Studienarbeit
(15-poängs-examensarbete)

SiRTiS

A Simulator for the evaluation of Real-Time Scheduling Algorithms

Rolf Stechert

LITH-IDA-Ex-ing-97/30

1997/04/23

This study work was written by Rolf Stechert (720622-P793), in the Real-Time Systems Laboratory RTSLAB of the University of Linköping, Sweden between September 1996 and April 1997.
Supervisor was Anders Törne (andto@ida.liu.se, tel.: +46-13-28 2365).
My address in Linköping is: Björnkärrsgatan 6A:13
S-584 36 Linköping
Tel.:+46-13-174882
It will be valid until at least end of Juli 1997.
My e-mail address rolf.stechert@ira.uka.de will be valid further on.
In Germany, I can be reached at my parents' address: Lichtensteinstr. 28, D-89551 Zang, Tel.:+49-7328-5073, FAX:+49-7328-6223.

Contents

1 Overview

This simulator SiRTiS has been developed to compare different scheduling algorithms with each other. Special emphasis has been placed on the requirements of real-time systems. This implies that the simulator also has to take precise timing constraints into account. Apart from that it is also possible to evaluate multi-processor systems.

The simulator is supplied with an amount of scheduling algorithms that can be described in a simple way. It takes as input one or more descriptions of the system, the processes that run on it, and the timing constraints in form of deadlines. After that, it simulates the specified scenario descriptions with each provided scheduler.

A lot of care has been taken to make the usage of this simulator as easy as possible and at the same time keep the restrictions for the simulated system and the scheduling algorithms very few. In order to keep modularity, the scheduling algorithms and the scenarios are declared completely separate. The simulator checks if the scenario satisfies the scheduler's requirements. In case these checks should not suffice, the scheduling algorithm can add its own tests.

During and after the simulation, logging information is printed out on the screen or, if desired, into separate files. One file is created for each combination of scheduler and scenario. This output can be evaluated afterwards with other tools, preferably with small programs written in a rather shell-like language suitable for easy text-processing as e.g. Perl.

The simulator-tool is intended to be used for two purposes:

- First, it can be used for student education to demonstrate the functionality and implementation of different scheduling algorithms.

- Secondly, it serves as a research-tool to evaluate the behaviour of different scheduling algorithms in various distributed real-time systems.

This work is the result of a 'Studienarbeit' (15-poängs-examensarbete) at the Real-Time Systems Laboratory, RTSLAB, at Linköpings University in Sweden.

The research within RTSLAB is focused on aspects of the embedded software component of real-time systems. The current projects include, among others, the following fields of research:

- Error recovery in manufacturing control systems
- Database technologies for real-time control and simulation
- Timing analysis and time related Petri Nets
- Uniform interface between different system engineering design tools
- Proving software correctness using formal methods

1.1 Contents of the chapters

Chapter 2 explains the most important basic knowledge about real-time systems and scheduling.

Chapter 3 motivates the decision to develop the simulator SiRTiS.

Chapter 4 presents the overall design, and compares SiRTiS with STRESS, another existing simulation tool.

Chapter 5 moves on to the implementation. It also describes all the details that are important to describe new process scenarios and new scheduling algorithms.

Chapter 6 gives an example of a process scenario and of a scheduling algorithm.

Chapter 7 lists those features, that might be important in some situations, but are not implemented in the current version of SiRTiS.

Appendix B serves as a reference to look up the most important type declarations in the simulator. They are important to write new scheduling algorithms.

Appendix C finally presents the small program, that has been used to extract all the source code in this document from the original program files.

2 Background

This chapter explains fundamental definitions that are of importance for the understanding of the whole simulator.

2.1 Real-Time Systems

- Any kind of software-hardware-combination, that has to perform tasks within a given time, is considered a real-time system.

Real-time systems are most important in all kinds of hardware management where it is important that the computer has to react on events within a given time. The significance of time, that passes in our real world, is the reason for the word *real-time systems*.

An example for such a system is the control of a robot. In this case the computer must be fast enough to react on the sensor signals from the robot, otherwise possibly wrong movements are performed which might result in malfunction or even worse damage. The timing constraints in such a system are rather severe.

But there are also real-time systems with less severe timing requirements, for example multimedia systems. They should run with a given speed so that the user does not have to wait too long. The deadlines of such a system are soft, i.e. they can be missed without resulting in an error or damage. The only bad effect is that the user gets the impression that the system is too slow.

The topology of the software part of a real-time system can vary, but usually it has the following configuration:

- It consists of an amount of n processes running on p processors. Each process has a specified task to fulfill, but in contrary to a non-real-time system the tasks have to be completed within a given *deadline*.

- The operating system has to supply facilities that enable the programmer to get control over timing behaviours within the system. It must be possible to guarantee in some way, that the given deadlines are kept or to force a special reaction if it becomes clear that a deadline will be or has been missed.

There are 2 general problems that make this requirement hard to satisfy:

- The processes are not at all independent of one another. They often have to wait for *events* in the system which are caused by other processes. These events might for example be released semaphores or signals coming from other processes. They will be explained later in this chapter.

  This makes it extremely difficult for the operating system to know the remaining computation time for the processes without having a deeper insight in the processes' behaviour.

- There are in general more processes than processors, i.e. $n > p$. This implies that the processes have to be executed one after another. The decision of which process gets how much CPU time and on which processor is done by a **scheduling algorithm** within the operating system. An increasing amount of processes results in a slowdown of the execution times and possibly missed deadlines.

  Remark: There are now evolving new languages and changes to existing languages for real-time systems. They make it possible to ensure that a program will be executed within a maximum time period, supposed it does not have to wait for any events.
  The main point is to prevent endless loops. They are stopped after a certain amount of iterations or after a given time. When this happens, often correction code will be executed, so that the program can continue. [Kligerman and Stoyenko, 1986] present a language that is especially developed for real-time systems.

2.1.1 Hard and soft real-time systems

There are in general two kinds of deadlines:

- Hard deadlines
- Soft deadlines

A hard deadline means, that there exists a maximum time within which an action must be finished, otherwise it will result in an error that might cause the system to fail. A signal, coming over a line, for example, has to be processed or stored before the next signal arrives, otherwise information loss will occur.

Soft deadlines are not as critical as hard deadlines. If they are missed, the system just becomes less useful. As an example, the user of a multimedia system wants to run his applications in real-time, yet delays will not be supposed a severe error.

According to these definitions a hard real-time system is one whose processes have hard deadlines, and a soft real-time system is one with soft deadlines. Unfortunately, in practice we will often have to deal with a mixture of these two, making it harder to schedule the processes.

[Burns, 1991] presents a way to represent hard and soft deadlines. Every process has its own utility function $f : \mathbb{R}^+ \to \mathbb{R}$ that specifies the utility $f(t)$ that the system obtains if the process completes after time t. Between 0 and the deadline $t_o > 0$ the utility has a constant positive value. Then it decreases. In the case of a hard deadline it directly switches to a value less than or equal to zero. A value less than zero means that the system is harmed by this late process.
On the contrary, the utility function of a soft deadline is continuous, it just decreases after the deadline.

2.2 Aspects of multitasking systems

A system that provides capabilities to run more that one process at the same time, is said to be capable of *multitasking*.

There are several general very important aspects of such systems that need to be explained:

2.2.1 Process states

A process may be in one of the following states:

- Non-existent: This process does not yet exist. It can be created by for example a compiler.
- Non-active: The process exists, but has not been started, or it has already finished.
- Active: The process has been started. This state can be divided into three substates:
  - Blocked: The process cannot continue, it is waiting for an event to occur or for a condition to become true.

- ◦ Ready-to-Run: The process can continue, but it has to wait for a free processor.
- ◦ Running: The process is currently being executed by a processor. This is the only state in which a process really continues its work.

In the rest of this document, the state 'Running' and 'Ready-to-Run' are both called for 'Runnable', because the only difference is, whether they are currently working on a processor or not.

2.2.2 Scheduler

In a multitasking system, there are usually more runnable processes than available processors[1]. A result of this is that not all runnable processes may be executing at one time. Instead, they have to share the processors. There occur so-called *task switches*, that assign another process to a processor. These task switches are performed very often, so that the user gets the impression that all processes are running at the same time.

The operating system has the control over the task switches. There are principally two possibilities, when these switches occur:

- **Non-Preemptive:** The running processes have to call a routine in the operating system that might initiate task switch. There are no task switches if the process does not call any operating system routines. Therefore, it is possible that one process uses the system for a longer time, and the other processes have to wait. The impression of a multitasking system gets lost, because the task switches occur too irregularly.

  This method was used in older operating systems, because it is easier to implement, as for example in Microsoft Windows up to version 3.0.

- **Preemptive:** The operating system may interrupt running processes and cause a task switch at any time. Most operating systems, above all UNIX systems, use this mechanism.

The decision about which of the runnable processes gets the right to run, is made by a scheduling algorithm, which is contained in the operating system. There are many different strategies for schedulers with different parameters. They all have their own advantages and disadvantages. There does not exist a scheduler, which is optimal for all possible systems.

Almost all scheduling problems are NP-hard, or even NP-complete, thus approximations have to be used.

[1] If the base processes, that are provided by the operating system, are also counted, there are always at least as many runnable processes as available processors. This will not be regarded in the rest of this paper

Above all in real-time systems, it is very important to have a scheduler that cares about the processes' timing constraints. The decision about the right scheduing algorithm is very complex and depends on many factors. That is one of the reasons to use this simulator.

Example of a scheduling algorithm:

One of the most known scheduling algorithms is the rate monotonic algorithm. It works for single-processor environments. It assumes that the system only has periodic processes, i.e. the processes are started after a fixed time-interval. All processes have their (hard) deadline equal to period, i.e. they have to be finished before they are started again.

The rate monotonic algorithm always schedules the running process with the shortest deadline (= period).

2.2.3 Semaphores

There are many resources in a system that may only be accessed by a given maximum amount of processes at one time. The typical case is that only one process may access them, i.e. the maximum is one. These system resources may either be

- Hardware, for example a printer: It is undesired that the printouts of two users are mixed together, they have to be printed one after the other.
- Memory: If some processes have access to shared variables, it is very important that all writing operations on them are only performed by one processor at a time.

In order to guarantee this exclusion, the operating system provides so-called *semaphores* for each resource. These are implemented as integer variables containing values greater than or equal to zero. If a process wants to access n instances of a resource, it locks the semaphore of that resource with the value n. This means that the value of the semaphore is decreased by n, if the operating system allows the locking. A lock-operation is never allowed if the value of the semaphore before the lock-operation was less than the required n, because it must not be less than zero after the lock-operation.

A process becomes blocked if it is not allowed to lock a semaphore.

After the access to the resource, the process must release the semaphore. The semaphore's value is increased by n, and the operating system tries to reactivate any of the other processes that are blocked due to this semaphore.

Often, the words *wait* and *signal* are used instead of lock and release.

Examples:

- ∘ A hard-disk can be specified with a binary semaphore, only one process may access it at a time.
- ∘ A computer memory of a size of 16000 KB for example can be simulated with a semaphore initialized with 16000. A process requiring 1000 KB memory allocates 1000 units of this semaphore.

2.2.4 Signals

Processes, that have to work together, often send signals to each other. They are used for two purposes:

- Inform the other process(es) about an event that has occured. For example, a hardware-event such as pressing a key on a keyboard results in a signal to the process that handles this kind of event.
- There exist multiprocessor-systems with local memory for each processor. The processors do not have any shared memory. Because of this, two processes, that run on different processors, cannot exchange data via a shared memory. Instead, they send *messages* to each other. These messages consist of a signal that is extended with the data that has to be transmitted.

The most common way of a signal-comminication is that of sending asynchronously and receiving synchronously. This means, that the sender just sends the signal and continues without waiting that the signal has been received. The receiver has to wait until the signal has come. For that time, the receiving process is held in the blocked state.

2.2.5 Remote Procedure Call (RPC)

Multitasking systems provide possibilities to program procedures that can be called from other processes. A call to such a procedure is called remote procedure call. The process that has initiated the call is named *client*, the process executing the remote procedure is the *server*. A remote procedure call implies much more overhead than a call within one process:

- The processes have to be synchronized.
- The client has to be blocked and the server activated.
- On one-processor systems a task-switch needs to be performed.

- Parameters and results have to be transmitted, possibly across a network.

- The transmitted data has to be converted, if client and server are executing on different hardware architectures. They might for example use a different floating-point format.

Examples:

- The access to a hard-disk drive in a computer happens via a remote procedure call. Thus the low-level routines for handling the hardware exist only once in the system.

- The pages in a WWW-browser are fetched with a remote procedure call across the internet.

2.2.6 Deadlocks

A problem that arises from process dependencies is the danger of reaching deadlocks, which means that an amount of processes are dependent on each other in a way that none of them can continue unless the other processes continue.

Example:

Suppose a system with one modem and one loudspeaker. There are two processes, process 1 and process 2, that intend to load a sound from somewhere via the modem and play it on the loudspeaker at the same time. In order to do this, each of them has to lock as well the modem as the loudspeaker. This can cause a deadlock:

	Process 1	**Process 2**
1.	lock modem	
2.		lock loudspeaker
3.		lock modem
4.		play sound
5.		release modem and loudspeaker
6.	lock loudspeaker	
7.	play sound	
8.	release modem and loudspeaker	

In this situation, a task switch has occured after instruction 1. Therefore both processes cannot continue, process 1 is waiting for the loudspeaker and process 2 for the modem.

There are three ways of handling deadlocks.

1. **Deadlock prevention:**
   The programmer has to take special care of the matter. I.e. there are constraints in using semaphores.

2. **Deadlock avoidance:**
   The operating system must have information about the maximum requirements of semaphores of each process and the signals that will be sent among them. With this information, it is possible to decide whether a process may lock a semaphore without the risk of a deadlock in the future.

   An example for an algorithm that works that way, is *Banker's Algorithm*.

3. **Deadlock detection and recovery:**
   Deadlocks can easily be detected in time $O(n+m)$, with n processes and m dependencies among them. When the operating system detects a deadlock, it has to perform actions against this deadlock. It can for example kill one of the processes to release the semaphores that this process has locked.

[Silberschatz and Galvin, 1994] describe deadlock prevention and deadlock avoidance in more detail.

3 Simulation of Real-Time Systems

This section will explain the motivation to use a simulator for the comparison of different scheduling algorithms.

System programmers for real-time systems have to take special care of the timing constraints of the problem. Above all two aspects are of importance:

- ∘ The behaviour of the processes and
- ∘ The algorithm used to schedule the processes

It is very important that the scheduling algorithm can handle the sort of timing constraints arising in the system. Thus the scheduler and the processes have to be regarded together. There are in general three ways to check the usability of a scheduling algorithm for a given scenario of tasks, which are presented below:

3.1 Code everything and try it in its real environment

The idea is to develop the whole system and after that perform test executions. There are important disadvantages to this approach:

- It will give sufficient results only in small systems, where many or all of the possible situations can be checked. This is impossible for larger systems because the amount of possible system states grows exponentially with the size of the system. It would take too much time to test every situation.
- It requires that the whole system has to be programmed before it can be tested. This means, that the right scheduler can only be selected after the whole system is complete. However, this selection is a design decision, and should be clear before the implementation.
- It is often inconvenient to use the real environment for testing, as it might result in high hardware costs or even damage.

It is for example impossible to test a medical control-system in practice. The patients might not survive this.

3.2 Theoretical Proof

In order to prove that the system will work, we have to know all possible timing behaviours of the programmes in question. This information can be generated for real-time languages by the compiler. However, there are two important reasons contradicting the theoretical way:

- All the analysis is for the worst case, because the final behaviour of the programs is not known, and therefore the longest possible execution times have to be presumed. This often results in a real-time system with a very low CPU load, as the actual execution times are in average significantly less than the worst case times.

  The difference between the worst case and the average case can easily grow up to large factors, such as 100. This implies that such a system would run with a CPU-load of only 1%, which is too low for most engineering purposes.

  Alternatively, the estimated average execution times can be used instead of the worst case. This will result in a higher CPU-load, but the proof cannot guarantee, that the timing constraints are obeyed.

  [Liu and Ha, 1995] show efficient proofs for timing constraints in real-time systems. These proofs all put constraints on the system.

  [Leinbaugh and Yamini, 1986] present a proof for hard real-time systems with signal communications between the processes.

- As soon as the system grows, the theoretical proofs come up to a level of complexity, that often cannot be reasonably handled any more. In order to be able to prove a system to be correct, a lot of restrictions have to be laid upon it. These restrictions are often too strong to be used for realistic real-time systems. For larger systems, it is often only possible to prove parts of it, and not the system as a whole.

  However, a theoretical proof is necessary for very critical systems, where all timing constraints have to be obeyed in every possible case.

3.3 Simulation

A Simulation executes the processes without the real environment they are dedicated for. It has to be chosen, when the first two possibilities cannot be used.

The decision, which scheduling algorithm will be used, should be made during the design-phase. Test runs cannot be done so early, because they need a running implementation. Formal proofs require very detailed information about the timing behaviour of the processes. This information is not available in the design phase, because the details depend on the final implementation. The only possibility is a simulation .

There are mainly two goals in using a simulation:

- Try out a new scenario:

  When it is not so important which kind of scheduler is used but rather to simulate the precise behaviour of a given processes, this simulator might not be the right one, because it only accepts a rather coarse description of the processes.

  There are simulators available for this problem.

  [Gustafsson and Liss, 1985] describe the simulation package DEMOS for the programming language Simula.
  [Sjölund et al., 1993] present a more extended package for ADA, called ADAM.
  These two simulation packages have their own scheduler included, and they are not designed to work with different kinds of scheduling algorithms.

- Try out a new scheduling algorithm:

  SiRTiS is especially addressed to this kind of problem. It is possible to specify a given scenario of processes abstractly without the need to program every detail. Thus, complex systems that are too large to be proved theoretically can be examined with SiRTiS. There is an interface to the simulater that makes it possible to plug any sort of scheduling algorithm to it, plug and work `:-)`

3.4 Summary

This section has shown the advantages and disadvantages of the three possible methods for comparing scheduling algorithms. All of them are usable, but simulation offers simplicity and easy investigation of the properties of the scheduling algorithm.

4 Design

4.1 Design of the scenario description

There are some general decisions that have to be taken to make this simulator small but yet powerful enough to fit to the different kinds of processes and above all scheduling algorithms.

There exists a special small language that shall be introduced later. It has the possibilities and restrictions listed below:

4.1.1 The Simulated System

The following properties of the hardware of the simulated system can be specified:

- There exists only one type of processor, i.e. the processes' execution time does not depend on the processor, that it is scheduled on.

- There only exists preemptive multitasking. The older non-preemptive multitasking is not supported.

- The system consists of a number of nodes with the same amount of processors on each node. Signals that are sent to another node need a longer time than those sent within one node. A process-move to another node takes longer than one to another processor within the same node.

- The network is assumed to be a point-to-point network, i.e. it connects every node (processor) with every other node (processor on the same node). Therefore the delays between any two nodes (processors) are the same.

  Future releases of SiRTiS might contain possibilities to specify other network topologies, but this makes the design of general scheduling algorithms mor complicated.

4.1.2 The Processes

- *Process coordination and communication:*

  The processes are coordinating and communicating via semaphores and signals. The remote procedure-call is not provided, because it can easily be described with two signals.

  - Semaphores do not have an upper bound, any amount of resources can be specified.

  - There are no differences of what a semaphore is used for. This means, that a special semaphore for shared data, that allows either one process to write or many processes to read, cannot be specified directly. Instead, the following method has to be used:

    There is one read-semaphore, initialized with $n > 1$, and one write-semaphore, initialized with 1. A read-process only locks one read-semaphore, whereas a write-process has to lock all of the n read-semaphores, and the write-semaphore, too.

    This method does not look very nice, but future releases of SiRTiS might provide this kind of semaphore.

  - Signals are sent asynchronously and received synchronously.

  - Signals can be assigned an additional delay in order to simulate the additional data, that has to be transmitted if it is a message.

  - Signals are only sent to a process type, not to an instance of a process. Therefore, any running instance of that process may receive that signal.

  - There are no channels, which could be used for signals from or to more than one processes.

- *Time:*

  The time is measured in virtual time units. All timing specifications are meant to refer to the virtual time of the simulated system. There are two different times:

  - Execution time: This is used to describe the time that a process needs for execution. It measures only scheduled time, i.e. the time that a process is running on a processor.

- System time: This time begins with zero when the simulation starts and does not depend on the actions in the simulated systems. It represents the clock of the simulated system.

- *Distributions:*

  All the timing information in the process description can be specified as an *absolute number* or as a *statistical distribution.* Currently the following distributions are available:

  - `normal`, the Gauss distribution, with mean value and standard deviation.

  - `exp`, exponential, also with mean value and standard deviation

  - `uniform`, an equal probability of all values between the minimum and the maximum

  As the first two distributions are infinite, which is unlikely to fit on a real-time system, there is always a minimum and maximum required. The values are limited to this interval. The detailed implementation of the normal and the exponential distributions are explained in section 5.5 on page 42.

- *Process creation:*

  Processes can be started aperiodically at a certain time (fixed or given by a distribution) or by a signal sent by any other process or they are started periodically after a given time-interval.

- *Control flow in the processes:*

  The only control structures in processes are loops. They can iterate either a constant or a distribution-given amount of times, or infinite many times. This is the only place where a distribution can be used instead of a natural number.

4.1.3 The Timing constraints

All the timing constraints in the simulated system are expressed by means of deadlines. Deadlines always describe how much time may be spent between two events in the system. Usually, a deadline is assigned to the execution time of a process, i.e. the time between the process creation and the end of the process. Hard deadlines give a maximum time. If the deadline is missed, i.e. the time between the two events is longer than the given maximum time, the system fails in some respect. On the contrary, a soft deadline states a maximum time that should not be passed between the two events. A missed soft deadline is not as severe as a missed hard deadline, it does not automatically result in a failure.

- *Utility:*

  The method of utility-functions, that has been introduced in 2.1.1 on page 10, is used. This means that every deadline has a utility function. This function assigns a utility to the elapsed time between the start and end event of the deadline. This utility can be seen as a kind of credits, that the system gets when the processes are executing fast enough. The utility function is decreasing, according to the semantics that a slower execution cannot be more useful than a faster one. An example for a utility function is shown in figure 2 on page 47.

  Hard deadlines:

  Suppose t_0 is the time of the deadline, and $u_0 \in \mathbb{R}^+$ the initial utility of the process.
  A hard deadline has the form
  $$f(t) = \begin{cases} u_0 & \text{for } 0 \leq t \leq t_0 \\ 0 & \text{for } t > t_0 \end{cases}.$$

  Soft deadlines:

  A soft deadline is specified by $n+1$ pairs (t_i, u_i), $_{0 \leq i \leq n}$ for the utility at a time. n is greater than or equal to one. The function $f(t)$ is formed by interpolating the points (t_i, u_i) and (t_{i+1}, u_{i+1}). This gives the utility function the form
  $$f(t) = \begin{cases} u_0 & \text{for } 0 \leq t \leq t_0 \\ u_{i-1} + \frac{(t-t_{i-1})\cdot(u_i-u_{i-1})}{t_i-t_{i-1}} & \text{for } t_{i-1} < t \leq t_i \\ u_n & \text{for } t > t_n \end{cases}$$
  where $\forall\, 1 \leq i \leq n$: $(t_{i-1} < t_i) \wedge (u_{i-1} >= u_i)$.

- *Deadline specification:*

  To specify deadlines, labels are introduced as annotations in the process description. The deadlines can be specified between two labels within one process or along a chain of signals between several processes. In this way almost any timing constraint can be expressed.

  A new deadline is created, when a process reaches the start label, that has been specified in the deadline. This process is said to own the deadline. If a deadline is specified between different processes, it is transferred with the signal, that is sent from one process to the other. This means, that the deadline changes its owner.

- *Missed deadlines:*

  A deadline is said to be missed if it takes more than t_0 time units. A missed hard deadline always results in an abrupt end of the process currently owning the deadline. A soft deadline just reduces the utility function, and the process can continue. In addition it is possible for both to specify that a signal shall be sent to another process or that the simulation shall be stopped, in case that the the deadline is missed.

4.2 The specification of scheduling algorithms

The scheduling algorithm consists of a set of callback procedures, that can react on different events that occur during the simulation. These procedures get all the simulation data as a parameter so that they can make their decisions and change the simulation accordingly. After their return the simulation continues.

All non-scheduling operations are executed outside the scheduler algorithm.

- In order to make sure that the scheduling algorithm fits to the kind of problem that is described in the process definition file, the scheduler has to fill a record of booleans describing its abilities. The record `Abilities` contains for example variables that state if the scheduling algorithm can handle multiprocessor-systems or if it can handle hard deadlines.

  The simulator performs checks to ensure that the scheduling algorithm can handle the described scenario. The details of these checks are explained in section 5.3 on page 34. In addition, the scheduling algorithm can add more checks if it has special requirements that have not been regarded in this simulator.

- Because the algorithm is specified in a universal programming language, any scheduling algorithm can be specified.

4.3 The simulator STRESS

There exists a simulation package for scheduling algorithms in real-time systems. That program, called STRESS, has been developed at the University of York and is described in [Audsley et al., 1994].

It includes the following features:

- STRESS is designed only for hard real-time systems. There are no soft deadlines.

- STRESS introduces a small programming language. It is used for both, the specification of the process-scenario in the system and the corresponding scheduling algorithm.

- The description language provides variables, arrays, and a conditional control flow, including loops and conditional execution. Thus the processes can be specified quite in detail.

- STRESS has the possibility to give a uniform distribution to specify, how much CPU-time a process needs for its calculations. There are no other distributions.

- Always the process with the highest priority is running. Scheduling is done by changing a process' priority.

- STRESS provides some predefined schedulers.

- Scheduling is performed locally on the processors. Therefore no process will be moved to another processor or node.

- STRESS has no checks to verify that the scenario and a user-defined scheduling algorithm fit together. They are both specified together and thus one scheduling algorithm is only used for one scenario. It can be cumbersome to compare two different scheduling algorithms with the same scenario or to run one algorithm on two scenarios.

- A possibility for user-defined routines for the locking and unlocking of semaphores are provided. Priority inheritance can be applied to these routines.

- STRESS provides possibilities to specify a multiprocessor system. Processors are grouped together on nodes, and there can be more than one node.

- A network topology can be specified by limiting the given point-to-point connections of the nodes.

- Coordination and communication is done with semaphores and signals. The signals are sent to a node's or processor's mailbox.

- Timing constraints can only be specified as hard deadlines for the whole execution-time of one process. It is not possible to specify a deadline between any other events.

- Predefined tests can be performed with these deadlines for guaranteed schedulability.

- STRESS has a graphical user interface showing the simulation data in a graphical way. It is available for X-Windows on Sun3 workstations and for MS-Windows.

4.4 Comparison to STRESS

This section shows the important differences between STRESS, see section 4.3 and [Audsley et al., 1994], and SiRTiS.

- The most important difference between STRESS and SiRTiS is the specification of deadlines. STRESS only allows hard deadlines for the whole execution time of a process. SiRTiS knows about hard and soft deadlines. It even provides comfortable possibilities to specify them to refer to the time between almost any events that depend on one another.

- STRESS provides possibilities to specify network topologies, but it has only processor-local scheduling. Processes cannot migrate to other processors.

  SiRTiS has a fixed point-to-point network, but the scheduling can work globally. If special network topologies were introduced in SiRTiS, it would be difficult to write general scheduling algorithms which can handle different scenarios.

- The description language in STRESS provides possibilities to describe the processes more detailed than in SiRTiS. On the other hand, the process description in SiRTiS with only loops and different distributions is easier to understand and provides enough flexibility. Apart from that, it is simple enough that its semantic can be checked statically by the simulator, i.e. it can for example be tested whether a process will terminate or not. This can be useful to check, if a scheduling algorithm can work with the scenario.

- STRESS is independent on compilers and linkers, because it uses only its own specification language for both, the process description and the scheduling algorithms. Therefore it can be sold as a stand-alone product.

SiRTiS has to be provided as source code or at least as an amount of header-files and the corresponding object files. Of course this is not suitable for a commercial product. But for scientific experiments it is better, because the scheduling algorithm can use a higher programming language. However, the source code for scheduling algorithms in SiRTiS is longer than that of STRESS, because the programming language is not especially designed for the specification of scheduling algorithms.

- The modularization in SiRTiS is more consequent. Scheduling algorithms and scenarios are completely divided. This provides a more structured way to compare different scheduling algorithms with the same scenario.

- SiRTiS provides facilities to check the matching of a scheduling algorithm to the scenario. STRESS does not really need this, because the algorithm and the scenario are specified together.

- STRESS has a graphical user interface showing the simulation data in a graphical way. SiRTiS works completely in text mode and should run on any system which provides the corresponding compiler.

  A graphical presentation of the simulation data can be understood much easier, whereas a textual output provides more specific information.

5 Implementation

5.1 General implementation decisions

This section lists two general implementation decisions of SiRTiS.

- All numbers are natural numbers between 0 and $2^{31}-1$. No floating point numbers are used in this simulator!

- The Programming language:

  The whole simulator is written in Ada 95.

  It has been developped with the Gnu-Ada-Translator GNAT for Ada 95 and has been tested on Sun-OS, Linux for PC, and on MS-DOS. There should not be any severe problems in using it with another compiler or on another operating system. [Barnes, 1996] gives an easy introduction to Ada 95 which should suffice for most programmers. [Cohen, 1996] describes the whole language standard and expects the reader to be very familiar with imperative programming languages.

5.2 Scenario description

This section presents the small language that is used to specify a process scenario, i.e. the hardware of the system, the processes, and the deadlines.

An example of a process description can be found in section 6.1 on page 45.

For this presentation a kind of extended Backus-Naur form, EBNF, is used. It is composed of the following symbols:

$\langle name \rangle$	'name' is a token
$[x]$	x occurs zero or one times
$\{x\}^*$	x occurs n times, $n \geq 0$
$\{x\}^+$	x occurs n times, $n \geq 1$
$\{x\}^{a \ldots b}$	x occurs n times, $a \leq n \leq b$
`name`	'name' must appear as written here
$\langle a \ldots z\_0 \ldots 9 \rangle$	a letter, an underbar or a number
$\langle 0 \ldots 2^{31}-1 \rangle$	a number between 0 and $2^{31}-1$
$\{x \mid y\}$	either x or y

Here are general declarations that are used later:

$$
\begin{array}{lcl}
\langle Name \rangle & ::= & \langle a \ldots z\_ \rangle \{ \langle a \ldots z\_0 \ldots 9 \rangle \}^{0 \ldots 31} \\
\langle NameList \rangle & ::= & \langle Name \rangle \; [\texttt{,} \; \langle NameList \rangle] \\
\langle SemaphoreName \rangle & ::= & \langle Name \rangle \\
\langle SignalName \rangle & ::= & \langle Name \rangle \\
\langle ProcessName \rangle & ::= & \langle Name \rangle \\
\langle ProcessNameList \rangle & ::= & \langle NameList \rangle \\
\langle LabelName \rangle & ::= & \langle Name \rangle \\
\langle NatNr \rangle & ::= & \langle 0 \ldots 2^{31}-1 \rangle \\
\langle PosNr \rangle & ::= & \langle 1 \ldots 2^{31}-1 \rangle \\
\langle Dist \rangle & ::= & \texttt{dist normal (}\langle Min \rangle\texttt{,} \; \langle Max \rangle\texttt{,} \; \langle MeanValue \rangle\texttt{,} \\
 & & \qquad \langle StandardDeviation \rangle\texttt{)} \\
 & \mid & \texttt{dist exp (}\langle Min \rangle\texttt{,} \; \langle Max \rangle\texttt{,} \; \langle MeanValue \rangle\texttt{)} \\
 & \mid & \texttt{dist uniform (}\langle Min \rangle\texttt{,} \; \langle Max \rangle\texttt{)} \\
\langle Time \rangle & ::= & \langle NatNr \rangle \\
\langle TimeDist \rangle & ::= & \{ \langle NatNr \rangle | \langle Dist \rangle \} \\
\langle TimeDistSig \rangle & ::= & [TimeDist] | \texttt{signal} \; \langle SignalName \rangle \\
\langle TimeDistSigVal \rangle & ::= & \texttt{(}\langle TimeDistSig \rangle\texttt{,} \; \langle NatNr \rangle\texttt{)} \\
\langle Comment \rangle & ::= & \texttt{\#}\langle any\_characters\_till\_end\_of\_line \rangle
\end{array}
$$

$\langle Dist \rangle$ describes a distribution function as introduced in section 4.1.2.

Now the four parts, that can be specified, are presented:

$$
\begin{array}{lcl}
\langle Specification \rangle & ::= & [\langle Node \rangle] \; [\langle Processor \rangle] \; [\langle Semaphores \rangle] \\
 & & \qquad \langle Processes \rangle \; \{ \langle Deadline \rangle \mid \langle Log \rangle \}^*
\end{array}
$$

5.2.1 Specifying the Hardware

This part presents the description of the system, that the processes are running on. Above all, it contains the amount of *nodes* and *processors* on each node.

The `signaltime` specifies the time a signal takes from one processor or node to another, the `switchtime` specifies the time a task switch to another processor takes. If the new process comes from another node, the switchtime for nodes is used.
The speed is a percentage value n that makes it possible to raise or lower the system speed. It is implemented in a way that all the time values in the deadlines are multiplied by $\frac{n}{100}$.

$$\begin{array}{lcl} \langle \mathit{Node} \rangle & ::= & \texttt{node}\ [\texttt{amount}\ \langle \mathit{PosNr} \rangle]\ [\texttt{signaltime}\ \langle \mathit{Time} \rangle] \\ & & \qquad [\texttt{switchtime}\ \langle \mathit{Time} \rangle]\ [\texttt{speed}\ \langle \mathit{PosNr} \rangle]\ \texttt{;} \\ \langle \mathit{Processor} \rangle & ::= & \texttt{processor}\ [\texttt{amount}\ \langle \mathit{PosNr} \rangle]\ [\texttt{signaltime}\ \langle \mathit{Time} \rangle] \\ & & \qquad [\texttt{switchtime}\ \langle \mathit{Time} \rangle]\ \texttt{;} \end{array}$$

5.2.2 Declaration of Semaphores

The names of *semaphores*, and their initial value which defaults to zero have to be declared. The signals are implicitly declared in the process descriptions.

$$\begin{array}{lcl} \langle \mathit{Semaphores} \rangle & ::= & \texttt{semaphore}\ \langle \mathit{Semaphore} \rangle\ \{\texttt{,}\langle \mathit{Semaphore} \rangle\}^{*}\texttt{;} \\ & & \qquad [\langle \mathit{Semaphores} \rangle] \\ \langle \mathit{Semaphore} \rangle & ::= & \langle \mathit{SemaphoreName} \rangle\ \{\langle \mathit{PosNr} \rangle\} \end{array}$$

5.2.3 Specifying the Processes

The largest part of the scenario description specifies the processes.

$$\begin{array}{lcl} \langle \mathit{Processes} \rangle & ::= & \{\langle \mathit{Process} \rangle\}^{+} \\ \langle \mathit{Process} \rangle & ::= & \texttt{process}\ \langle \mathit{ProcessNamelist} \rangle\texttt{;} \\ & & [\texttt{node}\ \langle \mathit{PosNr} \rangle\texttt{;}] \\ & & [\texttt{processor}\ \langle \mathit{PosNr} \rangle\texttt{;}] \\ & & [\texttt{priority}\ \langle \mathit{PosNr} \rangle\texttt{;}] \\ & & \texttt{start}\ [\texttt{reentrant}] \\ & & \qquad \{\texttt{aperiodic signal}\ \langle \mathit{NameList} \rangle \\ & & \qquad \ \ \mid \{\texttt{periodic}|\texttt{aperiodic}\}\ \langle \mathit{TimeDist} \rangle\}\texttt{;} \\ & & \langle \mathit{Program} \rangle \\ \langle \mathit{Program} \rangle & ::= & \texttt{begin} \\ & & \{\{\langle \mathit{LabelName} \rangle\texttt{:}\}\ \langle \mathit{Action} \rangle\texttt{;}\}^{*} \\ & & \texttt{end;} \end{array}$$

The `node`– and `processor`-entry specify if this process can only be run on one special node or even only on one special processor.

The optional `priority` is not used by the simulator. It is intended for scheduling algorithms, which need it. The rule, that a higher number stands for a higher priority, should be obeyed.

The `start`-line describes the time when the process will be created and then has to be scheduled. If the process is `reentrant`, it may be running more than once, otherwise an exception is raised in the simulator if it is started before the previous instance of it has finished. A `periodic` process is, in contrary to an `aperiodic`, created at the specified time after the last creation. This is the reason why it cannot be started by a signal, because the signal would implicitly have to be periodic.

The part between `begin` and `end` describes the process' behaviour. Every command is executed exactly once and after the last command the process terminates. The labels are used to describe the exact range of deadlines, see below.

An $\langle Action \rangle$ can be one of the following:

$$
\begin{array}{lcl}
\langle \mathit{Action} \rangle & ::= & \texttt{work}\ \langle \mathit{TimeDist} \rangle \\
& | & \texttt{wait}\ \langle \mathit{SemaphoreName} \rangle\ [\langle \mathit{PosNr} \rangle] \\
& & \quad \{\texttt{,}\ \langle \mathit{SemaphoreName} \rangle\ [\langle \mathit{PosNr} \rangle]\}^* \\
& | & \texttt{signal}\ \langle \mathit{SemaphoreName} \rangle\ [\langle \mathit{PosNr} \rangle] \\
& | & \texttt{send}\ [\texttt{delay}\ \langle \mathit{Time} \rangle]\ \texttt{(}\langle \mathit{SignalName} \rangle\texttt{,} \\
& & \quad \{\langle \mathit{ProcessName} \rangle | \texttt{client}\}\texttt{)} \\
& | & \texttt{receive}\ \langle \mathit{SignalName} \rangle \\
& | & \texttt{alarmloop} \\
& & \quad \texttt{(}\langle \mathit{TimeDist} \rangle\texttt{,}\ \langle \mathit{SignalName} \rangle\texttt{,}\ \langle \mathit{ProcessName} \rangle\texttt{)} \\
& | & \texttt{abort} \\
& | & \texttt{loop}\ [\langle \mathit{PosNr} \rangle\ |\ \langle \mathit{Dist} \rangle] \\
& | & \texttt{endloop}
\end{array}
$$

`work` stands for any calculations the process has to do. The $\langle TimeDist \rangle$ stands of course not for system time but for execution time, i.e. the time that the process is scheduled on a processor.

`wait` and `signal` block respectively release one or more semaphores. The number $\langle PosNr \rangle$ stands for the amount of that semaphore. To help avoiding deadlocks, `wait` can lock different semaphores at once. Thus no semaphore is blocked until all the semaphores are available. There are no delay times for semaphores although they are global and can be used on different processors and nodes.

`send` and `receive` are used for signals. The optional `delay` is added to the time the signal takes to be transferred. The special process name `client` is used in aperiodic processes, that are created by receiving a signal. `client` refers to the parent or client process, i.e. the process that has sent the first signal of the list of starting signals in the ⟨*Process*⟩ description. Signals are sent asynchronously and received synchronously, i.e. that the sending process directly continues and the receiving process has to wait until a signal comes.

The command `alarmloop` is used to install an alarm that periodically sends a signal to a process. This can be very useful to simulate events coming from the outside world. An alarm will never finish, so care has to be taken not to put an alarm into a loop with too many iterations because each iteration will install a new alarm. An alarm could also be simulated by a process that sends this signal. But then it would be dependent on the scheduling of that process.

`abort` stops the whole simulation. This can be used to simulate a scenario that is not infinite.

`loop` starts a loop with ⟨*PosNr*⟩ (∞ if omitted) iterations up to the corresponding `endloop`. The number of iterations can also be a distribution.

5.2.4 The Deadlines

The way that deadlines are specified might seem a bit complicated. It is therefore recommended to have a look at the example in section 6.1 on page 45.

In addition to the deadline of one single process, almost any deadline between two events can be given. Hard and soft deadlines are possible, and signals can be sent to start an action in case a deadline is missed.

First of all, here is the way, that the start and end point of a deadline are specified:

$$\langle \mathit{LabelName} \rangle \quad ::= \quad \langle \mathit{ProcessName} \rangle \texttt{.}\{\texttt{start} \mid \texttt{end} \mid \langle \mathit{LabelName} \rangle\}$$

The reserved labels `start` and `end` refer to the creation resp. the end of the process. ⟨*LabelName*⟩ represents a label in front of any of the commands of the process. It works in the same way as a label in imperative programming languages. It devotes the time before that action. For example, a label in front of a `receive`-command means the time just before the `receive`, i.e. before the process might become blocked due to the waiting for that signal.

Now it is possible to introduce the syntax for the deadline description:

$$
\begin{array}{lcl}
\langle \mathit{Utility} \rangle & ::= & \texttt{(}\langle \mathit{Time} \rangle\texttt{,}\ \langle \mathit{NatNr} \rangle\texttt{)}\ [\texttt{,}\ \langle \mathit{Utility} \rangle] \\
\langle \mathit{ProcessDead} \rangle & ::= & \texttt{process}\ \langle \mathit{ProcessName} \rangle \\
\langle \mathit{GeneralDead} \rangle & ::= & [\langle \mathit{DeadlineName} \rangle] \\
 & & \quad \texttt{from}\ \langle \mathit{LabelName} \rangle \\
 & & \quad [\texttt{via}\ \langle \mathit{LabelName} \rangle \{\texttt{,}\ \langle \mathit{LabelName} \rangle\}^{*}] \\
 & & \quad \texttt{to}\ \langle \mathit{LabelName} \rangle \\
\langle \mathit{Deadline} \rangle & ::= & [\texttt{hard} \mid \texttt{soft}]\ \texttt{deadline} \\
 & & \quad \{\langle \mathit{ProcessDead} \rangle \mid \langle \mathit{GeneralDead} \rangle\} \\
 & & \quad \texttt{:}\ \langle \mathit{Utility} \rangle \\
 & & \quad [\texttt{ifmissed}\ \{\texttt{abort} \\
 & & \qquad \mid \texttt{send}\ [\texttt{delay}\ \langle \mathit{Time} \rangle] \\
 & & \qquad\quad \texttt{(}\langle \mathit{SignalName} \rangle\texttt{,}\ \langle \mathit{ProcessName} \rangle\texttt{)}\}]\texttt{;} \\
\langle \mathit{Log} \rangle & ::= & \texttt{log}\ \{\langle \mathit{ProcessDead} \rangle \mid \langle \mathit{GeneralDead} \rangle\}\texttt{;}
\end{array}
$$

The $\langle \mathit{Log} \rangle$ is not a real deadline rather than just a statement to make the simulator log the time elapsed between two events.

The first possibility in a deadline with `utility process` is a short form to describe the overall execution time of one single process, i.e. from `start` to `end`.

$\langle \mathit{Utility} \rangle$ describes the time t_0 of the deadline. For a hard deadline this is enough. In case of a soft deadline the pairs (t_i, u_i) are listed afterwards as a comma-separated list.

The `via`-part is used to describe a deadline which corresponds to events in different processes which send or receive signals from each other. All the listed labels stand for `send`-commands, the corresponding receive commands will automatically be assumed.

The idea of the `via`-part is the following:

- Every time, a process reaches the start-label of a deadline, an instance of that deadline is created and attached to the running process.
- When the process reaches the next label in the `via`-part, the deadline is taken away from that process, and attached to the `send`-command at that label. The process, that receives the signal, will also automatically receive the deadline.
- The next label in the deadline has to be a valid label in the receiving process.

Any labels are allowed in the `from`– and `to`-part, whereas only labels in front of `send`-commands are allowed in the `via`-part. In the `via`-part, no send-to-client-commands may be given, the destination has to be a process-name. This restriction might be removed in future versions of SiRTiS.

If a deadline starts at a `send` command and shall continue at the receiver of this sent signal, the label has to be listed in the `from`– as well as in the `via`-part.

The `ifmissed`-part will be executed during simulation in case the deadline is missed.

5.3 Writing scheduling algorithms

All the procedures belonging to one scheduling algorithm should be put together into a single package. There exist two files, namely `psched_base.ads` and `psched_base.adb`, that can be copied and the copies can be changed accordingly. The data, that the scheduler needs, shall be placed into a record that is derived from `Priv_Environment`, which itself contains all the scheduling data. Every callback procedure will get this derived record as its first parameter. Yet the scheduler cannot access any data in the environment because it is declared as private. Instead it has to call the function `Env` and get the public record `Pub_Environment` that it can inspect, but this is only a copy, so that no changes made to it will be recognized by the simulator. Instead interface-procedures can be called with the whole `Priv_Environment` as parameter.

There is a package `PMain` with a `with`-clause for every scheduling algorithm that is used. The main-program calls the procedure `Read_File` to read in the process specification and afterwards `Run_Simulation` to start it.

`Run_Simulation` will get a record `Abilities` and a record `Scheduler_Functions_Type`.

`Abilities` contains the scheduler's abilities as boolean values and has the following components:

```
type Abilities is
   record
      Hard_Deadline, Soft_Deadline,
            ⇝ can handle hard resp. soft deadlines
        Dynamic_Creation,
            ⇝ processes can be created during runtime
        Dynamic_Finish,
            ⇝ processes can finish during runtime
        Infinite_Processes,
            ⇝ supports processes with an infinite loop
        Variable_Process_Runtime,
            ⇝ supports processes with a distribution at a work– or loop-command
        Internal_Process_Dependencies,
            ⇝ supports processes that send signals not as their last actions, or
```

```
            receive signals, or use semaphores
      Unique_Signals,
         ⇝ this means that every signal is only sent/received at exactly one
            place, which is important to create a control flow graph.
      Multi_Node, Multi_Processor,
         ⇝ supports more than one node or processor
      Needs_Priority
         ⇝ if this value is true, then the scheduler needs a priority specified for
            every process
      :  Boolean := False;
   end record;
```

All the values are set to false by default, so that the scheduler only has to set those values to true that it can handle.

`Scheduler_Functions_Type` contains a set of access-types to procedures that are called in case a special event has occured. This provides a possibility for the scheduler to react on events and do scheduling. To achieve this it has to initialize those values in the record to which it has its own handlers, otherwise the defaults of the simulator are used.

After the call to `Run_Simulation` the simulation will continue until one of the situations below occurs:

- an error occurs
- an `abort` command has been reached
- the scheduler calls `Stop_Simulation`
- the system comes to a stand, i.e. all processes are blocked or have finished and no alarms have been initialized.

After that, the main program may call `Print_Stat` to output statistics about the simulation.

There are 2 exceptions that can occur:

- `Process_Description`: The description of the processes contains an error. An error-message explaining the problem will be printed on the screen.
- `Simulation_Error`: There occured a problem in the simulation. The most common reason is that the scheduling algorithm cannot handle a property of the described processes. An error message will show the details.

  This exception is also raised in case that the scheduler calls a procedure in the package `PSimulator` with wrong parameters or any other non-recoverable situation occurs.

And here are all the callback-procedures that the scheduling algorithm may have:

```
type Scheduler_Functions_Type is
   record
      Initialize                : Sched_Env_Type;
            ⇝ This one is called to give the scheduler a possibility to initialize itself
              before the simulation, in particularly no process is yet created, the
              timetable contains the process creation data. This is quite useful for
              static schedulers that have to do all the decisions here and have the
              other procedures just to execute these decisions.
      Get_Node_For_New_Process: Get_Node_For_Process_Type;
            ⇝ Is called everytime a new process is created. Decides which processor
              and node is to be used by a certain newly created process, the default
              is the first processor on the first node. If the process has a default
              setting, that one is used and the corresponding return values of this
              procedure are ignored.
      Missed_Deadline           : Sched_Env_DID_Type;
            ⇝ When a deadline is missed, this one is called with the deadline's ID
      Process_Blocked           : Sched_Env_PID_Type;
            ⇝ When a process becomes blocked. The simulator automatically ad-
              vances the next process at this processor, so that it will be executed.
              If the scheduler does not want this, it has to select another process
              on this processor or fetch a process from another processor or node.
              If the processor shall be pausing, the procedure Processor_Pausing
              has to be called, but remember to reactivate the processor again.
      Process_Finished          : Sched_Env_PID_Type;
            ⇝ A note that a process has finished. The process has already been taken
              away from the queue of its processor. If it was the running process
              on that processor, the next process is automatically activated. The
              scheduler may change this.
      Process_Unblocked         : Sched_Env_PID_Type;
            ⇝ When a process is running again. Yet it is possible that the process
              is not running on a processor but waiting in the running-queue
      Reached_Processor         : Sched_Env_PID_Type;
            ⇝ The specified process coming from another processor has reached its
              destination.
      Alarm_Handler             : Sched_Env_Type;
            ⇝ Is called in case the scheduler has invoked Call_Me_Back to change
              something after a given time period. It is an error to call Call_Me_Back
              without having an own handler here.
   end record;
```

5.4 The different packages

The whole simulator consists of a set of packages whose names begin with an uppercase letter 'P'. There is a list of basic packages that are used by the other ones at many places:

PTypes:

Here, the basic types of the simulator are defined. The only important point to mention here is the declaration of the function `"+"`. It just returns an access-type to the string that is given as a parameter. This function is used for all initialization of strings with a variable length.

PVar_Array:

This generic package provides an array of any element type. It has a variable length and is automatically extended if the desired index, given in the procedures to get or to set an element, is too high. There are also functions to get an element, but they can only have in-parameters in Ada and thus cannot change the array-length. They will raise an exception if the index is too high.

This array is used at many places in the simulator so that there are principally no restrictions in array lengths in the simulator.

PQueue_Stack:

This package is also generic and contains records that implement variable-size queues and stacks that extend automatically when filled with elements. This package was supposed to be used by the simulator but was not needed afterwards, maybe it is of any use to a special scheduling algorithm.

PEnglish:

All the texts for error and logging messages are stored here. These texts are used by `PError` when printing out text. Thus it is very easy to change the language of the simulator to another that English.

The only exception are some output messages printed in `pmain.adb` and in `psimulator-print_stat.adb`.

PError:

Here are routines to output error messages or logging information. The procedures usually receive the index of the error-message and up to 3 strings that are printed after each of the three parts of the message. This makes it possible to print whole sentences containing up to three variable parts.

The routines to print out an error in the process description receive an additional parameter telling the line and column where the error occured in the file.

PString_Storage:

This package provides possibilities to store all strings and returning a key, a so-called `String_Index`, that can be used to extract the original string. Equivalent strings automatically get the same key.

The string storage is used in order to have the same representation for the same strings, that occur in the process description. So, instead of handling strings in the whole simulator, only the keys are used. In this way, a string comparison of for example process names is only a comparison of two keys.

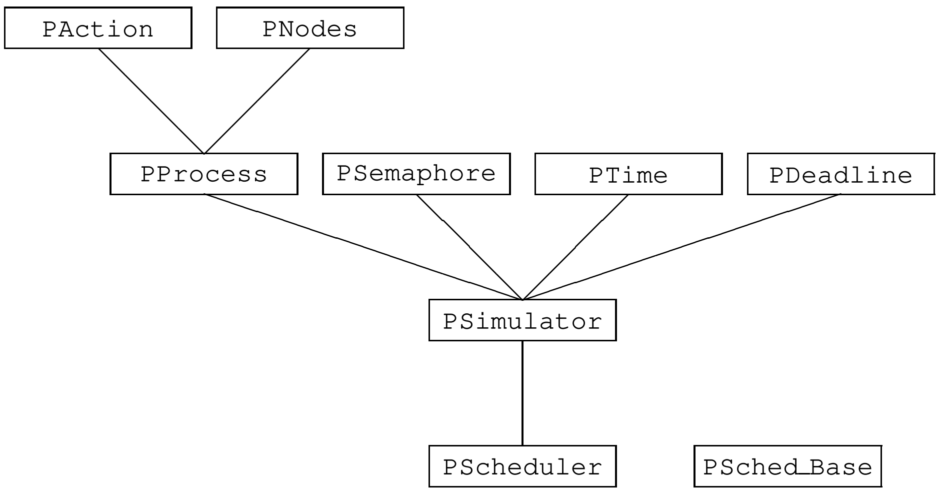
The strings are stored in a hashtable of size n with each entry containing a variable length list of strings with their corresponding keys.

PLex:

This package provides facilities to read in the process description file and to return the next input token. It does not recognize the grammar, but only the keywords and symbols. In other words, it is no parser but rarely a scanner. The parsing is done by the other packages that store the corresponding data, so, for example `PProcess` parses the description of a process, `PAction` that of a single command, an action, within a process and so on.

The record component `Success` is set to false when a scan– or parse-error has been encountered. This makes it possible to have just the Lex-structure and the internal data structure that has to be filled as parameters to the various procedures. Otherwise a success parameter or an exception would be needed to tell the caller that something went wrong, and this would make it more difficult to continue parsing also after an error occurence, the program would be much more difficult to understand.

The following packages build the sceletton of the simulator:



Here is a small overview of their functionality:

PAction:

The record `Action` is used to store one single command of a process, e.g. a command to send a signal. The `Action_Array` contains an array of actions that contains all actions of one process.

PNodes:

Here, the system information is stored, i.e. the system speed, the amount of nodes and processors on each node, the time that a signal needs to reach the receiver, and the time to move one process to another processor or node.

PProcess:

This package declares several record-types:

- `Process_Type` contains the definition of one process and also an array of signals that have reached this process and thus can be received by running instances of that process.

  `Processes` is an array of type `Process_Type`.

- `Run_Process` is used to store all information for a running instance of a process. An `Index` references to the corresponding description in the `Processes`-array.

  `Run_Processes` is the according array for a list of running processes.

- `Processor_Queue` contains the queues of running and blocked processes on one processor. It is also used to log the processor load and to define whether the processor is pausing. The latter might be used by a round-robin scheduler.

  `PID_Array3` is an array of arrays of `Processor_Queue`. It contains the state and the process queues for each processor on every node.

PSemaphore:

A `Semaphore` stores one semaphore's name and its current amount.

`Semaphores` represents an array with element-type `Semaphore`.

The exception `Undeclared_Semaphore` is raised in case a semaphore with an invalid name is used.

PTime:

This package provides the record `Sim_Timer` that is used for the numbers that can also be specified in terms of a distribution. `Peek_Time` provides the next value without deleting it and `Get_Time` provides and after that deletes a value so that the next value will be a new random number.

To make the simulator behave in the same way at every run, the initialization of the random number generator in the body of `PTime` has to be commented out.

PDeadlines:

The record `Deadline` holds all information specified for one deadline. The from– the via– and the to-part are stored together in the array `Where` with the first element designating the from-part and the last one for the to-part. If the array-length is only two, there is no via-part.

The record is also filled with logging information.

PSimulator:

Unfortunately, this package has become very big, but it did not seem useful to divide it into several packages. It contains all the data and routines to run the whole simulation.

First of all the different records:

Simulation_Spec: This one gathers all the specifications in the process description. It is only filled once and not changed thereafter.

Pub_Environment: It extends `Simulation_Spec` with all the data of a running simulation that the scheduler should also be able to read. Especially the following data structures are contained:

Virtual_Time: A natural number representing the time in the simulated system, it starts at 0.

Timetable: An array of `Events`, sorted after their planned arrival time. Everything in the simulation that cannot be executed directly but has to happen at a later time, is represented as an event. For example a signal takes some time to receive its recipient and therefore there exists an event that tells when it shall be put into the signal queue of the receiving process.

The private procedure `Next_Step` processes all the next events in the timetable, that occur at the same time, yet only one after another.

Priv_Environment: All information that is used during the simulation, but which is unimportant for the scheduler, is stored in this record.

It actually contains the `Pub_Environment`, but this is not visible to the scheduler. Thus the scheduler can get the `Priv_Environment` as an in-out parameter without being able to destroy the data in it. Instead it has to use procedures within `PSimulator`, that check the semantic of the call and raise a `Simulation_Error` when something is wrong. However, to provide arbitrary read-access to the simulation data, the function `Env` returns a `Pub_Environment` out of a `Priv_Environment`.

Of course, any changes made to this copy do not affect the original `Priv_Environment`.

The two records `Abilities` and `Scheduler_Functions_Type` were described in 5.3 on page 34.

PScheduler

This package just declares the record `Scheduler_Base` that has to be filled by the scheduler with its own description. First of all a specification of its name is needed, eventually also a filename for a logfile, if the name is too long or contains characters that are not allowed for filenames. It also has to state its ablilities concerning the handling of different scenarios, the callback functions that the simulator has to call, and last but not least its own environment. This may be just an instance of `Priv_Environment`, but as well a derived type to store additional information that the scheduler needs.

The scheduler should not use any static variables to store its data but only the environment, so that it is reentrant. This makes it possible to enhance SiRTiS to run several simulations at one time. It has been developped with regard to this, the only static variable within the whole simulator is the random-number generator in `PTime`.

PSched_Base

It is recommended to use a separate package for every different scheduling algorithm.

`psched_base.ads` and `psched_base.adb` are two files that can be copied and changed afterwards to form a new scheduling algorithm. This makes it easy to remember all the details that have to be specified in a new scheduling algorithm. After that, only a with-statement for this new scheduler has to be added in `pmain.adb` and the record `Scheduler` filled by the scheduling algorithm has to be added in the array `PMain.Scheds`.

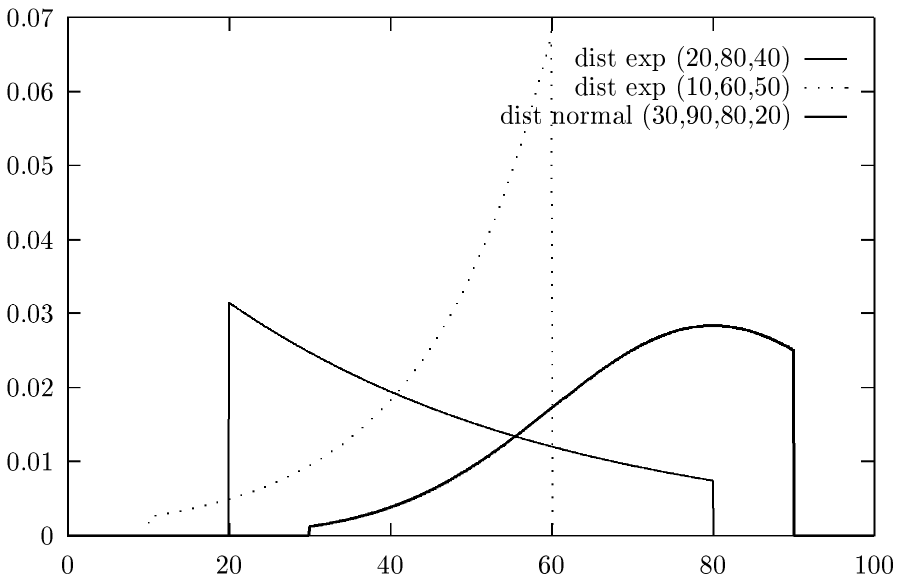


Figure 1: Examples for the density functions of some distributions

5.5 The distributions

For the uniform distribution there is no problem in creating a random value. The other two possibilities are a little more tricky. In general the condition $min < mean < max$ is assumed, and $R \in [0,1]$ stands for a random value with a uniform distribution provided by the system.

Figure 1 shows some examples of distributions.

Exponential distribution:

For this distribution, a denstity function of the form $f(x) := \varphi \cdot e^{\theta x}$ is used where x is the actual value. It is defined between min and max. Let $F(x) := \int_{min}^{x} f(x)\, dx$. Then $F(mean) = \frac{1}{2}$ and $F(max) = 1$. This implies that θ is $\begin{cases} < 0 & \text{for } mean < \frac{min+max}{2} \\ = 0 & \text{for } mean = \frac{min+max}{2} \\ > 0 & \text{for } mean > \frac{min+max}{2} \end{cases}$.
If $\theta = 0$, the exponential distribution is the same as the uniform distribution.

In order to avoid an overflow, first the minimum or maximum is changed, if necessary, so that $\frac{1}{5} \leq \left|\frac{max-mean}{mean-min}\right| \leq 5$. This cuts off those values that are very unprobable, but ensures that no overflow will occur in the following calculations. After that, a transformation is made to guarantee, that the function values remain in the representable range of floating point numbers:

$$\begin{array}{lll} \gamma & := & \frac{1}{max - min} \\ t & := & \frac{x-min}{\gamma}, \quad 0 \leq t \leq 1 \\ c & := & (mean - min) \cdot \gamma \\ g(t) & := & \alpha \cdot e^{\beta t} \\ G(t) & := & \int_0^t g(t)\, dt = \frac{\alpha}{\beta} e^{\beta t} - \frac{\alpha}{\beta} \end{array}$$

$g(t)$ is a new density function, and $G(t)$ gives the probability to get a random value between 0 and t. If $c = \frac{1}{2}$, the rest is skipped and an equal distribution is used. Otherwise the following calculations are made to calculate the values for α and β:

$$\begin{array}{ll} & G(c) = \frac{1}{2} \;\wedge\; G(1) = 1 \\ \Longrightarrow & 2 \cdot \frac{\alpha}{\beta} e^{\beta c} - 2 \cdot \frac{\alpha}{\beta} = 1 \;\wedge\; \frac{\alpha}{\beta} e^{\beta} - \frac{\alpha}{\beta} = 1 \\ \Longrightarrow & h(\beta) := 2 \cdot e^{\beta c} - e^{\beta} - 1 = 0 \end{array}$$

With a start value of $\beta_0 = \begin{cases} -20 & \text{for } c > \frac{1}{2} \\ 20 & \text{for } c < \frac{1}{2} \end{cases}$ a Newton-iteration is done, i.e. $\beta_{n+1} := \beta_n - \frac{h(\beta_n)}{h'(\beta_n)}$. This gives the β with $h(\beta) = 0$. In order to reach the final $\beta \neq 0$, additional conditions are tested in that loop.

With the condition $G(1) = 1$ the value of α is evaluated as $\alpha := \frac{\beta}{e^{\beta} - 1}$.

A random value x can then be calculated as

$$x := \left\lfloor \frac{\log\left(\frac{R \cdot \beta}{\alpha} + 1\right)}{\beta \cdot \gamma} + min \right\rfloor$$

Gauss' normal distribution:

The usual density function for this is $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ with mean-value μ and deviation σ, but as the function in the simulator is limited to $min \leq x \leq max$, another function is used instead:

$$g(x) := \alpha \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

α is set so that $\int_{min}^{max} g(x)\, dx = 1$, and thus a random value x can be provided given the relation $\int_{min}^{x} g(x)\, dx \leq R \leq \int_{min}^{x+1} g(x)\, dx$.

The search for this value x is done by divide-and-conquer, which results in $\log_2(max - min + 1)$ necessary integrations.

The integration itself is calculated using Romberg's method:

Romberg-integration:

Given the definitions

$$\begin{aligned} T_{0,k} &:= \frac{max - min}{2^k} \cdot \left(\frac{f(min)+f(max)}{2} + \sum_{j=1}^{2^k-1} f(min + \frac{j}{2^k})\right) \\ T_{n,k} &:= \frac{4^n\, T_{n-1,k+1} - T_{n-1,k}}{4^n - 1} \end{aligned}$$

the integral evaluates to $T_{n,0}$. This method is extremely useful as it approximates the integral with an error of maximal $\frac{\int}{2^{n \cdot 2^{n+1}}}$.

The simulator uses a single array to store the old values of $T_{n.k}$ instead of a 2-dimensional array, because the old values $(T_{n-1,k-1})$ are not needed any more. It iterates over this array increasing n in every iteration until the last two calculated integrals $T_{n,0}$ and $T_{n-1,0}$ differ in a value smaller than $\frac{max - min}{10000}$.

6 Examples

6.1 A sample scenario

The following process description demonstrates some facilities available in the simulator to specify a real-time system:

```
stopafter 3000;

semaphore Hardware_Sem 1;

process Hardware_Controller;
  start reentrant aperiodic signal Start_Move;
 begin
   wait Hardware_Sem;
  Begin_Work:
   work 100;
   signal Hardware_Sem;
   send (Ready_Move, client);
 end;

process User_1;
  start aperiodic 0;
 begin
  loop 20;
    work dist uniform (40, 200);
   Hardware_Start:
    send (Start_Move, Hardware_Controller);
    receive Ready_Move;
   Hardware_End:
  endloop;
 end;

process User_2;
  start periodic 600;
 begin
  send (Start_Move, Hardware_Controller);
  receive Ready_Move;
```

```
  end;

soft deadline process User_2: (250, 10), (350, 3);

hard deadline User1_Wait
  from User_1.Hardware_Start
  to User_1.Hardware_End : (250, 10)
                           ifmissed abort;

log User1_Until_Hardware_Action
  from User_1.start
  via User_1.Hardware_Start
  to Hardware_Controller.Begin_Work;
```

This scenario describes a system that uses a kind of hardware that can only be accessed by one process at a time. This might be for example a tape drive to restore a backup.

First of all, the time that the simulation will run, is limited to 3000 time units. That should always be specified, otherwise an infinite simulation might be possible if there is one scheduling algorithm that does not finish it, and no `abort`-command within the simulation is reached.

The hardware is accessed by the process '**Hardware_Controller**' which is started by sending it the signal 'Start_Move'. It is reentrant so that the simulator does not complain about more than one process trying to access the hardware at a time. First it locks the semaphore '**Hardware_Sem**' to get exclusive access to the hardware. 'Begin_Work' is a label that will be explained later. After that, the hardware operation takes 100 time units, then the semaphore is released, and an acknowledge signal is sent to the process that has sent the start signal Start_Move. This method presents the way that remote procedure calls can be written in the process description.

The process '**User_1**' starts at time 0 and thus when the simulation is starting. It iterates 20 times in a loop that contains some work for a time uniformly distributed between 40 and 200 time units, and then a remote procedure call to the Hardware_Controller. The call is expressed as a (asynchronous) `send`-command and a (synchronous) `receive`-command.

'**User_2**' starts periodically after every 600 time units. It only contains one call to access the hardware.

Finally, the timing constraints are specified:

- The soft deadline '**User_2**' specifies, that the whole execution of process User_2 gives the system a utility of 10, if it finishes after 250 time units or earlier. After that, the utility decreases down to 3 at time 350 and remains

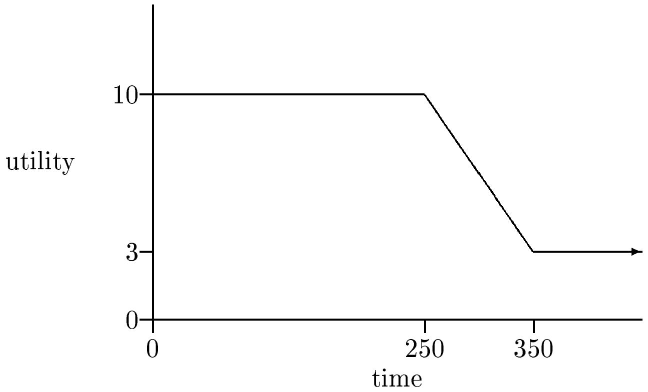


Figure 2: Utility function for `(250,10), (350,3)`

at that value. The graph in figure 2 shows this.

- '**User1_Wait**' is a hard deadline because it has only one timing constraint, '(250, 10)'. The keywords *hard* and *soft* are optional. If the hard deadline has not been finished after 250 time units, the process User_1, that currently owns the deadline, is killed and the optional command is executed. In this case the simulation is aborted. Another possibility would be to send a signal to a process.

- '**User1_Until_Hardware_Action**' is an example for a little more complicated description with a *via*-part. It is just a log, i.e. it only logs the time and thus does not need any utility-specification.

6.2 The earliest-deadline-first scheduler

This example shows a possible implementation of the earliest-deadline-first scheduling algorithm. That algorithm always activates the non-blocked process whose deadline will expire next. This implementation works only for one-processor environments and makes no difference between hard and soft deadlines.

This scheduling algorithm has been programmed by copying the following two files:

`psched_base.ads` to `psched_early_dead.ads`
`psched_base.adb` to `psched_early_dead.adb`

The contents of the copies are modified accordingly. The existence of the incomplete base-files makes it easier to write a new scheduling algorithm because they remind one of every part that has to be specified for this simulator.

First of all the package declaration is presented:

PSched_Early_Dead — Created on 20.02.97

Package declaration for the earliest-deadline-first scheduling algorithm

```
with PSimulator; use PSimulator;
with PScheduler; use PScheduler;

package PSched_Early_Dead is
   pragma Elaborate_Body;    ⇝ necessary, because the package declaration con-
                               tains neither functions nor procedures

   type My_Env is new Priv_Environment with
      record
         null;    ⇝ do not add anything because this scheduling algorithm does
                    not need any additional data
      end record;

   Scheduler:  Scheduler_Base;  ⇝ the data that is used by the main program
                                  to be able to use this scheduler
end PSched_Early_Dead;
```

This file looks very simple. In fact, the only difference that can be seen here between different scheduling algorithms is, that some of them will extend the record `Priv_Environment` with data that they need on their own. All the important scheduling procedures need only be listed in the package body, which looks like this:

```
package body PSched_Early_Dead is
   ⇝ The routines to perform the scheduling are omitted, they are presented after-
     wards.
...........
   ⇝ And here are the most important initialization routines. They have the purpose
     to fill the record Scheduler with the important data needed to use the scheduler.
   This_Env:  aliased My_Env;
begin
   Scheduler.Name := +"Earliest deadline first";
   Scheduler.Filename := +"early";   ⇝ This filename is used if logging into
                                       files is desired.

   ⇝ If This_Env has additional data that needs to be initialized, this is the right
     place. If the initialization is dependent on the process scenario then it has to be
     put into the procedure Initialize.
   Scheduler.Env := This_Env'Access;

   Scheduler.Ab.Hard_Deadline                    := True;
   Scheduler.Ab.Soft_Deadline                    := True;
   Scheduler.Ab.Dynamic_Creation                 := True;
   Scheduler.Ab.Dynamic_Finish                   := True;
   Scheduler.Ab.Variable_Process_Runtime         := True;
   Scheduler.Ab.Internal_Process_Dependencies := True;
```

```
    Scheduler.Funcs.Initialize
      := Initialize'Access;
    Scheduler.Funcs.Get_Node_For_New_Process
      := New_Process'Access;
    Scheduler.Funcs.Missed_Deadline
      := Missed_Deadline'Access;
    Scheduler.Funcs.Process_Blocked
      := Reschedule'Access;
    Scheduler.Funcs.Process_Finished
      := Reschedule'Access;
    Scheduler.Funcs.Process_Unblocked
      := Reschedule'Access;
    ⇝ No handler is needed for Reached_Processor and Alarm_Handler
end PSched_Early_Dead;
```

Here are the routines that perform the scheduling:

- Initialize

  ⇝ The function **Initialize** could be omitted, yet it is listed here to show how it would work. This procedure can perform additional checks that the simulator does not perform to ensure the compatibility between the scheduling algorithm and the scenario.
  In case they do not fit together, an error message with an explanation has to be printed and then the procedure `Stop_Simulation` has to be called. After that, `Initialize` shall return normally.

```
procedure Initialize
  (E:in out Priv_Environment'Class) is
begin
   null;
end Initialize;
```

- Select_Process

  ⇝ **Select_Process** returns the PID (process ID) of the non-blocked process with the earliest deadline. If there are no deadlines it returns the first process, and if there are no runnable processes at all it returns 0.

```
procedure Select_Process (E: Priv_Environment'Class;
                          Next_PID: out PID_Type) is
   Earliest_Deadline :  Natural := Natural'Last;
   This_Soft,
     This_Hard       :  Natural;
   This_PID          : PID_Type;
begin
   for N in 1 ..  Length(Env(E).Proc_Queues(1)(1)
                         .Running)
   loop
      This_PID
        := Get(Env(E).Proc_Queues(1)(1).Running, N);
      Nearest_Deadline_Time
        (E, This_PID, This_Soft, This_Hard);
      if This_Soft > 0 and This_Hard > 0
```

```
      then This_Soft := Integer'Min(This_Soft,
                                    This_Hard);
      else This_Soft := Integer'Max(This_Soft,
                                    This_Hard);
      end if;
      if This_Soft > 0 and  This_Soft < Earliest_Deadline
      then
         Earliest_Deadline := This_Soft;
         Next_PID := This_PID;
      end if;
   end loop;
   if Earliest_Deadline < Natural'Last
   then
      return;
   elsif Length(Env(E).Proc_Queues(1)(1).Running) > 0
   then
      Next_PID
        := Get  (Env(E).Proc_Queues(1)(1).Running, 1);
   else
      Next_PID := 0;
   end if;
end Select_Process;
```

◦ New_Process

⇝ **New_Process** is called, when a new process is created. It checks if this process has the earliest deadline and sets `Shall_Be_First` accordingly.

```
procedure New_Process (E: in out Priv_Environment'Class;
                       Process_ID: PID_Type;
                       Node, Processor:  out Positive;
                       Shall_Be_First:  out Boolean) is
   Min_PID : PID_Type;
   Min_Soft, Min_Hard,
     This_Soft, This_Hard :  Natural;
begin
   Node := 1;        ⇝ default settings for node and processor
   Processor := 1;
   Select_Process (E, Min_PID);
   ⇝ Select_Process only looks for processes on the first processor and
     thus does not see this new one, because it is not yet assigned to any
     processor. It is therefore necessary to check if the new process has a
     earlier deadline.
   if Min_PID = 0
   then
      Shall_Be_First := True;  ⇝ this is the only runnable process
      return;
   end if;
   Nearest_Deadline_Time (E, Min_PID, Min_Soft, Min_Hard);
   Nearest_Deadline_Time (E, Process_ID,
                          This_Soft, This_Hard);
   if Min_Soft > 0 and Min_Hard > 0
   then Min_Soft := Integer'Min (Min_Soft, Min_Hard);
```

```
    else Min_Soft := Integer'Max (Min_Soft, Min_Hard);
    end if;
    if This_Soft > 0 and This_Hard > 0
    then This_Soft := Integer'Min (This_Soft, This_Hard);
    else This_Soft := Integer'Max (This_Soft, This_Hard);
    end if;
```

⇝ Now Min_Soft and This_Soft contain the next deadline time, or zero, if there is no deadline.

```
    if This_Soft > 0 and This_Soft < Min_Soft
    then
       Shall_Be_First := True;
    else
       Shall_Be_First := False;
    end if;
end New_Process;
```

◦ Reschedule

⇝ **Reschedule** is called every time a process becomes blocked or unblocked or when a process finishes. Then it activates the right process again.

```
procedure Reschedule (E: in out Priv_Environment'Class;
                      Dummy_PID: PID_Type) is
   New_PID: PID_Type;
begin
   Select_Process (E, New_PID);
   if New_PID > 0
     and then (Get(Env(E).Proc_Queues(1)(1).Running, 1)
              /= New_PID)
   then
      Activate_Process (E, New_PID);
   end if;
end Reschedule;
```

◦ Missed_Deadline

⇝ **Missed_Deadline** is just a front end for Reschedule for missed deadlines.

```
procedure Missed_Deadline
  (E:in out Priv_Environment'Class;
   DID: DID_Type) is
begin
   Reschedule (E, 0);
end Missed_Deadline;
```

After the declaration of this new scheduling algorithm it has to be put into the simulation by putting the statement '`with PSched_Early_Dead`' into the file `pmain.adb` and adding the record `Scheduler` to the array `PMain.Scheds`.

6.3 A simulation

This section shows the logging information that is obtained by a sample simulation. The process scenario and the earliest-deadline-first scheduler, that were shown on the last pages, have been used for this simulation.

The output was sent into a log-file, so that only the following information was shown on the screen:

```
Simulating processes in file robot.txt
Using scheduler Earliest deadline first
Logging into file early.log
```

The file `early.log` begins with a lot of logging message, that were printed, when an event occured. All the log messages begin with a number, showing the virtual time, followed by '`Log:`', and the text of the log-message.

The following lines show show a part of log information in the file:

0 Log: Initializing new instance of deadline User1_Until_First_Backup_Scheduled

0 Log: Process User_1 (1) created

170 Log: Initializing new instance of deadline User1_Wait

170 Log: Deadline User1_Until_First_Backup_Scheduled transferred from process User_1 (1) to process Hardware_Controller

171 Log: Process User_1 (1) blocked at instruction 4

172 Log: Process Hardware_Controller (2) created

174 Log: Log User1_Until_First_Backup_Scheduled used 174 time units

276 Log: Process Hardware_Controller (2) finished.

277 Log: Process User_1 (1) is unblocked again

278 Log: Deadline User1_Wait finished, raising its utility by 10 to the sum of 10

336 Log: Initializing new instance of deadline User1_Wait

⋮

3000 Log: The simulation ends.

The number in parentheses after a process name shows the prosess ID. Every process gets a unique ID, that has to be used by the scheduler to address a running process.

The simulation ends after 3000 time units as specified in the process description, because the deadline User1_Wait has not been missed, and the scheduler does not stop it earlier.

After the simulation, the main program takes the environment, and prints out a summary of the whole simulation. It contains the following information:

- `Simulation finished.`

  The state of the simulation is *finished*. This indicates that the following logging summary was printed after the simulation.

- `Overall achieved utility: 146`

  This is the sum of the utilities of the deadlines.

- ```
  Processor loads:
   Node 1, Processor 1: 99%
  ```

  This shows that the system was running with full load. The last 1% has been used for process communication and process creation.

- ```
  Deadlines:
   User_2: 2 missed, 2 kept, utility = 26,
       average time = 294, min= 204, max= 389
   User1_Wait: 0 missed, 12 kept, utility = 120,
       average time = 142, min= 108, max= 212
   User1_Until_First_Backup_Scheduled:
       1 measured events,
       average time = 174, min= 174, max= 174
  ```

  This is a summary of the two deadlines and the log, specified in the process description. For each of them, the minimum, average, and maximum time are given. For deadlines, the achieved utility is also shown.

- ```
  1 existing processes:
   User_1 (1) at instruction 2
  ```

  The processes have not all finished, because the simulation was stopped after 3000 time units.

- ```
  1 existing semaphores:
   Hardware_Sem: 1
  ```

  The hardware-semaphore has a value of 1.

7 Future work

There are a lot of aspects in simulating real-time systems, that have not been taken into account in SiRTiS. This appendix gives an overview over the most important extensions that could be added in future releases.

7.1 Possible amendments to the design of the scenario description

- The use of channels for signals would make it possible to send a signal to unknown processes.
- It might be meaningful to add the possibility of specifying network topologies. However, this would make the programming of scheduling algorithms a lot more difficult. There are too many aspects in scheduling a process to a specific processor in a network.
- The whole hardware is assumed to be 100% reliable. It might be possible to specify the occurence of errors in the network. A lot of additional problems for the usage of SiRTiS would be introduced by this.
- The discrete Poisson distribution could be added, if needed.
- Virtual semaphores with preemption could be used to simulate for example virtual memory.
- Semaphores with different types of locking might be provided. This would provide a possibility for a semaphore that allows either one process to write or many processes to read shared date.
- Semaphores could also be assignable to one single node or processor. There has not been found any use for that, as it is also possible to assign a process to a special node or processor.
- A simple programming language with variables, conditional execution and

maybe also some sort of list– or array-processing would enable even more realistic scenarios.

- Use an init process to set an alarm before the scheduling algorithm is initialized

- Make it possible for separate loops within one processor to have exactly the same number of iterations, although it might be a distribution. This would be automatically possible, if a small programming language with variables can be used.

- Allow a send-to-client in the `via`-part of a deadline. This has been forbidden to make the semantic analysis, that proves the description correct, easier.

  For the analysis, there are two cases to be examined. If the deadline came from another process, it has to go back there, i.e. the process names of the labels before and after the actual `via`-label have to be the same. If the deadline has just started, it has to be checked at every possible client.

- Provide a command `sleep`, that makes a process sleep for a while. This means, that the process is blocked for the time period given in the `sleep`-command.

7.2 Semantic control of processes

- The number of processes possibly running at one time should be limited. If, for example, process A creates two processes B which in turn create one process A each, the number of running processes will increase.

- There are a lot of possibilities to perform an analysis for guaranteed schedulability of the processes, i.e. that no deadlines are missed. These tests would substitute a simulation, because a simulation is not needed any more, if a test guarantees schedulability. SiRTiS is intended for the simulation of complex systems with potential overload. Such systems cannot be easily analyzed. Therefore, no analysis for schedulability is implemented in SiRTiS. If it is desired, other tools have to be used.

  [Liu and Ha, 1995] present some possibilities for analysis, that work for restricted process-scenarios. [Stoyenko et al., 1991] present analysis methods that have been developped for compilers for real-time programming languages. These methods have been successfully tested on the language

Euclid ([Kligerman and Stoyenko, 1986]).

- The simulator should contain routines to detect deadlocks and recover from them. Up to now the user has to ensure that the processes are working in a way that no deadlocks can occur.

7.3 Extensions to the scheduler's abilities

- It should be possible to pass arguments to the schedulers via the command-line and/or the process description.
- Until now, the time, that the scheduling itself takes, is not regarded. This is the most important extension for SiRTiS.
- The Scheduler could be enabled to interact with the scenario by for example starting or killing processes. On the other hand, this would introduce a dependency between the scheduler and the scenario and thus decrease modularity.
- The scheduler might be enabled to decide, which process has the right to lock a semaphore or to receive a signal.
- Checks could be made, to detect deadlines that will be missed for sure. This information could be passed to the scheduler to influence its decisions or to enable it to start an actions. It might also be possible to name an action in the process description.

7.4 General aspects to the main program

- Do some sort of memory management with `Unchecked_Deallocation`! Up to now, the simulator relies upon the garbage collection of the compiler, which does not yet exist, the memory just fills up. Future releases of Ada 95 compilers will have garbage collection implemented.
- Provide more command-line options to the simulator. Up to now, every process description, that is listed on the command-line, is used with every scheduling algorithm, currently compiled into SiRTiS.

  The only command line switch is to write logging information into files instead of onto the screen.

8 Summary

SiRTiS provides a powerful and easy-to-handle tool to simulate rather arbitrary real-time systems. The system is described in an abstract way that provides possibilities to specify process behaviours without having to deal with the details of the processes themselves. Only the dependencies between processes have to be specified in detail.

The simulation algorithms are written in a high-level programming language. The simulator provides a comfortable interface to the scheduling algorithms that ensures consistency during the simulation without constraining the possibilities of the scheduler.

The distinction between the process description and the scheduling algorithms makes it easy to exchange them and run various scheduling algorithms on different scenarios.

A Acknowledgements

First of all, I would like to thank my supervisor Anders Törne for the idea and the support for this work. The atmosphere in the RTSLAB has been a very comfortable one.

I am very grateful to Tore Risch, who let me use his ERASMUS-program. A warm *kram* for my parents without whom I would never have been able to come to Sweden. Thanks also to Linköpings university, above all to Anja and Elisabet, for the uncomplicated application as an exchange student and the whole lot of social activities, they have offered me.

There are many others that I must not forget at all. Thanks a lot to Benoit and Holger for their comments on all this, Vild for a whole lot of funny days, Gesine for teaching me the secret of the 'å', Sylvia and Axel for the shaker, LSIF for the best jympa, Johan for the liver, Patrik and Torbjörn for always motivating me, Thomas and Gabrielle for the moskito-tour, Mathias for the telemark-instructions, Rebecca for the funny babysitter-job, Stefan for not understanding, Konrad for the most impressive laugh, Petra for not throwing me out at 3 o'clock in the morning, Pia for the best new-year party, Åsa for the axe, Wolfi and Tina for the orientation-aid, Eriwan and Manni for their 75 years old experience, Tom for his ‚Schinkenhörnle', Alme for saving the future, York for the dork, Kiki for being there when I need her, Wolfgang for fun in the sauna, Armin for the salt in the bath, Sebb for the action on the parties, Långben for the fika all day, Oskar for taking it easy, Linda for the cookies, Samy for longer birthdays, Koppe for the discussions, Uwe for the music in my life, Volkmar for driving, Hartmut for his knowledge, Markus for his e-mails, Sven for his ‚harhar', and last but not least the elks for showing up!

B Parts of the package declarations

These source-code listings present the most important parts of some package declarations. Any source-code lines, that are left out, are marked with a little dotted line.

B.1 The Package PSimulator

PSimulator — Created on 25.11.96 (last Update 19.02.97)

all the data of the simulation is collected here

....

```
package PSimulator is

   Process_Description:  exception; ⇝ This exception is raised by Read_File if
                                       there is an error in the process descrip-
                                       tion.
   Simulation_Error:  exception;      ⇝ This one is raised by different simulator
                                       functions if called with wrong parame-
                                       ters
   ...........

   ⇝ The data structures provided to the scheduling algorithm:
   type Simulation_Spec is tagged     ⇝ The whole process description is stored
                                       here.
      record
         Nodes           :  Node_Type;
         Procs           :  Processes;
         Sem             :  Semaphores;
         Dead            :  Deadlines;
         Stop_Simulation :  Natural:=0;
                ⇝ This is the virtual time when the simulation will be finished, at that
                  time Stop_Simulation will be called. A value of 0 stands for infinite
                  run-time, i.e. until an abort command or until the scheduler stops
                  the simulation.
      end record;
   type Pub_Environment is new Simulation_Spec with
             ⇝ Here all the additional data for running the simulation is stored. Every-
               thing that is not of interest to the scheduling algorithm is deferred to
               Priv_Environment which is in fact a child of Pub_Environment. The
               scheduling algorithm gets read access to Pub_Environment via the func-
               tion Get_Pub_Environment and writa access to Priv_Environment via
```

```
           procedures.
   record
      Virtual_Time     :  Natural;
      Run_Procs        :  Run_Processes;
           ⇝ The scheduling algorithm must not rely on that an index in that
             array will be valid, use the PID of the process instead!!
      Timetable        :  Events;
      Run_Sem          :  Semaphores;
      Alarms           :  Loop_Alarms;
           ⇝ All the looping alarms initiated until now.
      Proc_Queues      :  PID_Array3_Access;
           ⇝ the queues of processes working on the processors
      Utility          :  Integer;
           ⇝ The overall utility that has been achieved by kept deadlines
   end record;
type Priv_Environment is tagged private;
      ⇝ It is important to hide the part of Pub_Environment so that the scheduling
        algorithm cannot make any changes to it.
type Priv_Environment_Access is
   access all Priv_Environment'Class;

⇝ The type Abilities defines what kind of processes this scheduling algorithm can
  manage. By default all values are set to false so that the scheduler just needs to set
  those to true that it can handle.
type Abilities is
   record
      Hard_Deadline, Soft_Deadline,
              ⇝ can handle hard resp. soft deadlines
        Dynamic_Creation,
              ⇝ processes can be created during runtime
        Dynamic_Finish,
              ⇝ processes can finish during runtime
        Infinite_Processes,
              ⇝ supports processes with an infinite loop
        Variable_Process_Runtime,
              ⇝ supports processes with a distribution at a work− or loop-command
        Internal_Process_Dependencies,
              ⇝ supports processes that send signals not as their last actions, or
                receive signals, or use semaphores
        Unique_Signals,
              ⇝ this means that every signal is only sent/received at exactly one
                place, which is important to create a control flow graph.
        Multi_Node, Multi_Processor,
              ⇝ supports more than one node or processor
        Needs_Priority
              ⇝ if this value is true, then the scheduler needs a priority specified for
                every process
        :  Boolean := False;
   end record;
```

⇝ Now come the functions that have to be supplied by the scheduling algorithm. The

scheduling algorithm has to set all the values in `Scheduler_Functions_Type` that it wants to fill with its own procedures.
The procedures have an in-out-argument `Priv_Environment'Class` so that the scheduler can use a derived class of `Priv_Environment` to store some own data it needs in it.
Yet one has to take care only to use this derived type, a CONSTRAINT_ERROR would be raised when converting the type `Priv_Environment'Class` to the derived type.

```
 ..........
type Get_Node_For_Process_Type is
   access procedure (E: in out Priv_Environment'Class;
                     Process_ID: PID_Type;
                     Node, Processor:  out Positive;
                     Shall_Be_First:  out Boolean);
type Sched_Env_PID_Type is
   access procedure (E: in out Priv_Environment'Class;
                     Process_ID: PID_Type);
type Sched_Env_DID_Type is
   access procedure (E: in out Priv_Environment'Class;
                     Deadline_ID: DID_Type);
type Sched_Env_Type is
   access procedure (E: in out Priv_Environment'Class);

type Scheduler_Functions_Type is
```

⇝ If I provide default values here, the program will not be compiled on the GNAT-Compiler for Linux, because the actual values (procedure addresses) can only be set in while linking and that does not work, on GNAT for Sun-OS it is okay.

```
   record
      Initialize              :  Sched_Env_Type;
      Get_Node_For_New_Process:  Get_Node_For_Process_Type;
      Missed_Deadline         :  Sched_Env_DID_Type;
      Process_Blocked         :  Sched_Env_PID_Type;
      Process_Finished        :  Sched_Env_PID_Type;
      Process_Unblocked       :  Sched_Env_PID_Type;
      Reached_Processor       :  Sched_Env_PID_Type;
      Alarm_Handler           :  Sched_Env_Type;
   end record;


procedure Read_File (S: in out Priv_Environment'Class;
                     Filename:  String);
```

⇝ Reads in the process-specifications.

```
procedure Run_Simulation
  (E:in out Priv_Environment'Class;
   Ab:  Abilities; Sched_Func:  Scheduler_Functions_Type);
```

⇝ This procedure runs the whole simulation. It first checks if the scheduling algorithm has the abilities required by the process description and then goes on. The data needed for the evaluation is stored in the `Environment`.

```
procedure Print_Stat (E: Priv_Environment'Class);
```

⇝ Prints out some statistics on the screen. This procedure may also be called during the simulation.

```
procedure Stop_Simulation (E: in out Priv_Environment'Class);
```

⇝ This procedure can be called by the scheduler at any time to stop the simulation. The calling procedure must not perform any changes in the simulation data but just return to its caller, i.e. `Run_Simulation`. In this way no data is destroyed, and so the evaluation of the simulation can be started after `Run_Simulation` has returned with `Success`

```
function Env (E: Priv_Environment'Class) return Pub_Environment;
```

⇝ This function enables the scheduler to have read-access to the data that is important for it.

```
procedure Activate_Process (E:in out Priv_Environment'Class;
                            PID: PID_Type);
```

⇝ If this process is not blocked then it replaces the currently running process on its processor.

```
procedure Call_Me_Back (E:in out Priv_Environment;
                        After:  Positive);
```

⇝ Makes the simulator call the scheduler via its alarm handler after the given amount of simulation time.

```
function Get_String (E: Priv_Environment'Class;
                     Nr:  String_Index) return String;
```

⇝ returns the string for the given `String_Index`

⇝ **Now come some procedures to help the scheduler a little.**

⇝ The following two procedures return the overall runtimes resp. the remaining runtime of the process, supposed that the process will always be scheduled to run and is never blocked:

```
function Get_Run_Process_Index (E: Priv_Environment'Class;
                                PID: PID_Type) return Positive;
```

⇝ returns the process' index in `E.Run_Procs`

```
procedure Get_Runtime (E: Priv_Environment'Class;
                       Name:  String_Index;  ⇝ the process' name
                       Min, Max, Average:  out Natural;
                       First_Action_Nr:  Positive := 1);
procedure Get_Remain_Runtime
   (E: Priv_Environment'Class;
    PID: PID_Type;        ⇝ the PID of a running process
    Min, Max, Average:  out Natural);

procedure Move_Process (E: in out Priv_Environment'Class;
                        Pid:  PID_Type;
                        To_Node, To_Processor:  Natural);
```

⇝ Moves the process to the specified processor, on the old processor. If the process is declared to belong to a node or process other than the parameters, `Simulation_Error` is raised. The next process is automatically activated. This can be changed by the scheduler after the return of `Move_Process`.

```
procedure Nearest_Deadline_Index
  (E: Priv_Environment'Class;
   PID: PID_Type;
   Soft_Ind, Hard_Ind:  out Natural);
```

⇝ Returns the index in the array `Run_Process.Run_Dead` of the next hard and soft deadline that will expire. Soft deadlines that already have expired are not regarded. A 0 means that there is no such deadline. Of course, after the next instruction this index will not be valid any more.

```
procedure Nearest_Deadline_Time (E: Priv_Environment'Class;
                                 PID: PID_Type;
                                 Soft, Hard:  out Natural);
```

⇝ Returns the virtual time of the next corresponding deadline or 0 if there is no deadline left.

```
function Running_Processes (E: Priv_Environment'Class;
                            Name:  String_Index)
                            return PID_Array;
```

⇝ Looks for all instances of that process.

```
procedure Waiting_For (E:Priv_Environment'Class;
                       PID: PID_Type;
                       Running:  out PID_Array;
                       Is_Enough:  out Boolean);
```

⇝ Sets the array `Running` to the PIDs of the processes that will send a signal or release a semaphore that is needed and not available right now by `PID`. If `PID` is not blocked right now then the array is empty.
No selection is taken within the other processes, so it might happen that not all of them have to be scheduled.
`Is_Enough` is set to false if all the running processes cannot satisfy the need.

```
function Process_Will_Release_Semaphore
  (E: Priv_Environment'Class;
   PID: PID_Type;
   Sem_Name:  String_Index) return Boolean;
```

⇝ Returns true if this process will release the given semaphore in the future by a `signal`-command. Returns true even if this process will block it with a `wait`-command before.

```
function Process_Will_Send_Signal (E: Priv_Environment'Class;
                                   PID: PID_Type;
                                   Signal_Name:  String_Index)
                                   return Boolean;
```

⇝ Returns true if the process will send this signal somewhen in future time.

```
procedure Processor_Pausing (E: in out Priv_Environment'Class;
                             Node, Processor:  Positive;
                             Pausing:  Boolean);
```

⇝ Activates/stops this processor. Do not forget to set `Pausing` to false again later, otherwise the processor will never be activated again.

```
function Clients (E: Priv_Environment'Class;
                  Proc_Name:  String_Index) return Name_List;
```

⇝ Returns an array of process-names that are client to this process

```
private
```

All the private functions are skipped, they are not important for the understanding of the functionality of the simulator.

```
...........
    ⇝ This is the private environment that is used by the simulator:
    type Priv_Environment is tagged
       record
          Pub             :  Pub_Environment;
          Sched_Functions :  Scheduler_Functions_Type;
          Is_Running      :  Boolean := False;
          Next_PID        : PID_Type;
                ⇝ The PID that the next process will get
          Next_DID        : DID_Type;
                ⇝ The deadline-ID that the next running deadline will get
          Need_Next_Step  :  Boolean;
                ⇝ is set to true if the simulation shall stop at the next time unit to
                  execute the next command of some process, this saves a whole lot of
                  events that would otherwise be inserted into the event queue
          Stringtable     :  Unique_Strings;
                ⇝ Here are the strings for the String_Index values
       end record;


end PSimulator;
```

B.2 The Package PProcess

PProcess Created on 26.11.96 (last Update 30.01.97)
This package contains the behaviour of all processes.

```
package PProcess is
   package PName_List    is new PVar_Array (String_Index);
   type Name_List    is new PName_List.Var_Array with null record;

   type Signal_Address is
      record
         Name             :  String_Index;
         From_Process     :  String_Index;
         Transferred_Dead :  Run_Deadlines;
      end record;
   package PSignal_Address_Array is
     new PVar_Array (Signal_Address);
   type Signal_Address_Array is new PSignal_Address_Array.Var_Array
      with null record;
```

⇝ This record contains all the data for the specification of a process, i.e. everything that the running instances have in common.

```
   type Process_Type is
      record
         Name                :  String_Index;
```

⇝ The whole `Process_Type` is copied if a process is specified with several names.

```
         Priority            :  Natural := 0;
```

⇝ This is > 0 if a value has been specified. The simulator does not use this value, it is just provided for the scheduling algorithm if it might need it. The absolute value should not be important, but just the relative value compared to the other processes. A higher value stands for higher priority.

```
         Assigned_Node       :  Natural := 0;
```

⇝ if this process has to work on a special node or processor, these values are set to the according number (> 0)

```
         Assigned_Processor:  Natural := 0;
         Is_Reentrant        :  Boolean     := False;
         Is_Periodic         :  Boolean;
         Start_Signal        :  Name_List;
```

⇝ if the array length is > 0 the (aperiodic) process is started by the according signals, otherwise at the given time

```
         Start_Time          :  Sim_Timer;
         Actions             :  Action_Array;

         Received_Signals    :  Signal_Address_Array;
      end record;
      ..........
   Process_Error:  exception;
```

⇝ raised by `Where_Receive_Signal`

```
   function Where_Receive_Signal (P: Process_Type;
                                  Signal:  String_Index)
                                  return Natural;
```

⇝ returns 0 if the signal is a start signal or otherwise the index in the

`Actions` array. If the signal is not received at all, `Process_Error` is raised.

```
function Receives_Signal (P: Process_Type;
                          Signal:  String_Index)
                          return Boolean;
function Sends_Signal (P: Process_Type;
                       Signal, To_Process:  String_Index)
                       return Boolean;
```

⇝ Checks if this process sends the specified signal. If `To_Process` is 0 then the destiny is the client instead of a specified process.

```
type PID_Type is new Natural;
```
⇝ the process identification

⇝ **This is a running instance of a process:**

```
type Process_State is
   (
    Running,
```
⇝ running on a processor, yet the process is only being executed if it is also the first running process in the processor queue
```
    Waiting,
```
⇝ waiting either for a semaphore or a signal
```
    Moving,
```
⇝ currently being moved from one processor to another, there is an event in the timetable that tells the process has reached its destiny
```
    Finished
```
⇝ This process has finished and will be taken out of the list.
```
    );
    ..........
type Run_Process is
   record
      PID              : PID_Type;
      Index            :  Positive;
```
⇝ the index in the array `Processes_Type`
```
      Client           :  String_Index;
      State            :  Process_State;
      Run_Dead         :  Run_Deadlines;
      Assigned_Processor,
        Assigned_Node  :  Positive;
```
⇝ the place where this process is working or will be working after `Moving`
```
      PC               : Positive;
```
⇝ the index of the next command to be executed or the `work` command that is now being executed (program counter)
```
      Rest_Work        :  Natural;
```
⇝ if the process is at a `work`-command this counter is set once and decreased until it reaches 0
```
      Loop_Level       :  Natural;
```
⇝ tells in what level of loops the process is currently executing, is increased at a `loop` and decreased at `endloop`
```
      Loop_Remain      :  Loop_Remain_Array_Access;
```
⇝ Stores the amount of remaining iterations for the different loops
```
   end record;
   ..........
```

⇝ The following arrays are used to store the PIDs of the processes running on a specified processor. The first PID in every `Running/PID_Array` is used for the running process.

```
   package PPID_Array is new PVar_Array (PID_Type);
   type PID_Array is new PPID_Array.Var_Array with null record;
   type Processor_Queue is
      record
         Pausing   :  Boolean := False;
              ⇝ if true, then this processor is not doing anything, useful for e.g. round
                robin scheduling when the running process is blocked
         Work_Time :  Natural := 0;
              ⇝ to log the time that the processor has been working until now
         Running   :  PID_Array;
              ⇝ all the running processes
         Blocked   :  PID_Array;
              ⇝ all the blocked processes waiting for a semaphore or a signal
      end record;
   type Processor_Queue_Access is access all Processor_Queue;
   type PID_Array2 is array (Positive range <>)
     of aliased Processor_Queue;
   type PID_Array2_Access is access PID_Array2;
   type PID_Array3 is array (Positive range <>)
     of PID_Array2_Access;
   type PID_Array3_Access is access PID_Array3;
        ⇝ Now this is a 3-dimensional array of PIDs, i.e. an array of PIDs for every
          processor on every node.
private
..........
end PProcess;
```

C DOCU — Converting source code to LaTeX

All the source code listings in this document have been automatically created from the source code by a small Perl-program named '`docu`'. It reads in all the source files listed on the command line and parses them for commands to create LaTeX-files. The command '`docu -h`' gives a short help.

The output files are stored in the directory '`./docu/`'. This can be changed with the command line switch '`-o` ⟨*dir*⟩'.

Programs with the extension '`.ad?`' are regarded as Ada source, all other programs as C source.

All the commands start with one or two letters directly following the comment sign ('`--`' or '`/*`'). Thus it is important to leave at least one blank after the comment sign if the comment shall not be interpreted as a command for `docu`.

The C++ comment sign '`//`' is ignored.

There are several commands that spread over more than one line in the source code, so-called multi-line commands. For C-code they just end at the corresponding '`*/`', and for Ada at the first line that does not start with a comment or that starts with a command, i.e. a comment sign '`--`' directly followed by a letter.

Here are all the commands that have been added up to now:

fb ⟨*filename*⟩

All the output files that are created by `docu` are held on a stack. This command (file begin) adds a file with the name '⟨*filename*⟩`.tex`' on the top of the stack. This will furtheron be regarded as the *last* file. The *first* file is the one that has been opened first and is thus represented at the bottom of the stack.

The old contents of the output file will be deleted.

fh ⟨*filename*⟩ *(multi-line)*

This command also creates a new output file, but it starts with a header describing its name and optionally the date of the creation and the last update, the author, and its functionality.

For this purpose the LaTeX-macros \doccreate, \docupdate, \docauthor, and \docfunction are defined in the style file `docu.sty`.

fp

Pauses the output into the last file, it is popped from the stack.

fc ⟨*filename*⟩

Resumes the ouput into a file that has previously been paused, it is pushed onto the stack, and therefore becomes the last file.

fe

Ends the output into the last file. After that, the file is ready to be read in into a LaTeX-document with the command '\input {⟨*filename*⟩}'. The style `docu.sty` has to be used.

Everey output file **must** be closed with **this** command.

ca/cf/cl *(multi-line)*

These three commands start a comment. `ca` writes it into all output files, `cf` only into the first one, and `cl` into the last one.

All the rest of the comment is used as LaTeX text. The comment is indented in the following way:

- If the command ca, cf, or cl is directly followed by a non-blank, then the comment is indented to the first character of the comment sign in the source code, i.e. '–' for Ada and '/' for C.
- Otherwise, it is indented as far as the first non-blank in the comment.

ta/tf/tl *(multi-line)*

The following text is directly copied into the output file(s), that means all of them, only the first or only the last. Therefore this text is not indented in any way.

s $\lambda\, X\, \lambda\, Y\, \lambda$

λ may be any character, and X and Y are regular expressions according to Perl rules.

This command substitutes the text X in the code-line preceding this comment by the text Y. It can be used to create bugs in the code-listing to prevent other persons from copying code, however this is not a very serious method.

C.1 Example

The following code shows a little Ada program containing docu-commands:

```
with Ada.Text_IO;    use Ada.Text_IO;

--fh Example
-- \doccreate{12.03.97}
-- \docauthor{Rolf}
-- \docfunction{This is a small example for
-- the documentation program \texttt{docu}
-- that creates \LaTeX{}-files.}
--fb Example2
procedure Hello_World is
   --fp
begin
   --fc Example2
   Put_Line ("Hello world!");
   --caThis is a comment in all output files.
   --cl     This comment appears only
   -- in \texttt{Example2.tex}.
   -- It begins below the opening parenthes.
   --fe
end Hello_World;--caThis comment starts
-- directly after the semicolon.
--fe
```

This code segment creates the two output files:

`Example.tex` and `Example2.tex`

They look like this:

Example.tex

EXAMPLE ———— Created on 12.03.97 by Rolf

This is a small example for the documentation program docu that creates LaTeX-files.

```
procedure Hello_World is
begin
   Put_Line ("Hello world!");
   ⇝ This is a comment in all output files.
end Hello_World; ⇝ This comment starts directly after the semicolon.
```

Example2.tex

```
procedure Hello_World is
..........
   Put_Line ("Hello world!");
```

⇝ This is a comment in all output files.

⇝ This comment appears only in `Example2.tex`. It begins below the opening parenthes.

References

[Audsley et al., 1994] Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1994). Stress: A simulator for hard real-time systems. In *Software–Practice and Experience*, volume 24(6), pages 543–564.

[Barnes, 1996] Barnes, J. (1996). *Programming in Ada 95*. Addison Wesley.

[Burns, 1991] Burns, A. (1991). Scheduling hard realtime systems: A review. In *Software Engineering Journal*, pages 116–128.

[Cohen, 1996] Cohen, N. H. (1996). *Ada as a second language*. McGraw-Hill.

[Gustafsson and Liss, 1985] Gustafsson, L. and Liss, A. (1985). *Modellbyggnad och Simulering i DEMOS*. Uppsala universitet.

[Kligerman and Stoyenko, 1986] Kligerman, E. and Stoyenko, A. D. (1986). Real-time euclid: A language for reliable real-time systems. In *IEEE Transactions On Software Engineering*, volume SE–12, pages 941–949.

[Leinbaugh and Yamini, 1986] Leinbaugh, D. W. and Yamini, M.-R. (1986). Guaranteed response times in a distributed hard realtime environment. In *IEEE Transactions On Software Engineering*, volume SE–12, pages 1139–1144.

[Liu and Ha, 1995] Liu, W. S. and Ha, R. (1995). Efficient methods of validating timing-constraints. In [Son, 1995], chapter 9, pages 199–224.

[Silberschatz and Galvin, 1994] Silberschatz and Galvin (1994). *Operating System Concepts*. Addison Wesley, fourth edition.

[Sjölund et al., 1993] Sjölund, M., Thyselius, R., and Sjölund, B. (1993). *Adam—A Library With Reusable Ada Components For Simulation*. User Manual.

[Son, 1995] Son, S. H. (1995). *Advances In Realtime-Systems*. Prentice Hall.

[Stoyenko et al., 1991] Stoyenko, A. D., Hamacher, V. C., and Hol, R. C. (1991). Analyzing hard real-time programs for guaranteed schedulability. In *IEEE Transactions On Software Engineering*, volume 17, pages 737–750.