Uppsala Master's Thesis
in Computer Science 95
1995-09-01
ISSN 1100-1836

# Discrete Event Simulation in Erlang

Andreas Ermedahl

Department of Computer Systems, Uppsala University
Box 325, S-751 05 Uppsala, Sweden
+46-18-18 25 00

Supervisor: Robert Virding, Ellemtel AB
Examiner: Hans Hansson, Department of Computer Systems, Uppsala University

Passed : Date:

## Abstract

The report will show how process-oriented discrete event simulation can be performed in the programming language Erlang. A small introduction to simulation and Erlang is given. The model used for discrete event simulation in Erlang and mechanisms for interaction in-between simulation processes are described. The author also illustrates how an extension to distributed simulation can be made. Descriptive examples and larger applications are presented. In the last passage of the report function calls for the simulator are given.

# Contents

# 1 Introduction

This project was made as a Master's Thesis in Computer Science. The project took place at Ellemtel in Älvsjö in Sweden. The supervisor during the project was Robert Virding.

## Erlang

Erlang is a declarative language for programming concurrent systems which was developed by Joe Armstrong, Robert Virding, and Mike Williams, at the Ericsson and Ellemtel Computer Science Laboratories.

## Assignment

The assignment was:

- to examine the possibility to make discrete event simulation in Erlang.
- to develop and implement a model for simulation suited for Erlang.
- to show that the developed model can be used in one (or several) applications.
- to examine the possibility to create a distributed simulation model.

I think that all these task have been accomplished. The sections in the report will follow the order of the tasks.

Notice that this document is intended both to be a master theses report and a user manual. Therefore large program examples (for the user) and deep simulation semantics (for the examiner) are both given in the report.

## Background

Discrete event simulation has mostly been implemented on top of object oriented languages. By building the a discrete event package in Erlang, a language with a process-based model of concurrency, it will be easy both to parallelize and distribute the simulation. When the package is written completely in Erlang it will be possible to run the simulation on all platforms where Erlang is ported. The idea of using Erlang in discrete event simulation can be traced back to Robert Virding and Mike Williams.

## Outline

The report is divided into 20 chapters:
1. **Introduction**, this chapter, gives a short introduction of the assignment.
2. **What is Erlang?** gives the reader a short introduction to Erlang.
3. **Simulation**, an introduction to the ideas and concepts of simulation.

4. **Discrete Event Simulation models**, discuss existing parallel and sequential simulation implementations.
5. **Discrete event simulation in Erlang**, explains how simulation can be performed in Erlang.
6. **The eventhandler**, describes how the process that controls the simulation is constructed.
7. **To steer the simulation**, gives a description on how the simulation can be controlled before and during run time.
8. **Processes**, describes how the simulation state is changed by the simulation processes.
9. **Resources**, presents special processes used to collect data and synchronize the simulated processes.
10. **Probability distributions**, shows how different probability distributions can be generated.
11. **A radio traffic simulation application**, describes the first large simulation application I made.
12. **A mining simulation application**, describes the second large simulation application I made.
13. **Distributed simulation**, a description of how the simulation model can be distributed.
14. **Erlang as simulation glue**, explains how Erlang can be used to parallelize and distribute other simulation systems.
15. **AppendixA**, simulator and process commands.
16. **AppendixB**, resource commands.
17. **AppendixC**, probability distributions commands.
18. **AppendixD**, distributed simulation commands.
19. **AppendixE**, explanation of figures.

## Acknowledgments

I would like give acknowledgement to Robert Virding and the rest of the personnel at the Ellemtel Computer Science Laboratory, it has been real fun working with you. The personnel at ERA-t and Ångpanneföreningen gave me big support and inspiration with their simulation applications. I will also give Graham Birtwistle acknowledgement for all the simulation papers that he sent to me, they became a great influence.

# 2 What is Erlang?

This section should give the reader enough knowledge in the programming language Erlang to let him understand the code in this report. If the reader already is familiar with Erlang he can skip this section. For deeper studies in Erlang, the reader is referred to the book 'Concurrent Programming in ERLANG' [Armstrong, Virding, Williams - 93].

## 2.1 Erlang

Erlang is a programming language which was designed for programming concurrent, real-time, distributed fault-tolerant systems. The Erlang programming language has been developed at Ericsson and Ellemtel Computer Science Laboratories. Development started in the early eighties with experiments of programming telecom applications using different languages. The experiments showed that no existing language had both these following features:

- being a high level declarative language
- having primitives for concurrency and error recovery

Therefore, Erlang, a new language with these features, was developed. In the beginning of the nineties the first implementation of Erlang was released. Today, Erlang is available for most operating systems and platforms, and is developed, maintained and marketed by Erlang Systems Division.

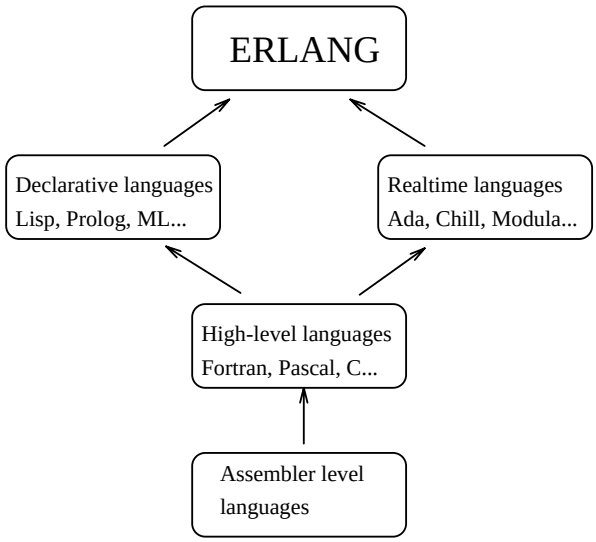



Figure 2.1 *The origin of Erlang*

Erlang is a functional, concurrent language designed for implementing reliable real-time systems. It is a small but powerful language which a person with some knowledge of programming can learn in a few days.

## 2.2 Modules, Functions and Clauses

An Erlang program is divided into modules. A module consists of one or more functions which, in turn, consists of clauses. The functions are hidden inside the modules, except for the exported functions which can be called from the outside.

```
-module(mathlib).
-export([factorial/1]).

factorial(0) ->
    1;
factorial(N) ->
    N * factorial(N-1).
```

The module `mathlib` above consists of one function, `factorial/1`, which means the function has an arity (number of arguments) of 1. Factorial consists of two clauses separated by a semicolon and ended by a full stop. The function factorial is exported and can be called from outside the module. Functions that not are exported are local and can only be called from inside the module.

An external call to `factorial/1` may have the following syntax:

```
mathlib:factorial(3).
```

This call would result in the answer 6.

It is also possible to import functions so that they can be used as if they were local functions. This is done by using `import`.

## 2.3 Data Types

Erlang provides the following data types:

- Constant data types. These are data types which cannot be split into more primitive subtypes. These are:

  1. Numbers: `125, -2.5, 7.8e2`

  2. Atoms: `abc, "kossan mu"`

- Compound data types. These are used to group together other data types. These are:

  1. Tuples: `{},{5.3, 7, abc, -5}`. Tuples are used for storing fixed number of items and are similar to records or structures in conventional programming languages.

  2. Lists: `[], [5.3, 7, abc, -5]`. Lists are used for storing variable number of items. Components of list and tuples can themselves be any Erlang data item - arbritary complex structures can be created.

## 2.4 Pattern Matching

The value of an Erlang data type can be stored in a variable. Variables always starts with an upper case letter. A variable is assigned a value with the `=` operator.

Pattern matching is used for assigning values to variables and for controlling the flow of a program.

```
> X = 1.
1
> Y= {a, b, c}.
{a, b, c}
> Z = [X, Y, foo].
[1, {a, b, c}, foo]
```

Erlang is a single assignment language. This means that once a variable have been assigned a value, the value can never be changed.

```
> X = 1.
1
> X = 2.
error.
```

Pattern matching is used to match patterns with terms. If the term have the same shape then the match will succeed and any variable occurring in the pattern will be bound to the data structure. The match primitive can be used to unpack items from complex structures.

```
> X = {4711, foobar}.
{4711, foobar}
> {A, B} = X.
{4711, foobar}
> A.
4711
> B.
foobar
```

Constructions for choosing between different alternatives exists, for example the `if` and `case` expressions. The syntax for the `case` expression is showed below.

```
case Expr of
    Pat1 -> Seq1;
    Pat2 -> Seq2;
    ...
    PatN -> SeqN
end
```

In the `case` expression the `Expr` is first evaluated and then the pattern which fits the evaluation is selected and the corresponding sequence is run. If no match occurs an error will be reported.

The pattern matching mechanisms are also used when a function is called and one of its clauses shall be chosen. The first clause which matches the specific pattern will always

be selected and the corresponding body will be executed. If no match occurs an error will be reported.

In the example function below `factorial(0)` will only be matched when the argument is `0`, otherwise the second clause will be chosen. When a match occurs the expression on the right-hand-side of the `->` will be evaluated.

```
-module(mathlib).
-export([factorial/1]).

factorial(0) ->
    1;
factorial(N) ->
    N * factorial(N-1).
```

## 2.5 Concurrency

Erlang is a concurrent programming language. This means that parallel activities (processes) can be programmed directly in Erlang and that the parallelism is provided by Erlang and not the host operating system.

In order to control a set of concurrent activities Erlang has some primitives for multiprocessing.

- spawn starts a parallel computation (called a process).
- send sends a message to a process.
- receive receives a message from a process.

`spawn/3` starts execution of a parallel process and returns an process identifier (pid) which may be used to send messages to and receive messages from the process. Spawn has the syntax:

```
Pid = spawn(Module, Function, [Arg1, Arg2, ...])
```

The syntax for sending a message to a process is:

```
Pid ! Message
```

`Pid` is an expression or constant which must evaluate to a process identifier. `Message` is the message which is to be sent to the process that corresponds to `Pid`. `Message` can be any Erlang term.

The primitive `receive` is used to receive messages. Receive has the following syntax:

```
receive
    Pattern1 -> Seq1;
    Pattern2 -> Seq2;
    ...
    PatternN -> SeqN
end
```

A receive expression will suspend the process until a message that will match any of the patterns has arrived. If a match occurs the code after the `->` is evaluated.

Message passing between processes is asynchronous. Send can be said to send messages to the mailbox of a process and receive tries to remove a message from the mailbox. The message in a process mailbox are matched in FIFO order, meaning that receive is selective. If a message does not match it is left in the mailbox until a match for that message is found. Instead the next message in the mailbox is checked.

The `after Time` expression can be used in receive expressions to time-out waiting processes.

## 2.6 Error Handling

Erlang has been designed with mechanisms to handle errors and to help the programmer write robust systems. These mechanisms are:

- Monitoring the evaluation of an expression.
- Monitoring the behaviour of other processes.
- Trapping calls to undefined functions.

Monitoring the evaluation of an expression is done by the primitive `catch` which will "catch" a failure of the expression. The normal effect of a failure in the evaluation of an expression is to cause the process evaluating the expression to terminate abnormally. This default behaviour can be changed using `catch`. This is done by writing `catch Expression`.

If failure does not occur in the evaluation of the expression `catch Expression` the value of the expression will be returned.

If failure occur the tuple `{'EXIT', Reason}` will be returned. `Reason` will be an atom indicating what went wrong. By examine the `Reason` different actions can be made.

```
case catch Expression of
    {'EXIT', Reason} -> Seq1;
    Other -> Seq2
end
```

Monitoring the behaviour of other processes is done by linking processes together. This is done by the functions `link` or `spawn_link` which results in that when a process terminates, an exit signal will be sent to all processes that are linked to that process. Exit signals for processes have the format `{'EXIT', Pid, Reason}` when trapped. If `Reason` is `normal` it means that the process has finished its execution, otherwise it has terminated abnormally. The default behaviour is that a linked process dies if it receives an abnormal exit. If the process instead "traps exits" it can take measures against abnormally terminating processes.

Trapping evaluation of an undefined function is taken care of by an error handler process. The error handler will search for the module and function elsewhere and if the

module not is found the process will terminate and an error message will be returned. If the module is found it will be loaded into the system and the function will be called. The error handler can be rewritten by the user.

## 2.7 External Communication

Erlang communicates with the outside world through ports. A port in Erlang is treated as a process (i.e. we can send messages, link and get exit signals from it) with which bit streams are sent and received. A port can be used for interaction with hardware, window systems or programs in other languages.

## 2.8 Code Management

In Erlang it is possible to load new code at runtime. This dynamic loading of code uses multiple versions, meaning that all calls to the changed module will be to the new version while all processes running on the old version will continue to do so. The old version can be removed when no processes run that version. Erlang also provides functions for killing all processes that runs the old version.

Code management is very useful when working with large applications which cannot be shut down for software updates, for example a telephone system.

## 2.9 Distribution

Erlang was designed from the beginning with concurrency and distribution in mind. This leads to language constructs for distribution and concurrency which are seamlessly integrated with the language.

Distribution in Erlang builds on the node concept. A node is an executing Erlang system which has told its nodename and network address to a network server. The node where a process executes is given with the `node()` call. Processes can be spawned on remote nodes as on local nodes.

```
Pid = spawn(Node, Module, Function, [Arg1, Arg2, ...])
```

Processes can send messages and create links to processes on remote nodes. The sending of a message to a process on a remote node is syntactically and semantically identical to sending to a process on a local node.

It is also possible to register a process globally so that all processes on the nodes in the network can access the process.

At the language level there is no concept of a connection between Erlang nodes. The programmer does not need to be concerned with details of the setup of connections between nodes. Whenever one node needs to communicate with another node the system takes care of setting up a connection.

The coupling of nodes in Erlang is extremely loose. Nodes may come and go dynamically in a manner similar to processes. For deeper studies in Erlangs distributions mechanisms the reader is referred to the article [Wikström -93].

# 3 Simulation

This chapter gives the reader a short introduction to general simulation and a more detailed description of the simulation form called discrete event simulation.

## 3.1 Introduction

Simulation is the process of modeling a proposed or real dynamic system and observing its behavior over time. A simulation study is often constructed in order to understand the behavior of a system, or to evaluate the effects of various parameters or operating policies. The simulation can be made to predict and understand the behavior of everything from a nuclear power station, a chemical reaction to a computer operating system. Simulation can be classified into three basic types [Pollacia -88]: continuous time, discrete time and discrete event. I have myself added a fourth competitor, analytic modeling.

**Continuous time simulation**
: Continuous time simulations arise from models described by ordinary differential or partial differential equations. The state of the system is, in contrast with discrete simulation, continuously evolving.

**Discrete time simulation**
: Discrete time simulations arise from models which can be in the form of difference equations. The state of the system may change through a countable sequence of events, each equally spaced in time.

**Discrete event simulation**
: Discrete event simulation arise from models describing physical systems that changes their state at variable intervals. The system evolves through a countable sequence of events, each occurring at some point in time. The intervals between events are variable and the system can therefore not be solved with a standard difference or differential equation. Several events can occur at the same point (simultaneous events). The occurrence of events can also spawn or cancel other events.

  The discrete event simulation is the type of simulation I have focused on and the one that is most suited for computer applications.

**Analytic modeling**
: Another competitor is analytic modeling, which simply consists in providing a mathematical model of the physical system under study, and solving the model equations.

## 3.2 Discrete event simulation

Discrete event simulation is a widely used modeling technique for studying the dynamic behavior of many real-world systems. Generally these systems are to complex to be

modeled using analytic methods. Examples of discrete event systems include queuing systems and most manufacturing methods.

As previously stated, a discrete event simulation models a system whose state may change only at discrete points in time. The system is composed of objects normally called *entities* that have certain properties called *attributes*. The *system state* is the collection of attributes or *state variables* that represent the entities of the system. An *event* is an instantaneous occurrence in time that may alter the state of the system. An event initiates an *activity*, which is a length of time during which entities engage in some operation. A *process* is a sequence of events that may encompass several activities. Entities, attributes, events, activities, and the interrelationship between these components are defined in the model of the system.

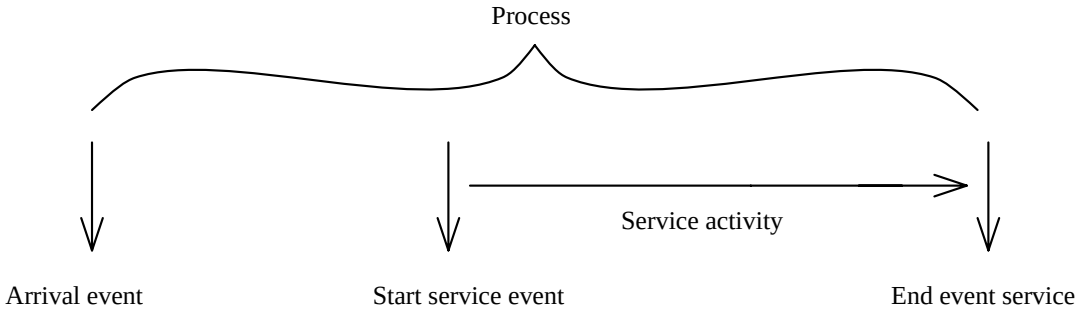



Figure 3.1 *Relation between Process, Event and Activity*

Fundamental to every simulation study is a mechanism to model the passage of time. Thus every model contains a state variable called the *event time*. Simulated time is advanced using two approaches: *fixed-increment* time advance (discrete time simulation), or *next-event* time advance (discrete event simulation).

In my simulator package it is, as I will later show, both before and during the simulation, possible to switch between discrete time and discrete event simulation.

In the fixed-incremented time advance method, the clock is always updated by the same time increment. After each clock update, all events that were scheduled to occur during this interval are identified. These events are considered as if they had occurred at the end of the interval. This method is useful for a system in which events only occur at intervals of some fixed time.

The next-time advance is the more commonly used approach. Time is advanced from the time of the current event to the time of the next scheduled event. The simulation thus skip periods of inactivity. There is a calendar of events that contains the time and type of all scheduled events, usually arranged in chronological order. The simulation program must contain an executive routine for management of the clock and the calendar, eg. a routine responsible for sequencing events and driving the simulation forward.

It is the behavior and construction of this 'executive routine' that has been the real problem when the simulation package was constructed.

# 4 Discrete event simulation models

The first task in my assignment was to investigate the possibility of doing discrete event simulation in Erlang and to develop a model which could be implemented in Erlang. I investigated both techniques for sequential and parallel discrete event simulation before making a decision.

## 4.1 Parallel discrete event simulation

The different parallel and distributed algorithms for discrete event simulation that were examined all builds on the elimination of the globally shared eventlist used in sequential (uniprocessor) simulation. Some algorithm is required to ensure that no event $e$ is simulated until all events that affect $e$ have been simulated. Such algorithms are frequently classified as either *conservative* or *optimistic* algorithms. [Cota, Sargent -88], [Madisetti, Walrand, Messer -89]

### 4.1.1 Conservative algorithms

Conservative algorithms block the simulation of an event until it is verified that the event is safe. The simulated process is only allowed to update its internal clock when all the relevant information is available.

A conservative approach to simulation in Erlang can be made using Erlangs message primitives. If a process P1 simulates an event that influences P2, P1 is said to be the *predecessor* of P2 and P2 will be said to be a *successor* of P1. By calculating, in advance, which processes that will be predecessors and successors of which processes a dependence graph can be constructed. When, in Erlang, a process P wants to perform an event e, it must first receive messages from all the processes that runs events that are predecessors of the event e. When all messages have arrived the process P is safe to proceed with event e. When the process P is finished with event e it must inform all his successors (using messages) that it is ready.

```
%% Conservative parallel simulation in Erlang.
    process_loop([{Event, Action, Predecessors, Successors} | EventList]) ->
        wait_for_messages(Predecessors),
        do_some_action(Action),
        send_message_ready(Event, Successors),
        process_loop(EventList).
```

The big problem is how to construct the dependence graph. These algorithms may be very complicated and time consuming. It may not always be possible to predict, in advance, every action and interference a process may take.

### 4.1.2 Optimistic algorithms

Optimistic algorithms allow events to be simulated without verifying that they are safe, by frequently save the states of the participating processes. When an event is simulated incorrectly, the error is hopefully discovered and the process is restored to the state

before the error occurred. The internal clock is "rolled back" and antimessages are sent to all process that have been affected by the process.

An optimistic approach to simulation can be made using Erlangs message primitives. Every process have an internal clock which is updated by itself. and a set of variables that it is currently aware of. A timestamp is added to every message a process sends which indicates the current time of the sending process. Every action a process performs, the state of the variables at the moment and every message that is sent are saved in a internal list indexed on the current timestamp. When a message arrives with a timestamp smaller than the own internal time of the process, the process must roll back to the state it was in before the time of the timestamp. All messages that have been sent to other processes must be unsent with antimessages that rolls back the state of those processes.

```
%% Optimistic parallel simulation in Erlang.
    process_loop(Event, Action, Variables, Performed) ->
        {VariableChanges, Rollback} = receive_messages(),
        case Rollback of
            false ->
                ChangedVariables =
                    perform_changes(VariableChanges, Variables),
                {NewEvent, NewAction, NewVariables, Messages} =
                    do_some_action(Action, ChangedVariables),
                process_loop(NewEvent, NewAction, NewVariables,
                    [{Event, Action, Variables, Messages} | Performed]);
            {true, OldTime} ->
                {OldEvent, OldAction, OldVariables, OldPerformed} =
                    roll_back_and_unsend_messages(OldTime, Performed),
                ChangedVariables =
                    perform_changes(VariableChanges, OldVariables),
                process_loop(OldEvent, OldAction, ChangedVariables,
                    OldPerformed)
        end.
```

Two big problems are present: 1. The list of performed actions and sent messages have a tendency to grow very fast. A mechanism for guaranteeing when a state is safe must be added. The mechanism would tell the processes when it is safe to remove old actions and messages from the list. 2. The fraction of time spent on rolling back instead of simulation forward may be very large, some algorithm for minimizing the risks for rollback must be used.

## 4.2 Sequential discrete event simulation

The work I studied for sequential discrete event simulation was made in Simula and called Demos. The name convention in my `demo` simulation package follows the names in the Demos package. Most of the ideas found in Demos can also be found in my package.

### 4.2.1 Simula

Simula was designed in 1967, under the name Simula 67, by Ole-Johan Dahl and Krysten Nyygrad from the University of Oslo and the Norwegian Computing Center. The name reflects continuity with a previous simulation language, Simula 1. This is however somewhat misleading since Simula 67 is really a general-purpose language, of which simulation is just one application. The name was shortened to Simula in 1986.

Simula is an object-oriented extension of Algol 60. Most correct Algol 60 programs are also correct Simula programs. The basic control structures, loop, conditional, switch and the basic data types (integer, real etc.) are like in the most imperative languages. Simula support the class concept of object-oriented languages.

As Algol 60, Simula is a classical language with a notion of main program. An executable program is a main program containing a number of program units (routines or classes).

In Simula, entities of non-basic types denotes references to class instances. Classes inheritance is supported (not multiple inheritance). Information hiding in classes is offered with `protected` and `hidden` declarations. Program units such as classes may be nested.

A simple form of parallelism in Simula is supported with the coroutine concept. It makes it possible to halt the execution of an object procedure and start a procedure on another object instead. The first procedure can later be restarted from were it was halted. The coroutine concept is used to schedule events between processes that are participating in a simulation. [Meyer -88].

### 4.2.2 Simulation in Simula

Simula also includes a set of primitives for discrete event simulation. This is made by letting all objects that participates in the simulation inherit from a class called `simulation`. The `simulation` class contains among many things the declaration of a class `process`, which describes processes of the physical system. Each object that inherits from `process` has the attributes of a process and the individual own defined attributes. Each process can be in one of four states: 1. Active, or currently executing. 2. Suspended, or waiting to be resumed. 3. Idle, or not part of the system. 4. Terminated.

Each object that inherits from `process` has access to a list of event notices, an *eventlist*. An eventnotice is a pair {*process, activation_time*}, where the activation time indicates when the process must be activated. The eventlist is sorted by increasing activation time; the first process is active and all the others are suspended. Non-terminated processes which are not in the eventlist are idle.

It is important to notice that a process in Simula not is the same thing as a process in Erlang. In Erlang a processes is a parallel activity that can execute at the same time as other processes. In Simula a process is an object with some special attributes.

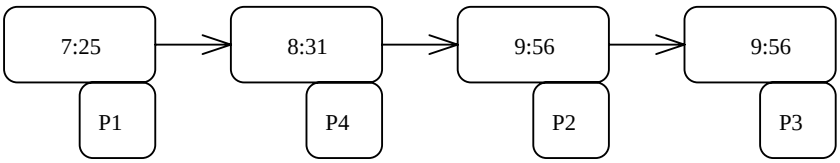



Figure 4.1 *The eventlist in Simula*

There is only one real process executing in a Simula program. The control is moved from one process to another. The main operation on processes is `active`, which schedules a process to become active at a certain time by inserting a suitable event notice for the process into a eventlist. The new event notice will be inserted after any other event already present in the list with the same activation time, unless the priority attribute for the process is specified.

The basic form of the call, to schedule a process in the eventlist, is `activate process delay`. This will schedule the process to be activated at the current eventtime + `delay`. This is a technique to represent a the duration of a task that takes some system time. For example the fact that a process carries out a 180-second task, is represented with a `activate self 180` call. To avoid having explicit self-references the procedure call `hold(time)` has exactly the same effect as above. Procedure `hold` is part of the class `simulation` and uses the coroutine concept. The effect of a `hold` call may approximately be described as:

```
hold(Time) :-
    -- insert a new event notice into the eventlist at the
       position determined by its time.
    -- get the first element in the eventlist and remove it.
    -- activate the chosen process, advancing eventtime if necessary.
```

The thread can be said to go from the process that holds, to the eventlist, to the process that is scheduled at the next event.

```
%% An example program in Simula of a worker that may be asked
%% to do either one of two tasks. Both tasks may take a variable
%% amount of time; the second task requires that the worker switch
%% on a machine m, which takes 300 seconds, and wait for the
%% machine to have done its job.
    PROCESS class WORKER
        begin
            while true do
                begin
                    ``get next task type i and taskduration d''
                    if i = 1 then
                        hold(d)
                    else
                        begin
                            activate m delay 300
                            reactivate this WORKER after m;
                            hold(d)
                        end
                end while
    end WORKER
```

### 4.2.3 Demos

Demos is a simulation package built on Simula. Demos extend the predefined class `simulation` by a few basic standardized approaches for solutions to a wide range of discrete event problems. The process concept is extended to something called *entities*. Datacollecting devices and synchronization mechanisms are added and eventtracing mechanisms for debugging is introduced. The mechanisms in Demos became a great influence when I constructed my own simulation package, `demo`. Much of the names and functionality can be derived from Demos.

The largest problem I found of the Simula approach was that parallelism were nonexistent. Only one logical process can be executing at the same time even though many processes are scheduled at the same event. My solution was to take the best pieces from the Demos package, adapt it to Erlang and extend it with parallelism.

# 5 Discrete event simulation in Erlang

After investigating the different possibilities to make discrete event simulation, I was ready to choose a simulation model for Erlang.

## 5.1 Simulation model

The model used for discrete event simulation in Erlang is the same model normally used in sequential languages but extended with parallelism. Instead of using simulated processes, as in Simula, a process is represented with a real Erlang process. By using Erlangs light-weight processes several processes can execute in parallel at the same time.

The processes interacts and are scheduled by using Erlangs message passing mechanisms. Instead of having a global eventlist that all simulation processes have access to, as in Simula, a special process is created whose only purpose is to schedule the simulation processes. This scheduler process will be called the *eventhandler* in the following pages. By letting the eventhandler be responsible for creating and scheduling processes that shall participate in the simulation it is now, by investigating the state of the eventhandler process, possible to get a compact view of the whole simulation. The state of the eventhandler process will therefore be called the *state of the simulation*.

Notice that in ordinary discrete event simulation it is possible to have several events at the same time, but processes that are scheduled at the events will only be able to run one at a time. In my package it is possible to merge all events at the same time into a single event. Depending on if several processes can be running in parallel at the same event we will said that the simulation is running in *sequential* (only one) respective *parallel* (many) mode. Large time gains can be made by letting several processes execute in parallel, for example: one process can execute while another is waiting for user input. But the parallelism also introduces, as we shall see, some new problems.

The major problem with the eventhandler process solution is that every state transformation must be handled by the eventhandler. This will lead to a bottleneck when all simulation processes will communicate with the eventhandler. I thought that this was a problem but not a insurmountable one, for several reasons: 1. Message passing in Erlang is very fast. 2. In a normal application it is not probable that all processes want to talk to the eventhandler at the same time. 3. By distributing the simulation the burden for the eventhandler can be reduced.

Two were other reasons for using a simulation concept like Demos: 1. The tools for synchronization and data collection that exists in Demos are desirable in most simulation applications. 2. This was what my supervisor Robert had in mind when he gave me the task :).

## 5.2 Semantics of the simulation

In a simulation model based on process interaction the state of the simulation is made up of the states of all the interacting processes. A process is a set of statements describing the operations in which an entity will engage during its lifetime.

All the processes $p1, p2, ..., pN$ that a simulation system consists of can be grouped together in a set called $E$. Each state transformation $pi \rightarrow pi'$ that occurs at an event in a process $pi$ that is included in $E$ will also result in a state transformation $E \rightarrow E'$.

When the state of the simulation may only change at discrete points in time, the current eventtime $Time$ of the simulation can be introduced as a parameter that affects the state of the simulation. The simulation model can now be expressed by the 2-tuple:

$$\{E, Time\}$$

where $E$ is the set of simulation processes and $Time$ is the current eventtime in the simulation.

By using $Time$ and the fact that each process will be 1. active or 2. suspended and rescheduled at the eventtime when it next shall be activated, $E$ can be divided in two sets: $Current$ and $EvTree$. The simulation model can now be expressed by the 4-tuple:

$$\{E, Current, EvTree, Time\}$$

where $Current$ is the set of processes that are indexed on the current eventtime $Time$, (active) and $EvTree$ is the set of process that not are indexed on the current eventtime (not active).

Finally, a set of processes, $R$, which not are scheduled by the eventhandler but still can change the state of the simulation, are introduced. $R$ processes are called *resources* and are useful tools for the interaction between simulation processes. Resources will be deeper explained in chapter 9. The state of a simulation can now be represented by a 5-tuple:

$$\{E, Current, EvTree, R, Time\}$$

Each new event causes the simulation to change from one state to another:

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E', Current', EvTree', R', Time'\}$$

Each time $Current$ becomes empty, i.e. all active processes have been rescheduled or terminated, the processes that are scheduled at the next event in $EvTree$ are activated. If $P$ is the set of processes in $EvTree$ that are scheduled next at the time $Time'$, then the state transformation is:

$$\{E, [], EvTree, R, Time\} \rightarrow \{E, P, EvTree - P, R, Time'\}$$

If $EvTree$ gets empty there are no more processes to activate and the simulation is complete.

When a process moves to $Current$ it will be active and start executing. An executing process can perform actions that may change the state of the simulation. An executing process $p$ can be:

suspended on a resource:

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E, Current - p, EvTree, R + p, Time\}$$

suspended on another process:

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E + p, Current - p, EvTree, R, Time\}$$

rescheduled in the $EvTree$ at a new Event:

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E, Current - p, EvTree + p, R, Time\}$$

creating a new process p':

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E + p', Current + p', EvTree, R, Time\}$$

creating a new resource r':

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E, Current, EvTree, R + r', Time\}$$

terminated, the process is removed from the simulation:

$$\{E, Current, EvTree, R, Time\} \rightarrow \{E - p, Current - p, EvTree, R, Time\}$$

When all processes have been removed from $Current$ the eventhandler will update the state to the next eventtime, i.e. start all over again.

The simulation model in the `demo` package is a little bit more complicated than this, but the basic principles are the same.

# 6 The eventhandler

The eventhandler is the process that is responsible for maintaining the state of the simulation. Each process that participates in the simulation must be scheduled by the eventhandler. The eventhandler schedules and communicates with processes via messages. By demanding a special message from a process each time a process makes an action that shall change the state of the simulation, the eventhandler can hold the correct simulation state.

## 6.1 Representation

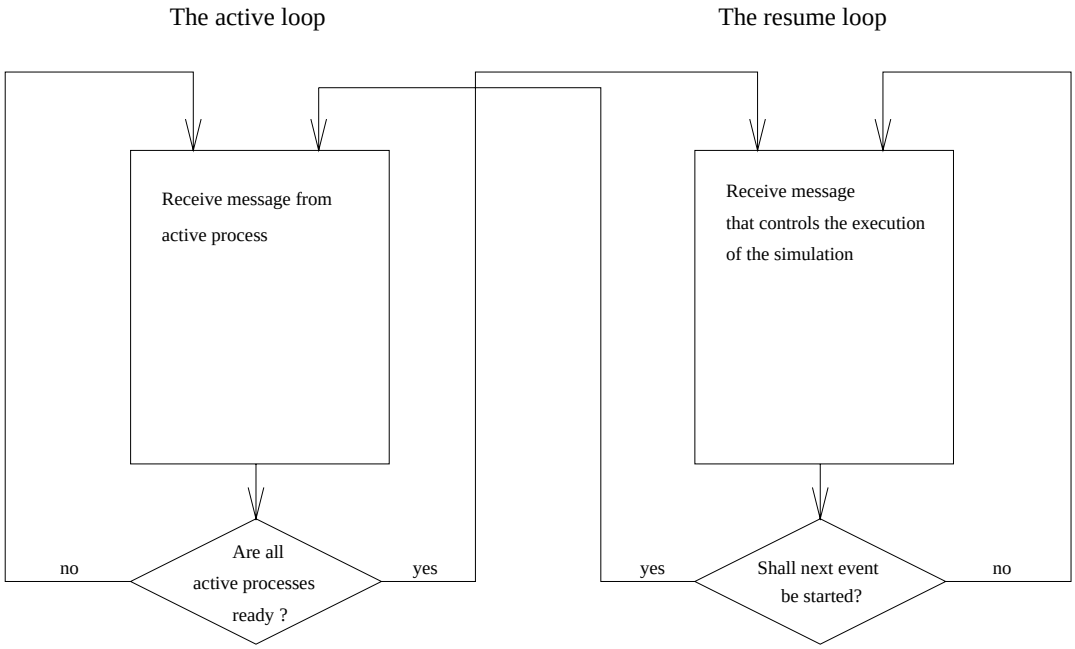



Figure 6.1 *The eventhandler process*

### 6.1.1 Two loops

The eventhandler process can be said to consist of two large loops, see figure 6.1. Each loop is a large `receive` statement where the eventhandler can receive different messages from processes. As the mechanism for receiving messages in Erlang is selective, (you can choose which messages you want to receive), it is possible to separate between two kind of messages that the eventhandler will receive and manage.

The two different kind of messages are:

- Messages made by an executing simulation process that may change the state of the simulation. The process tells the eventhandler about its state change and the eventhandler sends, if necessary, a confirmation message back to the process.

These are calls that only can be sent and received when there are processes executing, ie. during an event.

- Messages that do not have to be sent by a registered process and that may change the way that the simulation shall proceeds in the following events. For example: stopping the simulation, change granularity etc. These calls can only be received between events, ie. when no process is executing.

By demanding that the eventhandler only can change the simulation when an event is finished it is easier to keep control of the simulation. For example: Imagine that the simulation is running in *parallel* mode (several processes are active at the same event). If the user resets the simulation mode to *sequential* (only one process active at the same event) there is no way that the eventhandler can force the processes to stop executing. The only solution is to wait until all active processes are ready and then change the mode from *parallel* to *sequential*. The next event in the simulation will then be executed in *sequential* mode.

## 6.2 Extended semantics

The eventhandlers internal state (an representation of the simulation state) can be described by a 7-tuple:

$$\{EvTree, R, E, Current, RealTime, SList, Trace\}$$

Every action that is made in the simulation will change the states of one or more items in the tuple. The $EvTree, R, E, Current$ and $RealTime$ functionality are almost as simple as specified in chapter 5. The $SList$ variable is an attributelist for the eventhandler that specifies certain functionality of the simulation, for example if the simulation shall run in *sequential* or *parallel* mode. The variable $Trace$ tells the eventhandler if tracing shall be on or off.

The eventhandler keeps the representation of each process by the process identifier (pid) of the process. The following sections will give a deeper description of the physical representation of the internal state variables of the eventhandler.

### 6.2.1 EvTree, the set of events

$EvTree$ is the structure where all suspended (not executing) processes are placed. The eventhandler inserts processes indexed on the eventtime that they shall be activated again. It shall both be easy to insert a process in the structure and to retrieve and activate the processes with the lowest eventtime when the simulation shall be updated to the next event. $EvTree$ can be represented in two different ways, as a tree and as a list.

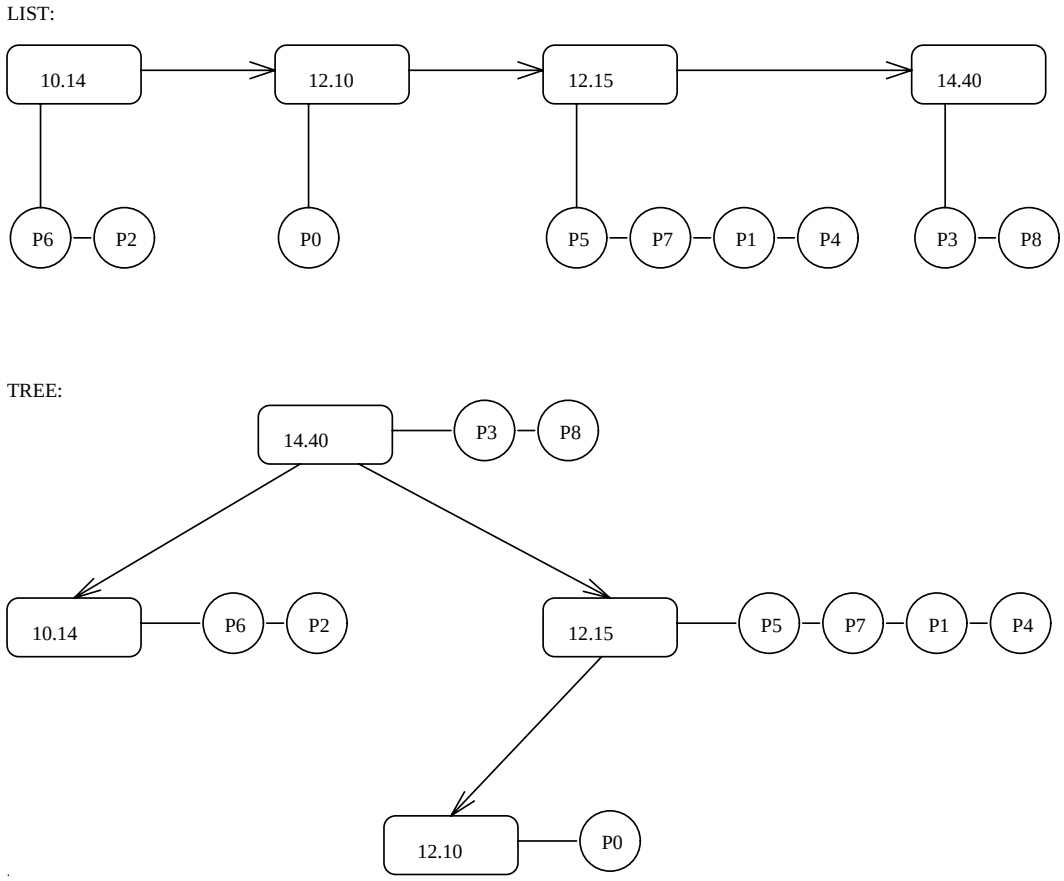



Figure 6.2 *The EvTree as a List and as a Tree*

**EvTree as a Tree**

$EvTree$, when represented as a tree, is a tree structure of all the scheduled processes, ranked according to the time of their next scheduled event. An event in the tree is represented with a $\{EventTime, PidList\}$ tuple. The $EventTime$ is the time the event shall be scheduled and $PidList$ is a list of pids belonging to processes that shall be activated at the event. The tree is a leftist priority tree, making insertion and deletion of items in the tree O(log n), where n = number of items in the tree.
Two conditions holds for the tree:
1. If a nodes left subtree is empty $\rightarrow$ the nodes right subtree is also empty.
2. All nodes in the left subtree $<$ all nodes in the right subtree $<$ the node.
Traversion of the tree is made: Left, Right, Node. All operations that are defined on the lptree structure are found in the module `lptree2`. The operations could be insertion and deletion of objects in the tree, returning of the first element in the tree, traversing the tree and find a certain pid in the tree.

**EvTree as a List**

As list operations in Erlang are very fast it is more effective to use a list structure when

the number of scheduled items are small. The list will be a list of $\{EventTime, PidList\}$ tuples, sorted on EventTime. The PidList is the list of processes that shall be activated at the event EventTime. Insertion of items are O(n) but normal deletion is O(1), where n = number of items in the tree. All operations that are defined on the lptree structure are found in the module `lptree`. By default the listversion of the lptree is used in the simulation, but this can be changed by exchanging names of the module files.

### 6.2.2 Current, the set of active processes

*Current* is the set processes that are currently executing.

*Current* is implemented as a list of pids belonging to the processes that are active. By receiving calls from the active processes the state transformations of the simulation processes can be registered by the eventhandler process. For example, if a process suspends on a resource, suspends on another process or is rescheduled in $EvTree$ the process is removed from *Current*. If an active process terminates this is reported to the eventhandler and the process is removed from *Current* and also from the whole simulation.

When *Current* gets empty all, at the event activated, processes are suspended or terminated. The eventhandler will then proceed in the resume loop. When all messages in the resume loop has been treated the simulation will start all over again. The processes that are scheduled at the next event in $EvTree$ are removed from $EvTree$, activated and inserted in the *Current* set.

### 6.2.3 R, the set of resources

$R$ is the set of resource processes that the eventhandler is aware of. $R$ is represented as a list of pids. Resources are participating in the simulation but are not scheduled by the eventhandler. A process that is executing can be suspended on a resource and will then be removed from *Current* (and not inserted in $EvTree$). The eventhandler will then register that the process is suspended on the resource and will not let anyone activate the process except the resource it is suspended on. More information about resources can be found in chapter 9.

### 6.2.4 E, the set of registered processes

$E$ is the set of processes that are participating in the simulation (except resources). $E$ is represented as a list of pids. If a process suspends itself on another process this is registered both on the process that it gets suspended on and on itself. A suspended process is removed from *Current*.

### 6.2.5 SList, the set of simulation variables

$SList$ is the set of variables that decides how the simulation shall proceed. $Slist$ is implemented as a list of $\{Attribute, Value\}$ tuples. Things like if the simulation shall be stopped, when it shall be stopped, the granularity size and if the simulation shall run in *sequential* or *parallel* mode are stored in $SList$.

### 6.2.6 RealTime, the eventtime variable

*RealTime* is the variable that holds the current eventtime of the simulation. Once the eventhandler process updates to a new event the variable *RealTime* is updated to the time of that event. A better name for *RealTime* would maybe have been "EventTime". When an event has been updated it is impossible to decrease the *RealTime* variable, this is because it shall not be possible to travel back in time.

### 6.2.7 Trace, the tracing variable

*Trace* is the variable that is set to *on* if the eventhandler shall trace all actions that is made on the state of the simulation. If *Trace* is *off* no tracing shall be made. The *Trace* variable could have been placed in the *Slist* but when it will be tested at every state transformaton it will be quicker to let it be a single item. See section 7.6 for more information about tracing.

# 7 To steer the simulation

All calls that changes the way the simulation proceeds can be made during runtime. The calls will only be handled by the eventhandler between events, i.e. when no simulation process is executing.

## 7.1 Representation

When the eventhandler may change the way the simulation proceeds it is said to be in the resume loop.

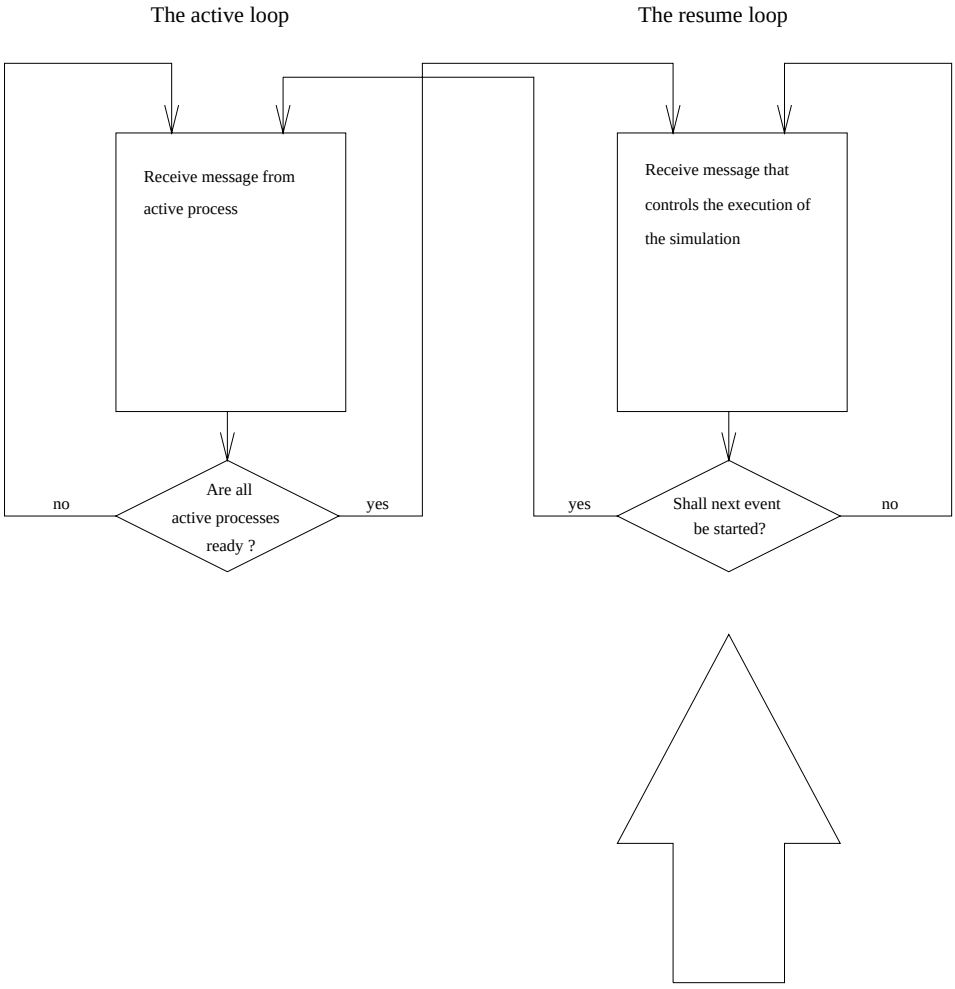



Figure 7.1 *The eventhandler process is in the resume loop*

## 7.2 Start and stop simulation

The simulation is started with a `demo:start` call from a process or user. If no other eventhandler process is running on the current node an eventhandler process will be started. The eventhandler will be registered under the name *demo_server*.

The simulation is stopped with a `demo:stop` call from a process or an user. This will terminate all simulation processes, all resources, all opened files and finally the eventhandler process. If `demo:reset` is called everything will be terminated as above except the eventhandler process.

## 7.3 Resume and suspend simulation

The simulation can be *suspended*, i.e. stopped after an event, by a `demo:suspend` call. Then the user has a possibility to investigate the state of the simulation. The simulation can also be stopped at regular time intervals, at every new event or at a certain eventtime.

When the simulation is suspended the only way to start executing the next events is by a `demo:resume` call. This call will make the simulation running until the next `demo:suspend` call is made or until all processes are terminated.

All suspend- and resume calls can be made during runtime.

The simulation will, by default, be in suspended mode when the simulation is started. The reason for that is all initialization of processes must have a chance to be made before the execution of the first event. To understand why this is a problem, look at the example below:

```
-module(ex).
-export([p1/0, p2/0]).

p1() ->
    demo:start(),
    demo:new_P(ex, p2, []),
    p3(),
    demo:new_P(ex, p2, []).

p2() ->
    hold(1).

p3() ->
    receive
        Something ->
            true
    after 100000 ->
        true
    end.
```

Depending on if the mailbox of the calling process is empty or not a `ex:p1()` call may give two different results. If the mailbox is empty it is possible that the

eventhandler will update the eventtime to the next event before the second `p2` process is created. The problem arises from the fact that the first simulation process (and the eventhandler process) must be created by a process that not are participating in the simulation. Normally this will be made by a function call in the shell process. But the shell process will not return a `{'EXIT', Pid, Reason}` signal when the execution of the function is ready. Instead it will go back to handle inputs from the shell. The eventhandler process have no way of telling when the shell process is finished and no more simulation processes will be created, i.e. the eventhandler is not able to start the first event in the simulation.

To handle this synchronization problem you better:

- Only call `demo:resume()` when you are sure that all initializations have been made.
- Only let simulation processes (processes created with a `demo:new_P` call) create simulation processes. In the example above `p1` shall be rewritten like:

```
p1() ->
    demo:start(),
    demo:new_P(ex, p1_1, []).

p1_1() ->
    demo:new_P(ex, p2, []),
    p3(),
    demo:new_P(ex, p2, []).
```

## 7.4 Concurrency

The concurrency mode decides if there shall be any parallelism in the simulation. The concurrency can be in one of two different modes: *parallel* and *sequential*.

If the concurrency mode is *sequential* only one process can be active at the same event. If two processes have the same eventtime one is first activated and the eventhandler waits until it gets suspended or terminated before it activates the second process. By giving the processes different priority it is possible to steer which process that shall be activated first. This sort of simulation corresponds to the simulation that exists in Simula.

If the concurrency mode is *parallel* several processes can be active at the same event. All events, with the same timestamp is merged together into a single event, resulting in parallel execution of processes. My belief is that this is more like events in the real life, when two things happens at the same time they are really happening in parallel. The *parallel* mode is the default mode of the concurrency. By running the simulation in parallel large time profits can be made. For example, if two processes, P1 and P2, are running in parallel and P1 stops waiting for a user input, P2 can still continue executing. This would not automatically have been the case if P1 and P2 were running sequentially after each other.

The concurrency of the simulation is changed with the `demo:set_concurrency(Form)` call where `Form` can be `sequential` or `parallel`.

The calls that changes the concurrency can all be made during runtime.

### 7.5 Granularity

The granularity is a way to change the granularity of the simulation by letting several events, not equal in time, be merged together into one event. For example, if the granularity is 10 and the current eventtime is 0 the eventhandler will the first time start all processes that are scheduled from 0 to 10, the second time all events from 10 to 20, etc. Big granularity may give large parallel executions but also introduces some new problems.

If the eventhandler starts all events between time T1 and time T2 (the granularity is T2 - T1) two things can happen: 1. No events shall be activated between T1 and T2: The eventhandler updates to eventtime to T2 and start all over again. 2. One or more processes are activated at an event between T1 and T2: The eventtime is updated to T2 and all processes are activated. If one of the processes gets suspended on the eventhandler two things can happen: 2.1 The time the process wants to be passivated to is > T2: Passivate the process on the eventhandler as normal. 2.2 The time the process wants to be passivated to is <= T2. The process must be reactivated again. I have solved this by letting each process have an internal time that it shall be scheduled after.

The granularity of the simulation can be changed by a user or a process with the `demo:set_granularity` command. The default value of the granularity is 0.

By changing the granularity of the simulation we are approaching the discrete time approach of simulation; 'update the simulation at regularly intervals'. We are also getting closer to the parallel approach to discrete event simulation, 'Let all the processes proceed with an internal time but only proceed to a state when it is safe.' (see section 4.1.1). To achieve security between processes you can use Erlangs message passing primitives. For example: The process P1 with internal time T1 and process P2 with internal time T2 becomes activated at the same time T when T1 < T2 =< T. To be sure that P2 not will proceed until P1 is ready put P2 in a receive statement. When P1 is finished it will send a start message to P2, P2 receives the message and can thereafter start executing.

```
%% P2 have to wait for P1 to be ready
    p1() ->
        do_something(),
        P2 ! ready
        do_something_else().

    p2() ->
        receive
            ready ->
                true
        end,
        do_something().
```

The main reason for using granularity is to increase the parallelism and to test what happens if the limits between the events becomes weaker. For example, test of unexpected delays.

## 7.6 Tracing

Tracing is supported for getting a clearer view of the simulation By a `demo:trace` respective `demo:notrace` call the user can turn on and off the tracing before and during the simulation. The tracing is also a way of detecting errors in the simulation process. Every tracing comment is written as: *pid of the simulation process* and *the action that the process makes.*

```
%% Tracing example. A bin resource is created, one process requests items
%% and have to wait on the resource for his request to be fulfilled.
10              -------------> Activates <0.17.1>
                <0.17.1>       Creates resource <0.13.1> with newR at time 0
                <0.17.1>       Calls resource <0.13.1> with call_R(<0.13.1>,
                                                                   init, [800])
                <0.17.1>       Holds 14 until time 24

24              -------------> Activates <0.17.1> <0.21.1>
                <0.17.1>       Holds 76 until time 100
                <0.21.1>       Calls resource <0.13.1> with call_R(<0.13.1>,
                                                                   take, [900])

100             -------------> Activates <0.17.1>
                <0.17.1>       Calls resource <0.13.1> with call_R(<0.13.1>,
                                                                   give, [100])
                <0.21.1>       Activates from resource <0.13.1>
                <0.21.1>       Holds 10 until time 110
                <0.17.1>       Process closed
```

The trace is default switched off.

# 8 Processes

Three types of processes that are useful in a simulation can be distinguished:

- Simulation processes. This is processes that are created by a `demo:new_P` call and are scheduled by the eventhandler. A simulation process is almost what normally in discrete event simulation is called an entity.
- Resources. This is processes used to synchronize simulation processes and collect data from the simulation. Resources are created with a `demo:new_R` call. Resources will be treated in the next chapter.
- Normal processes. This is processes that have been created with a `spawn` call (and not a `demo:new_P` call). These processes are not part of the simulation but can still be very useful.

## 8.1 Simulation processes

Only a simulation process can be scheduled by the eventhandler, i.e. be part of the simulation. A simulation process can compete with other simulation processes for resources, cooperate over stretches of time, or even interrupt one another. A simulation process can be in one of the following five states:

- Active, currently executing.
- Suspended, waiting for the eventhandler to reactive it.
- Passivated, on a resource or another process.
- Idle, not part of the simulation.
- Terminated.

The eventhandler can only receive calls from simulation processes when it is in the active loop. A simulation process can only change the state of the simulation when it is executing or terminates. Each state changing action in a simulation process will result in that a message is sent to the eventhandler. The eventhandler will always receive the messages from a process in the same order as they were sent.

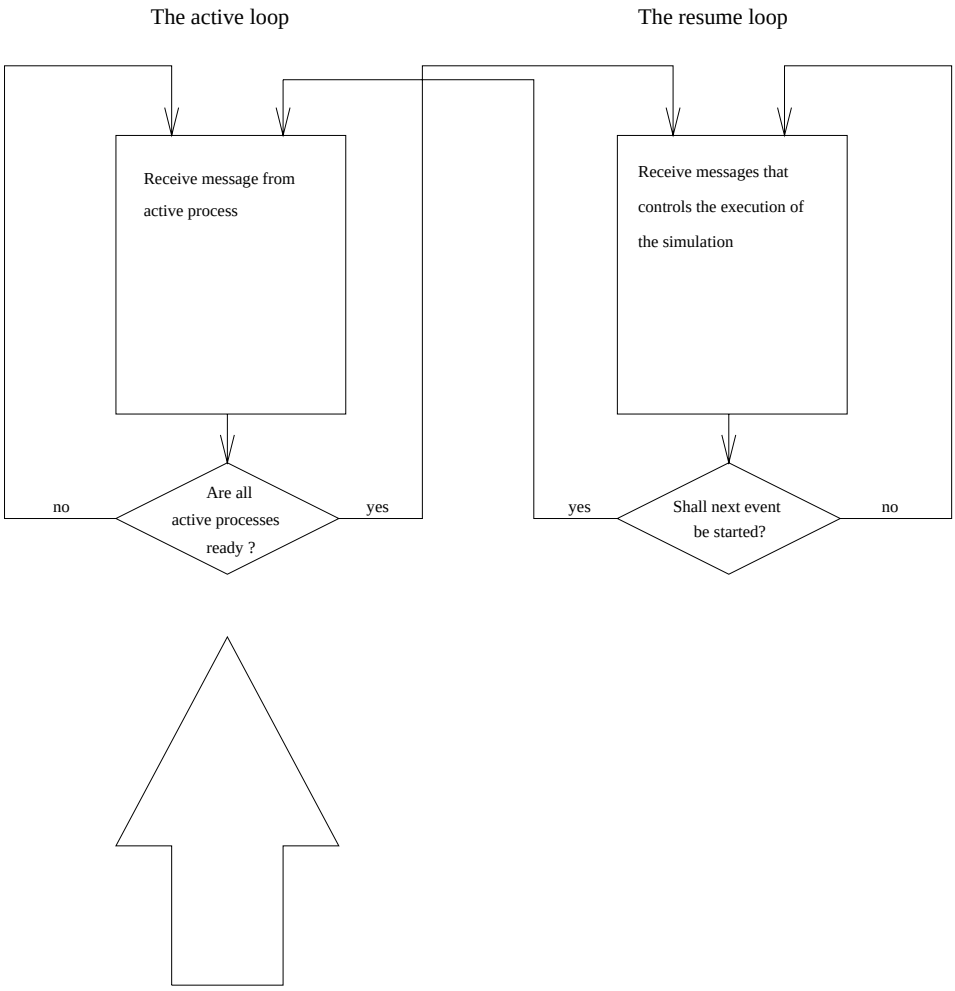



Figure 8.1 *The eventhandler process is in the active loop*

## 8.2 Process states

The different actions a simulation process can make will here be reviewed.

### 8.2.1 Create a process

A simulation process is created with a `demo:new_P` call and can be created by a simulation or normal process. The `demo:new_P` call returns the process identifier (pid) of the created process. The pid can be used by other processes who wants to communicate with the process.

It is requested, before a simulation process can be created, that the eventhandler process has been started (`demo:start`). A `demo:new_P` call will result in a message sent to the eventhandler. The eventhandler makes a `spawn_link` call that creates the wanted process and activates or schedules the process in the *EvTree*. By letting the

eventhandler create the process and link to it, the eventhandler will get the necessary information of the process that is needed for controlling and scheduling it. For example, when a process terminates, a `{'EXIT', Pid, Reason}` call is made to the eventhandler that will remove the process from the simulation.

```
%% An example that creates a ferry process:
    Ferry = demo:new_P(ferry_module, ferry_function, [Cargo]).

%% Compare that with a normal spawn call:
    Ferry = spawn(ferry_module, ferry_function, [Cargo]).
```

### 8.2.2 Suspend a process

The `demo:hold(Time)` is used to represent the duration of an activity. Seen from the process itself it represents a period of time in the same state and holding of resources until it take up its actions again. The *Time* parameter in the `demo:hold(Time)` call represents the delay in events before the process will be activated again, i.e. the process will be activated again at $eventtime + Time$.

A non active process is not really passive in terms of Erlang semantics, instead it is waiting in a `receive` statement to be reactivated by the eventhandler. A `demo:hold(Time)` call from a simulation process results in the following actions: 1. Send a message to the eventhandler that informs the eventhandler that the process will wait *Time* events before it gets activated again. 2. Wait in a receive statement for a message from the eventhandler that will activate the process again.

```
%% How hold is implemented.
    hold(Time) ->
        eventhandler ! {hold, Time, self()},
        receive
            startagain ->
                true
        end.
```

When the eventhandler receives a `demo:hold(Time)` call from a simulation process its actions are: Remove the processes from the list of current executing processes ($Current$) and reschedule the process at time $eventtime + Time$ in the $EvTree$.

All actions that handles suspension or passivation of processes works in the same way. For example, if a process passivates on a resource, the resource will send a message to the eventhandler that informs it that the process has been passivated on the resource. The passivated process will wait in a receive statement to be activated from the resource. When the resource is able to reactivate the process it sends a message to the eventhandler that tells it that the process has been released from the resource and can be activated. The eventhandler will reactivate the process by sending a startmessage to it.

### 8.2.3 Passivate a process on another process

A process that is active (executing) can be passivated on another process. This means that the suspended process leaves all control to the process it is suspended on. The

suspension is useful when two or more processes shall cooperate over a period of time. Instead of having several processes moving down the event list together, it is simpler to single out one process as the master and let it coopt the others for the period in question.

A process can only be passivated on another process by a `demo:passivate(Pid)` call from itself. The only one that can activate the suspended process again is the process it is passivated on. Activation will be made with a `demo:activate` call.

A process P1 can not suspend another process P2 without P2 is agreeing. This is because that if P2 is executing there is no way to stop his execution before it is ready. When P2 is ready it may be passivated on another process, on a resource or the eventhandler, terminated etc. The programmer have to deal with passivation of processes himself with the help of Erlangs message passing primitives.

The reason why a process P1 not is able to activate a process PS that is suspended on another process P2, is that we not are knowing if the process P2 currently is using PS. For example, P2 finds out that PS is suspended on itself and wants to activate PS. The process P1 may be quicker than P2 and activates PS before P2. When P2 tries to activate PS this may cause an error because PS no longer is passivated on P2.

For the same reason that it is forbidden, for a process to activate on another process suspended process, it is also forbidden for a process, without using the calls that the resource offers, to activate a process that is suspended on a resource.

### 8.2.4 Passivate a process on a resource

When a process gives a request call to a resource it is not certain that the resource can fulfil the request right away. The process will then be suspended on the resource waiting for the the resource to release it (when the request is fulfilled). This is an easy way to achieve synchronization between processes. When a process suspends itself on a resource it is only the resource that can reactivate the process. For deeper studies of resources see chapter 9.

### 8.2.5 Make a process idle

If a process no longer wants to be part of the simulation the easiest thing is to let the process terminate. But, if the process not wants to be scheduled by the eventhandler, but still be alive, it can make a `demo:cancel` call. The process is then removed from the simulation but not terminated. The process can be reactivated with a `demo:interrupt` call from a simulation process.

Processes that from the beginning not shall be scheduled by the eventhandler are created with a `spawn` call instead of a `demo:new_P` call.

### 8.2.6 Process termination

A simulation process terminates normally if it completes the evaluation of the expression for which it was spawned or if it evaluates the built in function `exit(Reason)`. The eventhandler process is linked to all simulation processes and will therefore receive a `{'EXIT', Pid, Reason}`. signal from each simulation process that terminates. The

process with the pid `Pid` will be removed from *Current* and *E* by the eventhandler, i.e. removed from the whole simulation. The eventhandler may also control if the process terminated abnormally by investigating the *Reason* variable. If the termination reason not where `normal` a warning message is written to the *error_log*. The error_log is where all errors discovered by the eventhandler will be written . The default value is `standard_io` for the error_log but can be changed to a file or a process instead.

The mechanism of catching run-time errors from executing processes makes the simulation error secure. The whole simulation will not fail just because one process causes an error. By setting the `stop_when_error` variable to `true` or `false` the user can chose if the simulation shall stop when a process terminates due to an abnormal error.

If a process terminates and still has processes suspended on itself, the suspended processes are also terminated by the eventhandler. The reason for that is that a process P1 that were suspended on the terminated process P2 was waiting for the process P2 to fulfil some task before it could get activated again. Now P1 can not be sure that the task it was waiting for have been fulfilled and therefore it is not possible to restart P1 again. For example, let P1 represent a car and P2 represent a carferry and let the ferry transport the car from a harbour A to another harbour B. If the ferry process terminates before it has released the car it is not possible to know if the ferry has transported the car to harbour B or still is out on the stormy sea. The information about the ferry location is held by the ferry process internal state and is not accessible for the eventhandler. The only way to accomplish security is by terminating the suspended processes too. Observe that this can lead to chain reactions of terminated processes if a suspended process had processes suspended on itself.

If a resource process terminates (God forbid) all processes that are suspended on the resource will also be terminated.

## 8.3 More about processes

The following sections will give a deeper explanation of the different attributes a simulation process has and the different actions a simulation process can take.

### 8.3.1 Process priority

A way to decide which processes that shall be first activated from a queue, a resource etc. is by the priority of the process. If a simulation executes in *sequential* mode the priority is also a way to decide which process that shall execute first if several processes are scheduled at the same event. If the simulation runs in *parallel* mode is this not the case, this is because all processes with the same eventtime will be activated in parallel. For processes that are suspended on resources the priority is much more important, this is because a process with higher priority will be placed before a process with lower priority in the queue (and be released much earlier).

Only the process itself can set its own priority. This is made with a `demo:set_priority (Priority)` call from the process. The Priority is default 0 for all processes. The priority of a process can be obtained by a `demo:get_priority` call.

Observe that if the simulation runs in *sequential* mode and a process P1 activates a process P2 that has a priority that is higher then P1s priority, P2 will not start executing before P1 is finished. A process can only be suspended if it explicitly desires it.

### 8.3.2 Process interrupts

A process can only interrupt another process if the process is suspended on the eventhandler or idle. The interruption of a process is made with the `demo:interrupt(Pid, Message)` call. `Pid` is the process to be interrupted and `Message` is the interrupt message that the interrupted process shall receive. If the interrupt succeeds is *true* returned, else *false*.

When a process gets interrupted from the eventhandler it will start executing right away (if the simulation is running in parallel mode, otherwize it will wait). The interrupted process can check who interrupted it with a `demo:get_interrupt` call. If the process was not interrupted *undefined* is returned, otherwise the interrupt message *Message* from the process that interrupted is returned. The interrupt message must be reset by the process himself with the `demo:set_interrupt` call. A process can turn on or off the possibility for another process to interrupt it with a `demo:turn_interrupt` call.

All calls to interrupt a process goes via the eventhandler. The eventhandler checks that the process that shall be interrupted is participating in the simulation and not have turned off the ability be interrupted. If a process is interrupted it will be activated according to the concurrency mode.

```
%% An example of a program where two processes interrupt each other,
%% resulting in that 'Hello world!!' is written to the screen.
    p1() ->
        demo:hold(1),
        Message = demo:get_interrupt(),
        io:format('~w',[Message]),
        demo:interrupt(P2, 'world!!').

    p2() ->
        demo:interrupt(P1, 'Hello '),
        demo:hold(1),
        Message = demo:get_interrupt(),
        io:format('~w',[Message]).
```

### 8.3.3 Process attributes

One difficult problem is how to transfer information about a process to another process. In Simula this is simple, an object consist of attributes and be accessed by a <object reference>.<attribute name> call (there is no information hiding within the objects). For example, the cargo on an object ferry may be accessed by a `ferry.cargo` call.

In Erlang we have processes instead of objects. A process can normally only get information about other processes through messages. I support two different kind of attribute exchanges between processes, *message passing* and *demo:attribute* calls.

### Message Passing

Message passing is the normal way to exchange information between processes in Erlang.

The messages can be used for synchronization between processes. For example: If process P1 sends a message with a value of an attribute to process P2, then P2 can be sure to get the value of the attribute that P1 wants it to have. If we look at the ferry example again it is possible that the ferry process for a moment is loading cargo. When the ferry process is ready it will update its cargo variable and send a message to a requesting process about its new cargo. This would lead to that requesting process gets the value of the cargo that the ferry process wants it to have. The solution is preferable but have two serious draftbacks: 1. The requesting process is passivated in a receive statement until it gets the message from the ferry, we loose some parallelism. 2. The ferry process must have information about all processes that wants to get the cargo attribute from it, and this is not always the case.

```
%% Synchronized attribute exchange between two processes
    p1() ->
        do_something(),
        P2 ! ready

    p2() ->
        receive
            ready ->
                true
        end,
        do_something().
```

### Attribute calls

Because of the problems above with normal message passing one more way for processes to exchange information is present, the `demo:attribute` calls. A process P1 can set an attribute with a `demo:set_attribute` call. A process is only allowed to set its own attributes. Another process P2 can get information about the attribute by an `demo:get_attribute` call. The call returns the value of the attribute or `undefined` if the attribute have not been set by the P1 process. This form of information exchange is not synchronized and the process P2 can not be sure that it gets the value of the attribute that P1 wants it to have. If we return to the ferry example, a process can get the cargo value both before, during and after the cargo loading have been made.

```
%% Nonsynchronized attribute exchange between two processes
    p1() ->
        do_something(),
        demo:set_attribute(ready, true)

    p2() ->
        case demo:get_attribute(P1, ready) of
            ready ->
                do_something();
            undefined ->
                do_something_else()
        end.
```

The `demo:attribute` calls is implemented using the process dictionary and the `process_info/2` call. This is a use of the process dictionary that is '*strongly discouraged*' by the writers of Erlang but I was not able to implement it so smoothly in any other way, (sorry Robert).

One way to handle the problem of using the process dictionary would have been a attribute server process. The simulation processes will set and get attributes by sending messages to the attribute server. This can be a acceptable solution when the number of simulation processes are small but will be a bottleneck when the number of processes increases.

The problem of getting the right attribute from a process is a consequence of the policy to let processes that has the same eventtime execute in parallel. If we only let the simulation execute in *sequential* mode and steer the process scheduling with the priority parameter the problem will not occur. My suggestion to the application programmer is to use normal message passing for synchronized 'want the process attribute when it is ready' calls and normal messages and the `demo:attribute` calls for nonsynchronized 'want the process attribute immediately' calls.

# 9 Resources

In the process oriented approach to discrete event simulation, programs are collections of interacting processes which compete with other processes for resources before a task can be undertaken.

## 9.1 What is a resource

Resources can be divided into two separate set:

- Datacollecting resources. These are devices used to collect data and statistics from simulation processes during the simulation.
- Synchronization resources. These are processes that are used to synchronize simulation processes during the simulation.

### 9.1.1 Implementation

A resource is implemented as a separate process that not is scheduled by the event-handler. A resource is created with a `demo:new_R(Module, ArgList)` call from a process. The first argument shall be the module name of the resource, and the second, if any, shall be a list of the initial arguments. The calls to a resource from a process is made with `demo:call_R` or `demo:report_R` calls. By making all resources of the same format and restricting the operations that can be made on them, the resources are much more easier to control.

All function calls to resources are made with `demo:call_R` calls. For example a call to a bin resource `demo:take(BinPid, 5)` is transformed to a call `demo:call_R( Bin Pid, take, [5], Priority)`. When a process calls a resource it will automatically be passivated in a receive loop and leave the control to the resource. It is now up to the resource to reactivate the process again. If the resource is a datacollecting resource it will just return a value to the calling process and let it be released immediately. If the resource is a synchronize resource it must also be able to passivate and later restart processes.

A few predefined function calls, that every resource shall support, is made with the function `demo:report_R`. Instead of using the resources *Module* variable (see below) a predefined report module is used. All resources can now give a report of its internal state in a similar manner.

## 9.2 Semantics of a resource

More specifically is a resource a looping process waiting for calls in a `receive` statement. There are three parameters in the loop: $AList$, $Trace$ and $Module$.

The state of a resource can abstractly be viewed as a 3-tuple:

$$\{AList, Trace, Module\}$$

Each time a process calls the resource a transformation is made on the resource state:

A `demo:call_R` call:
$\{AList, Trace, Module\} \rightarrow \{AList', Trace, Module\}$

A `demo:report_R/3` call:
$\{AList, Trace, Module\} \rightarrow \{AList, Trace, Module\}$

A `Trace` call from the eventhandler:
$\{AList, Trace, Module\} \rightarrow \{AList, Trace', Module\}$

```
%% A mutual exclusion example. A library has some books and readers want
%% to borrow them. The number of books are constant. The library is
%% represented by a Res resource. A reader process borrows some books,
%% reads them and returns them finally.
    -module(res_ex).
    -export([init/0, main/0, reader/2]).

%% init/0 starts the simulation, starts the tracer and activates the main
%% process.
    init() ->
        demo:start(),                       %% start simulation
        demo:trace(),                       %% start tracer
        demo:new_P(res_ex, main, []).       %% create main process

%% main/0 creates the Res resource that representates the library, and three
%% reader processes. It lets the simulation proceed for 100 events and
%% turns then of the simulation.
    main() ->
        Library = demo:new_R(res, [10]),    %% start the library res
        demo:new_P(res_ex, reader, [Library]), %% create readers
        demo:new_P(res_ex, reader, [Library]),
        demo:new_P(res_ex, reader, [Library]),
        demo:hold(100),                     %% wait 100 events
        demo:stop().                        %% stop simulation

%% reader/1 acquire a number of books of the library. It reads
%% one book per day and returns the books to the library.
    reader(Library) ->
        Number_Of_Books = random:uniform(10), %% how many books?
        demo:acquire(Library, Number_Of_Books), %% aquire books from library
        Days_To_Read = Number_Of_Books,
        demo:hold(Days_To_Read),            %% read the books
        demo:release(Library, Number_Of_Books), %% return the books
        reader(Library).                    %% loop
```

Below is each state variable explained in more detail.

### 9.2.1 AList, the attribute list

The *AList* is the resource attributelist, representing the state of the resource. All relevant data about the resource are contained in the *AList*, for example, its passivated processes, length of the different queues and number of made observations. When a process makes a call that changes the resource state it is actually changing

the *AList*. The *AList* is represented as a list of tuples consisting of $\{Index, Value\}$ pairs. For example: the tuple $\{resettime, Resettime\}$ holds the eventtime for the last resetting of the resource. The *AList* is available for other processes with calls like: `demo:get_attributeR`, `demo:get_all_attributesR` or `demo:report_R`. By using these commands it is possible to examine and gather information about resources.

### 9.2.2 Trace, the tracing variable

Trace is set by the eventhandler and specifies if the resource shall trace its actions or not. The trace can be turned on or off by calls from to eventhandler process, i.e. with `demo:trace` and `demo:notrace` calls.

### 9.2.3 Module, export functions

Module is the module name where all operations on the resource, (i.e. on the *AList*), are defined. The functions exported from the module Module are the functions that can be made, by an active process, to change the state of the resource. The module limits the access to the resource by only allowing exported functions of *Module* to be called. If a nonexported function call is made to the resource, no state transformation is made, instead an error message is written and false is returned. A resource can only have one module.

### 9.2.4 Why a process

By implementing the resource as a separate process, much of the responsibility of using the modules is moved from the calling process to the implementator of the resource. If a calling process had total control over the items that belongs to a resource it would also have been much easier to do something wrong. The user can change the *AList* with `demo:set_attributesR` calls, but that is not recommended.

It is not allowed for other processes to interrupt a process that is passivated on a resource. The reasons for that are several:

- The interrupted process can not know if the goal it is waiting for in the resource has been fulfilled.
- I have implemented waiting processes different in different resources and it is therefore difficult to write common release of processes primitives that suits each resource.
- The resources were constructed in the beginning of my examwork before I investigated the idea of letting processes interrupt each other.

### 9.2.5 How to activate passivated processes

A special situation appears when a process makes a call to a resource that shall activate a process that is passivated on the resource, which process shall be reactivated first? If the simulation is in *parallel* mode this is not a problem, both processes will be activated simultaneously. If the simulation is in *sequential* mode the situation is more problematic.

I have solved this by always letting the process that called the resource continue executing and let the released process be passivated in the eventhandler . It will the compete with other processes of which one that shall be first started when the current process is rescheduled or terminates. I tried to solve this problem by letting the process with the highest priority be first scheduled but soon noticed that it made the scheduling very complicated. I thought it would be better to have an easy and understandable algorithm.

All process passivated on a resource are normally scheduled by priority, which let a process with high priority be activated before a process with low priority. It is often possible to set parameters in the resource that changes how passivated processes shall be activated.

### 9.2.6 Resource reports

All resources has an extra predefined module that handles report generation from the resource. When a `demo:report_R(Resource, Operation, [Arg])` call is made the *Module* module that belongs to the resource is not used, instead the `demo_report` module is called. Every resource has in its *AList* a $\{report, ReportList\}$ tuple. The *ReportList* is a list of items that shall be included in the reports from the resource. A process can reset, add and remove items from the *ReportList*.

The user can set the file to write reports to with the `demo:set_reportfileR` command, the default value is `standard_io`.

### 9.2.7 Influences

The different resources constructed all corresponds against the predefined objects that exists in the Demos simulation package, both by name and functionality.

## 9.3 Datacollecting resources

Four data collecting devices for recording profiles of input sequences are defined:
1. Count, for incidences
2. Tally, for timeindependent sequences.
3. Histogram, for timeindependent sequences.
4. Accumulate, for timedependent input sequences.

### 9.3.1 Count

A Count resource is used to record incidences. Given a input sequence of numbers N1, N2, ..., Nm the Count resource records their sum N1 + N2 + ... + Nm.

**Commands**
The count resource is created with a `demo:new_R(count, [])` call from a process.

The function calls `demo:update(Count)` or `demo:update(Count, N)` updates the Count.

The function call `demo:reset(Count)` resets the Count. Reports and facts from the count resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

### 9.3.2 Tally

A Tally resource is used to record a profile of values. Given the input sequence of numbers N1, N2, ..., Nm the Tally resource records the number of observations (m), their sum so far (N1 + N2 + ... + Nm), the sum of their squares so far (N1 * N1 + N2 * N2 + ... + Nm * Nm), their range (the smallest and largest value so far), their mean value, their variance and standard deviation. The Tally resource is an extension of the Count resource.

**Commands**
The Tally resource is created with a `demo:new_R(tally, [])` call from a process.

The function calls `demo:update(Tally)` or `demo:update(Tally, N)` update the Tally.

The function call `demo:reset(Tally)` resets the tally. Reports and facts from the tally resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

```
%% A tally example. A program that records the average through
%% times of costumers in a supermarket.
    -module(tally_ex).
    -export([init/1, main/1, costumer/2]).

%% init/0 starts the simulation and activates the main process.
    init(N) ->
        demo:start(),                       %% start simulation
        demo:new_P(tally_ex, main, [N]).    %% create main process

%% main/1 creates a tally resource and some costumers. It lets the
%% simulation proceed for 100 events and asks the tally resource to
%% print its observations. Finally the simulation is stopped.
    main(N) ->
        Tally = demo:new_R(tally, []),      %% create tally resource
        create_costumers(Tally, N),         %% create costumers
        demo:hold(100),                     %% wait 100 events
        demo:report_headingR(Tally),        %% print tallys heading
        demo:reportR(Tally),                %% print tallys variables
        demo:stop().                        %% stop simulation

%% create_costumers/1 creates a number of costumer processes.
    create_costumers(Tally, 0) ->
        true;
    create_costumers(Tally, N) ->
        demo:new_P(tally_ex, costumer, [Tally, N]),
        create_costumers(Tally, N - 1).

%% costumer/2 represent the behavior of a costumer in a supermarket.
    costumer(Tally, N) ->
        random:seed(N, N, N),               %% initialize random func
        ArrivalTime = demo:event_time(),    %% costumer arrivaltime
        demo:hold(random:uniform(100)),     %% do some shopping
        LeaveTime = demo:event_time(),      %% costumer leaving time
        ShoppingTime = LeaveTime - ArrivalTime, %% calculate shoppingtime
        demo:update(Tally, ShoppingTime).   %% update tally
```

### 9.3.3 Histogram

A Histogram resource is used to record a rough profile of a sequence of values, N1, N2, ..., Nm by asking in advance for their (expected) lower and upper bound and the number of recording cells. The range from lower to upper is then divided into a number of slots. Each slot has the width (upper - lower)/cells. When an input value Ni is recorded, the appropriate slot incidence count is incremented with 1. Underflow values, Ni < lower, and overflow values, Ni > upper, are recorded separately.

#### Commands

The Histogram resource is created with a `demo:new_R(histogram, [Upper, Lower, Cells])` call from a process.

The function call `demo:update(Histogram, N)` update the Histogram.

The function call `demo:reset(Histogram)` resets the Histogram. Reports and facts

from the Histogram resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

### 9.3.4 Accumulate

An Accumulate resource is used to record a profile of a time dependent variables. The object records the time integral of a sequence of values. Given a input sequence of numbers N1, N2, ..., Nm with their input eventtimes T1, T2, .., Tm the Accumulate resource is calculating the (T2 - T1) * N1 + (T3 - T2) * N2 + ... + (Tm - Tm-1) * Nm-1 value. The resource builds a timeintegral under the assumption that the recorded value is constant during a timeinterval until the next inputvalue arrives.

**Commands**
The Accumulate resource is created with a `demo:new_R(accumulate, [])` call from a process.

The function calls `demo:update(Accumulate)` or `demo:update(Accumulate, N)` updates the Accumulate resource.

The function call `demo:reset(Accumulate)` resets the Accumulate resource. Reports and facts from the Accumulate resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

## 9.4 Synchronization resources

Five synchronization resources to be used for synchronisation problems between simulation processes are defined:
1. Bin, for producer/consumer problems.
2. Res, for mutual exclusion problems.
3. Condq, for wait until problems.
4. Waitq, for master/slave cooperation problems
5. Queue, for process synchronization.

### 9.4.1 Bin

Bin resources are used to model the producer/consumer synchronization between processes. A *producer* process makes items which are to be used by *consumer* processes. A *consumer* process is passivated on the Bin resource if no items are available when requested.

**Commands**
The Bin resource is created with a `demo:new_R(bin, [Avail])` call from a process. `Avail` is the number of items that shall be available from the beginning.

The function call `demo:give (Bin, Items)` is made by a producer process that gives away some items to the Bin resource. The resource seeks among the passivated consumer processes and reactivates any processes if possible.

The function call `demo:take(Bin, Items)` is called by a consumerprocess that wants some items. If the process demands more items than the Bin resource has available or if any other process with higher priority is waiting in the list of passivated processes the process is passivated on the Bin resource. If there are items available, and no process with higher priority is waiting ,the items are released to the calling process. The process has no indicator of the number of items it has been given, it will only know that its demands have been fulfilled when it is released from the resource.

The `demo:reset(Bin)` resets the Bin resource. Reports and facts from the Bin resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

```
%% A producer-consumer example. A producer process produces items and a
%% consumer process consumes them. A consumer process is blocked if no
%% item is available. The pool of available items is represented by a Bin resource.
    -module(bin_ex).
    -export([init/0, main/0, producer/1, consumer/1]).

%% init/0 starts the simulation and the tracing and activates the main
%% process.
    init() ->
        demo:start(),                       %% start simulation
        demo:trace(),                       %% start tracer
```

```
        demo:new_P(bin_ex, main, []),        %% create main process
        true.

%% main/0 creates a bin resource, one producer process and three consumer
%% processes. It lets the simulation proceed for 100 events and turns then
%% of the simulation.
    main() ->
        Bin = demo:new_R(bin, [0]),          %% start the Bin resource
        demo:new_P(bin_ex, producer, [Bin]), %% create the producer
        demo:new_P(bin_ex, consumer, [Bin]), %% create consumers
        demo:new_P(bin_ex, consumer, [Bin]),
        demo:new_P(bin_ex, consumer, [Bin]),
        demo:hold(100),                      %% wait 100 events
        demo:stop().                         %% stop simulation

%% producer/1 makes an item and gives it to the Bin resource, and
%% then restarts the production.
    producer(Bin) ->
        Time_To_Make_Item = random:uniform(4),
        demo:hold(Time_To_Make_Item),        %% create an item
        demo:give(Bin, 1),                   %% give the item to bin
        producer(Bin).                       %% loop

%% consumer/1 consumes items. If the Bin resource is empty when the
%% consumer wants an item the consumer suspended until an item appears.
    consumer(Bin) ->
        demo:take(Bin, 1),                   %% demand an item
        Time_To_Consume_Item = random:uniform(10),
        demo:hold(Time_To_Consume_Item),     %% consume an item
        consumer(Bin).                       %% loop
```

### 9.4.2 Res

Res resources are used to model the mutual exclusion synchronization between processes. There are some items available from the beginning in the resource which processes can acquire. In contrast with the Bin resource, the number of available items are constant, i.e. no new items can be created. If a request from a process can be fulfilled and no other process with higher priority is waiting for items, the process is given the items and is released. Otherwize the requesting process will be passivated on the Res resource until its request can be fulfilled. The Res resource keep tracks over which processes that have achieved how many items. A process can not create new items, only return items that it before has achieved from the resource. When a process returns some taken items the passivated processes are tested to see whose request that can be fulfilled.

#### Commands

The Res resource is created with a `demo:new_R(res, [Capacity])` call from a process, where `Capacity` is the number of items available from the beginning.

The function call `demo:acquire(Res, Items)` is called by a process that wants some items. The process are not allowed to ask for more items than the `Capacity` of

the Res resource. If the are less items available then the process demanded or if any other process with higher priority is waiting for items the calling process is passivated on the Res resource. If there are items available and no process with higher priority is waiting, the items are released to the calling process.

The `demo:release(Res, Items)` call releases some items from the calling process and returns them to the resource. A process can only release items if it before has achieved more or equal number of items with a `demo:acquire` call. If any process demand among the passivated processes can be fulfilled due to the new items, the corresponding process is released.

The `demo:reset(Res)` call resets the Res resource. Reports and facts from the Res resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

```
%% A mutual exclusion example. A library has some books and readers want
%% to borrow them. The number of books are constant. The library is
%% represented by a Res resource. A reader process borrows some books,
%% reads them and returns them finally.
    -module(res_ex).
    -export([init/0, main/0, reader/2]).

%% init/0 starts the simulation, starts the tracer and activates the main
%% process.
    init() ->
        demo:start(),                       %% start simulation
        demo:trace(),                       %% start tracer
        demo:new_P(res_ex, main, []).       %% create main process

%% main/0 creates the Res resource that representates the library, and three
%% reader processes. It lets the simulation proceed for 100 events and
%% turns then of the simulation.
    main() ->
        Library = demo:new_R(res, [10]),    %% start the library res
        demo:new_P(res_ex, reader, [Library]), %% create readers
        demo:new_P(res_ex, reader, [Library]),
        demo:new_P(res_ex, reader, [Library]),
        demo:hold(100),                     %% wait 100 events
        demo:stop().                        %% stop simulation

%% reader/1 acquire a number of books of the library. It reads
%% one book per day and returns the books to the library.
    reader(Library) ->
        Number_Of_Books = random:uniform(10), %% how many books?
        demo:acquire(Library, Number_Of_Books), %% aquire books from library
        Days_To_Read = Number_Of_Books,
        demo:hold(Days_To_Read),            %% read the books
        demo:release(Library, Number_Of_Books), %% return the books
        reader(Library).                    %% loop
```

### 9.4.3 Condq

Condqs are used to implement *waits_until* in the simulation. When a process wants to be passivated until a condition is true, it is passivated on a Condq. This is done by a waituntil call that evaluates a function and tests if the result is equal to the wanted result. If so, the process is activated right away, otherwize the process is passivated on the Condq resource until the condition is fulfilled. The condition is tested each time a process calls the Condq with a signal call. The condition test is made by the passivated process itself.

**Commands**

The Condq is created with a `demo:new_R(condq, [])` call from a process.

The function calls `demo:wait_ until (Condq, Module, Func, Arg)` and `demo:wait_until(Condq, Module, Func, Arg, Result)` enters a process into the Condq. The function `Module: Func(Args)` is evaluated to see if it evaluates to *true* or *Result*. If so , the process is immediately released from the Condq. Otherwise the process is suspended on the Condq until `Module:Func(Args)` is tested and evaluated to the wanted result.

The functioncall `demo:signal(Condq)` let all,on the Condq passivated processes test their conditions. If any of the conditions of the passivated processes, (`Module:Func(Args)`) are evaluated to the wanted result the corresponding process is released from the Condq.

The `demo:reset(Condq)` resets the Condq resource. Reports and facts from the Condq resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

It is worth noticing that if the simulation is running in parallel mode a condition can be tested and evaluated to wanted result by a process, but before the process has made some other actions another process can change the condition so that it no longer is true. A Condq resource is therefore best suited for simulation in sequential mode. For example, if the simulation is running in parallel mode it is not possible to implement mutual exclusion with a Condq
See appendix B for more information about the resource.

```
%% A dining philosopher example. The philosophers are Number many and
%% sitting at a round table. A philosophers life consists of eating and thinking.
%% To be able to eat a philosopher must have access to two forks.
%% A fork is shared by two philosophers that sits next to each other at
%% the table. Eg. two neighbour philosophers can not eat at the same time.
%% A Condq is used for testing if both forks are available.
    -module(condq_ex).
    -export([init/1, main/1, philosopher/4, forks_available/2]).

%% init/1 starts the simulation, starts the tracer, set concurrency to
%% sequential and activates the main process. The argument to init is the
%% number of philosphers
    init(Number) ->
        demo:start(),                       %% start simulation
        demo:trace(),                       %% start tracer
        demo:set_concurrency(sequential),   %% make simulation sequential
```

```
        demo:new_P(condq_ex, main, [Number]),  %% create main process
        true.

%% main/1 creates a condq, some forks and some philosphers. The number of
%% forks and philosphers are equal. It lets the simulation proceed for 100
%% events and turns then of the simulation.
    main(Number) ->
        Condq = demo:new_R(condq, []),       %% create condq
        demo:set_attributeR(Condq, test, all), %% test all philosophers
        ForkList = create_forks(Number, []), %% create fork resources
        create_philosophers(Number, Condq, ForkList), %% create philosophers
        demo:hold(100),                      %% wait 100 events
        demo:stop().                         %% stop simulation

%% create_forks/2 creates a number of forks. Each fork is represented with
%% a Res resource. The available amount of items in each Res are 1.
    create_forks(0, ForkList) ->
        Fork = hd(ForkList),                 %% the first fork is
        lists:append(ForkList, [Fork]);      %% also the last fork
    create_forks(N, ForkList) ->
        Fork = demo:new_R(res, [1]),         %% create a fork
        create_forks(N - 1, [Fork | ForkList]).

%% create_philosophers/3 creates a number of philosophers. Each philosopher
%% is a simulation process that is created with a new_P call.
    create_philosophers(Number, Condq, [Fork]) ->
        true;
    create_philosophers(Number, Condq, [Fork1, Fork2 | ForkList]) ->
        demo:new_P(condq_ex, philosopher, [Condq, Fork1, Fork2]),
        create_philosophers(Number - 1, Condq, [Fork2 | ForkList]).

%% philosopher/3 represents the behaviour of a philosopher. The
%% philosopher has a simple life, he first thinks, then eats and starts all
%% over again. In order to eat, a philosopher requires the fork on his left
%% and the fork on his right. When he has eaten both forks are released and can
%% be used by the neighbour philosophers. To test if a forks is free
%% the philosopher is placed in a Condq.
    philosopher(Condq, Fork1, Fork2) ->
        ThinkTime = dis:randint(10, 20),     %% get time to think
        demo:hold(ThinkTime),                %% think
                                             %% wait until forks are available
        demo:wait_until(Condq, condq_ex, forks_available, [Fork1, Fork2]),
        demo:acquire(Fork1, 1),              %% get the forks
        demo:acquire(Fork2, 1),
        EatTime = dis:randint(5, 10),        %% get time to eat
        demo:hold(EatTime),                  %% eat
        demo:release(Fork1, 1),              %% release the forks
        demo:release(Fork2, 1),
        demo:signal(Condq),                  %% tell condq
        philosopher(Condq, Fork1, Fork2).

%% forks_available/2 is called by a philosopher in a Condq to test if
%% his two forks are available.
    forks_available(Fork1, Fork2) ->
        Avail1 = demo:get_attributeR(Fork1, avail),  %% is fork available
```

```
        Avail2 = demo:get_attributeR(Fork2, avail),  %% is fork available
        test_avail(Avail1, Avail2).

    test_avail(1, 1) ->
        true;
    test_avail(_, _) ->
        false
```

### 9.4.4 Waitq

Waitq resources are used in the master/slave synchronizations in which several processes cooperate together over a period of time. Instead of having several processes moving down the event list together, it is simplier to single out one process as the master and let the other processes be passivated on the master for the time in question. The master is responsible for releasing its passivated resources when it is ready. The Waitq is responsible for passivating processes on master processes.

**Commands**

The Waitq resource is created with a `demo:new_R(waitq, [])` call from a process.

The functioncall `demo:wait(Waitq)` is made by a slaveprocess that wants to find a masterprocess that can enslave it. If there are no masterprocesses available or if the slave not can fulfil the masters demands of processes the slave passivated on the Waitq resource. If there are masterprocesses available, that has a demand of processes that would be fulfilled with the new process, the slave processes is passivated on the masterprocess and the masterprocess is activated.

The functioncalls `demo:coopt(Waitq)` and `demo:coopt(Waitq, N)` is called by a masterprocess that wants to enslave one respective N number of processes. If the demanded slaveprocesses not are available or if a masterprocess with higher priority is waiting on slaveprocesses the process is passivated on the Waitq resource. If the masterprocess demand can be fulfilled the wanted number of passivated slaveprocesses are moved from the resource to the masterprocess and the masterprocess is reactivated. The masterprocess will release the passivated processes with `demo:activate` calls.

The `demo:reset(Waitq)` resets the Waitq resource. Reports and facts from the Condq resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

```
%% A car and carferry example. The ferry goes between two islands and can take
%% two cars across each travel. The ferry starts by waiting for some
%% cars, load them, cross the water with the cars and finally unload them.
%% The cars comes to the harbour and wait in a Waitq for the ferry to take
%% and transport the car over to the other island. When the car has arrived
%% it will tour the island for a while and then return to the harbour.
%% Each harbour is represented by a Waitq resource.
    -module(waitq_ex).
    -export([init/0, main/0, ferry/3, car/2]).
```

```
%% init/0 starts the simulation, starts the tracer and activates the main
%% process.
    init() ->
        demo:start(),                       %% start simulation
        demo:trace(),                       %% start tracer
        demo:new_P(waitq_ex, main, []).     %% creat main process

%% main/0 creates two waitq resources, one ferry process and seven car
%% processes. It lets the simulation proceed for 100 events and turns then
%% of the simulation.
    main() ->
        Harbour1 = demo:new_R(waitq, []),   %% create wqueue
        Harbour2 = demo:new_R(waitq, []),
                                            %% create ferry process
        Ferry = demo:new_P(waitq_ex, ferry, [2, Harbour1, Harbour2]),
                                            %% create car processes
        Car1 = demo:new_P(waitq_ex, car, [Harbour1, Harbour2]),
        Car2 = demo:new_P(waitq_ex, car, [Harbour1, Harbour2]),
        Car3 = demo:new_P(waitq_ex, car, [Harbour2, Harbour1]),
        Car4 = demo:new_P(waitq_ex, car, [Harbour2, Harbour1]),
        demo:hold(100),                     %% wait 100 events
        demo:stop().                        %% stop simulation

%% ferry_loop/3 is the function that represents the ferry. The ferry first
%% avaits cars. When the cars has arrived the ferry will start loading,
%% cross the water and unloading the cars. During the time the cars are
%% passivated on the ferry. Finally the cars are released from the ferry.
    ferry(Capacity, CurrentHarbour, NextHarbour) ->
        CarList = demo:coopt(CurrentHarbour, Capacity), %% Get a list of cars
        Load = Capacity,
        demo:hold(Load),                    %% Load for some time
        Crossing = random:uniform(10),
        demo:hold(Crossing),                %% Cross the sea
        Unload = Capacity,
        demo:hold(Unload),                  %% Unload for some time
        release_cars(CarList),              %% Activate the cars
        ferry(Capacity, NextHarbour, CurrentHarbour). %% Loop

%% release_cars/1 activates all cars that has been passivated on the
%% ferry. This could have been done with a demo:activate_all() call.
    release_cars([]) ->
        true;
    release_cars([Pid | Rest]) ->
        demo:activate(Pid),                 %% Activate the car
        release_cars(Rest).

%% car_loop/2 is the function that represents a car. First the car arrive
%% to the harbour and waits for a ferry to take it to the other island.
%% When the crossing is made the car will be activated and doing some
%% sightseeing on the island. Finally the car will return back to the harbour.
    car(CurrentHarbour, NextHarbour) ->
        demo:wait(CurrentHarbour),          %% Wait for ferry
        Tour_Island = random:uniform(10),
        demo:hold(Tour_Island),             %% Do some sightseeing
        car(NextHarbour, CurrentHarbour).   %% Loop
```

### 9.4.5 Queue

A Queue resource can be used to chain several processes. A Queue is a resource where processes can passivate themselves with a push command. A passivated process is activated with a pop command from another process. Depending on what sort of queue the Queue is: FIFO, LIFO or PRIORITY the processes are released in different ways.

#### Commands

The queue resource is created with a `demo:new_R(queue, [Test])` call from a process. `Test` is the way that the process shall passivate processes and could be `fifo`, `lifo` or `priority`.

If a process makes a `demo:push(Queue)` call it will be passivated on the Queue resource.

The functioncall `demo:pop(Queue)` tries to release a process from the Queue resource. If there is a passivated processes in the queue it is released and the calling process receives the passivated process pid as return value. If there were no passivated process on the queue `false` is returned to the calling process. Observe that the released process will not be passivated on the calling process.

The `demo:reset(Queue)` call resets the queue resource. Reports and facts from the Queue resource can be achieved with `demo:report_R` or `demo:get_attributeR` calls.

See appendix B for more information about the resource.

```
%% Simulates a liftsystem with lifts, liftqueues, skiers and mountains.
%% There are two skilifts with different capacity and transport time and
%% two corresponding liftqueues where skiers have to wait. The input
%% argument is the number of skiers.
    -module(queue_ex).
    -export([init/1, main/1, skilift/4, skier/1]).

%% init/1 starts the simulation, and activates the main process.
    init(Number) ->
        demo:start(),                       %% start simulation
        demo:new_P(queue_ex, main, [Number]), %% create main process
        true.

%% main/1 starts two queue processes that represents liftqueues, two tallys
%% to record how many skiers each skilift has transported and two skilifts
%% that will transport the skiers. A number of processes representing skiers
%% are created. The simulation is allowed to proceed for 100 events and is
%% then turned off.
    main(Number) ->
        SkiQueue1 = demo:new_R(queue, [fifo]), %% create skiqueues
        SkiQueue2 = demo:new_R(queue, [fifo]),
        Tally1 = demo:new_R(tally, []),     %% create tallys
        Tally2 = demo:new_R(tally, []),
                                            %% create skilifts
        SkiLift1 = demo:new_P(queue_ex, skilift, [SkiQueue1, Tally1, 5, 10]),
        SkiLift2 = demo:new_P(queue_ex, skilift, [SkiQueue2, Tally2, 2, 4]),
                                            %% create skiers
        create_skiers(Number, [{SkiQueue1, SkiLift1}, {SkiQueue2, SkiLift2}]),
        demo:hold(100),                     %% wait 100 events
        io:format('skilift 1 transported ~w skiers ~n', [demo:get_attributeR(Tally1, sum)]),
        io:format('skilift 2 transported ~w skiers ~n', [demo:get_attributeR(Tally2, sum)]),
        demo:stop().                        %% stop simulation

%% skilift/4 is a process representing a skilift. The arguments to skilift/4 are
%% a skiqueue where the skiers shall be waiting, a tally resource to report
%% transports, the time to transport the lift and a capacity of the lift (the number
%% of skiers that can go with it). The process tries to release a number of skiers
%% that are waiting in the skiqueue and puts them in the skilift. The lift is
%% transported to the top, the skiers are relased and the skilift is transported
%% back down to the liftqueue.
    skilift(SkiQueue, Tally, LiftTime, Capacity) ->
        SkierList = get_skiers(Capacity, SkiQueue),  %% load skiers to transport
        demo:update(Tally, [length(SkierList)]).     %% update tally with skiers
        demo:hold(LiftTime),                %% go to the top
        demo:activate_all().                %% release all skiers
        demo:hold(LiftTime),                %% go down again
        skilift(SkiQueue, Tally, LiftTime, Capacity).

%% get_skiers/2 releases the skiers that currently are passivated on the
%% current skiqueue. The skilift can not take more skiers than its capacity.
    get_skiers(Capacity, SkiQueue) when Capacity =< 0 ->
        [];                                 %% can not take more skiers
    get_skiers(Capacity, SkiQueue) ->
        case demo:pop(SkiQueue) of          %% try to get one more
            false ->                        %% failed
```

```
                [];
            Pid ->                          %% succeeded
                [Pid | get_skiers(Capacity - 1, SkiQueue)]
        end.

%% create_skiers/2 creates a number of skiers. Each skier has knowledge of the
%% skilifts and skiqueues that exists.
    create_skiers(0, _) ->
        true;
    create_skiers(Number, QueueLiftList) ->
        demo:new_P(queue_ex, skier, [QueueLiftList]),
        create_skiers(Number - 1, QueueLiftList).

%% skier_loop/1 represents the behaviour of a skier. The skier first selects
%% a skilift to go with and a skiqueue to wait in. When the skier has been
%% transported to the top it will take some time to get down the mountain
%% again.
    skier(QueueList) ->
        {SkiQueue, SkiLift} =               %% select lift and queue
            get_queue_and_lift(QueueList),
        demo:push(SkiQueue).                %% wait in skiqueue
        demo:passivate(SkiLift).            %% go with skilift
        Time_DownHill = dis:randint(10, 20),

%% A skilift to go with and a skiqueue to wait in. When the skier has been
%% transported to the top will it take some time to get down the mountain
%% again.
    skier(QueueList) ->
        {SkiQueue, SkiLift} =               %% select lift and queue
            get_queue_and_lift(QueueList),
        demo:push(SkiQueue).                %% wait in skiqueue
        demo:passivate(SkiLift).            %% go with skilift
        Time_DownHill = dis:randint(10, 20),
        demo:hold(Time_DownHill),           %% ski downhill
        skier(QueueList).

%% get_queue_and_lift/1 selects which lift and queue the skier shall go with.
get_queue_and_lift(QueueLiftList) ->
        Nr_Of_QLs = length(QueueLiftList),  %% select a tuple at
        QL_Number = (random:uniform(10000) rem Nr_Of_QLs) + 1, %% random
        lists:nth(QL_Number, QueueLiftList).
```

## 9.5 Construct own resources

The user is free to make resources on his own by using my concept or any other idea. To help him there are some functions that the `demo_resource` module exports. The user may use the same base functions that are used by my resources for synchronization between processes and a resources. The functions that the resources uses are:

`demo_resource:passivateR/1`
: which tells the eventhandler that a process has been passivated on the resource.

`demo_resource:activateR/2`
: which tells the eventhandler that a process that is passivated on the resource shall be activated. It is up to the eventhandler when the process shall be activated again.

`demo_resource:unpassivateR/2`
: which takes a process that is passivated on the resource and passivates it on a process instead.

`demo_resource:repassivateR/2`
: which passivates a process, that not yet has been passivated on the resource, on another process.

The call to `demo:new_R` that creates the resource is not necessary, it just informs the eventhandler about its existence and let the eventhandler link to it. The eventhandler can also take calls from unregistered resources if they are made in the right order. For example shall a resource passivate a process before it can release it.

An example of an owndefined resource is a selector resource. The resource gets information by messages about attributes and demands from processes and singles out the best combination of processes. For example: a marriage counsel process can get demands from both men processes and women processes to put together the best couples.

# 10 Probability distributions

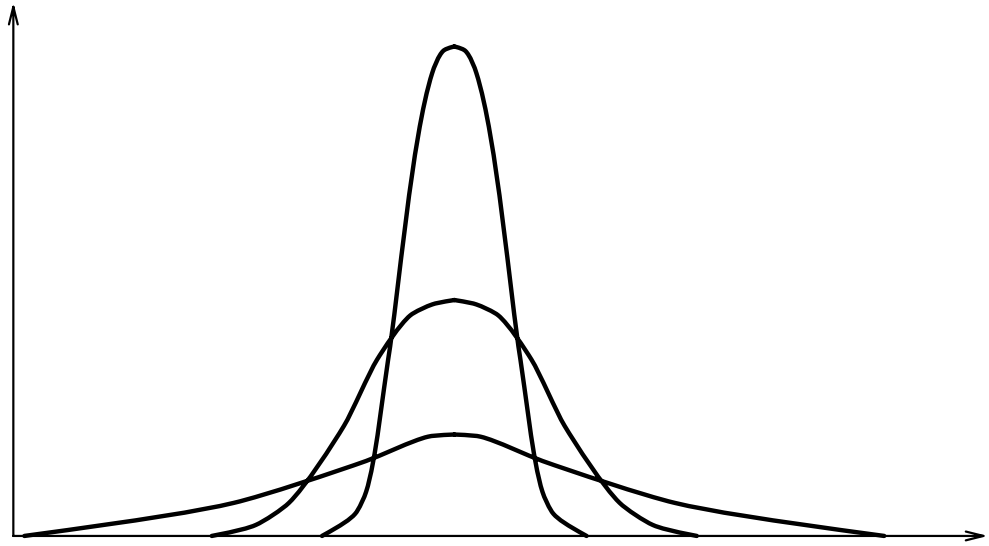


Figure 10.1 *Different normal distributions*

Operations to generate probability distributions are also defined in the simulation package. The distributions can be used by processes, for example, to decide how long time a work shall be take, the lifetime of a component, etc.

## 10.1 Module random

The basic random number generator used in my simulation package is built on the *random* module in Erlang. The random generator must be initialized with a `random:seed()` or a `random:seed(Nr1, Nr2, N3)` call. A `random:uniform()` call returns a random float between 0 or 1 and a `random:uniform(Nr)` call returns a random integer between 1 and Nr.

All distributions that can be generated built on the `random:uniform` call and it is therefore demanded that the a `random:seed` call has been made by the same process that wants a value from a distribution. All calls are made in the module *dis*.

## 10.2 An example

Repeated calls to the function `dis:normal(Nr1, Nr2)` will return drawings from the normal distribution, using the Box-Müller method, with the mean = *Nr1* and variance = *Nr2 * Nr2*.

See appendixC for more information about probability distributions.

# 11 A radio traffic simulation application

When a working simulation package have been constructed the next step where to test it with some applications. The applications should be able to detect missing functionality, and to compare the performance of my simulation package with others. Two different companies that are working with simulation were contacted, ERA-t and Ångpanneföreningen. The results from the application made for Ångpanneföreningen will be given in the next chapter. The ERA application will be presented in this chapter.

## 11.1 Assignment

ERA-t is a department at Ericsson in Sweden that is working with radio traffic simulations. A working model called SIMON implemented in C++ were present at ERA-t. A short description of SIMON can be found in the papers from ERA-t:

'The purpose of the SIMON simulator is to collect all that is common among mobile telephony systems simulation in a category of classes. These classes describe basic behaviors and features of all mobile telephony systems. These classes are then used when writing an application for simulating a particular kind of system.'

My assignment bacame to implement a minor part of the SIMON model in Erlang, using my simulation package.

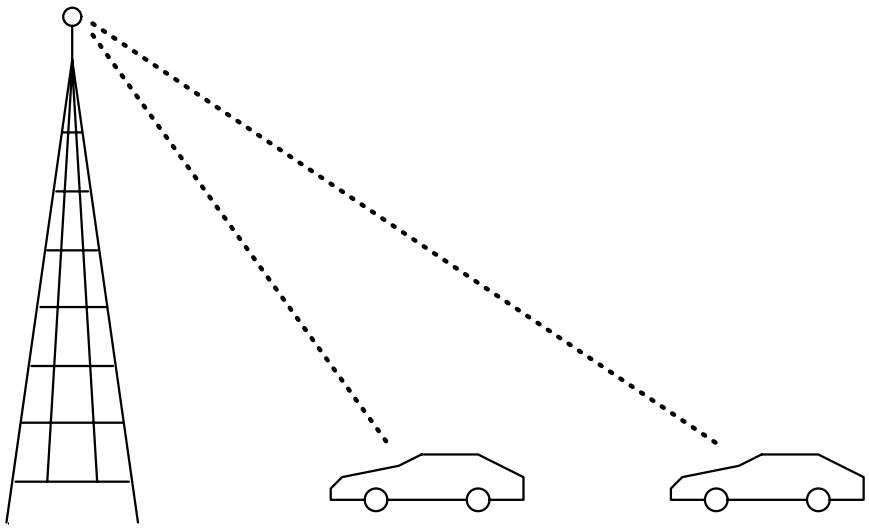


Figure 11.1 *Mobiles connected to a radio station*

## 11.2 The radio traffic model

The radio traffic world consists of a moving *mobiles* that connects to *antennas* over an area. Each place where an antenna is placed is called a *site*. The area that an antenna covers (the area where a mobile can connect to the site) is called a *cell*. An antenna and a mobile can connect on several special *frequencies*. Each frequency is diveded

into a number of *timeslices*. When a mobile connects to a site it is made on a special frequency and timeslice on that frequency. Many mobiles can therefore be connected to one site at the same time. A mobile moves, connects and makes measurements towards sites.

To generate mobile traffic different traffic generators may be used. By having different traffic generators that coveres different areas and can be overlapping each other, different traffic intensities can be achieved.

A mobile has a position, an antenna and a distance to a connected site. These parameters decides the *signal strength* of the connection between a site and a mobile.

Values measured during a simulation are:

**Signal strength (SS) value**
: Several sites (not neighbours) can send on the same frequncy. The SS value is the sum of contributions in signal strength from all sites that sends on the same frequency. To complicate the picture a neighbour frequency will also give a smaller contribution to the SS value. When a mobile has calculated the SS values on all different possible frequencies it will try to make a connection to the closest site that sends on the frequency with the best SS value. If all time slices on the frequency are occupied by connections a mobile can be refused a connection.

**Carrier over interference (C/I) value**
: The C/I value is the relation between the calculated SS value (C) and the sum of the disturbing connections (I) on the same frequency and timeslice but on different sites.

**Carrier over adjacent (C/A) value**
: The C/A value is the relation between the calculated SS value (C) and the sum of the connections on neighbour frequencies on the same timeslice (A).

**Up- and downlink**
: In reality the mobile and the site has two different connections, one uplink- (mobile to site) and one downlink- (site to mobile) connection. All values must be therefore be calculated in both directions.

**Handover**
: When a mobile moves across the site cell border the SS value toward the connected site may become to low. The mobile will therefore try to connect to another site. This is called a *handover*. When a mobile makes a handover it will first remove the present connection and try to make a new connection to another site. If the mobile fails to make the new connection it is called a *handover fail*.

## 11.3 The ERA-t model

In the ERA-t model only one mobile can be active at the same time. When the mobile has moved and measurements have been made, the mobile will be rescheduled to a later time (make a `hold` call) and placed at the end of the list of waiting mobiles.

During each activation of a mobile it will calculate the SS, C/I and C/A values, make handovers etc. The SS-value is calculated through a large two-dimensional radio path matrix where mobiles are indexed on one side and site on the other. Each radio path is calculated with help of a path loss value, L0 and a fading value, Lf.

The largest problem with the ERA-t model I found were that only one mobile can be active and make measurements at the same time, only one mobile can use the array at the same time and that only a few of the radiopath values in the array are used when a mobile calaculates its current values.

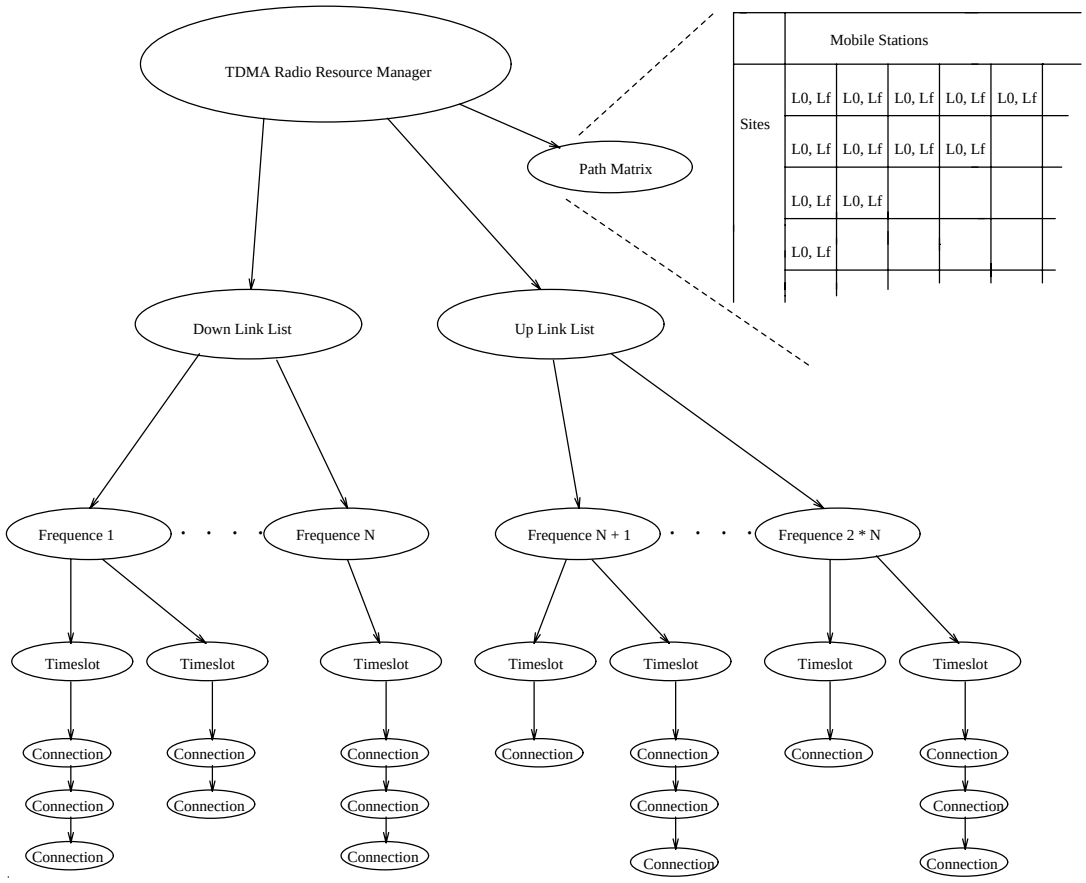



Figure 11.2 *The radio resource manager model in the ERA-t model*

## 11.4 The Erlang model

The Erlang model uses the fact that all facts about a site are static. This means that the sites position value and antenna value will be constant during the whole simulation.

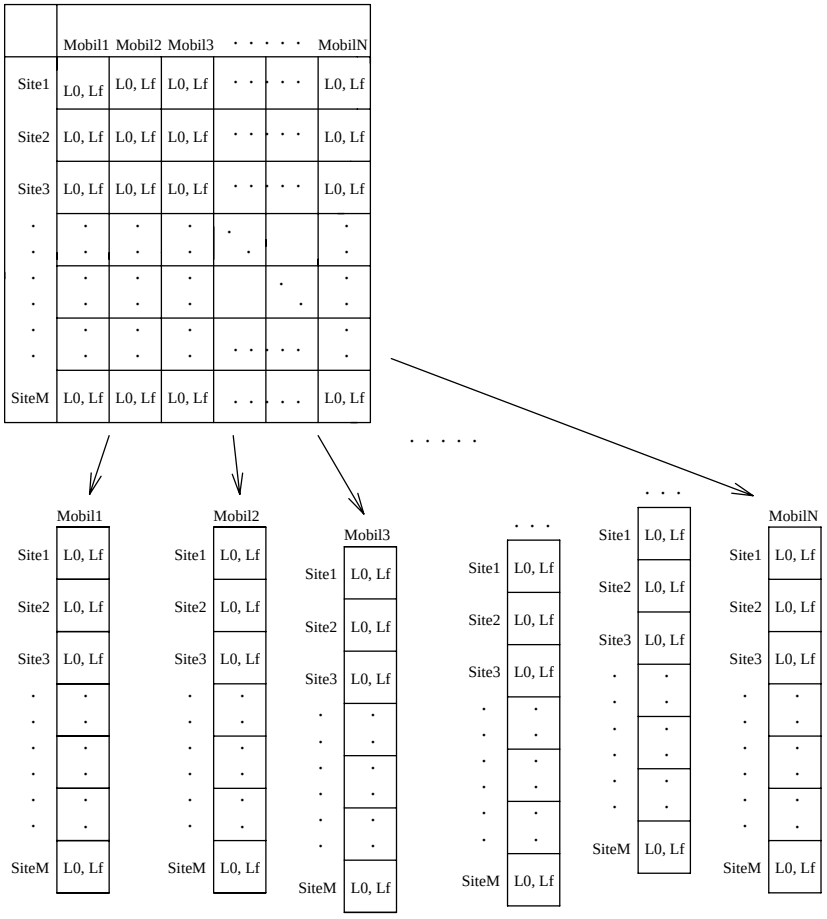



Figure 11.3 *The path matrix are separated among the mobiles*

### 11.4.1 Mobile

Each mobile is represented by two Erlang processes. One of the processes (the *mover* process) is scheduled by the eventhandler and generates the moving pattern of the mobile. The other process (the *mobile* process) is the process that calculates the SS -values, handles up-and down connections towards the sites and talks with radio resources (see below). By separating a mobile in two processes it is possible to get information from a mobile during the simulation even though it is suspended by the eventhandler. Each mobile will have information about all the sites in the system, (called the *path list*). This will lead to that the same information is duplicated among several mobile processes, but will remove the bottleneck with one large radio path matrix.

It is not necessary to represent the mobile with two processes. To reduce the number of processes in the simulation is it easy to transform the mover and mobile process to one process.

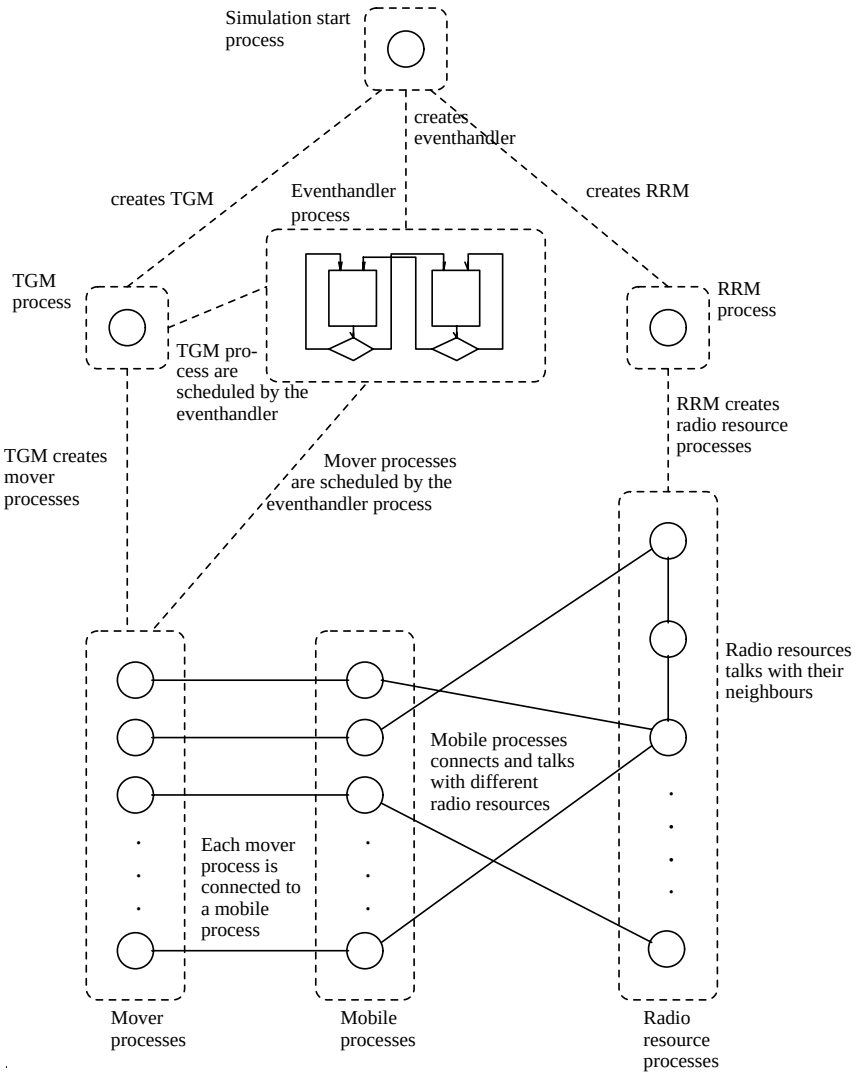



Figure 11.4 *The Erlang process oriented radio traffic simulation model*

### 11.4.2 Radio resource

Each frequency will be represented with a process and be called a radio resource process. A radio resource will handle all connections between mobiles and sites on the its corresponding frequency.

### 11.4.3 Manager

There are a few processes called managers that are used to start the simulation, creates mobiles (traffic generator manager, TGM) and radio resources (radio resource manager, RRM). By giving the start manager different input parameters the simulation can be made in a certain direction.

### 11.4.4 Action at an event

Several mover processes can be activated at the same time. When a mover process gets activated it will calculate where to move and send the new position to the mobile process. The mobile process uses the new position value and its radio path list to calculate the new SS value to the connected site. The SS value will be sent to the radio resource that manage the frequency. After that the mobile will check if it shall do a handover to another site. If it shall is a 'remove connection' message sent to the radio resource with the old frequency and a 'add connection' message sent to the radio resource with the new frequency. When the mobile process is ready it will send a 'ready' message to the mover process. The moverprocess will then make a `demo:hold` call that suspends the process on the eventhandler until it shall be activated again.

The radio resource process are not scheduled by the eventhandler. It can receive messages from mobile processes telling it to 'add' or 'remove' connections and may also receive SS values from mobiles that already are connected. The radio resource uses the SS value to calculate the C/I and C/A values. The C/I value can be calculated by the radio resource itself, but the C/A value must be calculated by cooperation of radio resources. The adjacent values to a frequency is held by the radio resources that represents the neighbour frequencies. The radio resource process will send a request for the adjacent value and the neighbour radio resource process will answer with the its calculated value.

## 11.5 Differences between the models

Differences between the Erlang model and ERA-t model are several. In the Elang model several mobiles can be activated at the same event, the large radio path matrix is eliminated and is instead spread out on the mobiles, the radio resource structure is represented by several processes instead of one large structure, the calculations made by a radio resource can be made during or after the calculations made by a mobile process. By splitting the calculations made by mobiles and radio resources the simulation becomes very suited for distribution.

Problems with the Erlang model are few but still important to notice. I can not guarantee that cars activated at the same event will send their messages in a certain order. This means that we can not be quite sure what values that the C/I and C/A values are calculated from. If we only allow one mobile to be running at the same event (*sequential* mode), like in the ERA-t model, this will not be a problem.

## 11.6 Conclusions

Conclusion and the thoughts of the radio traffic simulation application are:

- Radio mobile telephony showed to be much more complicated than I first could have guessed. This were the largest reason to that the radio traffic model I build had to be reconstructed from the beginning several times. My belief is that if I had got all relevant facts in an understandable way from the beginning my upstart

time would have been a little bit bigger, but the whole application construction time smaller. The construction of the model took therefore too much time.

- The functionality of the ERA-t model is much larger than my model but my task were just to make a minor part of their model and not a complete copy.
- I received much information about classes in SIMON but didn't get any wiser. If this depends on C++ or on ERA-t is though hard to tell.
- The model only used the `demo:hold` call of the simulation primitives in my package. It would have been much more fun to test my simulation package more throughout. The work became more a work in Erlang than a work in simulation.
- By distributing the simulation the radio resource processes can be placed on one node and the mobiles be placed one or several other nodes. This distribution model can though not be guaranteed to fit every radio traffic application.

# 12 A mining simulation application

A department at ÅF-industriteknik AB in Sweden that is working with simulations was contacted to give me the needed information for the second application. The department implements their simulation applications mostly in Simula but also in C++. It is mostly known for their train simulations made for SJ (Swedish railroads).

## 12.1 Assignment

It was decided that an application already made by the department should be remade, in Erlang, by me. The first problem therefore became to choose an application that were both interesting and small enough for the remaining time of the examwork. A mining simulation application made in Simula for Boliden Mindeco were decided as test application.

The application should simulate a mine with galleries, machines and mine workers. When a gallery shall be extracted it is made through a number of recurrent stages. Each stage consists of a number of working steps that must be performed in certain order. For example, the steps to extract a stage in a gallery could be: 1. bore the stage, 2. loaded the stage with explosive charges, 3. blow the stage into pieces, 4. safety control the stage, 5. carry away the largest stones from the stage, 6. clear the stage wall and ceiling, 7. remove the rest of the stones from the stage. For each step special machines and competent personal must be present.

The model should be able to answer questions on how the mining will be influenced by changes in the simulation world parameters. For example, machines or personal can be added or removed, personal can be further educated, personal working shift can be changed or machines can break down.

The purpose of the application can be read in the papers from Ångpanneföreningen: 'The purpose of the simulation model is to extract methods, under preserved or increased production, to reduces the number of galleries and the total waiting- inclusive transport time'.

My assignment became, by using the `demo` package, to implement the whole mining simulation model in Erlang.

## 12.2 The Simula model

The already existing model was implemented in Simula by personnel from Ångpanneföreningen. The model consists of a two dimensional area were a mine with one or several galleries are present. The figure below shows a mine with four galleries, A, B, C and D. Each gallery is divided into a number of stages, for example B is divided into four stages. A gallery extracts its stages in the order of the numbers. Some galleries has to wait for other galleries to be ready with their stages, for example gallery D can not be extracted before gallery C is finished with its third stage.

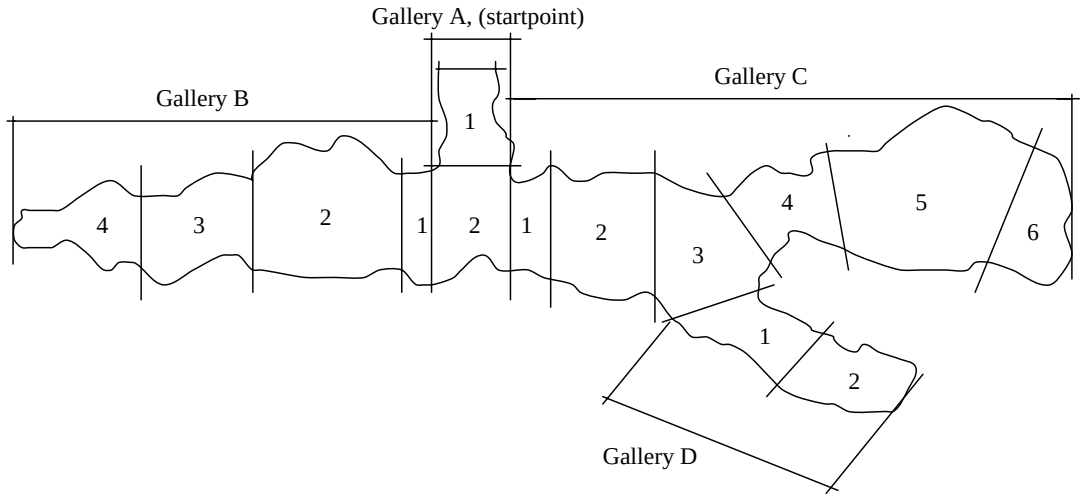



Figure 12.1 *A mine with four galleries*

Each gallery and machine is represented as an object in the Simula model. The workers are represented as one object that adds or deletes manpower depending on shift time or sickness of the workers. There are also an separate object that keeps the distance between the other objects in the simulation.

Every time a gallery shall begin the next step in the current stage a number of machines and workers must be achieved. For each achievement the gallery must both try to maximize the machine utilization and minimize the waiting- and transport time. This means that the model must be able to plan for the future and to check when machines and workers becomes available. For example it may be smarter to wait for a machine to be free near a gallery, instead of having another, at the moment free machine, be transported to the gallery from a distant place. A priority system is able to give certain combinations of galleries, machines and workers larger weight than others. The user is also able to stop the simulation to choose a certain combination. A gallery is also able to interrupt another gallery if it wants machines or manpower that the other gallery currently has taken.

The decision of which machines and workers that shall be given to a gallery is made by the gallery itself. The decision is made by examine attributes of the other galleries, machines, manpower object and distance object, and is a bit complicated. A gallery shall also react in a predefined manner when it gets interrupted.

The user is able to steer the mining simulation by setting a large number of input parameters.

## 12.3 The Erlang model

The Erlang mining simulation model should have the same functionality as the Simula model. If you runs the different applications with the same input parameters the resulting simulation values shall be almost equal.

I choose to represent each gallery, each machine and each worker as a separate pro-

cess. The distance between the different objects in the simulation (processes) are kept by an separate *distance* process. Instead of letting each gallery processes decide which machines and workers it shall achieve, the decision making to is moved to two separate processes, a *union*- and a *coordinator* process. Each time a gallery wants a machine or workers it will send requests to the decision processes. The decision processes keep records of the machines and workers that are free or taken and are therefore able to fulfil demands of different processes according to the priority system.

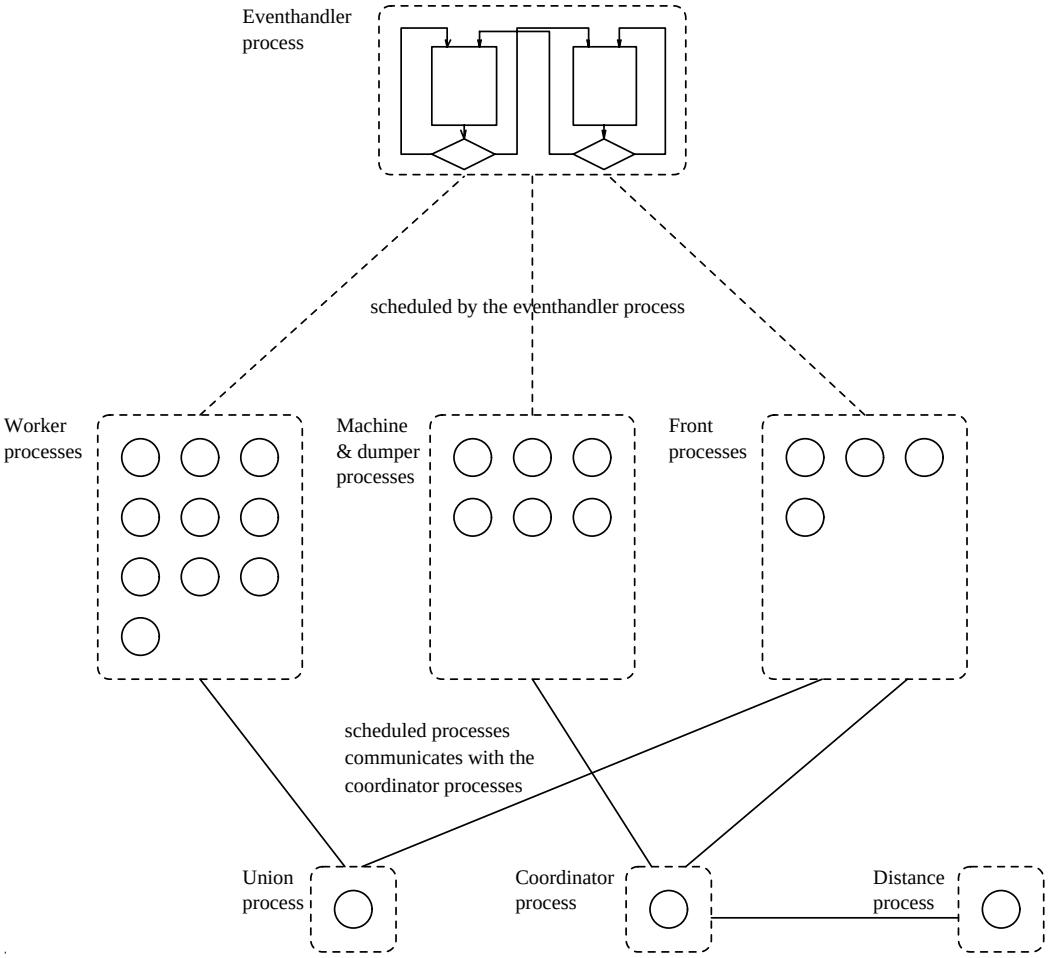



Figure 12.2 *The mine simulation model*

## 12.4 Conclusions

Conclusion and the thoughts of the application are:

- The time needed to construct an application of this size were clearly underestimated. Due to the fact that the ERA-t application took too much time, less time where available for the mining application. The application and the model are therefore just the guidelines of how I wanted to solve the different problems.
- The application were clearly distinguished from the ERA-t application. The ERA-t application only used `demo:hold` of all simulation package (`demo`) primitives. The Ångpanneföreningen application instead used almost everyone. The problem didn't lie in the Erlang programming as in the ERA-t application but were instead a simulation synchronization problem. A lot of missing functionality where detected, and had to be added to the `demo` package. For example the ability to interrupt processes and the `demo:attribute` calls. The mining application was not time critical, there was no need to distribute the simulation application.
- All the needed information for the simulation where given from the beginning. The upstart time where therefore smaller than in the ERA-t application.
- The Erlang code for the application, as far as I was able to proceed, became much simpler, shorter and survey able then the Simula code. This is a strong argument for using Erlang in simulation applications instead of imperative languages.

# 13 Distributed simulation

When the Erlang simulation package was ready the next task was to investigate the possibility to distribute the simulation. The idea was to extend the simulation package with distribution primitives that are built on the mechanisms for distribution that are available in Erlang.

## 13.1 Distribution in Erlang

Fundamental for understanding distribution is to understand the *node* concept. A node is an executing Erlang system which has told its nodename and network address to the network nameserver. Processes can be spawned on remote as on local nodes. Processes can send messages and create links to processes on remote nodes. The sending of a message to a process on a remote node is syntactically and semantically identical to sending a message to a process on a local node. It is also possible to register a process globally so that all processes over the network can access the process.

The coupling of nodes in Erlang is extremely loose. Nodes may come and go dynamically in manner similar to processes.

The programmer does not need to be concerned with details of the set-up of connections between nodes. All that has to be made is to use the node in a send expression or any other expression involving a remote process. Whenever a node needs to communicate with another node the Erlang system takes care of setting up the connection.

By using the distribution mechanisms present in Erlang the task to distribute the simulation did not became to hard to realize.

## 13.2 Why distribute the simulation?

When I looked at the possibility to distribute my simulation package, I looked at what must be the possible bottleneck (not a big one), the large communication between the simulation processes and the eventhandler. Some of the processes may also want to do large and time consuming calculations that the rest of the processes must wait for, which may slow down the simulation.

Two different alternatives to distribute the simulation package were investigated:

- One eventhandler, processes on different nodes.
- One global eventhandler, many servers and processes on different nodes.

In the following sections both investigated methods are described in more detail.

## 13.3 One eventhandler, processes on different nodes

The "one eventhandler process with processes located on different nodes" -model can be used when the number of processes are very large and wants to do time consuming calculations. The large problem with the model is that communication between processes not located on the same node increases. All operations, that influences the state of the simulation, must be made over the network. The nonlocal communication

is much more time consuming then normal message passing. The model is therefore most appropriate when processes not are using the predefined simulation primitives to much when they interact with one another. Also, the model will not be able to solve the bottleneck problem.

## 13.4 One global server, eventhandlers and processes on different nodes

It is often possible, in a simulation model, to separate between two or more groups of processes that will interact with one another much more than with the other processes during the simulation.

The approach is to let the constructor separate these *process groups* by himself. Each process group will be placed on a separate node and have an eventhandler on their own. All eventhandlers will be scheduled by a *global event handler* that makes sure that no eventhandler activates any process before the other eventhandlers allows it. If each node is located on a separate machine it is possible to make the simulation really distributed. The global eventhandler will not schedule any simulation processes but only handle the different eventhandlers.

A process can interact with a nonlocal process or resource in exactly the same way as with a local process. Almost all things that are possible when the simulation is running in non distributed mode is also possible when it runs in distributed mode. Due to the fact that process communication over the network takes longer time than normal communication, it is important to plan the processgroups so that nonlocal communication is minimized.

### 13.4.1 Start the global server

The distribution is started by the creation of the process that will schedule all the eventhandlers, this is done with the `demo_global:start(NodeList)` call. The created process will be called the *global eventhandler*. The global eventhandler process will be running at the node that is local to it (if the `demo_global:start/1` call not is spawned on a separate node). The input argument to the `demo_global:start` call is a list of nodes where eventhandlers shall be started. It is not possible to create two eventhandlers on the same node or to create an eventhandler on a node where a eventhandler already is running. The global eventhandler process will be globally registered by the name `demo_global` on all nodes where the eventhandlers are running.

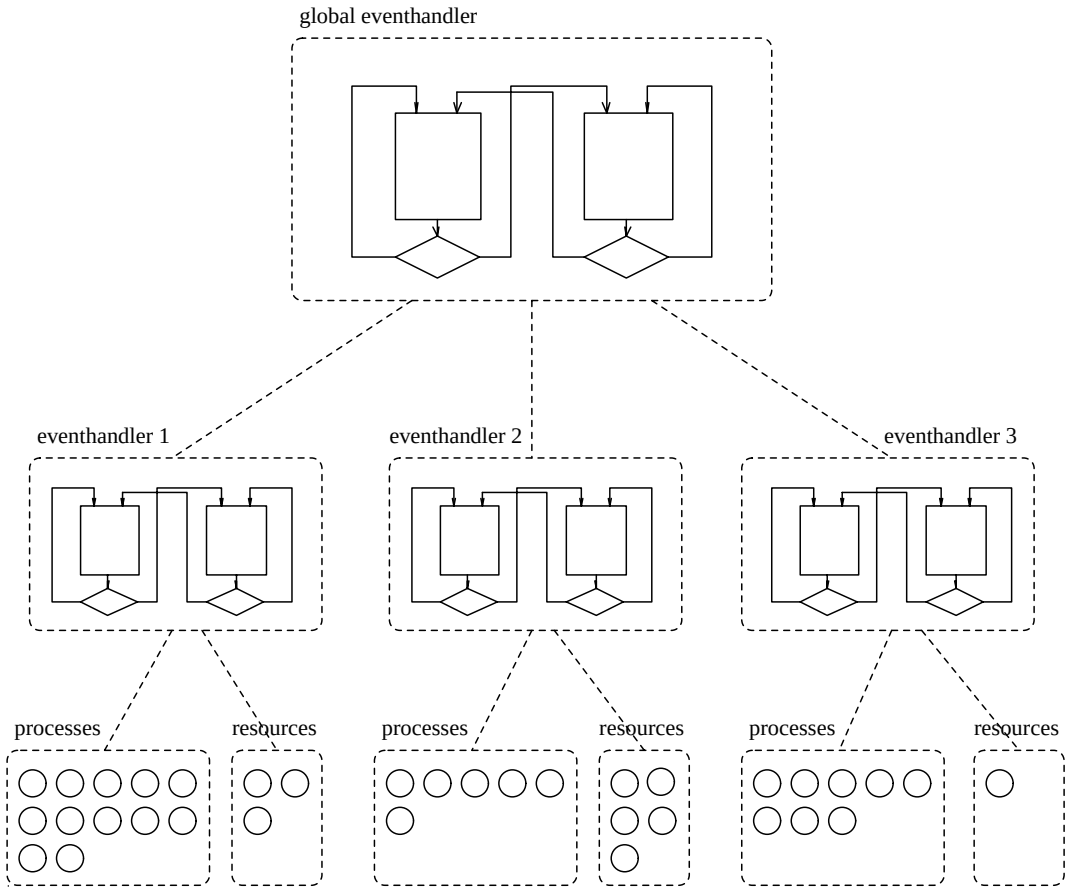



Figure 13.1 *Distributed simulation model*

### 13.4.2 Create a process on a nonlocal node

A process can be created on a remote node with a `demo_global:new_PN` call. The `demo_global:new_PN` corresponds to the `demo:new_P` calls that is made on a local node. The only difference is that the `demo_global:new_PN` call have the node where the process shall be spawned on and scheduled by the eventhandler as an extra argument.

### 13.4.3 Global calls

There are some calls that can be made to the global server that affects all the executing eventhandlers. For example it is possible to turn on or off the tracing on all eventhandlers. All these calls are made with the `demo_global` module. See appendixD for more information about `demo_global` calls.

### 13.4.4 Local calls

Almost all calls that can be made to steer the simulation on a local eventhandler when a simulation is running in nondistributed mode can also be made when the simulation is running in distributed mode. For example, it is possible to set the simulation to *sequential* mode on one eventhandler and still have the simulation running in *parallel* mode on the other eventhandlers.

All calls that handles interaction between processes can be made in exactly the same way as they were made in the local simulation. Processes can communicate with nonlocal processes, passivate on nonlocal processes and resources, get attributes from nonlocal processes etc.

The only calls that not are permitted when the simulation are running in distributed mode are local `demo:suspend` and `demo:resume` calls. The reason for that will be explained in the following section.

### 13.4.5 Implementation

The global eventhandler uses the `demo:suspend` and `demo:resume` calls to schedule all local eventhandlers. When an eventhandler has finished an event (all simulation processes are ready) it will examine at what event it shall stop the next time. The stop time and the current time in the eventhandler is thereafter sent to the global eventhandler. When the global eventhandler updates its time to the next event the eventhandlers that are scheduled to execute its next action on that event are restarted. You can compare this to the `demo:hold()` call that a process will do to an eventhandler when it shall be suspended.

### 13.4.6 Problems with distribution

The extension from "one node simulation" to "distributed simulation" became very simple. Though, some changes on the simulation package had to be made:

- To be sure that the `demo:suspend` or `demo:resume` calls only are made when they are allowed, tests to check if the global eventhandler is running must be implemented.
- All calls that involves interaction between two or more processes had to be remade as several calls. The reason for that is that the processes can be scheduled by different eventhandlers that want one separate call to change their internal state.
- The eventhandler must send information to the global eventhandler every time it is stopped or changes its next stoptime.
- Operations to create processes on nonlocal nodes had to be implemented.
- Operations that allows processes to migrate from one eventhandler to another had to be implemented. (Not real process migration, just change of eventhandler process).

The global eventhandler and the calls to change its way to steer the simulation had to be implemented.

### 13.4.7 Appropriate simulation applications

The distributed simulation model is best suited for large process simulations where many processes are activated at the same event. It is also preferable if the simulation processes can be separated in a number of process groups.

### 13.4.8 Even more distribution

If an application becomes very large it is fully possible to create an eventhandler that schedules a couple of global eventhandlers. Or if it gets really huge, create a eventhandler that schedules the eventhandlers that schedules the eventhandlers that schedules the eventhandlers that schedules the processes, etc.

```
%% An example of a distributed simulation program where two processes
%% interrupt each other, resulting in that 'Hello world!!' is written
%% on the screen. The processes are scheduled by two separate event-
%% handlers that are located on two separate nodes.
    -module(dist_test),
    -export([p0/2, p1/0, p2/0]).

    p0(Node1, Node2) ->                     %% start global server and eventservers
        demo_global:start([Node1, Node2]),  %% on the nodes Node1 and Node2.
        demo_global:trace_g(),              %% start trace on global server
                                            %% create process on node Node1
        P1 = demo_global:new_PN(Node1, dist_test, p1, []),
                                            %% create process on node Node2
        P2 = demo_global:new_PN(Node2, dist_test, p2, [P1]),
        P1 ! P2,
        demo_global:resume().               %% start simulation

    p1() ->
        receive
            P2 ->                           %% receive the pid of P2
                demo:interrupt(P2, 'Hello '),%% interrupt P2
                demo:hold(1),               %% wait to be interrupted
                Message = demo:get_interrupt(), %% get interrupt message
                io:format('~w',[Message]),  %% print message
        end.

    p2(P1) ->
        demo:hold(1),                       %% wait to be interrupted
        Message = demo:get_interrupt(),     %% get interrupt message
        io:format('~w',[Message]),          %% print message
        demo:interrupt(P1, 'world!!').      %% interrupt P1
```

# 14 Erlang as simulation glue

An interesting extension of Erlang's usefulness in simulation purposes is as a glue between non parallel simulation systems. An Erlang process can be assigned the task to coordinate the different simulation systems. The scheduled systems can, for example, be a couple of C++ or Simula systems or a mixture of both. If the different systems are located on remote nodes Erlang's distribution mechanisms can be used to distribute the simulation.

As you can see, real parallel execution can be achieved in a non parallel simulation language with the use of Erlang. Figure 14.1 shows a possible model for coordination of several different simulation systems.

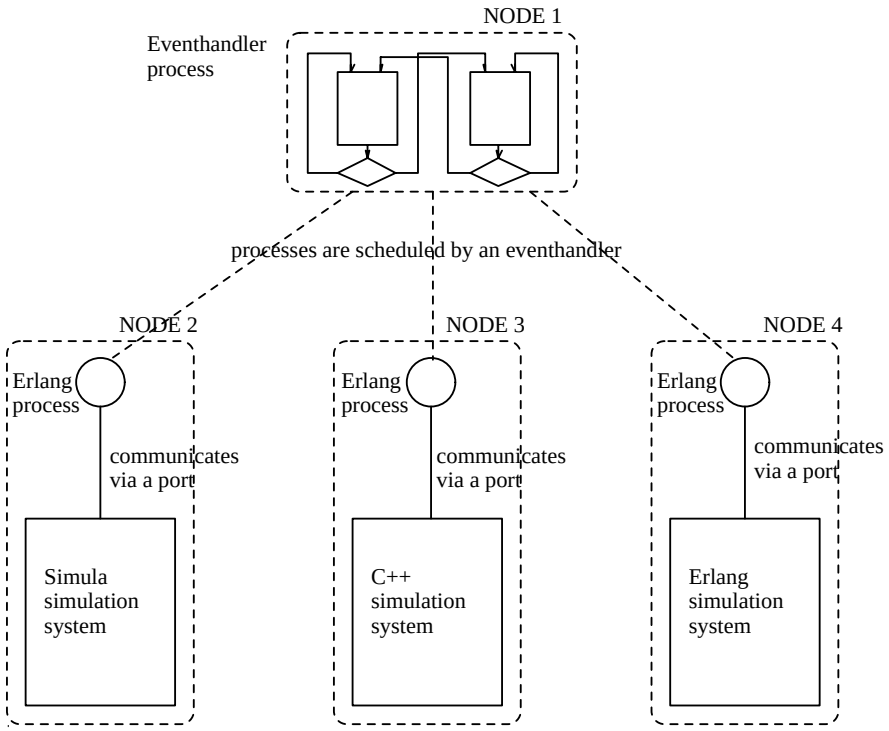



Figure 14.1 *Different simulation systems coordinated by Erlang*

The distribution granularity and the use of Erlang depends on the specific simulation application. It is therefore important that the different systems can be scheduled in a similar manner.

# 15 Conclusions

The conclusions that can be drawn from using Erlang in simulations are:

- Erlang is a language that is easy to learn and program. The light-weight process concept introduces the programmer to a new way of thinking, and the declarative programming style makes the code shorter and more error secure. By using Erlang the programmer doesn't have to handle hard programming problems (at least I didn't), like pointers in outer space, instead he could concentrate on the real simulation problems.

- The process oriented simulation model became a little more complicated then the comparable object oriented simulation model (Simula). This is because the simulation state is held by a single process in Erlang instead of several objects, as in Simula, and that simulation processes are allowed to execute in parallel. It has been my hope to hide as much as possible of the difficulties for the programmer, and instead let him concentrate of the real simulation problems.

- The Erlang simulation model is very suitable to large simulations with big parallelism. The simple distribution mechanisms in Erlang makes the extension to distributed simulation really easy.

- Both conservative and optimistic simulation models can easily be implemented in Erlang. The light-weight processes and the simple message passing primitives are very appropriate for these kind of simulations.

- Interesting effects are achieved by letting several simulation processes execute in parallel. It is my belief that this is more like in real life where several things can happen simultaneously. The

- By using Erlang's error handling mechanisms it is easy to achieve an error secure simulation. The whole simulation will not be stopped if a process terminates abnormally.

- The use of Erlang as simulation glue is an describing example of Erlang's many program eventualities.

# 16 AppendixA

Below is a list of the specific commands that the eventhandler can handle:

## Start Commands

`demo:start()`
: starts the eventhandler process. This must be made before any other call is made. If the eventhandler already is started an error message is printed and false is returned. Otherwize the eventhandler process is started and the file descriptor (pid) of the created eventhandler process is returned.

`demo:start(Module, Func, Args)`
: starts the eventhandler process with an executing process. The arguments are module, function and argumentlist of the function. The same result can be achieved by a `demo:start()` call and `demo:new_P(Mod, Func, Args)` call. The file descriptor (pid) of the created eventhandler process is returned.

`demo:start(Module, Func, Args, Time)`
: starts the eventhandler process with an executing process. The arguments are module, function, argumentlist and the eventtime to start the function. The same result can be achieved by making a `demo:start()` call and a `demo:new_P(Mod, Func, Args, Time)` call. The file descriptor (pid) of the created eventhandler process is returned.

`demo:start(Module, Func, Args, Time, Priority)`
: starts the eventhandler process with an executing process. The arguments are module, function, argumentlist, time to start the function and priority of the process. The same result can be achieved by making a `demo:start()` call and `demo:new_P(Mod, Func, Args, Time, Priority)` call. The file descriptor of the created eventhandler process is returned.

## Termination Commands

`demo:stop()`
: terminates all simulation processes, resources and the eventhandler process, i.e. ends the simulation. Returns `true`.

`demo:reset()`
: terminates all simulation processes and resources but not the eventhandler process. Returns `true`.

`demo:kill_current()`
: terminates all at the moment active simulation processes. This command is useful when one ore more processes has deadlocked. Returns `true`.

`demo:kill(Nr1, Nr2, Nr3)`
: Kills a process by sending it a exit message. The arguments are the process pidnumber (`<Nr1, Nr2, Nr3>` = Pid). Returns `true`.

## Suspend Commands

`demo:suspend(Time)`
: suspends the eventhandler process. The argument to `suspend` can be `next` which means that the simulation will stop at the next event, or a positive number which means that the simulation will stop at the current time + `Time`. Returns `true` or `false`.

`demo:suspend_at(Time)`
: suspends the eventhandler process at a specific time `Time`. The argument to `demo:suspend_at` is a positive number and shall be larger than the current time. Returns `true` or `false`.

`demo:suspend_reg(Time)`
: suspends the eventhandler process at regular time intervals. The argument to `suspend_reg` can be `next` which means that the eventhandler will stop regularly after each event, or a positive number which means that the simulation will stop regularly with a time interval of the number. Returns `true` or `false`.

`demo:suspend_del()`
: removes all suspendpoints from the eventhandler process. Returns `true`.

## Resume Commands

`demo:resume()`
: restarts the eventhandler when it has been suspended. `demo:resume()` shall also be called to start the simulation after all initializations have been made. Returns `true`.

`demo:resume_del()`
: restarts the eventhandler when it has been suspended and removes all suspendpoints. Returns `true`.

## Time Commands

`demo:event_time()`
: returns the current eventtime in the simulation.

## Information Commands

`demo:r()`
: returns all simulation resources, more for debugging reasons.

`demo:p()`
: returns all simulation processes, more for debugging reasons.

`demo:e()`
: returns the eventtree in the eventhandler, more for debugging reasons.

`demo:s()`
: returns the attributelist in the eventhandler, more for debugging reasons.

`demo:c()`
: returns a list of all active processes, more for debugging reasons.

## Concurrency Commands

`demo:set_concurrency(Mode)`
: sets how the eventhandler shall run processes scheduled at the same event. The `Mode` form can be `sequential` or `parallel`. `parallel` means that all processes scheduled at the same time will be merged together into a single event and be activated in parallel. `sequential` means that only one process can be active at the same time. If two process are scheduled at the same time one process will first execute, and when it is ready, the second process will be activated. In `sequential` mode a process with higher priority will start before a process with lower if their starttimes are equally. Default value is `parallel`. Returns `true` or `false`.

## Granularity Commands

`demo:set_granuality(Granularity)`
: sets the granularity in the eventhandler. The argument is a positive number which indicates in how large timesteps the eventhandler shall step forward. For example, if `Granuality` is 50 and the current time is 30 the eventhandler will start all scheduled processes between 30 and 80 at the same time. If a process starts, stops and starts again in the same granularity interval

this is done sequentially. An granularity larger then 0 implies that the concurrency mode will automatically be `parallel`. Default value is 0. Returns `true` or `false`.

## Trace Commands

`demo:trace()`
: turns on the tracer for the simulation. Returns `true`.

`demo:notrace()`
: turns off the tracer for the simulation. Returns `true`.

## Create Process Commands

`demo:new_P(Mod, Func, Args)`
: creates a new process to be scheduled by the eventhandler at the current time and with priority 0. The arguments are module, function and argumentlist to the function. Returns the file descriptor (pid) of the created process.

`demo:new_P(Mod, Func, Args, Time)`
: creates a new process to be scheduled by the eventhandler at the time specified in the fourth argument. The arguments are module, function, argumentlist and starttime. Returns the file descriptor (pid) of the created process.

`demo:new_P(Mod, Func, Args, Time, Priority)`
: creates a new process to be scheduled by the eventhandler with the priority as specified in the fifth argument. The arguments are module, function, argumentlist, starttime and priority. Returns the file descriptor (pid) of the created process.

## Filehandler Commands

`demo:open_file(Filename, Mode)`
: opens a file. The arguments are the name of the file and `Mode`. The `Mode` can be `read`, `write` or `read_write`. The command corresponds against the `file:open/2` command in Erlang. To make sure that the file will exist during the whole simulation, (and not exit with the process that creates it), the eventhandler will handle the opening of the file. Returns `{error, open}` if the opening failed or a file descriptor (pid) if the opening succeeded.

## Hold Commands

`demo:hold(Time)`
: passivates the calling process and schedules it to be activated at the current time + `Time`. `Time` shall be a positive number. Returns `true` or `false`.

## Cancel Commands

`demo:cancel()`
: removes the current process from the simulation. The only way the process can be reactivated is with a `demo:interrupt` call from another process. Returns `true` or `false`.

## Passivate Commands

`demo:passivate(Pid)`
: suspends the calling process on another process. The argument is the corresponding process descriptor (pid) of the process that the process shall be suspended on. The calling process is suspended until it gets activated (`demo:activate`) or terminated by the eventhandler. Returns `true` or `false`.

`demo:passivated()`
: returns a list of processes that are passivated on the calling process.

`demo:passivated(Pid)`
: returns a list of processes that are passivated on the process Pid.

## Repassivate Commands

`demo:repassivate(PidOn, Pid)`
: repassivates a process Pid, that already is passivated on on the calling process, on the process PidOn. Returns `false` if the process Pid not is passivated on it or if the PidOn not is registered, otherwize `true`.

`demo:repassivate_all(PidOn)`
: repassivates all process, that already are passivated on the own processes, on the process PidOn. Returns `false` if the process PidOn not is registered, otherwize `true`.

## Activate Commands

`demo:activate(Pid)`
: Immediately activates a process that are passivated on the calling process. Takes the pid of the process that shall be activated as argument. If the process not is passivated on the calling process the process is not activated and `false` returned. Otherwise the process is activated and `true` returned.

`demo:activate(Pid, Time)`
: activates, at the specified time, a process that are passivated on the calling process. Takes a file descriptor and a positive number as argument. If the process with the file descriptor not is passivated on the calling process no process is activated and `false` returned. Otherwise the process is activated at the current time + `Time`, and `true` is returned.

`demo:activate_all()`
: activates all processes that are passivated on the calling process. All passive processes are actitivated immediately. Returns `true`.

`demo:activate_all(Time)`
: activates, in time `Time`, all processes that are passivated on the calling process. Takes a positive number as argument, and activates all its passivated processes at current time + `Time`. Returns `true`.

## Priority Commands

`demo:get_priority()`
: returns the priority of the calling process. The default value is 0.

`demo:get_priority(Pid)`
: returns the priority of the process Pid. Returns `false` if the process not is a simulation process.

`demo:set_priority(Priority)`
: sets the priority of a process. A process can only set its own priority. Returns the old value of the priority.

## Interrupt Commands

`demo:interrupt(Pid, Message)`
: interrupts a process if the process is suspended on the eventhandler. If the process not is suspended in the eventhandler it can not be interrupted and `false` is returned. Otherwize the process will be activated with the interrupt message `Message`.

`demo:get_interrupt()`
: returns the message used to interrupt the calling process. If the calling process not were interrupted `undefined` or the previous interrupt message is returned.

`demo:set_interrupt(Message)`
: sets the interrupt variable on the calling process to message `Message`. Is useful when the process wants to reset its interrupt message to `undefined` after it has been interrupted. Returns the old interrupt message.

`demo:turn_interrupt(OnOff)`
: Turns off or on the possibility for another process to interrupt the process. The argument can be `on` (can be interrupted) or `off` (can not be interrupted). Returns the old value of the variable.

## Attribute Commands

`demo:set_attribute(Attribute, Value)`
: Sets the attribute `Attribute` to the value `Value`. Other processes can have access to the value via `demo:get_attribute` calls. The attribute can only be set by the own process. This is implemented by using the process dictionary.

`demo:get_attribute(Pid, Attribute)`
: finds the value of an attribute that belongs to another process. If the attribute not was found `undefined` is returned. If the value was found it is returned.

`demo:get_all_attributes(Pid)`
: returns all attributes that belongs to a process. The attributes are returned as a list of `{attribute, value}` tuples.

## Error Commands

`demo:stop_when_error(TrueOrFalse)`
: sets if the simulation shall stop when the eventhandler discovers an error or when an a registered process has exited abnormally. If the argument is `true` the simulation will stop. If the argument is `false` the simulation will not stop. The default value is `false`.

`demo:set_error_log(Arg)`
: sets where the eventhandler shall write generated error messages. The argument can be `standard_io` (standard output) or a process identifier (pid). The default value is `standard_io`.

# 17 AppendixB

Below is alist of the specific resource commands that the simulator can handle:

## General Resource Commands

`demo:new_R(Module, List)`
creates a new resource process and links it to the eventhandler. The arguments are module and initialization list. The resource is initialized with the initialization list, (`List`). Returns the file descriptor (pid) of the created resource.

`demo:call_R(Resource, Function, Args, Priority)`
calls a resource. Every call to an object, that not is a report call, must be made with

`call_R`. The arguments are process descriptor (pid) of the called resource process, function, argumentlist and priority. The module parameter were specified in the creation of the resource. Returns a value from the resource or passivates the calling process.

`demo:report_R(Resource, Function, Args)`
calls a resource with a report request. The arguments are file descriptor of the resource process, function and argumentlist. The resource writes the report to a specified output. The module used is the `demo_resource` module. The report functions are common functions to all resources. Returns a value from the function.

## Reports Commands

`demo:reportR(Resource)`
tells the resource to write a report to the reportfile.

`demo:report_headingR(Resource)`
tells the resource to write the indexes of the variables that shall be reported.

`demo:set_reportfileR(Resource, File)`
sets the file that the resource shall write its reports to.

`demo:reset_reportR(Resource, ReportFile)`
resets the variables that the resource shall report.

`demo:add_reportR(Resource, IndexVariable)`
adds a indexvariable to the ReportList of the resource.

`demo:del_reportR(Resource, IndexVariable)`
deletes the indexvariable from the ReportList of the resource.

## Information Commands

`get_all_attributesR(Resource)`
returns the AList of the resource.

`get_attributeR(Resource, Attribute)`
returns the value of the attribute thet corresponds to Attribute. If not found false is returned.

`set_attributeR(Resource, Attribute, Value)`
sets a value of an attribute in the AList of the Resource. To be used carefully.

## Count

### Variables

The count resource has five variables in the AList that represents the state of resource:
1. *observations*, the number of `demo:update` calls made to the count since the last call of `demo:reset`.
2. *sum*, the sum of input values since the last call of `demo:reset`.

3. *resettime*, the last time the count were resetted.
4. *report*, the list of items that shall be reported.
5. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(count, [])`
creates a count resource.

`demo:update(Count)` or `demo:update(Count, N)`
increments *sum* with 1 respective N and *observations* with 1.

`demo:reset(Count)`
sets *sum* and *observations* to 0 and *resettime* to the current eventtime.

## Tally

### Variables

The tally resource has eleven variables in the AList that represents the state of resource:
1. *observations*, the number of `demo:update` calls made to the tally since the last call of `demo:reset`.
2. *sum*, the sum of input values since the last call of `demo:reset`.
3. *sumsQ*, the square sum of input values since the last call of `demo:reset`.
4. *min*, the smallest value of the input values since the last call of `demo:reset`.
5. *max*, the largest value of the input values since the last call of `demo:reset`.
6. *mean*, the mean value of the input values since the last call of `demo:reset`.
7. *variance*, the variance of the input values since the last call of `demo:reset`.
8. *devation*, the devation of the input values since the last call of `demo:reset`.
9. *resettime*, the last time the tally were resetted.
10. *report*, the list of items that shall be reported.
11. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(tally, [])`
create a tally resource.

`demo:update(Tally)` or `demo:update(Tally, N)`
increments *observations* with 1, *sum* with N, and *sumsQ* with N * N, sets *min* to the smallest of N and the previous Min value, sets *max* to the largest of N and the previous Max value and calculates the new *mean*, *variance* and *devation* for the input values.

`demo:reset(Tally)`
sets *observations*, *sum*, *sumsQ* to 0 and *min*, *max*, *mean*, *variance* and *devation* to *no_value* and *resettime* to the current eventtime.

## Histogram

### Variables

The histogram resource has eight variables in the AList that represents the state of resource:
1. *observations*, the number of `demo:update` calls made to the tally since the last call of `demo:reset`.
2. *lower*, the expected lower bound of the input variables.
3. *upper*, the expected upper bound of the input variables.
4. *cells*, the chosen number of recording cells.
5. *table*, the table where all input values are recorded. It consists of a list of tuples. Each tuple looks like: {*LowLimit, UpLimit, Items*} where *LowLimit* is the lower limit of the cell, *UpLimit* is the upper limit of the cell and *Items* are the number of input values between LowLimit and UpLimit since the last call of `demo:reset`. Two separate cells for underflow and overflow values are also present.

6. *resettime*, the last time the histogram were resetted.
7. *report*, the list of items that shall be reported.
8. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(histogram,[Upper, Lower, Cells])`
creates a histogram resource.

`demo:update(Histogram, N)`
increments *observations* with 1, locates the appropriate cell in *table* and increments the cells *Items* by 1.

`demo:reset(Histogram)`
sets all the *Items* in all cells in *table* to 0 and *resettime* to the current eventtime.

## Accumulate

### Variables

The accumulate has ten variables in the AList that representates the state of resource:
1. *observations*, the number of `demo:update` calls made to the tally since the last call of `demo:reset`.
2. *sumT*, the time integral of the input values since the last call of `demo:reset`.
3. *sumsQT*, the time integral of the squares of input values since the last call of `demo:reset`.
4. *min*, the smallest value of the input values since the last call of `demo:reset`.
5. *max*, the largest value of the input values since the last call of `demo:reset`.
6. *lasttime*, the time of the last `demo:update` call.
7. *lastvalue*, the last input value.
8. *resettime*, the last time the accumulate were resetted.
9. *report*, the list of items that shall be reported.
10. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(accumulate, [])`
creates an accumulate resource.

`demo:update(Accumulate)` or `demo:update(Accumulate, N)`
increments *observations* with 1, *sumT* with *lastvalue* * (eventtime - *resettime*), *sumsQT* with *lastvalue* * *lastvalue* * (eventtime - *resettime*), sets *min* to the smallest of N and the previous Min value, *max* to the largest of N and the previous Max value, *lastvalue* to N and *lasttime* to eventtime.

`demo:reset(Accumulate)`
sets *observations*, *sumT*, *sumsQT* to 0 and *min*, *max* to *no_value* and *resettime* and *lasttime* to the current eventtime.

## Bin

### Variables

The bin has 16 variables in the AList that representates the state of resource:
1. *observations*, the number of `demo:take` calls made to the bin since the last call of `demo:reset`.
2. *wqueue*, a list of passivated processes with their demands.
3. *wlength*, the length of the *wqueue*.
4. *wlength_max*, the largest length of the *wqueue* since the last call of `demo:reset`.
5. *wlength_int*, records the timeintegral for the length of the *wqueue* since the last call of `demo:reset`.
6. *w_lasttime*, records the last time a process entered or left the *wqueue* since the last call of `demo:reset`.
7. *avail*, the currently available amount of the bin reseource.

8. *avail_min*, records the minimum value for *avail* since the last call of `demo:reset`.
9. *avail_int*, records the timeintegral for *avail* since the last call of `demo:reset`.
10. *a_lasttime*, the time when the last change to *avail* was made since the last call of `demo:reset`.
11. *capacity*, the amount available from the beginning.
12. *concurrency*, how the resource shall start passivated processes.
13. *test*. How the resource shall test passivated processes in the *wqueue* when a producer process gives some items to the bin. If *test == all* shall we test all passivated processes in the *wqueue*, else if *test == priority* shall we only test the first. Default *priority*.
14. *resettime*, the last time the bin were resetted.
15. *report*, the list of items that shall be reported.
16. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(bin, [Capacity])`
creates a bin resource.

`demo:give(Bin, Items)`
increase the *avail* variable with Items and tries to start passivated processes in *wqueue*. After that is the *wqueue* and *avail* variables updated.

`demo:take(Bin, Items)`
tries to give items to the calling process. If this not is possible is the calling process passivated on the *wqueue*. Otherwize, if items are avaialable is the process started again and *avail* is updated. The *wqueue* and the *avail* variables are updated.

`demo:reset(Bin)`
sets *observations*, *wlength_int*, and *wlength_max* to 0, *resettime*, *w_lasttime* and *a_lasttime* to the current eventtime, *wlength_max* to *wlength* and *avail_min* to *avail*. The rest of the variables are unchanged

## Res

### Variables

The res has 21 variables in the AList that representates the state of resource:
1. *observations*, the number of *demo:acquire* calls made to the res since the last call of `demo:reset`.
2. *wqueue*, a list of passivated processes with their demands.
3. *wlength*, the length of the *wqueue*.
4. *wlength_max*, the largest length of the *wqueue* since the last call of `demo:reset`.
5. *wlength_int*, records the timeintegral for the length of the *wqueue* since the last call of `demo:reset`.
6. *w_lasttime*, records the last time a process entered or left the *wqueue* since the last call of `demo:reset`.
7. *tqueue*, a list of processes holding items. 8. *tlength*, the length of the *tqueue*.
9. *tlength_max*, the largest length of the *tqueue* since the last call of `demo:reset`.
10. *tlength_int*, records the timeintegral for the length of the *tqueue* since the last call of `demo:reset`.
11. *t_lasttime*, records the last time a process entered or left the *tqueue* since the last call of `demo:reset`.
12. *avail*, the currently available amount of the res reseource.
13. *avail_min*, records the minimum value for avail since the last call of `demo:reset`.
14. *avail_int*, records the timeintegral for avail since the last call of `demo:reset`.
15. *a_lasttime*, the time when the last change to avail was made since the last call of `demo:reset`.
16. *capacity*, the amount available from the beginning.
17. *concurrency*, how the resource shall start passivated processes.
18. *test*. How the resource shall test passivated processes in the *wqueue* when a producer process gives some items to the rin. If *test == all* shall we test all passivated processes in the *wqueue*, else if *test == priority* shall we only test the first. Default is *priority*.
19. *resettime*, the last time the res were resetted.
20. *report*, the list of items that shall be reported.
21. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(res, [Capacity])`
creates a res resource.

`demo:acquire(Res, Items)`
tries to give items to the calling process. If this not is possible is the calling process passivated on the *wqueue*. Otherwize, if items are available is the process started again and *avail* and *tqueue* is updated. A process can not ask for more items the it has taken before.

`demo:release(Res, Items)`
removes some items from the process back to the resource. The process can only release items if it has taken more or equal items before with a *demo:acquire/2* call. If any process demand in the *wqueue* can be fullfilled with help of the new items is the passivated process started. The *tqueue*, *wqueue* and *avail* variables are updated.

`demo:reset(Res)`
sets *observations*, *wlength_int*, *wlength_max*, *tlength_int* and *tlength_max* to 0, *resettime*, *w_lasttime*, *t_lasttime* and *a_lasttime* sets to the current eventtime. *wlength_max* sets to *wlength*, *tlength_max* sets to *tlength*, and *avail_min* sets to *avail*. The rest of the variables are unchanged.

## Condq

### Variables

The condq has eleven variables in the AList that represents the state of resource:
1. *observations*, the number of `demo:update` calls made to the condq since the last call of `demo:reset`.
2. *condq*, a list of passivated processes with their condition demands.
3. *clength*, the length of the wqueue.
4. *clength_max*, the largest length of the *condq* since the last call of `demo:reset`.
5. *clength_int*, records the timeintegral for the length of the *condq* since the last call of `demo:reset`.
6. *c_lasttime*, records the last time a process entered or left the *condq* since the last call of `demo:reset`.
7. *concurrency*, how the resource shall start passivated processes.
8. *test*, how the resource shall test passivated processes in the condq when a process does a *demo:signal* call to the condq. If *test == all* shall we test all passivated processes in the *wqueue*, else if *test == priority* shall we only test the first. Default is *priority*.
9. *resettime*, the last time the condq were resetted.
10. *report*, the list of items that shall be reported.
11. *file*, the file to write reports to, default is standard_io.

### Commands

`demo:new_R(condq, [])`
creates a condq resource.

`demo:wait_until(Condq, Module, Func, Arg)` and `demo:wait_until(Condq, Module, Func, Arg, Result)`
enters a process into the condq. The function *Module:Func(Args)* are evaluated to see if it evaluates to *true* or *Result*. If so are the process immideatelly released from the condq. Otherwize is the process suspended on the condq until *Module:Func(Args)* is tested and evaluated to the wanted result. The *observations* and *cqueue* variables are updated.

`demo:signal(Condq)`
makes the condq let all his passivated processes test their condition. If the *Module:Func(Args)* that belongs to a process is evaluated to the wanted result is the passivated process released. Otherwize is the process repassivated on the condq. The *observations* and *cqueue* variables are updated.

`demo:reset(Condq)`
sets *observations* and *clength_int* to 0 and *clength_max* to *clength*, *resettime* and *c_lasttime* sets to the current eventtime. *clength_max* sets to *clength*. The rest of the variables are unchanged.

# Waitq

## Variables

The waitq has 16 variables in the AList that represents the state of resource:
1. *observations*, the number of `demo:update` calls made to the condq since the last call of `demo:reset`.
2. *squeue*, a list of slave process waiting for their master. 3. *slength*, the length of the *squeue*.
4. *slength_max*, the largest length of the *squeue* since the last call of `demo:reset`.
5. *slength_int*, records the timeintegral for the length of the *squeue* since the last call of `demo:reset`.
6. *s_lasttime*, records the last time a process entered or left the squeue since the last call of `demo:reset`.
7. *mqueue*, a list of master processes waiting for processes to enslave. 8. *mlength*, the length of the *mqueue*.
9. *mlength_max*, the largest length of the *mqueue* since the last call of `demo:reset`.
10. *mlength_int*, records the timeintegral for the length of the *mqueue* since the last call of `demo:reset`.
11. *m_lasttime*, records the last time a process entered or left the *mqueue* since the last call of `demo:reset`.
12. *concurrency*, how the resource shall start passivated processes.
13. *test*, how the resource shall test passivated processes in the *mqueue* when a slave process passivates on the waitq. If *test == all* shall we test all passivated processes in the *mqueue*, else if *test == priority* shall we only test the first. Default is *priority*.
14. *resettime*, the last time the waitq were resetted.
15. *report*, the list of items that shall be reported.
16. *file*, the file to write reports to, default is standard_io.

## Commands

`demo:new_R(waitq, [])`
: creates a waitq resource.

`demo:wait(Waitq)`
: If there are no master available or if the slave cant fullfill the masters demand of processes is the slave passivated in the *squeue*. If there are any masterprocess available thats demand of processes would be fullfilled with the new process is the demanded slave processes passivated on the masterprocess and the masterprocess is activated. The *squeue* and *wqueue* variables are updated.

`demo:coopt(Waitq)` and `demo:coopt(Waitq, N)`
: is called by a masterprocess that wants to enslave 1 or N number of processes. If the demanded slaveprocesses not are available or if a masterprocess with higher priority is waiting in the *mqueue* is the process passivated in the *mqueue*. If the masterprocess demand can be fullfilled are the wanted number of processes passivated on the masterprocess and the masterprocess is activated again.

`demo:reset(Waitq)`
: sets *observations*, *mlength_int*, slength_int to 0, *resettime*, *m_lasttime* or *s_lasttime* sets to the current eventtime. *mlength_max* sets to *mlenght*, *slength_max* sets to *slength*. The rest of the variables are unchanged.

# Queue

## Variables

The queue has eleven variables in the AList that represents the state of resource:
1. *observations*, the number of *demo:pop* calls made to the queue since the last call of `demo:reset`.
2. *queue*, a list of swaiting processes. 3. *qlength*, the length of the *queue*.
4. *qlength_max*, the largest length of the *queue* since the last call of `demo:reset`.
5. *qlength_int*, records the timeintegral for the length of the *queue* since the last call of `demo:reset`.
6. *q_lasttime*, records the last time a process entered or left the queue since the last call of `demo:reset`.

7. *concurrency*, how the resource shall start passivated processes.
8. *test*, how the resource shall passivate processes in the *queue* . If *test == fifo* are a first in first out policy used. If *test == lifo* are a last in first out policy used. If *test == priority* shall start the processes in priority order. 9. *resettime*, the last time the queue were resetted.
10. *report*, the list of items that shall be reported.
11. *file*, the file to write reports to, default is standard_io.

## Commands

`demo:new_R(queue, [Test])`
creates a queue resource.

`demo:push(Queue)`
puts the current process on the *queue*. The process are passivated until a process relaese it with a `demo:pop(Queue)` call. The *queue* variables are updated.

`demo:pop(Queue)`
tries to release a process from a queue. If there was a passivated process in the queue is it released and the calling process receives the passivated process id as a return value. If there were no passivated process on the queue is false returned to the calling process. Observe that the activated process not will be passivated on the calling process. The *queue* variables are updated.

`demo:reset(Queue)`
sets *observations* and *qlength_int* to 0, *resettime* and *q_lasttime* sets to the current eventtime. *qlength_max* sets to *qlength*, The rest of the variables are unchanged.

# 18 AppendixC

Below is a list of the different probability distributions that is supported in the Elang simulation package.

## Probability distributions

`dis:constant(A)`
: always return the same sample value A. This function is perhaps most useful in the early stages of the model building where a distribution can be replaced by its mean value.
Mean = A, Variance = 0.

`dis:erlang(A, B)`
: returns a drawing from the erlang distribution. Sums B independent random variables, each with a mean life of A / B. Demands that A > 0 and B > 0.
Mean = A, Variance = (A * A) / B.

`dis:empirical(List)`
: takes a list of {X, Prob(X)} tuples. Prob(Xi) indicates the probability that the value is equal or smaller than Xi. Prob(X1) must be and Prob(Xn) must be 1, if there are n tuples in the list. This condition must be fullfilled: Prob(Xi) < Prob(Xj) and Xi < Xj for i < j. The function returns a drawing from the distribution specified by X and Prob(X) values in the list.

`dis:normal(A, B)`
: returns a drawing from the normal distribution, using the Box-Mller method, with the mean `A` and the standard devation `B`. The formula is: X1 = sqrt(-2 * ln(U2)) * cos(2 * Pi * U1), X2 = sqrt(-2 * ln(U2)) * sin(2 * Pi * U1) where U1 and U2 is random variables, X1, X2 is the returned values, every second time is X1 respectively X2 returned. Demands that B > 0.
Mean = A and Variance = B * B.

`dis:binomial(N, P)`
: returns a drawing from the binomial distribution. The parameters indicates how many trails that shall be done and the probability to succeed in each trail. Returns the number of trails that succeeded. The formula used is: binomial(N, P) = sum(trunc(U[i] + P)) where 1 -> i -> N. Demands that N >= 1 and 0 =< P =< 1.
Mean = N * P, Variance = N * P * (1 - P).

`dis:negexp(A)`
: returns a drawing from a negative exponential distribution, with the specified arrival rate of A. The formula used is: negexp(A) = - (1 / A) * ln(U). Demands that A > 0.
Mean = 1 / A, Variance = 1 / (A * A).

`dis:gamma(N, A)`
: returns a drawing from the gamma distribution. Distribution of the sum of N independent random variables with an exponential distribution with parameter A. The formula used is: gamma(N, A) = (1 / A) * sum(ln(U[i])) where 1 -> i -> N. Demands that A > O and N >= 0.
Mean = N / A, Variance = N / (A * A).

`dis:weibull(A, B)`
: returns a drawing from the weibull distribution. The formula used is: weibull(A, B) = $(1/A) * (-ln(U))^{1/B}$. Demands that A > 0 and B > 0.
Mean = (1 / A) * gamma(1 + 1 / B), Variance = (1 / $A^2$) * (gamma(1 + 2 / B) - (gamma(1 + $1/B)^2$)).

`dis:draw(P)`
: returns a drawing from the probability distribution with chances P of returning true (1) and chances (1 - P) of returning false (0). Returns 1 for true and 0 for false. Demands that 0 =< P =< 1.
Mean = P, Variance = P * (1 - P).

`dis:uniform(A, B)`
: returns a drawing from the uniform (rectangular) distribution from a range with lower bound A and upper bound B. Returns a float. Demands that B >= A.
Mean = (A + B) / 2, Variance = (A - B) * (A - B) / 12.

`dis:randint(A, B)`
: returns a drawing from the uniform (rectangular) distribution from a range with lower bound A and upper bound B. Returns a integer. Demands that B >= A.
Mean = (A + B) / 2, Variance = ((A - B + 1) * (A - B + 1) - 1) / 12.

`dis:poisson(C)`
: returns a drawing from the poisson distribution. Distribution of number of points in random point process under certain simple assumptions. The functions uses the fact that in a poisson process with intensity C the number of points in a unit length interval has a poisson distribution with parameter C. Demands that C > 0.
Mean = C, Variance = C.

`dis:psi_square(M)`
: returns a drawing from the psi_square distribution with M number of degrees of freedom. The function uses the fact that if: X1 = gamma(N1, A), X2 = gamma(N2, A) and Y = 2 * A * (X1 + X2), then Y has the distribution: psi_square(2 * N1 + 2 * N2). Demands that M >= 0.
Mean = M, Variance = 2 * M.

`dis:t(R)`
: returns a drawing from the t distribution. The functions uses the fact that if: X1 = normal(0, 1), X2 = psi_square(R) and Y = X1 / $((X2/R)^{0.5}$ then Y has the distribution: t(R). Demands that R > 0.
Mean = 0 if R > 1, Variance = R / (R - 2) if R > 2.

`dis:beta(R1, R2)`
: returns a drawing from the beta distribution. The function uses the fact that if: X1 = gamma(R1, A), X2 = gamma(R2, A) and Y = X1 / (X1 + X2) then Y has the distribution: beta(R1, R2). Demands that R1 > 0 and R2 > 0.
Mean = R1 / (R1 + R2), Variance = (R1 * R2) / $((R1 + R2)^2$ * (R1 + R2 + 1)).

`dis:f(R1, R2)`
: returns a drawing from the F distribution. The function uses the fact that if: X1 = psi_square(R1), X2 = psi_square(R2) and Y = (X1 / R1) / (X2 / R2) then Y has the distribution: F(R1, R2). Demands that R1 > 0 and R2 > 0.
Mean = R2 / (R2 - 2) if R2 > 2, Variance = (2 * $(R2^2)$ * (R1 + R2 - 2)) / (R1 * $((R2 - 2)^2)$ * (R2 - 4)).

`dis:cauchy(A)`
: returns a drawing from the Cauchy distribution. The function uses the fact that if: U = uniform(-Pi / 2, Pi / 2) and Y = A * tan(U) then Y has the distribution: cauchy(A).
Mean: does not exist, Variance: does not exist.

# 19 AppendixD

Below is a list of the specific commands that the global eventhandler can handle:

## Start Commands

`demo_global:start(NodeList)`
: starts the global eventhandler and normal eventhandlers on the nodes given input argument list, `NodeList`. It is assumed that the nodes are existing and up. The call spawns the global eventhandler process on the current node and registers it as *demo_global*over the net

## Stop Commands

`demo_global:stop()`
: terminates all processes and eventservers and ends the simulation.

`demo_global:reset()`
: terminates all processes in the eventservers but doesn't end the simulation.

## Suspend Commands

`demo_global:suspend(Time)`
: suspends the whole simulation. The argument to `suspend` can be `next` which means that the simulation will stop at the next event, or a positive number which means that the simulation will stop at the current time + `Time`. Returns `true` or `false`.

`demo_global:suspend_at(Time)`
: suspends the whole simulation at a specific time `Time`. The argument to `demo:suspend_at` is a positive number and shall be larger than the current event time. Returns `true` or `false`.

`demo_global:suspend_reg(Time)`
: suspends the whole simulation at regulary time intervalls. The argument to `suspend_reg` can be `next` which means that the simulation will stop regulary at each event, or a positive number which means that the simulation will stop regulary with a timeintervall of the number. Returns `true` or `false`.

`demo_global:suspend_del()`
: removes all suspendpoints from the whole simulation. Returns `true`.

## Resume Commands

`demo_global:resume()`
: restarts the whole simulation when it has been suspended. Is also the command that starts the simulation after all initializations have been made. Returns `true`.

`demo_global:resume_del()`
: restarts the simulation when it has been suspended and removes all suspendpoints at the same time. Returns `true`.

## Time Commands

`demo_global:event_time()`
: returns the current eventtime in the global simulation.

## Concurrency Commands

`demo_global:set_concurrency(Form)`
: sets the way that the whole simulator shall start events with the same start time. The `Form` can be `sequential` or `parallel`. `sequential` means that processes with the same starttime will start one at a time, only one process can be active at the same time. A process with higher priority will start before a process with lower if their starttime is equal. `parallel` means that all processes with the same starttime will be activated in parallel. Defaultvalue is `parallel`. Returns `true` or `false`.

## Granularity Commands

`demo_global:set_granuality(Granularity)`
: sets the granularity in the whole simulation. The argument is a positive number which indicates how large timesteps the eventhandler will step forward. For example, if `Granularity` is 50 and the current time is 30 will the eventhandler starts all scheduled processes between 30 and 80 at the same time. If a process starts, stops and starts again in the same granuality intervall is this done sequentially. If granularity is larger then 0 does that imply that the concurrency mode is `parallel`. Defaultvalue is 0. Returns `true` or `false`.

## Trace Commands

`demo_global:trace()`
: turns on the tracer for the whole simulation. Returns `true`.

`demo_global:notrace()`
: turns off the tracer for the whole simulation. Returns `true`.

`demo_global:trace_g()`
: turns on the tracer for the global eventhandler process. Returns `true`.

`demo_global:notrace_g()`
: turns off the tracer for the global eventhandler process. Returns `true`.

## Create Process Commands

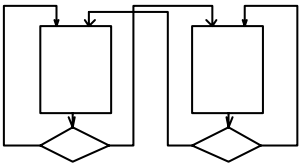
`demo_global:new_PN(Node, Mod, Func, Args)`
: creates a new process to be scheduled by the eventhandler located at the node at the current time and with priority 0. The arguments are node, module, function and argumentlist of the function. Returns the file descriptor (pid) of the created process.

`demo_global:new_PN(Node, Mod, Func, Args, Time)`
: creates a new process to be scheduled by the eventhandler located at the node at the time specified in the fourth argument. The arguments are node, module, function, argumentlist and starttingtime. Returns the file descriptor (pid) of the created process.

`demo_global:new_PN(Node, Mod, Func, Args, Time, Priority)`
: creates a new process to be scheduled by the eventhandler located at node with the priority as specified in the fifth argument. The arguments are node, module, function, argumentlist, starttime and priority. Returns the file descriptor (pid) of the created process.

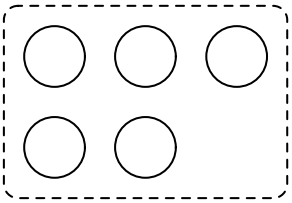
# 20 AppendixE

Below is a description of the standard figures in the report.

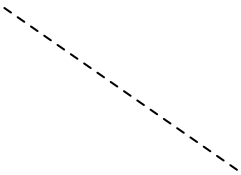


= an eventhandler process

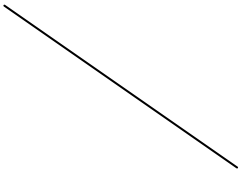
= a process



= a group of processes



= are scheduled or created by



= communicates with (through messages)

# 21 Bibliography


[Siklosi -80] Károly Siklósi, *Simula simulation*, 3rd ed., LiberTryck Stockholm, Inc., 1984.

[Sundblad, Romberger, Leringe -81] Yngve Sundblad, Staffan Romberger, Örjan Leringe, *Fortsatt programmering i Simula*, 2nd ed., LiberTryck Stockholm, Inc., 1984.

[Birtwistle -89] Graham Birtwistle, *Discrete event modeling in Simula*, 1st ed., The Macmillan Press Ltd, Inc., 1989.

[Birtwistle -81] Graham Birtwistle, *Demos reference manual*, 2rd ed.

[Birtwistle, Tofts -93] Graham Birtwistle, Chris Tofts, *Operational semantics of processoriented simulation languages, Part I: $\pi$Demos*, 1st ed.

[Birtwistle, Tofts -94] Graham Birtwistle, Chris Tofts, *Operational semantics of processoriented simulation languages, Part II: $\mu$Demos*, 1st ed.

[Blom -89] Gunnar Blom, *Sannolikhetsteori och statistikteori med tillämp- ningar (Bok C)*, 4th ed., Studentlitteratur Lund, 1989.

[Kreyszig -68] Erwin Kreyszig, *Advanced engineering mathematics*, 2nd ed., Wiley international, 1968.

[Råde, Westergren -90] Lennart Råde, Bertil Westergren, *BETA B Mathematics handbook*, 2nd ed., Studentlitteratur, 1990.

[Armstrong, Virding, Williams -93] Joe Armstrong, Robert Virding, Mike Williams, *Concurrent programming in Erlang*, 1st ed., Prentice Hall International (UK) Ltd, 1993.

[Erlang Manual -93] *Erlang manual*, Erlang System AB, 1993.

[Erlang Course -93] *Erlang course*, Erlang System AB, 1993.

[Wikström -93] Cleas Wikstöm, *Distributed Programming in Erlang*, Prentice Hall, 1993.

[Meyer -88] Bertrand Meyer, *Objectoriented Software Construction*, University Press, Cambridge, 1988.

[Garzia -89] Mario R. Garzia, *Discrete event simulation methodologies and formalisms*, Annual simulation symposium, 1989.

[Pollacia -88] Lissa F.Pollacia, *A survey of discrete event simulation and state-of-the-art discrete event languages*, Annual simulation symposium, 1989.

[Cota, Sargent -88] Bruce A. Cota, Robert G. Sargent, *An algorithm for parallell discrete event simulation using common memory*, Annual simulation symposium, 1989.

[Madisetti, Walrand, Messer -89] Vijay Madisetti, Jean Walrand, David Messerschmitt, *Efficient Distributed Simulation*, Annual simulation symposium, 1989.

[Lin -92] Yi-Bin Lin, *Parallelism analysers for Parallel Discrete Event Simulation*, ACM Transactions on Modeling and Computer Simulation, Vol 2, No. 3, July 1992, Pages 239-264.