Real-Time Action*

Padmanabhan Krishnan
Department of Computer Science
Ny Munkegade 540
Aarhus University
DK 8000, Aarhus C Denmark
E-mail: paddy@daimi.aau.dk

April 1991

Abstract

The behavior of a real-time system depends on the scheduler used. The order in which tasks are executed depends on its characteristics such as ready time, deadline etc. We describe a language in which the readiness and deadlines can be specified. A scheduling policy using the task characteristics can be defined. To study the effect of schedulers on a system a notation should allow for the specification of time, processes and scheduling. In this paper we show the applicability of the Action Notation for specifying real-time behavior.

1 Introduction

The importance of scheduling in real-time need not be reiterated. As pointed out in [4], a real-time system is one which, given limited resources, has performance requirements. The general problem of scheduling can be stated

*To appear in the Euromicro 1991 Workshop on Real-Time Systems, June 12-14, France.

as "how best to utilize the resources in order to meet the performance requirements." Time is one of the primary resources which has been studied extensively. Scheduling algorithms are techniques for managing the use of time given timing constraints such as deadlines. Therefore, the choice of scheduling algorithm could affect the timing behavior of the real-time system significantly. Hence when the behavior of a real-time system is described, it is necessary to specify the scheduler under which the behavior was observed.

Therefore, the programmer must have some control over the type of scheduler used in implementing the system being developed. This aspect has been largely ignored by programming language designers and the scheduling being left completely to the implementation. For example, the language Ada [1] only permits the specification of tasks and leaves the scheduling to the implementation. While the priority inheritance algorithm has been suggested in [3], it is neither mandated by the language nor is it under the control of the programmer. This leads to the situation where the programmer is completely at the mercy of the compiler and the programs developed may exhibit vastly different behavior on different systems thus affecting portability. Mandating a scheduler for a particular language will only restrict the types of systems that can be programmed in it. If the real-time system developer is allowed to specify a scheduler, the language need not mandate a scheduling policy nor will program behavior vary across implementations due to differences in the choice of scheduling policy. In [11] the need for 'schedulability check' for real-time systems is stated, which is ensuring that the constraints specified in the program can be satisfied by a scheduler. For this goal to be satisfied, the specification of the scheduler to be used must be allowed. This is similar to specifying the types of objects/operations. In this paper we outline a simple language which permits the specification of schedulers.

Given that scheduling affects the behavior of a program, it is important to have a unified framework, in which the effect of a program and the effect of a scheduler on it, can be defined. To define the effect of a real-time program requires a notion of time. In [5], the programmer is allowed to define a clock and the execution of the clock acts as the time-keeper for the system. We adopt a similar approach and define the ticking of time as a construct in our formalism. In this paper, we show that the Action Notation [9, 10] is a framework in which a scheduler, time and their effects on the behavior of a program can be described.

This paper outlines a simple language in which one can define tasks which are to be controlled by certain types of schedulers and its semantics in the Action Notation. The paper is organized as follows. In section 2 we describe a syntax to describe tasks with deadlines and readiness and the scheduler to be used. In section 3 we present a brief outline of the Action Notation and in section 4 the semantics for these constructs is explained. Appendices A and B contain the formal details. In appendix C the effect of the semantics on behavior is explained via two examples.

2 Model and Language

Traditionally, real-time systems have been assumed to consist of a set of tasks. Each of these tasks have an associated deadline. Tasks can either be repetitive or non-repetitive. Repetitive tasks can be periodic or sporadic. Periodic tasks in the system have an associated periodicity, while sporadic tasks have a particular arrival characteristic with a least bound on periodicity.

As we are interested in a general model, we assume that every task has an associated deadline which indicates the time by which a task must terminate and an associated readiness which indicates when the task is ready to execute. Both the deadline and the readiness are represented by general expressions thus enabling the specification of a wide variety of timed behaviors. A periodic task can be defined by specifying that is ready at the beginning of its period and the deadline is the end of period (or earlier). Similarly, a task whose readiness is a random variable can simulate a sporadic task.

Tasks can also be classified in two categories 1) Non Preemptable and 2) Preemptable. Tasks which one scheduled are to run to completion are non preemptable tasks while tasks which can be interrupted to schedule other tasks are preemptable. In preemptable tasks it is essential to know the granularity of execution. As we are describing schedulers, it is necessary to identify schedulable points, i.e., where schedulers can be invoked.

There have been a number of schedulers for systems of this nature, e.g. deadline-first, rate-monotonic, minimum laxity, FCFS to name a few. See [2] for details. In this paper we do not discuss the relative merits of the various schedulers. Our primary task to allow the programmer to specify a scheduler and not to use a fixed scheduler. As mentioned earlier, the language to

describe a scheduler will involve deadlines and readiness, i.e., schedulers in our language define an ordering on the execution based on the deadlines and periodicity of the process.

We assume some well defined language for defining the building blocks (Basic-Unit) of tasks and the expressions (Expression, Operator) that appear in the following abstract syntax for the language.

comment: Grammar

Basic-Unit = □

Expression = □

Operator = □

Body = ⟦Basic-Unit "->" Body⟧ | Basic-Unit

Task = ⟦"*"? Body "Within" I-Expression "Readyat" I-Expression⟧

I-Expression = ⟦"I"⟧ | Expression | ⟦I-Expression Operator I-Expression⟧

Feature = ⟦"D"⟧ | ⟦"R"⟧ | ⟦"T"⟧ | ⟦"I"⟧ | Expression

Evaluation = Feature | ⟦Evaluation Operator Evaluation⟧

Sched = Evaluation

Process = Task | ⟦Process ";" Process⟧

System = ⟦Process ":" Sched ⟧

A Basic-Unit represents the basic schedulable unit, i.e., a unit which cannot be preempted. Body is a sequence of basic-units defining the executable part of the task. A Task consists of the executable part, and the deadline/ready time. The expressions following Within and Readyat represents the deadline and ready time respectively. A "*" indicates iteration of the body with the ready-time/deadline computed before every iteration. The scheduler is defined as an expression over "D", "I", "R", "T" and other other

expressions. "D" and "R" extract the deadline and readiness respectively, while "T" refers to the current time and "I" in both I-expression and Feature refers to the iterative count of the task, which is initialized to 1 before the start of execution. The execution of the scheduler is as follows. The evaluation function is applied to *all tasks* in the system. The task which yields the lowest value is selected for execution. The evaluation of a task need not yield an integer value as a scheduler can be composed of general expressions. The specifier has to be provide a comparison function for the type of value used by the scheduler.

Example: 1

$[\![$ *"D"* $]\!]$ *represents earliest deadline first,*

$[\![[\![$ *"D"* $]\!] - [\![$ *"R"* $]\!]]\!]$ *represents minimum laxity first with laxity based on deadline and readiness*

$[\![[\![$ *"D"* $]\!] - [\![$ *" T"* $]\!]]\!]$ *represents minimum laxity first with laxity defined as difference of deadline and the current time*

$[\![$ *"R"* $]\!]$ *represents first come first serve.*

Example: 2 *Consider the definition* $[\![$"*" B; "Within" $[\![$E1 * $[\![$"I"$]\!]]\!]$ "Readyat" $[\![$E2 * $[\![[\![$"*I*" $]\!]$ - 1 $]\!]]\!]$ *For the first iteration the task is ready at time* 0 *and should finish by time E1. For the second iteration the task is ready at time E2 and should finish by 2*E1 etc. If E1 and E2 are equal constants, the usual definition of a periodic task is achieved. By defining E1 and E2 appropriately (using general expressions), a sporadic task can be defined.*

Example: 3 *A periodic task's ready time is defined as* $[\![C^*[\![$ *"I"* $]\!]]\!]$*, where C is a constant (the period) and its deadline is defined as* $[\![C^*[\![[\![$ *"I"* $]\!] + 1]\!]]\!]$*. That is, the deadline of an iteration is the same as the start of the next iteration. The expression* $[\![[\![$ *"R"* $]\!]$ *"/"* $[\![$ *"I"* $]\!]]\!]$*, extracts the period of the task. The rate monotonic algorithm [7] can be defined as follows. Define the evaluation function to be* $[\![[\![$ *"I"* $]\!]$ *"tuple" P* $]\!]$ *where, P is the period of the task. "tuple" is an operator which given two values returns a 2 tuple. The ordering on these values is (X1 "tuple" Y1) is less than (X2 "tuple" Y2) if and only if*

X1 is less than X2 or if X1 = X2 then Y1 is less than Y2. In appendix C, the eject of the scheduler on a periodic task system is defined.

3 The Action Notation

Action Notation which has evolved from Abstract Semantic Algebras [8] allows descriptions of realistic programming languages. Actions are objects which when "performed" process information and are used to represent semantics of programs. Actions can be combined using the action combinators to derive a compositional semantics.

Actions are classified into the following facets: 1) Control 2) Functional 3) Declarative 4) Imperative and 5) Communicative. We give a brief and informal introduction to the above facets.

The control actions include complete , diverge , fail escape , commit . complete is an action that always completes, while diverge always diverges. The fail action fails which corresponds to abandoning the current alternative. The commit action corresponds to cutting away all alternatives, while escape corresponds to raising an exception.

The combinators include or , and , and then and trap . or represents non-deterministic choice. An alternative to the chosen action is performed when the chosen action fails unless a commit has been performed. and is an combinator which performs the two actions with arbitrary interleaving. and then corresponds to sequential performance, while trap corresponds to trapping the exception.

The functional actions process transient (as opposed to input/output) data and give/are given data. The actions include give D which yields the datum D, regive which gives any data given to it. choose D gives an element of the data of sort D. The principal combinator is then . A1 then A2 corresponds to functional composition, i.e., A2 is given the data produced by A1.

The declarative actions process scoped information. The actions include bind T to D, which produces a binding of token T to datum D and rebind which reproduces all the bindings it received. The combinators include moreover , hence and before . A1 moreover A2 corresponds to letting bindings produced by A2 override those produced by A1. A1 hence A2 restricts the

bindings received by A2 to those produced by A1. A1 before A2 corresponds to letting bindings accumulate.

The imperative actions deal with storage or stable information. The actions include store and allocate . store D1 in D2 which stores the datum D1 in cell D2 and allocate D which corresponds to the allocation of a cell of sort D.

The action notation also provides primitives to model parallelism. Agents form the basic unit of parallelism. The actions for this facet include send D which sends a message of sort D, receive D which receives a message of sort D and subordinate D which corresponds to creating a agent of sort D which is then sent an message containing actions which are to be executed. However in this paper, we concentrate on uniprocessors and this facet is not used. A generalization of our scheme to handle distributed real-time systems is possible. **N.B.** The Action notation may appear informal, but it has a formal signature and is specified in [9].

4 Semantics

In this section we explain the basic structure of the semantics of the constructs. The notational details are presented in appendix A and B.

The semantics of the system consists of 1) compiling the defined processes 2) creating the scheduler and 3) starting the execution from an initial state.

The process of compiling a task is to create an object representing the task and inserting it into the list of tasks in the system. The creation is defined by the function Compile, while the insertion is defined by Register. There are two types of task objects O-tasks (for non-iterative (non-*) task to be executed once) and I-tasks (for iterative (with *) tasks). Both O-task and I-task objects have: 1) the abstraction representing the compiled body, 2) the abstraction for the expressions associated with the deadline and ready time, and 3) A field representing the iteration which is initialized to 1.

If one is considering only non-preemptable tasks, the iteration field for an O-task is not necessary (it can be assume to be 1). However, when a task is preempted, the 'rest of the task' is treated as a new noniterative task. As

the scheduling algorithm could involve the iteration field, the iteration field. is copied in the O-task. This ensures that the valuation of the task under the scheduling algorithm does not change due to preemption.

The process of generating a scheduler can be described as constructing an evaluation function. This function when applied to a task yields a value. The scheduler abstraction is bound to %scheduler, which can be thought of as generating the code and storing the start address in %scheduler.

The actual scheduling (defined by the function Scheduler) consists of the following:

1. Apply the scheduler function to all the tasks in the system. This is defined by Map-to-all.

2. Select the task which yielded least value. This is defined by Select-min-index. This requires the comparison of values returned by the scheduler. To achieve this a function Compare is required by the semantics. The precise definition depends on the type of values it handles. The only restriction is that such a function be 'transitive'.

3. Remove the task from the system which is defined by Remove.

4. Add the next iteration (if any) to the system which is defined by Next-Itr. If the object is an O-task, Next-Itr is a no-op, otherwise a new task is added to the system by Form-new-system.

5. Wait till the task is ready to run defined by Process. That is, as the selected task may not be ready (but the evaluation function yields the minimum value) the system idles till the task is ready to run.

6. Run the task withing the deadline defined by Run.

If the task being executed is preemptable, a scheduling point will be reached. The remainder of the task (if any) is put back into the system and the scheduler is run again on the system. The remainder of the task is treated as a 'new' non-iterative task. Reschedule and Form-new-system effect the above behavior.

As mentioned in the introduction, it is essential to define time against which scheduling is defined. A reference to an external notion of time, is

not acceptable. If this were allowed, different implementations could have different definitions of time thus leading to unpredictable behavior across implementations. If the semantics defines 'time', a 'correct implementation' will have to account for time when translating the operators in the Action Notation to machine code. Our approach is: 1) to represent the incrementing of time as an action, 2) to define a minimum time an atomic action can take and, 3) running composite actions within the available time.

The ticking of time is represented by the functions Tick and Tick-for. The execution of one Tick increments time by one unit. Tick-for takes an integer argument (n) and ticks for n-units. Delay-till n delays till the current time is greater than equal to n. Both Tick-for and Delay-till could have been defined to increment time directly; but we define it as a sequence of steps to maintain 'regularity' of time.

The function Run executes the given action within the specified deadline. We assume that each atomic action requires a fixed time to execute (threshold). If the execution of an action cannot be completed within the deadline, the result is fail. Due to this, our semantics can be considered to be describing hard real-time systems. One can define soft real-time systems by changing fail to trap (an exception) and specifying a handler for the trap. By varying the handler different behavior can be specified. For example, if the handler was a no-op imprecise computation is specified [6].

Composite actions are ordered. For example, the and combinator which permits arbitrary interleaving between the subactions is restricted to and then. The time available for a later action depends on the consumed by the preceding actions, thus modeling execution on a single processor.

5 Conclusion

We have presented a language in which schedulers can be specified. We have also given a semantics for real-time systems which incorporates time and scheduling. Our definitions can be basis for defining 'correctness' of a real-time compiler. It can also be used to perform schedulability check, viz., whether the given scheduler is adequate for the given set of tasks. This can be done using an interpreter as time is defined in the semantics and the interpretive overhead (wall clock time) does not interfere with the time taken

to execute the actions.

However, our approach has the following drawback. In our description the scheduler is applied to all the tasks at every scheduling point. This is because the language allows general expressions, due to which one is not certain if the ordering on tasks is preserved with time. If the scope of the expressions for specifying deadlines/readiness is restricted a more 'efficient' technique can be used.

We have assumed that the real-time system executes in a uni-processor environment. By defining a scheduler for each processor and mapping a process to a processor, distributed execution of real-time programs can be described.

Acknowledgement

The author is grateful to Peter Mosses for his comments and help in presenting the semantics.

References

[1] *Ada programming language (ANSI/MIL-STD-1815A)*, Washington, D.C. 20301, January 1983. AJPO : DOD, OUSD(R&D).

[2] S. C. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems: A brief survey. In J. A. Stankovic and K. Ramamritham, editors, *Tutorial Hard Real-Time Systems.* IEEE, 1988.

[3] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *Ada Letters*, 8(7), Fall 1988.

[4] M. Joseph and A. Goswami. What's 'Real' about real-time systems? In *IEEE Real-Time Systems Symposium*, pages 78–85, 1988.

[5] P. Krishnan and R. A. Volz. A distributed real-time language and its operational semantics. In *The 10th IEEE Real-Time Systems Symposium*, 1989.

[6] K. Lin, S. Natarajan, and J. W. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *IEEE Real-Time Systems Symposium*, pages 210–217, 1987.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, pages 46–61, February 1973.

[8] P. Mosses. Abstract semantics algebras. In D. Bjoerner, editor, *Proceeding of the IFIP TC2 Working Conference on Formal Description of Programming Concepts II*, pages 63–88. North Holland, 1982.

[9] P. D. Mosses. Action semantics. Technical report, DAIMI: Aarhus University, 1990.

[10] P. D. Mosses. *Action Semantics*. Cambridge Univesity Press (in the series Tracts in Theoretical Computer Science), to appear in 1991.

[11] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, pages 10–19, October 1988.

A Semantic Equations

A.1 Compilation

Complie :: Process → Act

Compile ⟦ B "Within" E1 "Readyat" E2 ⟧ =

```
| | give abstraction (P-execute B) label # 1
| and
| | give abstraction (Evaluate E1) label #2
| and
| | give abstraction (Evaluate E2) label #3
then
| give O-task(the abstraction #1,
|       the abstraction #2,
|       the abstraction #3, 1) then
| Register it
```

Compile ⟦ "*" B "Within" E1 "Readyat" E2 ⟧ =

```
| | give abstraction (P-execute B) label # 1
| and
| | give abstraction (Evaluate E1) label #2
| and
| | give abstraction (Evaluate E2) label #3
then
| give I-task(the abstraction #1,
|       the abstraction #2,
|       the abstraction #3, 1) then
| Register it
```

Compile ⟦ P1 ";" P2 ⟧ =

Compile P1 hence Compile P2

A.2 Compilation of Scheduler

```
Create-Scheduler :: Sched → Act
Create-Scheduler ⟦ Sched ⟧ =
        give abstraction (Evaluate Sched) then
        bind %scheduler to it
```

A.3 Expression Evaluation

```
Evaluate :: Evaluation → Act

Evaluate ⟦ "I" ⟧ = give Iteration(it)

Evaluate ⟦ "T" ⟧ = the datum bound to %time

Evaluate ⟦ "D" ⟧ = enact (Deadline it)

Evaluate ⟦ "R" ⟧ = enact (Readiness it)

Evaluate ⟦ F1 O F2 ⟧ =
        | | Evaluate F1 then give it label #1
        | and
        | | Evaluate F2 then give it label #2
        then
        | give result-of O
```

A.4 Execution

```
P-execute :: Body → Act

P-execute ⟦ Bu "->" B ⟧ = execute Bu and then
                          Reschedule B

P-execute Bu = execute Bu

Reschedule :: Body → Act
```

```
Reschedule B =
    | | give abstraction (P-execute B) label # 1
    | and

    | | | give (the datum bound to %current-task)
    | | then
    | | | | give Deadline label #2
    | | | and
    | | | | give Readiness(it) label #3
    | | | and
    | | | | give Iteration(it) label #4
    | | then
    | | give 0-task( the abstraction #1,
    | |              the abstraction #2,
    | |              the abstraction #3,
    | |              the number #4) then
    | | Register it
```

A.5 Running the System

```
Begin :: System → Act

Begin ⟦ P ":" S ⟧ = | | Initialize
                    | before
                    | | | Compile P
                    | | and
                    | | | Create-Scheduler S
                    | hence
                    | | Run-System
```

B Semantic Entities

B.1 Initialization

```
Initialize = bind %system to empty-list moreover
             bind %time to 0
```

B.2 Scheduler

```
Run-System =
        unfolding
        | | | check (the datum bound to %system
        | | |         is not the empty-list)
        | | and then
        | | | Scheduler hence unfold
        | or
        | | check (the datum bound to %system
        | |         is the empty-list)

Scheduler = give (the datum bound to %system) then
            | Map-to-all (the datum bound to %scheduler) to it
            then
            | | | Select-min-index from it then give it label # 1
            | | and
            | | | give (the datum bound to %system) label #2
            | then
            | | | Remove (the integer #1) (the list #2) then
            | | | | rebind
            | | | moreover
            | | | | bind %system to it
            | | hence
            | | | Process (the integer #1) at (the list #2)

Register :: Task → Act
```

```
Register T =
    | give (the datum bound to %system) label #1
    then
    | concatenate (list T) (the list #1) then
    | | rebind moreover bind %system to it

Process :: Task → Act

Process Tsk = | Next-Itr Tsk moreover (bind %current-task to Tsk)
              before
              | | enact (Readiness Tsk) then Delay-until it
              | before
              | | | enact (Deadline Tsk) then give it
              | | then
              | | | Run (Body Tsk) it

Next-Itr :: Task → Act

Next-Itr Tsk = | | choose (I-task & Tsk)
               | and then
               | | give Step-I-task Tsk then
               | | Form-new-system it
               or
               | choose (O-task & Tsk)
```

B.3 Time

```
Tick = indivisibly
    | give (the datum bound to %time) then
    | give (its successor) then
    | bind it to %time

Tick-for :: Integer → Act
```

```
Tick-for n =
        unfolding
        | | check (n is not 0) and then
        | | Tick 1
        | | before
        | | give (predecessor of n) then unfold
        | or
        | | check (n is 0)

Delay-till :: Integer → Act

Delay-till n =
        unfolding
        | | give the datum bound to %time
        | then
        | | check (it is less than n) and then
        | | Tick before unfold
        | or
        | | check (it is not less than n)

Run :: Act, integer → Act
```

```
Run (A:Atomic) Dline =
    indivisibly
    | | bind the current-transients to %save-data
    | before
    | | | give the datum bound to %time then
    | | | give sum it threshold
    | | then
    | | | check (the datum is less than Dline)
    | | and then
    | | | | Tick-for threshold
    | | | before
    | | | | | reflect the datum bound to %save-data
    | | | | then
    | | | | | A
    | | or
    | | | check (it is is greater than Dline) and then
    | | | fail
```

```
Run (A and then B) Dline = | Run A Dline
                           before
                           | Run A Dline
```

```
Run (A and B) Dline = | Run A Dline
                      before
                      | Run A Dline
```

comment: Other composite actions can be defined analogously

B.4 Tasks

I-task $\leq$ datum

I-task :: datum, datum, datum, integer $\rightarrow$ I-task

O-task $\leq$ datum

```
O-task :: datum, datum, datum → O-task

Task = I-Task | O-Task

I-Task(B,D,R,I) = T ⇒
        Body(T) = B, Deadline(T) = D,
        Readiness(T) = R, Itr(T) = 1

O-Task(B,D,R) = T ⇒
        Body(T) = B, Deadline(T) = D,
        Readiness(T) = R, Itr(T) = 1

Step-I-Task Tsk =
        | | give Body(Tsk) label #1
        | and
        | | give Deadline(Tsk) label #2
        | and
        | | give Readiness(Tsk) label #3
        | and
        | | give Itr(Tsk) then
        | | give its successor label #4
        then
        | give I-Task (the abstraction #1,
        |       the abstraction #2,
        |       the abstraction #3,the integer #4)

Form-new-system Tsk =
        | give (the datum bound to %system) # 1
        then
        | give contactenate (list Tsk) (the list #1) then
        | rebind moreover bind %system to it
```

B.5 List Manipulation

```
Remove ix L =
    | check ( ix is 1) and then give (tail L)
    or
    | | check ( ix is not 1 )
    | and then
    | | | give list (head ix) label #1
    | | and
    | | | Remove (predecessor of ix) (tail L) then
    | | | give it label #2
    | then
    | | give concatenate (list #1) (list #2)
```

```
Map-to-all A L = | check (L is empty-list) and then give L
                 or
                 | | check (L is not empty-list L)
                 | and then
                 | | | enact A with (head L) then give it label #1
                 | | and
                 | | | Map-to-all A (tail L) then give it label #2
                 | then
                 | | give concatenate (the list #1) (the list #2)
```

Compare :: value, value → truth-value

Compare(v1,v2) = true; Compare(v2,v3) = true
    ⇒ Compare(v1,v3) = true

Select-min-index L = I; J : [less than or equal][length of L]
    ⇒ Compare (L at I) (L at J) = true

C Examples

In this section we present two eximples which demonstrate the applicability of the semantics. To show this we use the operational intuition of the actions. The operational semantics for the notation is defined in [9] and this behavior can be formally derived from it. The first example describes a rate monotonic scheduler while the second describes a earliest deadline first scheduler.

C.1 Rate Monotonic Scheduler

Consider a system with two periodic (and hence iterative) tasks. Let the readiness associated with task T1 be 40 * "I" and deadline be 40*(I+1). Task T2's readiness is defined as 50*I and deadline as 50*(I+1). Define the rate monotonic scheduler as I "tuple R/I with the ordering defined as (X1,Y1) $\leq$ (X2,Y2) iff (X1 $\leq$ X2) or (X1 = X2 and y1 $\leq$ Y2). (We have omitted the brackets [[]]to enhance readability.)

Compiling (Compile) this system creates two I-task objects I-task(B1, D=40*(1+1), R=40*I, I=1) and I-task(B2, D=50*(I+1), R=50*I, I=1) where B1 and B2 are the compiled bodies which is bound to %system. The scheduler abstraction is bound to %scheduler.

Running the system (Run-System) executes the Scheduler. The scheduler applies the evaluation function to the two tasks and selects the first task for further processing. Process alters the state of the %system to I-task(B1, D=40*(I+1), R=40*I, 1=2) and I-task(B2, D=50*(I+1), R=50*I, I=1). The next iteration of the current task is added to the task set by Next-Itr. The current task I-task(B1, D=40*(I+1), R=40*I, I=1) is bound to %current-task. The readiness and the deadline expressions are evaluated and the task is run (Run). If B1 were preemptable (i.e., of the form B11 -> B12), after the execution of B11, Reschedule is executed. It adds 0-Task(B12, D=40*(I+1), R=40*I, I=1) to the system. Run-System operates on the modified system.

The scheduler does not differentiate between I-tasks and O-Tasks and it selects the O-Task for B12. The effect is thus to execute B1 without preemption as defined by the rate monotonic algorithm. When B1 terminates, the state of the system is I-task(B1, D=40*(I+1), R=40*I, I=2) and I-task(B2, D=50*(I+1), R=50*I, I=1). As task T2's iteration is still 1, it is selected over T1 and B2 is executed till completion and the cycle is repeated. The

above behavior assumes that the task body completed its execution within time. That is, Run B1 80, etc. did not fail. However, if one of the tasks does not finish its execution, the entire system terminates abnormally (fail).

If the first iteration of B1 took only 5 units of time, T2 will idle for 5 time units as it is ready to execute at time 50. If B2 took 10 units of time, the next iteration of T1 will idle for 20 units and so on.

C.2 Deadline First Scheduler

In this example we concentrate only on the behavior. The effect of the semantic equations at each step is as in the above example. Consider a system with two iterative tasks. Task 1 has readiness 40 *I and deadline 40*I+10, while task 2 has readiness 50*I and deadline 50*I+20. These are not periodic tasks as the beginning of one iteration is not the deadline of the previous iteration. Let the scheduler be D, viz. the deadline first scheduler. Initially there are two tasks in the system: I-task(B1, D=40*I+10, R=40*I, I=1) and I-task(B2, D=50*I+20, R=50*I, I=1) where B1 and B2 are the compiled bodies. Assume that B1 and B2 take 5 and 10 units of time respectively. Task 1 will be executed first and the system will idle for 5 time units. T2 will be executed next and at time 60 the state of the system is as foIlows: I-task(B1, D=40*I+10, R=40*I, 1=2) and I-task(B2, D=50*I+20, R=50*I, I=2). The system idles for 20 units and then executes task1 and so on. In state I-task(B1, D=40*I+10, R=40*I, I=4) I-task(B2, D=50*I+20, R=50*I, I=4), T1 is executed leading to I-task(B1, D=40*I+10, R=40*I, I=5) I-task(B2, D=50*I+20, R=50*I, I=4). T1's deadIine is 210 and T2's deadline is 220. Thus, T1 is executed, and the cyclic execution of the tasks is not exhibited.