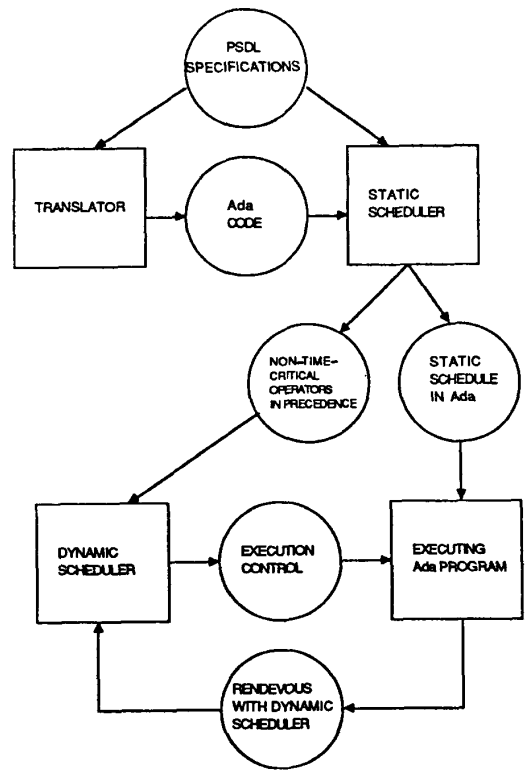


Fig. 3 Execution Support System

2. Translator

The Translator's primary responsibility is transforming part of the PSDL prototype source program into an executable Ada program that simulates the behavior of the prototype. The Translator has been constructed via the Kodiyak translator generator [7]. This translator was generated from the productions of the PSDL grammar with their associated attribute definition equations that represent the corresponding Ada program structures. These equations define the semantics of the translation using the structure of the PSDL grammar. Augmentation code for PSDL operators is embedded within the attribute definition equations. These augmentations implement PSDL data streams, PSDL operator conditional constraints and PSDL TIMER functions [14]. Fig. 4 illustrates the process used to generate the translator.

Once the executable translator is generated, it can be given any source program in PSDL and will output a source program in Ada. The translator scans an input file written in PSDL, parses it, locates syntax errors, and if no errors are present, produces an Ada translation in an output text file.

The output of the translator is just part of the implementation of a PSDL prototype. PSDL operators can be either atomic or composite. During the early prototype design phase, PSDL composite operators are decomposed into atomic operator networks. Atomic operators are realized by reusable modules from the CAPS software database, in the form of Ada program units. The code generated by the translator serves to adapt and connect the atomic operators realizing each composite operator. The complete Ada source program consists of the combination of the reusable modules and the augmentation code produced by the translator. This text file can be compiled, linked, and exported to the operating system for execution.

2.1. Mapping between PSDL and Ada Constructs

This section describes the mapping between PSDL and Ada used by the translator. This mapping gives the syntactic and semantic correspondence between the two languages.

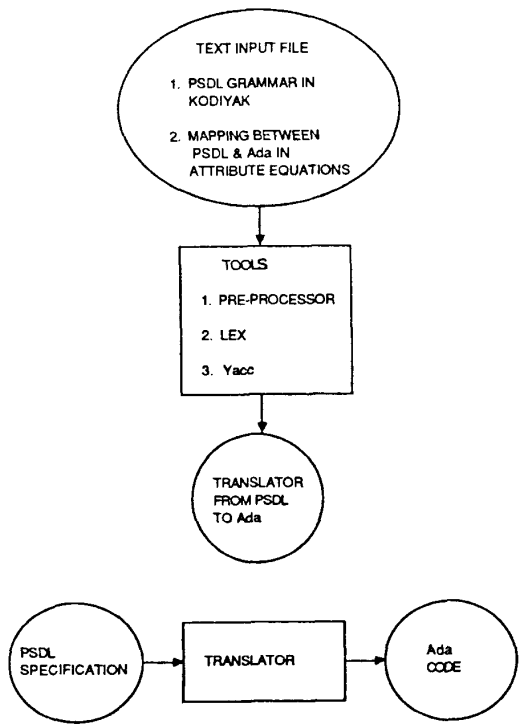


Fig. 4 Translator Generation Process

2.1.1. PSDL Operators

An **operator** is either a function or a state machine. When an operator fires, it reads one data object from each of its input streams, and writes at most one data object on each of its output streams. The output objects produced when a function fires depend only on the current set of input values. The output values produced when a state machine fires depend on the current set of input values and the current values of a finite number of internal state variables.

In the simplest case, we assume that all PSDL operators are implemented by Ada procedures. These procedures contain code to implement input and output to PSDL data streams, PSDL triggering conditions, and PSDL conditional output statements.

2.1.2. PSDL Data Streams

A **data stream** is a communication link connecting one or more producer operators to a consumer operator. Each stream carries a sequence of data values. Communication links with more than two ends are realized using copy operators.

There are two types of data streams - **dataflow** streams and **sampled** streams. A dataflow stream guarantees that none of the data values are lost or replicated, while a sampled stream guarantees the most recently generated data value is always available. Dataflow streams are used to connect operators that must coordinate corresponding input values from different producers. Sampled streams are used to connect operators that fire at incompatible frequencies.

A PSDL stream is mapped into a buffer capable of holding one data value. Since a buffer may be read by an operator executing independently of the operator writing into the buffer, it must be protected from data conflicts due to concurrent access. Consequently buffers are embedded in Ada tasks and read or written via task entries, to provide mutually exclusive access. Buffer manager tasks are declared inside generic packages to make it easy for the translator to create a separate buffer manager task for each PSDL data stream. Thus each PSDL data stream is implemented by an instance of a generic package.

Two kinds of buffers are needed, corresponding to the two kinds of data streams in PSDL. Sampled buffers are used to implement sampled streams and FIFO buffers are used to implement dataflow streams. The difference between the two kinds of buffers is that a FIFO buffer makes sure that every value written into the buffer is read exactly once before the next value is written into the buffer. Violations of this constraint are reported via Ada exception conditions. There are two possible exceptions: Underflow and Overflow. Underflow is raised if the consumer operator attempts to read the buffer before it has been updated by the producer operator. Overflow is raised if the producer attempts to write to the buffer before the consumer has read the previous data value. There are no constraints on the order a sampled buffer is accessed, and no associated exception conditions.

The translator must select the appropriate type for buffer for a given data stream according to the triggering conditions of the consumer operator associated with the stream. There are two types of **data triggers** for PSDL operators.

OPERATOR p TRIGGERED BY ALL x, y, z

OPERATOR q TRIGGERED BY SOME a, b

In the first example the operator p is ready to fire whenever new data values have arrived on all three of the input arcs x, y, and z. In the second example, the operator q fires when any of the inputs a and b gets a new value. If q has some other input c, the output of q can be based on old values of c, since q will not be triggered on a new value of c until after a new value for a or b arrives.

If an operator mentions an input data stream in a TRIGGERED BY ALL condition then the translator will use a FIFO buffer to realize the stream, and otherwise it will use a sampled buffer. The translator realizes each sampled buffer as an instance of the generic package "sampled_buffer" and each FIFO buffer as an instance of the generic pack-

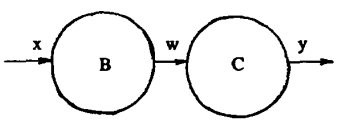
age "fifo_buffer". These generic packages are standard reusable components contained in the software base for PSDL.

The translator also generates the definition of a procedure for initializing the buffers realizing data streams whose initial values have been declared in PSDL. This is done by using the write operation provided by each buffer task. The procedure body contains one such statement for each data stream with a declared initial value, and can be empty if no initial values are declared.

2.1.3. An Example of the Mapping between PSDL and Ada

This section gives an example of the transformation performed by the Translator. Consider the following PSDL description of the composite operator A.

```
OPERATOR A
 SPECIFICATION
  INPUT x: integer
  OUTPUT y: integer
 IMPLEMENTATION
  GRAPH
```




```
  OPERATOR B
   TRIGGERED IF x > 0
   PERIOD 100 ms
END
```

The Ada code generated by the Translator for the operator B is used in the implementation of A is shown below.

```
x_buffer is new sampled_buffer(integer);  -- instance of generic package

procedure b_driver is
  x, w: integer;
begin
  if x_buffer.new_data then  -- new data in the buffer
    x_buffer.read(x);
    if x > 0 then  -- triggering condition is true
          b(x, w);
          w_buffer.write(w);
    end if;
  end if;
end b_driver;
```

This example assumes the procedure b is a reusable component implementing the PSDL operator B. The procedure b_driver contains the augmentation code generated by the Translator and is called from the "static_schedule" task every 100 milliseconds.

2.2. Attribute Equations

The Kodiyak translator generator requires an attribute grammar defining the translation. The attribute grammar consists of BNF rules for PSDL where each rule is associated with a set of equations defining the attribute values for the symbols appearing in the rule. The attribute equations map PSDL constructs to the constructs of the target language Ada according to the mappings described in the previous section. Fig. 5 illustrates some of the attribute equations used by the Kodiyak translator generator to produce the PSDL to Ada translator.

3. Static Scheduler

The Static Scheduler addresses only those operators with critical timing constraints whose precise performance determines whether the system as designed will meet the required timing specifications. The operators that do not have real time constraints are handled by the dynamic scheduler, as explained in the next section. The primary purpose of the Static Scheduler is creation of a static schedule which gives the precise execution order and timing of operators with hard real-time constraints, in such a manner that all timing constraints are guaranteed to be met [13], provided that such a schedule is possible for the given system specifications. The static schedule contains the pre-allocated starting time and execution time for each critical

```
time
   :NUMBER unit
    {time.trn = [NUMBER.%text, unit.trn]; }
   ;

unit
   :MICROSEC
    {unit.trn = ""; }
   |MS
    {unit.trn = "000"; }
   ;
```

Figure 5. Sample Attribute Equations for the PSDL Translator

operator. The construction and validation of a static schedule is a labor intensive process that cannot be carried out both rapidly and reliably without the benefit of automated software tools.

The initial input to the Static Scheduler is a text file containing the PSDL prototype program created jointly by the designer and user. An intermediate output to the Dynamic Scheduler is an Ada source file containing the non-time-critical operators in the PSDL program. The final output of the Static Scheduler to the compiler/linker/exporter (CLE) is an Ada source file containing the static schedule. The CLE compiles and links this program together with the compiled program produced by the Translator. This combined program is the executable Ada program used to demonstrate the prototype's performance. The data flow diagram shown in Fig. 6 illustrates the conceptual design of the Static Scheduler and outlines its five major functions [9, 10, 20].

The Read_PSDL function extracts operator identifiers, timing information, and link statements describing the data streams from the PSDL source file. The Pre-Process File function classifies the resulting information into three different categories and checks the following consistency and completeness properties.

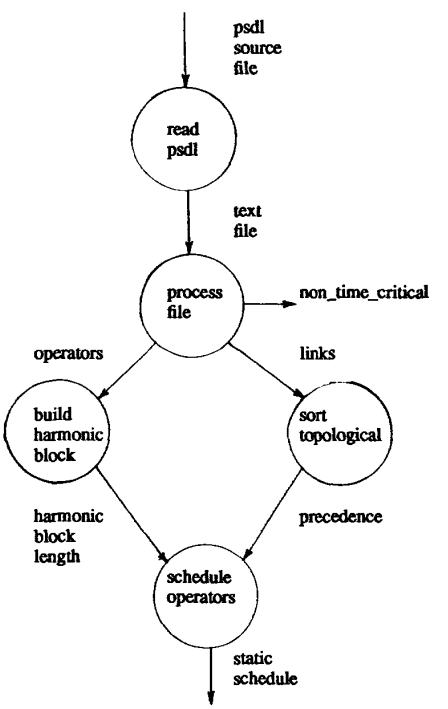


Fig. 6 DFD for the Static Scheduler

(1) Each sporadic operator with timing constraints must have values for MET, MRT, and MCP with MET <= MRT

(2) Each periodic operator must have MET <= PERIOD.

To enable the execution of all operators in a predictable manner that can be described by a static schedule, sporadic operators are implemented by their calculated periodic equivalents [18].

The calculation of an equivalent period requires that all sporadic operators have values for MCP, MRT, and MET satisfying the following relationships.

1. MET <= MRT
2. MET <= MCP

The first condition ensures (MRT - MET) is positive, while the second allows a single processor implementation. The equivalent period is given by

P = minimum(MCP, MRT - MET).

A single processor implementation also requires (MET <= P) to allow the operator to complete within the calculated period.

The function Sort_Topological sorts the link statements into an order consistent with the dataflow relationships, so that the producer of each stream appears before its consumers. The link statements form a directed acyclic graph because every cycle in the data flow graph must be broken by a link with a declared initial value, and such links are removed before the topological sort is performed.

The Build_Harmonic_Blocks function partitions the set of time-critical operators into non-overlapping HARMONIC BLOCKS, based on their periods or equivalent periods. A harmonic block is a set of operators with the following properties:

(1) The periods of all of the operators in the set are exact multiples of the BASE PERIOD.

(2) One of the operators in the set has a period equal to the BASE PERIOD.

Harmonic blocks are made disjoint by placing each operator that satisfies the constraints for more than one harmonic block into the block with the longest possible base period, since to help ease schedule congestion. Each harmonic block can be treated as an independent scheduling problem since operators with different periods must be connected by sampled data streams. We assume a separate processor will be used for each harmonic block. This assumption is reasonable because any schedule containing two operators with relatively prime periods is guaranteed to have periodic tight spots due to the beats between the two frequencies, leading to low utilization of the scheduled processor. In single processor implementation all operators are place into the same harmonic block by relaxing the first of the two properties given above.

The static schedule is constructed using algorithms similar to those of [17] by the "Schedule_Operators" function. This function uses the "Precedence_List" and "Harmonic_Block" files, which are generated by the "Sort_Topological" and "Build_harmonic_Blocks" functions, respectively. The Precedence_List file defines the required execution order, while the Harmonic_Block file defines the set of operators to be scheduled and the length of the static schedule. The resulting static schedule is a linear table giving the exact execution start time for each time-critical operator and the reserved maximum execution time (MET) within which the operator must complete execution.

The algorithm used in this implementation is a two step process. The first step allocates the initial execution interval for each operator [18] in the order given by the Precedence_List, using the relation

interval = (current_time, current_time + MET)

where the current_time is the beginning of the currently unallocated time interval in the block period. This step also creates a firing interval for each operator, during which the second step must schedule the next execution of the operator. The firing interval gives the lower and upper bound for the next possible starting time of the operator. For example, OP_2 in Fig. 7 is scheduled to start execution at time 2 and to complete by time 3, based on its MET of 1. Since OP_2 has a period of 10, it cannot fire again before time 12, the lower bound for its firing interval. OP_2 must fire by time 21, the upper bound, in order to ensure completion by the end of the second period at time 22.

Assume given

PRECEDENCE_LISTS (op_1, op_2, op_3, op_4)
HARMONIC_BLOCK_LENGTH = 20

and

OPERATOR_ID	MET	PERIOD
OP_1	2	10
OP_2	1	10
OP_3	3	20
OP_4	1	10

result

STATIC SCHEDULE

OPERATOR_ID	TIME START	END	FIRING	
OP_1	0	2	(10,18)	H11
OP_2	2	3	(12,21)	
OP_3	3	6	(23,40)	
OP_4	6	7	(16,25)	

and

HARMONIC BLOCK

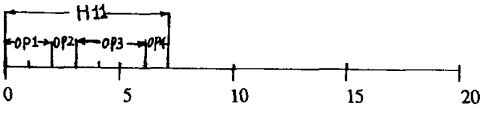


Fig. 7 Static Schedule after First Process

STATIC SCHEDULE

OPERATOR_ID	TIME START	END	FIRING		
OP_1	0	2	(10,18)	H11	H1
OP_2	2	3	(12,21)		
OP_3	3	6	(23,40)		
OP_4	6	7	(16,25)		
OP_1	10	12	(20,28)	H12	
OP_2	12	13	(22,31)		
OP_4	16	17	(26,35)		
OP_1	20	22	(30,38)	H21	H2
OP_2	22	23	(32,41)		
OP_3	23	26	(43,60)		
OP_4	26	27	(36,45)		
OP_1	30	32	(40,48)	H22	
OP_2	32	33	(42,51)		
OP_4	36	37	(46,55)		

and

HARMONIC BLOCK

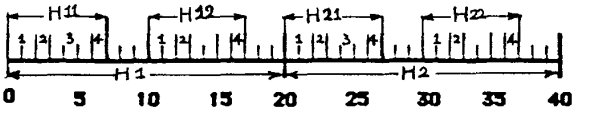


Fig. 8 Static Schedule for 2 Block Periods

The second step schedules subsequent firings of the operators, which are allocated in earliest-lower-bound-first order. In the example of Fig. 7 the operators are scheduled in the order [OP_1, OP_2, OP_4] during the first iteration of the second step. Since the period of OP_3 (20) is the same as the block period, it is scheduled to fire only once in the static schedule. As each operator is scheduled, this process verifies that both

1. current_time + MET <= block period, and
2. current_time <= upper bound.

Failure to meet either condition results in an infeasible schedule, resulting in an exception and an appropriate error message at the UI. These checks ensure that the process produces a feasible schedule whenever it terminates without an exception.

The process also calculates new firing intervals for each process scheduled. Fig. 8 shows the static schedule after three iterations of the process, along with a timing chart for two block periods.

In the example shown, the static schedule is complete after only one iteration of the second step, since at that point the lower bounds of all the firing intervals are greater than or equal to the block period.

4. Dynamic Scheduler

The dynamic scheduler is a run-time executive with three main purposes: to schedule operators that are not time critical, to provide debugging facilities, and to gather statistics about the run-time characteristics of the prototype. In the case of a distributed implementation, there is an instance of the dynamic scheduler running on each processor.

PSDL assumes that time constraints are absolute if they are given. This requires the static scheduler to allocate processor time based on worst case execution times and firing frequencies. This policy results in plenty of spare processor time on the average, because worst case loads tend to be rare. The dynamic scheduler uses a simple strategy to utilize this spare capacity for operations that are not time critical.

During each base period the dynamic scheduler invokes the time critical operators in the order in which they are scheduled. When it runs out of things to do, it checks to see if it has any time left, and if so it picks a non time-critical operator to execute. A simple round robin scheduling algorithm is used. Just before the end of the base period, the currently running operator is interrupted and the resumption point for the operator is saved. The interrupt is given sufficiently long before the end of the base period so that the currently running operator will have enough time to get out of any critical sections it may have entered. The only critical sections in the system are in the buffering primitives for reading values from data streams and writing values into data streams. These critical sections are short, and have fixed upper bounds on their execution times.

The debugging facilities are fairly conventional. Breakpoints can be attached to operators, and can be conditional with respect to a PSDL predicate. Selected inputs or outputs of an operator can be traced, resulting in a display of the values and their associated arrival or departure times. Commands for inserting and deleting values in data streams are provided. The facilities for gathering statistics include commands for monitoring both frequencies and timing information. Frequency statistics include the number of values that pass down a data stream, the number of times an operation fires, the number of times an exception occurs, etc. Timing statistics include minimum, average, standard deviation, and maximum times for the execution, response, or interval between firing of an operator. These statistics are intended primarily for feasibility and performance studies.

The dynamic scheduler has two parts, a pre-processor and a run-time executive. The purpose of the pre-processor is to generate a dynamic schedule, while the purpose of the run-time executive is to start and control the execution of the prototype.

The pre-processor must determine an order for invoking the non time-critical operators that is consistent with the implicit scheduling constraints of PSDL [2]. It is convenient to represent the dynamic schedule as an Ada task that invokes the non time-critical operators. The required order of invocation is embedded in the code of the task representing the dynamic schedule, which will be referred to as the dynamic schedule task.

The run-time executive part of the dynamic scheduler must initialize the system before it can allow any operators to fire. The initialization is performed by calling a procedure generated by the translator for that purpose. This procedure initializes all of the buffers corresponding to data streams with declared initial values.

It is also convenient to represent the static schedule as an Ada task, which will be referred to as the static schedule task. This task is responsible for invoking the time-critical operators at times determined by the static scheduler. Both the dynamic schedule task and the static schedule task start with an accept statement for an "initialization_complete" entry. This entry in each schedule task is called by the run-time executive after the initialization procedure terminates, to ensure that none of the operators get fired before initialization is complete. After the initial accept statement, both tasks enter an endless loop which invokes the operators under their control in the predetermined order.

The other responsibility of the run-time executive is to handle exceptions from the prototype and interrupts from the user. Some exceptions are used by the PSDL execution support system to report run-time errors such as buffer overflows or underflows. The current version of the execution support system prints a message and terminates the execution of the prototype when it encounters an unhandled exception or an interrupt from the user. Future versions will provide facilities for debugging and collecting statistics.

The important function of coordinating the static and dynamic schedules is provided implicitly by the built-in scheduling mechanism of Ada and careful assignment of task priorities. This is accomplished by assigning a low priority to the dynamic schedule task and a medium priority to the static schedule task. The static schedule task is responsible for executing a delay statement of the appropriate duration whenever a time-critical operator terminates and the static schedule requires a wait before the next time-critical operator can be invoked. Operators invoked by the dynamic schedule task can run only when the static schedule task is blocked at a delay statement. When the delay time is up, the execution of any active non time-critical operator must be suspended, because it will have the same low priority as the dynamic schedule task, and the scheduled time-critical operator gets to fire.

Such interruptions of the non time-critical operators cannot cause any inconsistencies, because all direct data references from the operator must be local in PSDL. However, care must be taken not to interrupt an operator in the middle of a read or write operation on a buffer implementing a data stream. This can be accomplished by encapsulating all of the buffers in tasks with a higher priority than either of the schedule tasks. The delay statements in the static schedule must also have durations that are a little shorter than the unused time slots in the static schedule, to give any active buffer operations time to complete before the static schedule must invoke the next time-critical operator.

The structure defined above can be readily extended from a single processor implementation to a multiple processor implementation by creating one static schedule task and one dynamic schedule task for each processor in the system. The implementation strategy outlined above will work correctly only if the number of medium priority tasks is less than or equal to the number of processors. The scheme also requires all of the processors in the system to be dedicated to prototype execution, and all of the clocks associated with different processors to be accurately synchronized.

5. Conclusions

The Computer Aided Prototyping System (CAPS) was introduced as a software engineering tool that is currently being designed. This tool will enable software designers to exploit rapid prototyping to its fullest by automating the construction of executable prototypes. The execution support system is the component within the CAPS that makes the prototype, written in the Prototype System Description Language (PSDL), executable. The major contribution of CAPS to the advancement of software engineering technology lies in the fact that the executable prototypes can be automatically generated by the use of specifications and reusable software components.

It is feasible to describe a prototype in PSDL and to use an automated facility to translate the prototype into Ada. The present translator lays a sound foundation for further development. It implements and recognizes the full syntax of PSDL as published in [11]. The fundamental conceptual design implementation of the major PSDL syntactical constructs has been completed and documented. The translator produces rudimentary Ada code for interconnection of reusable software program modules. Several additional research possibilities exist. First, the current translator is an empirical demonstration of the capability. Therefore, it should not be expected to function properly in all cases. Work must be undertaken to establish a

rigorous, formal definition of the relationship between the syntax/semantics of PSDL and the syntax/semantics of Ada. Once such a rigorous definition has been produced, it must be applied to the translator to produce a facility which works for general cases. Second is the issue of code optimization. Some programs may require optimization for speed of execution, while others require optimization for code size. The incorporation of such optimizations in the translator should be explored.

As it is currently designed, the static scheduler performs some validity checks on the timing information that is provided by the system designer and notifies the designer if any information is invalid. Execution of the prototype cannot continue without the designer altering the timing information as necessary and running the program again. It may be possible for the static scheduler not only to identify the problem, but also to correct it. The scheduler would have to pick a feasible value for whatever attribute is in question based on worst case criteria. The designer would still have to be notified of the situation; the difference is that execution would not be suspended.

The prototype design presented in this paper has assumed that all timing constraints for an operator have been supplied by the designer. A more sophisticated design could handle those instances where some required information is missing. Again, the static scheduler could assign a value based on some worst case criterion.

Tests could be done to indicate the feasibility of constructing a valid schedule once all operators had periods and were assigned to harmonic blocks. As with the simple validity checks, in the event it is determined that a valid static schedule in not feasible, program execution is discontinued. It is also possible in this situation to modify some timing constraints for the purpose of constructing the schedule rather than requiring the system designer to input all corrections. An exception would still be raised to notify the designer of the problem and what actions were taken to correct it. Only if attempts to modify timing information prove too difficult should the program be allowed to cease execution prior to completion.

The algorithm for scheduling operators within harmonic blocks is primarily for use in a single processor environment. It should only require slight adjustments to this algorithm to make it suitable for use with multiprocessor systems. One of the adjustments that is necessary is in the algorithm for scheduling the first operator in each harmonic block. Even though each harmonic block is a separate scheduling problem, there will be precedence relationships between some of the operators in separate blocks. For this reason, the first operator in every harmonic block will not necessarily be able to be scheduled to start at time t = 0. The algorithm needs to incorporate this possible situation.

Future enhancements identified in addition to the current Dynamic Scheduler design would provide debugging capabilities and statistical information. During execution of the prototype, the debugging capabilities would trace relevant information concerning operator execution. Computed values and their associated input and output times would display a record of events that occur during execution. Statistical information collected during execution would include frequency of operator firing, quantity of EXCEPTIONs occurring, and statistical data on timing parameters for critical operators [13]. When combined, these two enhancements would provide the designer and the user with precise information for measuring, analyzing and validating the prototype's performance. Recognizing the increased cost and importance of software development for Command and Control systems, the Secretary of the Navy (SECNAV) promulgated a new instruction addressing software development and acquisition [5]. This instruction documents SECNAV concern for defining a DON acquisition policy for software-intensive systems and increasing user involvement during the design and development stages. The policy combining these two concerns states: "To promote effective interaction between the user and the developer, software prototyping methods shall be used in the design and construction of C2 information systems. Early delivery of software systems is emphasized through the use of prototyping methods [5]". The instruction defines software prototypes identical to that used throughout this paper as "Software which stimulates the important interfaces and performs the main functions of the intended system while not being bound by the same hardware, speed, size or cost constraints. It may serve to demonstrate or provide a subset of the functions that would be required of software to meet a related, fully-validated requirement" [5]. Computer-aided rapid prototyping specifically addresses the concerns of the SECNAV. In particular, the CAPS stresses interaction between the software designer and the user early in the design and development stages. This allows validation of the prototype's ability to simulate the critical interfaces and functions of the envisioned system. The author agrees that the increased cost and complexity of developing software warrants a revised approach to the software acquisition procedures.

There are many aspects of software requirements which can be most effectively validated by user inspection of a running prototype, such as the appropriateness of a given user interface, or the correct description of an existing hardware interface. Executing prototypes of the novel or difficult parts of a complicated system can significantly increase the confidence that the system can in fact be built, before significant resources have been committed to the development effort. Cost estimates can be improved by using a prototype, since the cost of designing the intended system is usually proportional to the cost of the rapid prototype. Performance bottlenecks can be found during the execution of the prototype by collecting statistics on module execution frequencies.

Our initial investigation leads us to conclude that an execution support system for PSDL is feasible, and that such a software tool is currently the most practical way to support rapid prototyping for real-time systems. This together with the features of PSDL for large scale software design makes PSDL a good candidate for inclusion in an advanced Ada programming environment. At the current point in time, we have a conceptual design for the PSDL execution support system, and the implementation of the PSDL translator is under way.

The PSDL language, its associated methodology, and programming environment apply well to the design of Ada software systems. The demand for large scale Ada software systems is increasing dramatically. Real time systems have particularly strict requirements on accuracy and precision. A rapid prototyping environment for creating and modifying an executable prototype is needed. The design of PSDL, its prototyping methodology, and the use of reusable components from a software base make highly automated software tools practical. An experienced PSDL user should be able to rapidly construct a prototype significantly faster than an experienced Ada-user.

The use of PSDL for prototype construction should be much easier and simpler than the direct use of Ada. PSDL has selected and transformed all the good language features of Ada primitive constructs into a small and a simple set of PSDL language constructs which is convenient for the designer. It is simpler to describe the structure of a system and the relation between system components in PSDL than in Ada since PSDL allows a designer express his thoughts at a specification or a design level. The abstractions of PSDL are tailored to describing real-time systems, and allow the designer to express his thoughts clearly and quickly by eliminating many lower level details from his consideration. The computational model of PSDL forces all interactions between models to be explicit. All state variables are local to some component, thus confining the effects of state changes. This helps designer understanding by eliminating hidden interactions on the large scale, while allowing the efficiencies of imperative programming inside individual components. The important points are that the software tools and the prototyping methodology of PSDL lead to a well structured prototype and that the resulting PSDL prototype is executable. PSDL components can be mapped into Ada directly. Ada is a large and powerful programming language. It is a good underlying programming or an implementation language for PSDL. However, it is too hard and too cumbersome to use as a design language. The mapping between PSDL and Ada and the use of the reusable Ada components are the keys to making PSDL prototypes executable and useful in large Ada projects.

1. V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development", *Comm. of the ACM 29*, 5 (May 1986), 402-415.
2. V. Berzins and Luqi, "Semantics of a Real-Time Language", in *submitted to the the Real-Time Systems Symposium*, 1988.
3. V. Berzins, "The Design of Software Interfaces in Spec", in *to appear in Proceedings of the International Conference on Computer Languages*, Miami, Oct. 1988.
4. B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them", in *Proceedings of the Real-Time Systems Symposium*, IEEE, New Orleans, LA, Dec. 1982, 197-204.
5. "Acquisition of Software-Intensive C2 Information Systems", in *SECNAV INSTRUCTION 5200.37*, Department of the Navy, Jan 5, 1988.

6. S. Faulk and D. Parnas, "On Synchronization in Hard-Real-Time Systems", *Comm. of the ACM 31*, 3 (Mar. 1988), 274-187.

7. R. Herndon, "The Incomplete AG User's Guide and Reference Manual", 85-37, University of Minnesota, October, 1985.

8. F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Trans. on Software Eng. SE-12*, 9 (Sep. 1986), 890-904.

9. D. Janson, "A static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

10. D. Janson and Luqi, "A Static Scheduler for the Computer Aided Prototyping System", in *to appear in Proceedings of COMPASS 88*, Gaithersburg, MD, June 1988.

11. Luqi, "Rapid Prototyping for Large Software System Design", Ph. D. Thesis, University of Minnesota, 1986.

12. Luqi, *Normalized Specifications for Identifying Reusable Software*, Proc. of the ACM-IEEE 1987 Fall Joint Computer Conference, Dallas, Texas, October 1987.

13. Luqi, "Execution of Real-Time Prototypes", in *ACM First International Workshop on Computer-Aided Software Engineering*, vol. 2 , ACM, Cambridge, Massachusetts, May 1987, 870-884. Technical Report NPS 52-87-012.

14. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988.

15. Luqi and V. Berzins, "Rapid Prototyping of Real-Time Systems", *IEEE Software*, Sep. 1988.

16. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software*, March 1988.

17. A. K. Mok, *The Design of Real-Time Programming Systems Based on Process Models*, IEEE, 1984.

18. A. K. Mok, "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Proc. of the 1984 Real Time Systems Symposium*, Dec. 1984, 125-133.

19. A. Mok and S. Sutanthavibul, "Modeling and Scheduling of Dataflow Real-Time Systems", in *Proceedings of the Real-Time Systems Symposium*, IEEE, San Diego, CA, Dec. 1985, 178-187.

20. J. O'Hern, "A Conceptual Design of a Static Scheduler for Hard Real-Time Systems", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

Session 4
Language and Semantics
Chair: F. Schneider