HIT and HI-SLANG

An Introduction

Document Version 1.2.00

the
Hierarchical Evaluation Tool

HIT

Version 3.1.000

HIT AND HI-SLANG: AN INTRODUCTION

Norbert Weißenberg (Editor)
Achim Wilde (Editor)

Bruno Müller-Clostermann
Salwa Shaban
Wolfgang Dittrich

Part of the material has been taken from publications of the HIT staff. The document has been typed by Brigitte Adunola, Nathalie Münter and mainly by Iris Koch and the editor. Bruno Müller-Clostermann was the main author and editor of former versions of this paper, called "HIT. An Introduction".

Many updates have been considered for the current document version 1.1.00. For main changes to HIT version 3.1.000 please see Appendix D.

Abstract:

The system evaluation tool, HIT, is a software tool for model-based performance evaluations of computing systems during all phases of their life cycle. The hierarchical model description language, HI-SLANG, allows the construction of deeply structured models in a highly modular fashion. Quantitative model evaluations can be performed using simulative or a vast range of analytical methods.

HIT has been developed at the chair of Prof. Dr.-Ing. H. Beilner, Fachbereich Informatik, Universität Dortmund in cooperation with Nixdorf Computer AG and with partial support of the BMFT (German Federal Ministery of Research and Technology). The tool, HIT, is in industrial use at computer manufacturing companies since mid 1985. It is operational in Siemens BS2000, IBM VM/CMS, IBM MVS and several Unix environments (Sun/3, Sun/4 and Apollo workstations, PC '386, WX200, ...).

This document is the primary source for the HIT beginner. Corrections, comments, criticisms and suggestions for improvements relating to this document are welcome. For a complete language description the reader is referred to the HI-SLANG Reference Manual. Additionally the document on the HIT-OMA Object Management System and the graphical interface HITGRAPHIC will be helpful.

Address:
Universität Dortmund
Informatik IV
Prof. Dr.-Ing. H. Beilner
D-44221 Dortmund

Telefon: (Germany)-(231) 755-2411
Telefax: (Germany)-(231) 755-4730
E-Mail: hit@ls4.informatik.uni-dortmund.de

Part I

A FIRST LOOK ON HIT

Chapter 0

0. Introducing the Big Idea

0.1. Motivation and Main Ideas

The development of computing systems is associated with many problems due to the fact that both the performance requirements and the technological progress are increasing rapidly. It is impossible to master the increasing complexity of system architectures, the integration speed in the hardware, as well as the diversification in the software area without employing tools to analyse both planned and existing computing systems quantitatively and qualitatively.

In particular, modelling and evaluation should be done by the designer, consultant, salesman or engineer himself without being an expert in simulation, statistics, queueing theory, numerical analysis and related techniques for quantitative system evaluation.

Consequently, it was an important requirement for the HIT system, that a modelling problem can be solved by these persons themselves and must not be passed to a modelling specialist, thus eliminating the enormous communication overhead.

0.1.1. The Modelling Tool HIT and the Language HI-SLANG

HIT is a performance modelling tool which allows the structured specification and the quantitative evaluation of computing system models.

Nowadays system design and development are usually based on a layered model with functional abstraction. HIT employs therefore hierarchical modelling techniques allowing the separate specification and analysis of models, model components and their evaluations.

The software tool, HIT supports

- the specification of (models of) dynamic, discrete-event, stochastic systems using a particular model description language, HI-SLANG, for the description of model structures and the evaluation to be performed;

- the (performance) analysis of correspondingly specified models using a variety of techniques of the simulative, analytic-algebraical and analytic-numerical type.

Although originally developed for evaluating computing system performance, HIT also lends itself to the analysis of "similar" systems such as communication and office systems, transport and logistic systems and others of the specified (dynamic, discrete-event, stochastic) type.

The HIT model world is tailored upon the prevailing view of computing system structures which partitions a system

- **vertically**, into a sequence of layers and levels, communicating via function calls, and jointly representing a hierarchy of virtual machines;

- **horizontally**, into independent, mutually well-protected, information-hiding modules each one realising some subset of functions to be provided at a particular level.

The corresponding HI-SLANG specification maintains as far as possible the conventional, high-level-language (HLL) approach, assumed to be well-known to and convenient for the envisaged user community of the tool:

- Conventional functions/procedures (termed SERVICEs) serve as "patterns" for process/subprocesses "to-be-run". They are described in terms of traditional HLL control and data structures.

- SERVICEs can be packaged into modules (termed COMPONENTs). These services can be called upon by other (higher layer) SERVICEs, situated within other (higher layer) COMPONENTs. Imported (USEd) and exported (PROVIDEd) names of SERVICEs are explicitly linked in order to increase the independence of partial designs.

- Options for initiating processes in time-controlled or event-controlled mode complete the desired specification capabilities for describing systems of parallel processes.

From a software engineering point of view, HI-SLANG specification supports various design styles such as top-down, bottom-up and (realistically) ping-pong/yo-yo. From a modelling point of view, disjoint specifications of models (to be analysed) and experiments (to be performed with these models) greatly increase the flexibility of use. Additionally a modelling base is offered by HIT to support storage and retrieval of (partial) models and analysis results. This option eases to combine previous modelled parts into larger models as well as the development of team developed models.

HIT evaluation techniques include the following approaches:

- stochastic discrete-event simulation with appropriate, statistical result evaluation
- exact result evaluation for "separable networks" (with product-form solution) and approximate evaluation techniques for both "large" separable and certain "non-product-form" networks
- numerical evaluation of Markov chain representations of general models
- sub-model analysis and aggregation with the objective of generating "equivalent" higher level representations, to be used in structured and/or heterogeneous (total model) evaluation.

It must be emphasized that a HIT model specification is not directly influenced by the particular evaluation technique to be employed. There does, of course, exist an indirect influence whereby certain models will turn out not to be tractable by one or the other analysis technique, with simulation clearly offering the largest spectrum.

0.1.2. Areas of Application

HIT is a tool for the performance evaluation of computing systems during most phases of their life cycle.

- During the design of computers, computer components and operating systems, model evaluations can answer many of the arising questions. Design studies are usually undertaken by the producer of a system rather than by its users.

- During selection and configuration of systems, model evaluations can help to choose among the various available alternatives, for example, the most convenient alternative to a given application.

- During the operation phase of computing systems, model evaluations are helpful for tuning and upgrading purposes.

Apart from computing system modelling, HIT can be used for the modelling of communication systems, office systems, flexible manufacturing systems and others.

0.2. Sketching the HIT Model World

We sketch very briefly the most important features of HIT.

0.2.1. Machines and Loads

In each model layer, a usable machine and a using load face each other. A machine is composed of a set of components. Each of them provides certain usable services. The total set of services (of all machine components) defines the level upon wich a layer may be built. A load consists of a set of process patterns. Each one specifies a particular prescription for the dynamic use of any (usable) services. Processes which are obeying the rules of specific service can be instantiated in time-controlled or event-controlled mode within the load. Returning to the machine, components are normally declared as instances of certain component types. Predefined standard types are available, amongst them the component type, *server*, which provides a basic service, *request (amount: real)*. The parameter *amount* indicates the temporal duration of the service, *request.*

0.2.2. Layers, Models and Components

A model layer is formed by referring a load to a machine. This step includes an explicit linking of the various used services (of the load) to specific provided services (of the machine). The resulting, linked machine/load complex is termed a model. If the machine consists exclusively of standard *server* components, the traditional non-hierarchical model will be achieved, consisting only of a single layer. A model can be transformed into a component by declaring certain of its internally specified services as externally accessible, usable services. We thereby arrive at (part of) the next higher level, i.e., at (part of) the basis of a next higher layer. Consequent application of this concept results in arbitrarily multi-level/multi-layer models, which can be developed top-down or buttom-up or, more realistically, in a ping-pong strategie.

0.3. Quantitative Evaluation Techniques in HIT

HIT users do not have to bother about the analysis and the evaluation of their models. This is automatically done by HIT! The user must merely specify the experiments to be performed and the analysis technique to be used. Experiment specifications describe the results demanded from model or component analysis. As for model analysis, a corresponding experiment specification encloses

- instantiation of a model (of an earlier defined model type) inclusive of the setting of any parameters;
- specification of the model analysis technique;
- indication of all evaluation objects, i.e., model components, where measurements are to be taken;
- indication of all measurement streams, i.e., performance variables, of the above evaluation objects, for which evaluation is demanded;
- specification of measurement specifics; and
- indication of data gathering starting rules (if simulation is the selected analysis technique) and evaluation stopping rules.

Depending on whether the analytical or the simulative method is specified, the HIT system will transform the HI-SLANG representation of the model to analytical algorithms or to a simulation program. The figure depicted below shows the different solvers of the HIT system. At the tree´s leaves the method names used by HIT are given.

Please note that due to historical reasons there are different names for the same solution method. Some of these names refer to different algorithms used in that solver, since most solvers implement a collection of algorithms. The most appropriate one is selected at run time, and the reasons for this selection are given on the HIT listing.

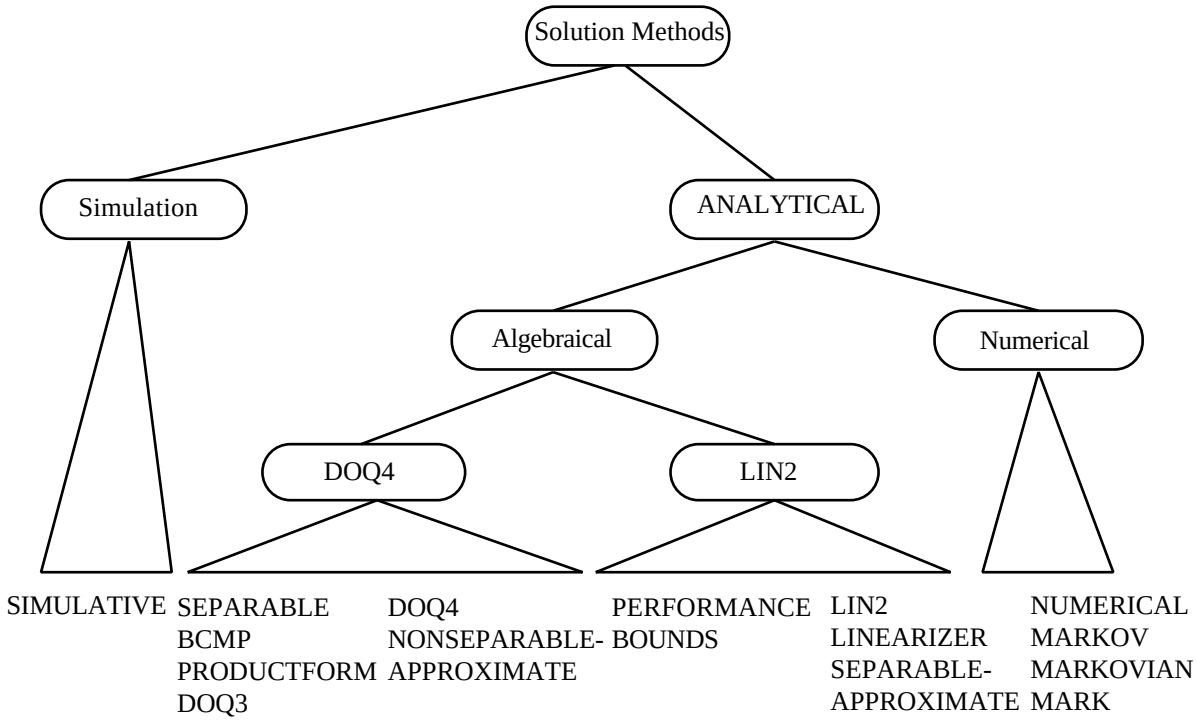

Figure 0.1: HIT Solvers

In the following we will use the method names DOQ4, LIN2 and NUMERICAL.

The analytic-algebraical algorithms can treat the so-called separable networks or product form networks, which form an important model class for quantitative performance evaluation. Separable models can either be solved exactly (via the DOQ4-algorithm) or approximately (via the Linearizer LIN2-algorithm). The second choice should be made in case of very large models. It may also deliver lower and upper bounds for performance values, called performance bounds.

A class of non-separable models including FCFS-scheduling with different service requests as well as priority scheduling disciplines can also be treated by the DOQ4-algorithm (which is obviously an extension of the older DOQ3!).

Models based on Markov chains can be evaluated by numerical techniques. These models include more general distribution functions, priority scheduling disciplines and features for the modelling of synchronization mechanisms.

The simulative method is based on the discrete event simulation concept of the host language, Standard SIMULA.

As a result of model evaluation we obtain performance values for selected model components or pre-analyzed component types. Pre-analyzed components are components that are analysed without being influenced by the rest of the model. This process leads to a flow-equivalent substitute of a component in the form of a state dependent *server* component. It can later be included in a model, replacing the original component, yielding (under certain conditions) approximately the same results as the original. Currently only the DOQ4-algorithm can be used for aggregation.

HIT offers the performance indices population, turnaround time, utilization, occupation and throughput. Additionally self-defined performance indices as well as scheduling and preemption rates can be evaluated simulative.

0.4. How this Document is Organized

We will gradually introduce various HI-SLANG subsets and the pertinent modelling possibilities followed by examples and general explications. We dedicate special sections to the following subsets of HI-SLANG:

- Subset for separable models and some extensions. There are two solution methods for this class of models:

  - METHOD ANALYTICAL "DOQ4" for the exact evaluation (and aggregation) of separable models as well as for the approximate evaluation (and aggregation) of extended separable models
  - METHOD ANALYTICAL "LIN2" for the approximate evaluation of large separable models (including performance bounds)

- Subset for METHOD ANALYTICAL "NUMERICAL" for the evaluation of Markov chain based models, which includes the subset for separable models (with some few exceptions)

- and the METHOD SIMULATIVE which is the most comprehensive model class. It includes both of the above subsets (with some few exceptions). But simulation in general needs much more cpu time than analytical solvers and the results are only estimated.

It is not our intention to give an exhaustive description of HIT, although we try to be complete in the parts which are of central importance.

Detailed information can be found in the HI-SLANG Reference Manual (/Weis92/) and, concerning the use of a modelling base, in the HIT-OMA User's Guide (/Weis91/). Moreover the graphical interface of HIT is described in the HITGRAPHIC User's Guide (/Heck91/).

In the next part of this manual (Part II) we deal with the subset of separable models. We recommend the HIT beginner to restrict himself to this subset. Numerical and simulative evaluation of HIT models should be postponed until the first modelling experiences with the HIT system have been made. They are described in Parts III and IV respectively.

The last part consists of several appendices. Appendix A. is devoted to the handling of the HIT system on different computer systems and should be used as reference material if necessary. Appendix B. summarizes the HI-SLANG syntax for experiments and Appendix C. lists HI-SLANG features not treated in this introduction. Some references and an index conclude this document.

Part II

SUBSET FOR SEPARABLE MODELS AND THEIR EXTENSIONS

Chapters

1 - 5

1. First Steps with HIT

1.1. Overview

The objective of this chapter is:

- to become acquainted with the basic elements of modelling with HIT;
- to become conversant with handling the HI-SLANG compiler;
- to answer, by performing experiments, some "what-if" questions, which will typically arise while evaluating several design alternatives and
- to comprehend the HI-SLANG features for the evaluation of "flat" (i.e., non-hierarchical) models.

To use HIT as soon as possible, we postpone an examination of isolated language constructs and start with a complete example.

1.2. A Basic Model

We consider a computing system, which satisfies the service requirements of certain tasks. The computing system is called the machine. The tasks to be processed are called the load. In HI-SLANG we address the tasks as processes. They are created according to a process pattern, called a service.

1.2.1. The Load

The load our computing system has to face behaves as the process pattern depicted below. After its creation it fulfills an initial computing requirement. Then it repeats a loop 9 times on the average. Within this loop read or write accesses to files are performed, followed by more calculations. Finally the process finishes.

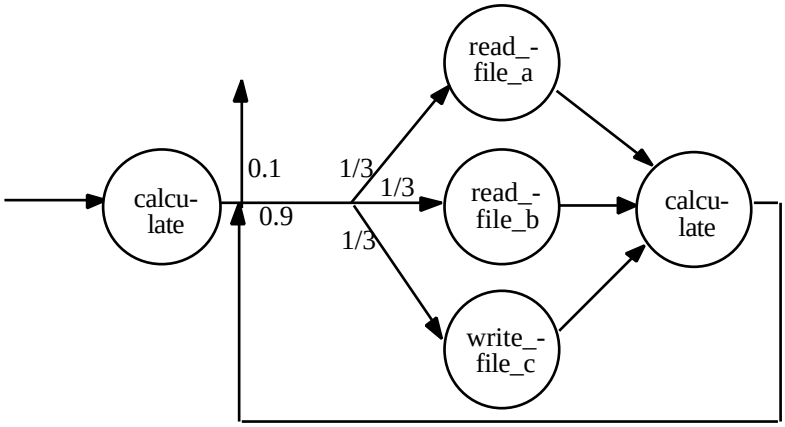

Figure 1.1: Simple Process Pattern

After describing the load in an informal manner we have to translate the informal description into a formal HI-SLANG service declaration. We use four service requests, one for calculations and three for file accesses. But before we can use them we have to name them in the USE declaration part of the corresponding service named *batch_task*:

```
TYPE batch_task SERVICE;

    USE SERVICE
        calculate       (amount: REAL);
        read_file_a(amount: REAL);
        read_file_b     (amount: REAL);
        write_file_c    (amount: REAL);
    END USE;
```

After we have finished this step, we can determine by control statements how the services are used.

```
BEGIN
    calculate (negexp(5.0));
    AVERAGE 9 TIMES
    LOOP
        BRANCH
            PROB 1/3    :   read_file_a(negexp(1/0.1));
            PROB 1/3    :   read_file_b     (negexp(1/0.2));
            ELSE        :   write_file_c    (negexp(1/0.4));
        END BRANCH;
        calculate (negexp(1/0.2));
    END LOOP;
END TYPE batch_task;
```

We have refined the file accesses by a BRANCH construct now. The files *a*, *b* and *c* are accessed with identical probabilities 1/3. The rest of the process pattern was directly translated into HI-SLANG.

Please note that we have used the random distribution function *negexp* to specify the amount of required service. Its parameter gives the rate R of a negative exponential distribution with mean 1/R.

1.2.2. The Machine

After specifying the load, the machine has to be described by means of units. We assume a CPU and three disk units. From now on, we refer to them as components. These components can accept and process tasks, and finally offer them to the environment. In the given context we don't intend to refine these components any further. Thus they are called elementary components or standard components of type server. You can think of them as the machine. A server provides a basic service *request* (amount: real), the parameter indicating the temporal duration of the service.

The progress of the service is governd by rules which can be specified by parameters of the components shown below. See also Section 2.2. for more information.

```
COMPONENT cpu: server
        (LET accept   :=  always,
         LET schedule:=  immediate,
         LET dispatch:=  shared,
         LET offer:=  all);

COMPONENT disk_a, disk_b, disk_c : server (LET schedule := fcfs);
```

The parameterization of the CPU has the following meaning (in the order of appearance):

- Service requests are always accepted without conditions.
- Scheduling is immediate, i.e., there is no waiting time.
- The processing capacity is shared between all processes.
- All completed processes are offered.

The parameterizations of the disks are different. Note that *fcfs* (first-come-first-scheduled) has been chosen as scheduling discipline. The other parameters have their default values, which are *always, equal(1.0)* and *all,* respectively.

1.2.3. Referring the Load to the Machine

We have described the machine by a set of components and the load by a set of services. Now we can specify from which machine components the load requirements ought to be fulfilled. Obviously the computing requirements should be referred to the CPU and the IO requirements should be referred to the disk units.

We say, that a load is referred to a machine for execution by explicit binding the various used services of the load to specific provided services of the machine. In HI-SLANG this has to be done by a REFER part, depicted below. Remember that the basic service of *cpu* and *disks* is *request.*

```
REFER batch_task TO cpu, disk_a, disk_b, disk_c
    EQUATING
        batch_task.calculate      WITH   cpu.request;
        batch_task.read_file_a    WITH   disk_a.request;
        batch_task.read_file_b    WITH   disk_b.request;
        batch_task.write_file_c   WITH   disk_c.request;
END REFER;
```

The figure below shows the HITGRAPHIC equivalent of the former REFER part.

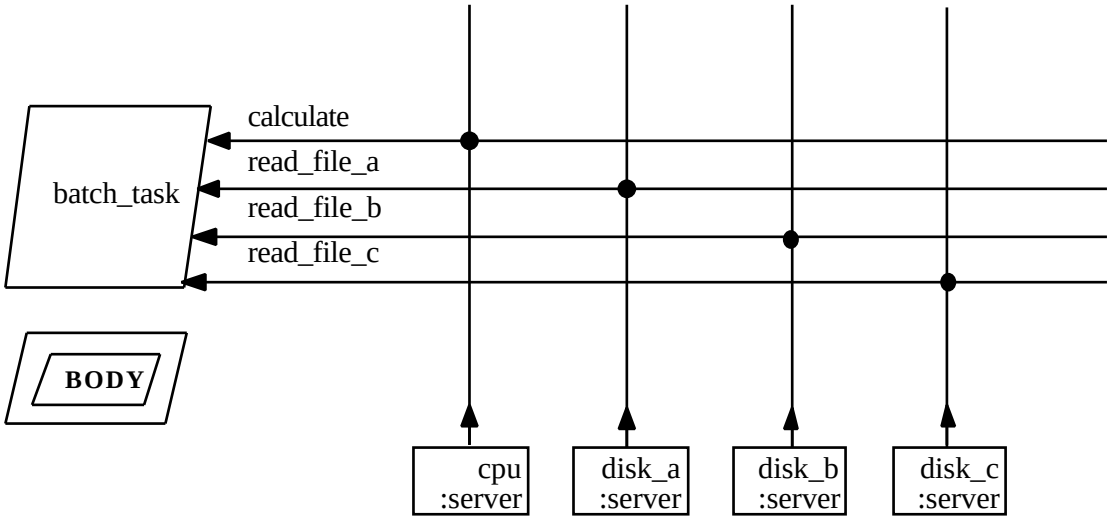

Figure 1.2: HITGRAPHIC Representation of a Component Type

1.2.4. Building the Complete Model

A complete model type is formed by concatenating the load, the machine and the REFER part, followed by a CREATE statement within a block which describes the arrival pattern of batch tasks, e.g., of processes of type *batch_task*. Note the parameter *tasks_per_second* of the model *example1.*

The parameterization of a model is not necessary. But it is advantageous in order to perform many similar experiments with different parameters. For example it is now possible to investigate which load our model cannot handle any more.

```
TYPE example1 MODEL (tasks_per_second: REAL);

    {description of load}
    {description of machine}
    {referring load to machine}

BEGIN
    CREATE 1 PROCESS batch_task EVERY negexp (tasks_per_second);
END TYPE example1;
```

The CREATE statement leads to the instantiation of individual processes of type *batch_task* at exponentially distributed inter-instantiation times with mean value *1/tasks_per_second*.

Note that the model *example1* consists exclusively of standard *server* components. The following figure illustrates the "flatness" of the model. Aside from flat models HIT offers also hierarchical models. Their construction and advantages will be shown in the succeeding chapters.

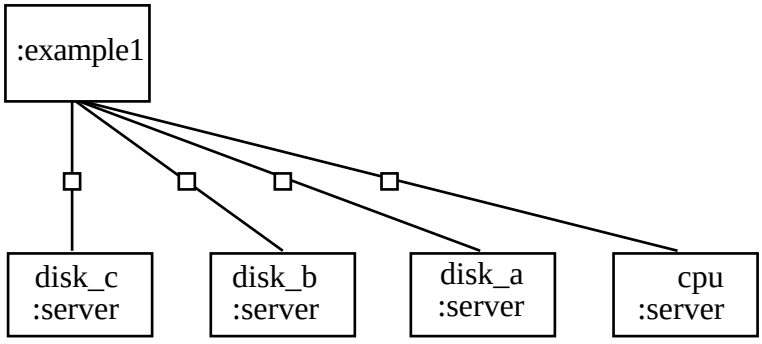

Figure 1.3: Model Structure

Up to now we have specified a model type only. In order to measure its performance indices, i.e., to perform an experiment we have to add an experiment block. This block specifies the lacking actual parameter *tasks_per_second* and determines the evaluation method to be used as well as the performance indices of interest.

1.2.5. Describing an Experiment

We have learned in the previous section that the parameters of a model type will be set in an experiment block. It consists of a declaration part and a statement part. The former may contain declarations of constants, variables and procedures. The latter describes the performance indices to be determined by means of an EVALUATE statement. In our example we are going to declare the variable *arrival_rate* of type REAL in the declaration part.

The EVALUATE statement generates an analysable model object by setting the actual model parameters and describes all performance indices of interest. An experiment block for our sample model is depicted below.

```
EXPERIMENT experiment1 METHOD ANALYTICAL "DOQ4";

    VARIABLE arrival_rate: REAL;

BEGIN
    FOR arrival_rate := 0.05, 0.1, 0.15, 0.2, 0.25, 0.3
    LOOP

        EVALUATE
            MODEL model1: example1 (arrival_rate);

            EVALUATIONOBJECT
                cpu VIA model1.cpu,
                disk VIA model1.disk_a;
        BEGIN
            MEASURE POPULATION, UTILIZATION, THROUGHPUT,
                TURNAROUNDTIME AT cpu;

            MEASURE POPULATION, UTILIZATION, THROUGHPUT,
                TURNAROUNDTIME AT disk;
        END EVALUATE;

    END LOOP;
END EXPERIMENT experiment1;
```

Concerning the experiment block we emphasize the following points:

- The FOR loop embraces an evaluation, starting with EVALUATE MODEL, which will be repeated for different model objects. We specified an evaluation series.
- The different models (all named *model1*) have the same type, namely *example1.* They only differ in the actual parameter values.
- The EVALUATIONOBJECT construct defines evaluation objects (*cpu* and *disk*) via model components. The MEASURE statements refer to these evaluation objects.
- As performance estimator the mean value is chosen (by ESTIMATOR MEAN). Indeed MEAN is the only choice in case of separable networks.
- The output is directed to a table file, as is the default. Otherwise an OUTPUT part has to be added (see Reference Manual).
- If you like you can simply switch to another evaluation method by replacing ANALYTICAL "DOQ4" by, e.g., ANALYTICAL "LIN2".
- Note that all HI-SLANG statements can be used for writing experiment bodies, independent of the evaluation method used.

1.2.6. The Whole Model

We now present the model and experiment in total. The following source text can be fed to the HIT system:

```
TYPE example1 MODEL (tasks_per_second:REAL);

    TYPE batch_task SERVICE;
        USE SERVICE
            calculate       (amount: REAL);
            read_file_a(amount: REAL);
            read_file_b     (amount: REAL);
            write_file_c    (amount: REAL);
        END USE;

    BEGIN
        calculate (negexp(5.0));

        AVERAGE 9 TIMES
        LOOP
            BRANCH
                PROB 1/3    :   read_file_a(negexp(1/0.1));
                PROB 1/3    :   read_file_b(negexp(1/0.2));
                ELSE        :   write_file_c    (negexp(1/0.4));
            END BRANCH;

            calculate (negexp(1/0.2));
        END LOOP;
    END TYPE batch_task;


    COMPONENT
        cpu: server
            (LET accept       :=  always,
             LET schedule     :=  immediate,
             LET dispatch:=  shared,
             LET offer        :=  all);

    COMPONENT
        disk_a,
        disk_b,
        disk_c : server (LET schedule := fcfs);


    REFER batch_task TO cpu, disk_a, disk_b, disk_c
    EQUATING
        batch_task.calculate  WITH  cpu.request;
        batch_task.read_file_a    WITH  disk_a.request;
        batch_task.read_file_b    WITH  disk_b.request;
        batch_task.write_file_c   WITH  disk_c.request;
    END REFER;

BEGIN
    CREATE 1 PROCESS batch_task EVERY negexp (tasks_per_second);
END TYPE example1;
```

```
EXPERIMENT experiment1 METHOD ANALYTICAL "DOQ4";

    VARIABLE arrival_rate: REAL;

BEGIN
    FOR arrival_rate := 0.05, 0.1, 0.15, 0.2, 0.25, 0.3
    LOOP

        EVALUATE
            MODEL model1: example1 (arrival_rate);

            EVALUATIONOBJECT
                cpu     VIA model1.cpu,
                disk    VIA model1.disk_a;

        BEGIN

            MEASURE POPULATION, UTILIZATION, THROUGHPUT,
                TURNAROUNDTIME AT cpu;

            MEASURE POPULATION, UTILIZATION, THROUGHPUT,
                TURNAROUNDTIME AT disk;

        END EVALUATE;

    END LOOP;
END EXPERIMENT experiment1;
```

1.3. Handling the HIT System

A model, which is written in HI-SLANG, will first be translated to Standard SIMULA by the HI-SLANG compiler and will finally be translated to executable code by the SIMULA compiler. In order to control the operation of the HIT system a control part can be written, which is either a separate file or the beginning of the HI-SLANG source. How to specify the control part is described in a later section.

Now we have to run the model described in the preceeding section. The evaluation of our model *example1* can be done without any control part.

1.3.1. How to Call HIT

After calling HIT in the operating system environment HIT responds with the line

```
Please enter name of Compiler SOURCE or CONTROL file:
```

demanding for the name of the control or the source file. Since we do not need a control file we supply the name of the source file.

HIT is normally processed in dialog mode, so you can expect a response within a short time (provided you use METHOD ANALYTICAL). At the end of a HIT run you return into operating system mode. Of course, HIT can also be processed in batch mode. In any case, when you use HIT the first time, we recommend to contact the administrator of the HIT installation on your host system. Some advices are given in Appendix A.

1.3.2. HIT Output

Now you have activated HIT. But where are the results? By default all HIT output is written to files named by the HIT file name generator by suffixing the control file name with the kind of output. After a successful compilation from HI-SLANG via SIMULA to executable code, followed by the execution of the model, the required values for the performance indices will be supplied in a separate file. The results are normally represented as tables, but they can as well be given in a simple graphical form. Please see the Reference Manual for more information.

If the input file contains errors, (syntax errors, logical errors) we recommend to inspect the listing file for finding and correcting these errors. In addition to your formatted HI-SLANG source text you may find a cross reference listing (if demanded in the control file) and the (let us hope empty) list of error messages. Moreover information about the model solution process is appended to the listing.

1.4. Performing First Experiments

We first explain the performance indices, which are provided by HIT as results of an experiment. Then we introduce some questions which can be answered with the help of a model similar to *example1.*

1.4.1. Performance Indices

HIT provides standard performance indices for all components of a model. In case of *example1* these performance indices can be determined for all used services of the service *batch_task* and will be represented as a table in a separate file. Only those performance indices will be given which are specified in the MEASURE statement of the experiment block.

One can choose between the following performance indices, which are obtained by evaluation of the so-called standard streams. Note that the results given by the analytical method are mean values!

- **THROUGHPUT**

This is the number of processes leaving the considered component per time unit. If a component provides several services, throughputs for the specific services can be distinguished.

- **TURNAROUNDTIME**

This is the total time a process spends in the component until it is completed. Sometimes response time or system time is used as a synonym.

- **POPULATION**

This is the number of processes present in a component. Another word for population is filling. Sometimes the misleading term queue length is used in the literature. Population includes the queued processes as well as the processes in service!

- **UTILIZATION**

This performance index is only permitted for components of type *server* and specifies the use of that *server* on the average (utilization). Due to assigned service speed or the simultaneous requests of different processes values greater than one may occur.

1.4.2. Some What-If Questions

It is very important for you to make your own modelling experiences. For that reason we recommend to perform some experiments with the model of *example1.* This will give you a first impression how to handle some typical what-if questions appearing, e.g., in system design and configuration.

Note that the model type or the experiment block have to be changed according to the questions imposed. Now try to solve the following problems on basis of *example1*:

- What is the maximum load intensity (given by the arrival rate) which the system can still handle?

- Find the bottleneck of the system and examine whether increasing the speed of the CPU or of the disk units will improve the system throughput or not. (How to specify the speed of a standard component is described in Section 2.5.1.)

- How does a 30% faster CPU affect the system performance?

- How are the system performance measures affected if the access to the disk devices is not uniformly distributed? Consider, for example, an unbalanced case where the probabilities of disk accesses are given by 0.2, 0.3 and 0.5 !

1.5. Solvers for Separable Models and their Extensions

HIT offers mainly two possibilities for solving separable models, an exact and an approximate technique.

1.5.1. DOQ4

By means of the analytic-algebraical solver DOQ4 separable models as well as certain non-separable models can be analysed or aggregated. A check of restrictions will partly be performed at run time. A choice between approximate or exact evaluation will be automatically made during the execution of the experiment. Whenever possible the exact solver will be chosen.

1.5.2. LIN2

The algorithm which calculates approximate solutions of separable models called LIN2 is able to calculate the mean as well as the so-called performance bounds of performance indices.

The algorithm for performance bounds calculates upper and lower bounds for performance indices of separable models. Performance bounds are an appropriate alternative or supplement for exact and approximate evaluation methods, respectively. The PBH method by Eager/Sevcik as well as the integral method by McKenna and Mitra are implemented. Both can analyse separable models with state dependent and state independent services.

The PBH method's state dependency is limited to monotonous increasing speed functions and is dependent on the number of tasks in the concerning *server.* On the other side it is required for McKenna/Mitra's method that every closed chain has to contain at least one infinite server, which must not be overloaded. The state dependency is limited to the speed which is itself dependent on the number of tasks in a *server.* Therefore both methods cannot manage state dependent *server*s which speed is dependent on the population vector, i.e., they cannot handle aggregated components. Open chains visiting state dependent *server*s are not allowed. Influence can be taken on the quality of the bounds to be calculated by means of the stop condition ACCURACY in the control block. For accuracy only values ≤ 4 are allowed. Dependent on the size of your models LIN2 will decrease the accuracy automatically in order to limit the effort. In such a case a warning will be given.

Real values for accuracy will be rounded to integer values. If accuracy is smaller than 0.5 or the ACCURACY stop condition is omitted, no bounds will be calculated. The stop condition CPUTIME will be ignored.

Performance bounds will be selected if you choose the LIN2 solver. Use ESTIMATOR BOUNDS in the EVALUATE statement and ACCURACY as stop condition.

2. HI-SLANG Subset for Flat Models

2.1. Overview

The objective of this chapter is:

- to comprehend how a component deals with service requests;
- to learn how *server* components are used for modelling;
- to become conversant with services and model types;
- to get familiar with *spend, hold, negexp* and *draw* and
- to learn HI-SLANG control statements

All this is necessary to specify flat models with HI-SLANG.

2.2. Component Control Procedures

A component deals service requests in the following manner:

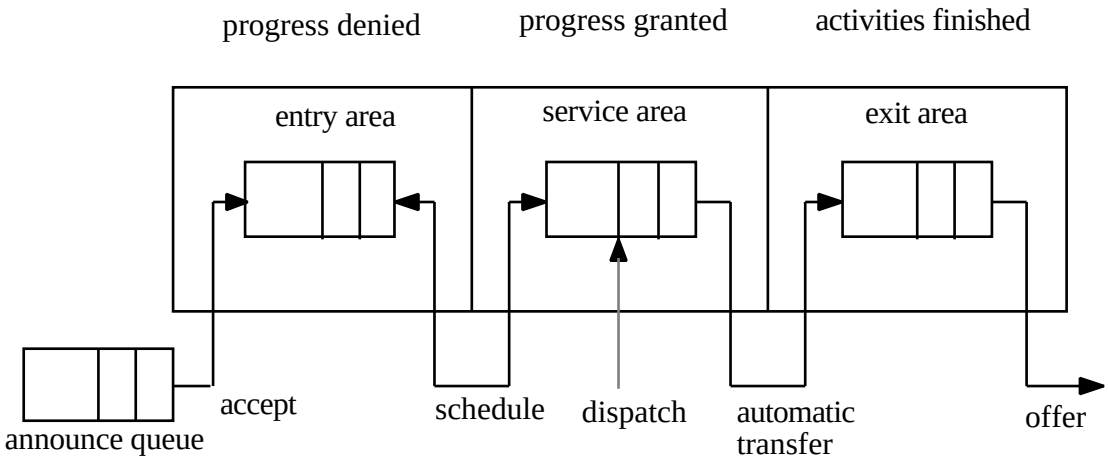

Figure 2.1: The Component Control Mechanism

- All service requests are stored in the so-called *announce queue*.
- First such requests will be accepted in the *entry area*. For a standard component there are no limitations or restrictions concerning the acceptance capacity (LET accept := always).
- Then they are transported out of the *entry area* into the *service area* according to the schedule strategy chosen.
- In the *service area* they will be processed according to the *dispatch* procedure.
- And finally, in the *exit area*, they will be offered. Note that a standard component offers all tasks to its environment (LET offer := all).

The organization of the processes' progress is governed by four behavior rules: *accept, schedule, dispatch* and *offer*. Upon declaration every component is being parameterized in accordance with the explications given below.

We now describe the function of these four rules and the possible parameter values admitted for separable models. For numerical and simulative evaluation more behavior rules are allowed.

2.2.1. Accept

The *accept* procedure describes under which conditions new service requests will be accepted. The default value *always* is the only possibility for the HI-SLANG subset treated by METHOD ANALYTICAL.

2.2.2. Schedule

A *schedule* procedure describes under which conditions a new process will be transferred into (and out of) the *service area*. The available scheduling strategies for separable models are:

- **fcfs:** first-come-first-scheduled (LET schedule:=fcfs)

The processes will be transferred from the *entry area* into the *service area* in the order of their arrival. Imagine that *fcfs* is realized by a "queue".

- **lcfspr:** last-come-first-scheduled-preemptive-resume (LET schedule:=lcfspr)

Priority is given to processes just arriving. Therefore processes, which were longer present are preempted, e.g., they are driven out of the service area. The services of the preempted processes will be continued ("resumed") at the "break point". Imagine that *lcfspr* is realized by a "stack".

- **immediate:** immediate scheduling

Arriving processes will be directly (immediately) transferred into the *service area.* Therefore the processes do not suffer any waiting time in the *entry area.* The procedure *immediate* is the default *schedule* procedure.

2.2.3. Dispatch

The *dispatch* procedure determines whether the service capacity is shared among all processes present in the *service area* or whether each process is served with equal speed.

There are various options of dispatching strategies which are fixed by the following parameter values:

- ***shared:*** sharing the service capacity

```
        LET dispatch := shared
or      LET dispatch := shared(S)
```

Each of the *N* processes in the *service area* is served with speed *1/N* or with speed *S/N*, respectively. Note that *S* may only have values > 0.0 .

- ***equal:*** equal service for all processes

```
        LET dispatch := equal
or      LET dispatch := equal(S)
```

All processes in the *service area* are served with the same speed 1.0 or with the same speed *S*, respectively. It seems as if each process possesses its own processor permanently. If a process does not have to spend some waiting time in the *entry area* (immediate scheduling!), its service request time will be identical to its turnaround time. If *m* denotes the service request, the turnaround time will be given by *m/S*.

- ***sdequal*** and ***sdshared:*** state dependent service speed

```
        LET dispatch := sdequal (a, c)
or      LET dispatch := sdshared(a, c)
```

"State dependent" means the service speed is a function of the component's population.

The first parameter a of *sdequal* and *sdshared* is a 2-dimensional array, which has to be specified as follows:

$[[N_1,N_2,\ldots,N_k], [S_1,S_2,\ldots,S_k]]$,

where $N_1,N_2,\ldots,N_k$ denote the component populations and $S_1,S_2,\ldots,S_k$ denote the associated speeds. Note that the following conditions must hold:

$N_1 = 1 \quad N_1 < N_2 < N_3 < \ldots < N_k$

and $S_1, S_2, S_3 \ldots S_k > 0.0$

If N denotes the actual population of a component, the associated speed is given according to the following interpretation.

$N_1 \leq N < N_2 : S_1$
$N_2 \leq N < N_3 : S_2$
.
.
.
$N_{k-1} \leq N < N_k : S_{k-1}$
$N_k \leq N : Sk$

This means that we don't have to specify speeds for all possible fillings. The gaps will be filled according to the interpretation given above. Examples which demonstrate the application of state dependent speeds are given in the following chapter.

The second parameter, *c,* specifies a norm speed. The default value of *c* is 1.0. The service speed can be increased and decreased by increasing or decreasing this parameter, respectively. Instead of *LET dispatch := sdshared (a,c)* we can also write *LET dispatch := sdshared (a, LET speed := c).*

The parameter *equal(1.0)* is the default value for the *dispatch* procedure.

2.2.4. Offer

The *offer* procedure describes under which conditions the finished processes will be offered by the current component. The default value is *all,* i.e., all "completed tasks" are offered without any conditions. *All* is the only *offer* parameter, treated by METHOD ANALYTICAL.

2.3. Examples for the Use of Servers

Predefined types are available, amongst them the standard component type *server* is the most important. A *server* provides the basic service *request (amount: REAL)*, the parameter *amount* indicates the temporal duration of the requested service.

2.3.1. Modelling CPUs (Sharing Service Capacity)

We consider a model of a time-sliced CPU. In this model we introduce the abstraction that all processes are allowed to make progress, without having to wait for the allocation of the CPU. Nevertheless they are sharing the service capacity equally.

The CPU can therefore be modelled as a standard component with the following parameters, which denote processor sharing:

```
COMPONENT cpu: server   (LET accept     :=  always,
                         LET schedule   :=  immediate,
                         LET dispatch   :=  shared,
                         LET offer   :=  all);
```

or shortly, using defaults:

```
COMPONENT cpu: server (LET dispatch     :=    shared);
```

2.3.2. Modelling Dialog Users (Infinite Servers)

Imagine a large computer system with many terminals. A dialog task is running for each terminal and spends a certain time at the terminal. This holding time is needed by the user for thinking and typing of commands, followed by striking the return key, such that processing is initiated or continued. Indeed, we take the view that a dialog process has a service request at a component *terminal_pool* which includes the terminals' hardware and software as well as the users themselves (a human sub-component, if you like!)

The dialog users of a computing system can therefore be modelled by a standard component without parameters. These are infinite servers:

```
COMPONENT terminal_pool: server;
```

2.3.3. Modelling Input/Output Devices (Queueing)

If tasks are not allowed to access a resource simultaneously, then the scheduling discipline must specify the criteria to be used for the selection of the next task.

For example a disk unit employing the discipline first-come-first-scheduled (*fcfs*) can typically be defined as follows:

```
COMPONENT disk: server (LET schedule := fcfs(1));
```

The parameter of the *fcfs*-scheduling discipline defines the number of processes which can be in the *entry area* simultaneously. In the given example at most one process can receive service. One is also the default value for *fcfs*.

2.3.4. Modelling Multi-Processors

A multi-processor can be modelled by selecting the speed proportional to the filling. This is done only up to a certain limit, which is governed by the number of available processors.

The following declaration defines a double processor system.

```
COMPONENT
   multiprocessor : server (LET dispatch := sdshared ([[1,2],[1.0,2.0]]));
```

If you like to model the total service rate of n processors which can be significantly less than n times the rate of a single processor because of competition for software locks and interference in accessing main memory, you should specify the effective service rate for each possible filling.

See for example the following "four-processor system" component declaration.

```
COMPONENT
    quad_processor : server
        (LET schedule     :=  immediate,
         LET dispatch:=   sdshared    ([[1,2,3,4],[1.0,1.7,2.2,2.6]]);
```

In both examples, multiprocessor and quad-processor, we omitted the norm speed parameter for the dispatch procedure. If we want to change the norm speed from 1.0 to, e.g., 1.5 we can write:

```
... LET dispatch := sdshared (..., LET speed := 1.5);
```

Instead of LET speed := 1.5 a real variable or a real constant can be used as actual parameter, too.

2.3.5. Modelling of Degradation due to System Overhead

There are systems which suffer from performance degradation due to heavy work load. Typical examples are thrashing in paging systems and Ethernet-like protocols under heavy traffic. In the following example we consider the Ethernet protocol.

The Ethernet protocol permits simultaneous access to the bus. The resulting conflicts are regulated by repeating the access after a certain delay time. First the throughput will increase with the load (number of packets to transmit) to a certain threshold (which depends on the Ethernet parameters), but will then drop sharply to a level of very poor performance.

To model this phenomenon, state-dependent speeds can be used in the following way:

```
COMPONENT
bus : server
        (LET dispatch := sdshared    ([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                                       [1.0,1.8,2.7,3.5,4.2,4.5,4.6,4.6,3.0,1.5]]));
```

This is only an example; the values given do **not** stem from a measurement.

2.4. Services

Services are used for the instantiation of processes obeying identical load patterns. Services may have parameters which are specified in a list of formal parameters.

Due to restrictions imposed by METHOD ANALYTICAL service parameters must not be used in conditions of control statements. And moreover you can use only the control statements listed below and service calls in your services. You have a richer choice of possibilities in simulation of course. We refer to Section 9.5. for more information. A service, which due to historical reasons is specified as a service type, has the following syntactical structure:

```
TYPE service_name SERVICE (formal_parameters);
    USE
        SERVICEservice1 (...);
                service2 (...);
    END USE;

BEGIN
    …
    {process pattern composed of control statements and}
    {calls of service1, service2,...}
    …
END TYPE service_name;
```

Services can be seen as "service types", as indicated by the HI-SLANG notation. Then "service objects" are processes which are generated dynamically during run time by means of:

```
CREATE 1 PROCESS service_name (...) EVERY negexp (1/m);
```

or

```
CREATE n PROCESS service_name (...);
```

In the first case processes are generated continuously according to a temporal pattern specified by *negexp (1/m)*, i.e., a new process is generated with exponentially distributed inter-instantiation times (mean *m*).

Note that the body of the service declaration must not contain an infinite loop, otherwise the population grows to infinity. In the second case, *n* processes are generated in the same time instant and have to remain in the system permanently provided their process pattern is of the infinite loop type.

2.5. Model Types

A model type forms the uppermost level of a model. It is an analysable unit, which looks like a component type except that a PROVIDE block is missing. Remember that a model does not provide services! Consider the following example of a model type:

```
TYPE model_type_name MODEL (formal_parameters);
    …
    {definition of the load, composed of one or more services}

    {definition of the machine, composed of one or more component objects}
    {of lower levels}

    REFER...{service names}...TO...{component names}...EQUATING
        …
        {referring the load to the machine}
        …
    END REFER;

BEGIN

    {initial statements to create processes}

END TYPE model_type_name;
```

A model object is generated in the experiment block (more precisely in the EVALUATE statement) by:

```
MODEL model_object_name : model_type_name (...{actual parameters}...);
```

2.6. Spend and Hold

The normal way to specify (either implicitly or explicitly) time durations in a HIT model is to use services from lower levels which finally lead to time consumption at the standard *servers.*

Another way to describe time delays is given by the predefined services *hold* and *spend*. For these calls the HIT system implicitely introduces *server* components. A call of *hold* causes the process to pause for a certain time interval. Note that the interpretation of *hold* is left to you! *Hold*, e.g., models the execution of a service or the passivation of the calling process. To ensure the treatability within separable models the parameter of hold is restricted to the *negexp* function.

The *spend* service works in a similar way. The main difference is that *spend* is controlled by the *dispatch* procedure of the component embracing the calling service. If the calling service is located at highest level, e.g., in the model type, *spend* has the same effect as *hold.*

Some examples will demonstrate the use of *spend* and *hold*:

```
hold    (5);              {the process sleeps for 5 time units}
hold    (negexp (1/2));   {mean sleeping time is 2 time units}
spend   (5);              {depending on the dispatch procedure of the embracing}
                          {service there may occur contention for spend}
```

2.7. Distribution Functions

2.7.1. Negexp

In order to model the random variation of the processing time, e.g., "service duration" or "time between process creations" the function *negexp* will be used. It has the form:

```
negexp (R)
```

The parameter R denotes the rate (number of events per time unit) of a negative exponential distribution with mean M=1/R. The expressions *negexp* (R) and *negexp (1/M)* are equivalent. The result of *negexp* is a positive real value, which is randomly chosen according to a negative exponential distribution. If R ≤ 0, a run time error will appear.

We give two examples for typical applications of *negexp*:

- *Negexp* must be used to specify a random pattern of arrivals (or pattern of creation) of temporary processes, for example as follows

  ```
  CREATE 1 PROCESS batch_task EVERY negexp (0.1);
  ```

  This statement has two interpretations which are completely equivalent:

  - batch jobs of service *batch_task* are created and "appear" in the model at a rate of 0.1, or
  - the time between two successive creations is in the average 10 time units.

- *Negexp* must be used to specify the amount of required service from a standard component.

  The statement *read_file (negexp (5.0))*;
  or equivalently *read_file (negexp (1/0.2))*;

  requires the use of a provided service bound to *read_file* for 0.2 time units on the average. Because in general several processes compete for a single component, the mean turnaround time (composed of waiting time and service time) for the *read_file* process will be larger than 0.2.

2.7.2. Draw

The function *draw* is used to choose in a probabilistic manner between different alternatives, e.g., access to disk units or as a second example alternative routings in a communication network.

draw is a boolean function with a parameter *p* of type REAL:

If 0<p<1, the value will be TRUE with probability *p*, FALSE with probability 1-*p*.
If p=0, the value will be always FALSE.
If p=1, the value will be always TRUE.

The function *draw* is typically used within the conditions of control statements.

2.8. Control Statements

2.8.1. The Infinite Loop

Syntax: LOOP {statements} END LOOP;

An infinite LOOP statement is used when we wish to repeat the same sequence of statements forever. A typical use of this LOOP construct is the modelling of processes moving permanently through the system. These processes are sometimes called permanent processes or cyclic processes. The type declaration for such processes usually contain an infinite LOOP construct without exit!

2.8.2. The TIMES Loop

Syntax: AVERAGE n TIMES LOOP {statements} END LOOP;

The TIMES loop causes the repetition of a sequence of instructions. The mean number of repetition is given by n. It can be a real expression with value >= 0.0. The LOOP and END LOOP act as a paranthesis and bracket the group of statements to be repeated.

The distribution behind AVERAGE is geometric; it is interpreted as

```
X := n/(n+1); WHILE draw (X) LOOP {statements} END LOOP;
```

2.8.3. The WHILE Loop

Syntax: WHILE draw (prob) LOOP {statements} END LOOP;

The WHILE clause decides whether or not processing is to continue. If the value of the random drawing procedure *draw* is true, the sequence of statements between LOOP and END LOOP will be executed. The boolean procedure *draw* (p) returns the value TRUE with probability *p, 0≤p≤1*, or the value FALSE with probability 1-*p*. The mean number of iterations is *p/(1-p)*. For *p*=1 the loop is endless.

Note: Don´t use service parameters within expression *prob.* This advice is valid for all control statement conditions.

We should add, that in case of simulative evaluation other forms of the WHILE loop are possible. The restriction to the *draw* function and the service parameters is due to properties of METHOD ANALYTICAL.

2.8.4. The UNTIL Loop

Syntax: LOOP {statements} END LOOP UNTIL draw (prob);

The condition stipulated in the UNTIL clause is evaluated and, if it is false, the loop will be repeated. Otherwise execution will be terminated at this point. Unlike the WHILE loop, the UNTIL loop is at least executed once. The mean number of iterations is *1/prob*. For *prob*=0 we have an infinite loop!

2.8.5. The IF Statement

Syntax: IF draw (prob) THEN {statements} ELSE {statements} END IF;

The IF statement enables the choice between two alternative courses of action. According to the value of *draw (p)* the appropriate course is selected. The statements after THEN or the statements after ELSE are executed, depending on whether or not *draw (p)* is true. The ELSE clause is optional.

2.8.6. The BRANCH Statement

The BRANCH statement enables the user to choose between many alternative courses of action:

```
BRANCH
   PROB p1 :   {statements}
   PROB p2 :   {statements}
      ...
   PROB pn :   {statements}
   ELSE    :   {statements}
END BRANCH;
```

The statements to be executed are chosen depending on probability values $p_1,p_2,\ldots,p_n$ given after the keyword PROB. The sum of these probability values must be less or equal one. The ELSE clause is optional. If it exists, it will sum the probabilities to one, otherwise the body of the BRANCH statement will be skipped with probability $1\text{-}p_1\text{-}p_2\text{-}\ldots\text{-}p_n$.

We provide a simple example which illustrates the BRANCH statement.

```
BRANCH
   PROB 0.1:  calculate  (....);
              store      (....);
   PROB 0.5:  calculate  (....);
   ELSE    :  store      (....);
END BRANCH;
```

With probability 0.1 the sequence "*calculate-store*" will be chosen, with probability 0.5 and 0.4 *calculate* and *store* will be used, respectively.

2.8.7. The CHAIN Statements

Note that each HI-SLANG service describes a chain of a corresponding queuing network. If a queuing network is given as a start point it might be difficult to construct the corresponding HI-SLANG model. Therefore a recent addition to HI-SLANG, the CHAIN statements allow to describe queuing systems directly in the following way:

```
OPEN_CHAIN
    QNODE node_name
        PROB p1 :   node_name1
        PROB p2 :   node_name2
            …
        PROB pn :   node_namen
        ELSE    :   node_namen+1
    QNODE node_name2
        …
END OPEN_CHAIN;
```

The PROB parts describe the selection probabilities for the successor nodes and their names. In a CLOSED_CHAIN they have to sum up to one, while in an OPEN_CHAIN statement the remaining probability is the exit probability.

The service *batch_task* of *example1* can now as well be specified as follows:

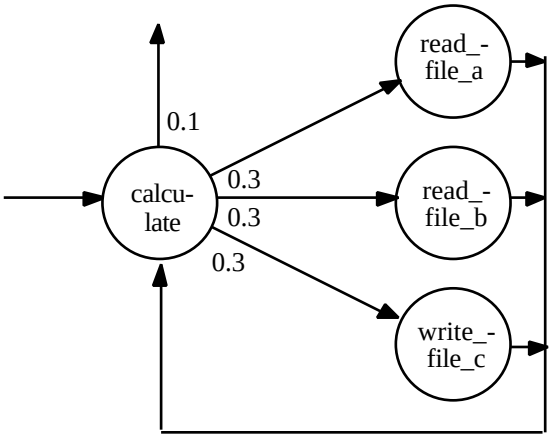

Figure 2.2: Specification of a Service using the CHAIN statement

Please compare the figure above with Figure 1.1. Of course the nodes may be drawn like stations of a queueing network, but here we preferred a more simple representation. The corresponding HI-SLANG representation is stright forward:

```
TYPE batch_task SERVICE;
    USE SERVICE
            calculate       (amount: REAL DEFAULT negexp(1/0.2));
            read_file_a     (amount: REAL DEFAULT negexp(1/0.1));
            read_file_b     (amount: REAL DEFAULT negexp(1/0.2));
            write_file_c    (amount: REAL DEFAULT negexp(1/0.4));
    END USE;

BEGIN
    OPEN_CHAIN
        QNODE calculate
            PROB 0.3 : read_file_a;
            PROB 0.3 : read_file_b;
            PROB 0.3 : write_file_c;
            {else (prob 0.1) exit the chain}
        QNODE read_file_a
            PROB 1.0 : calculate;
        QNODE read_file_b
            PROB 1.0 : calculate;
        QNODE write_file_c
            PROB 1.0 : calculate;
    END OPEN_CHAIN;
END TYPE batch_task;
```

3. Hierarchical Model Specification

3.1. Overview

The objective of this chapter is:

- to introduce the concept of hierarchical modelling;
- to discuss horizontal and vertical refinement;
- to illustrate this concept by means of an example and
- to explain more on component types.

In order to gain an overall view, we present the following figures. They illustrate the stepwise refinement of the model we are going to discuss in the following sections.

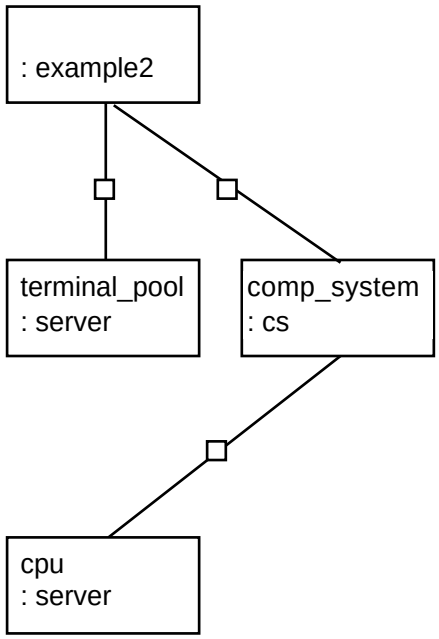

Figure 3.1: Gross Specification of a Two Level Model

The model, called *example2*, consists of a *server* and a component *cs*. The name *cs* is an abbreviation of central server and consists itself of a *server.*

The horizontal refinement of *cs* leads to a new component type named *cs_ref_hor*, consisting of five *servers*, see Figure 3.2. Note that the depth of the hierarchy does not change.

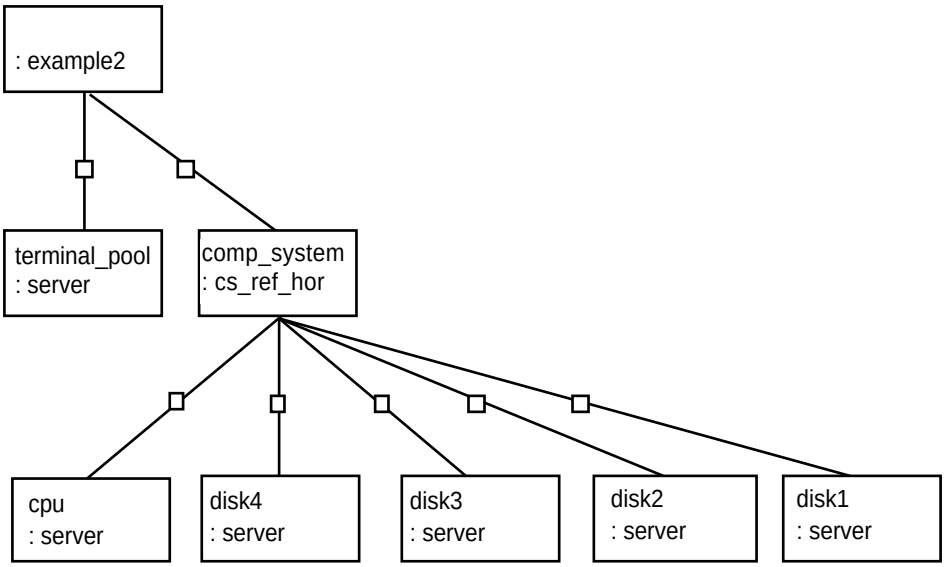

Figure 3.2: Horizontal Refinement of Figure 3.1

Finally this new *cs* component type *cs_ref_hor*, respectively its *disk4*, is refined vertically by changing its type from *server* to *io_system*. See Figure 3.3.

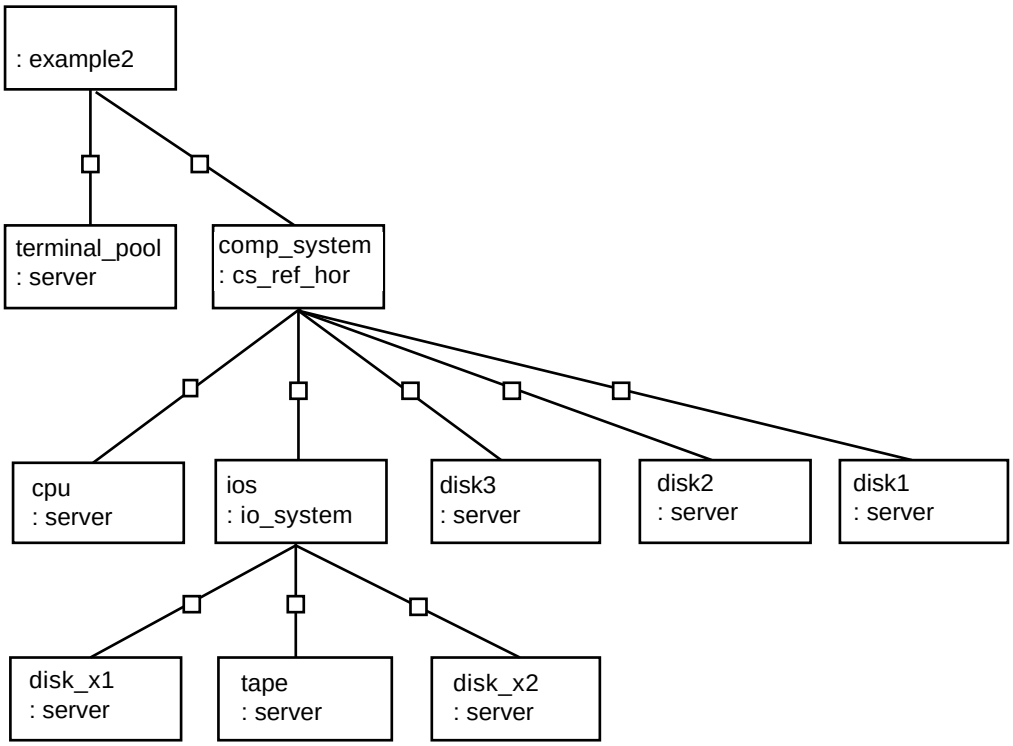

Figure 3.3: Vertical Refinement of Figure 3.2

3.2. A Hierarchical Model

In order to ease the design of multi-level/multi-layered models and to enable a piecewise specification by different people, the HIT system provides some features for organizing a model into vertical levels/layers.

In this chapter we are primarily concerned with these features. They are best discussed in terms of an example. Before we define our model, let us briefly consider the following HIT features.

3.2.1. Transforming a Model into a Component

We have learned that if a machine consists exclusively of standard components, we will attain a flat model (see *example1*). In order to build hierarchical models of arbitrary height we must create components. One way to do this is to transform a model type into a component type by declaring certain of its internally specified services as externally usable. Initial statements (e.g., CREATE) can, but must not be removed. We thereby arrive at the basis of a next higher layer. Consequent application of this concept leads to arbitrarily deep multi-level/multi-layered models with an uppermost layer and a lowermost layer. The former consists of a load/machine complex without any externally accessible services. The latter is directly based on standard components.

3.2.2. A Two-Level Model

In our example, called *example2*, we deal with a two-level model, which is defined as follows:

- The load consists of two different kinds of tasks, which are described by the services *cmd1* and *cmd2*, respectively. Users submit these tasks (*dialog* jobs) after some thinking time.

- The machine is composed of the components *terminal_pool* and *comp_system* (abbreviation for computer system). The component *terminal_pool* will be directly represented in the model by the standard component type *server*, whereas the component *comp_system* forms a further hierarchical level, providing the services *cmd1_processing* and *cmd2_processing* as externally usable.

We will write down this model in HI-SLANG. We describe the component *comp_system* in a gross way neglecting any details first. In a second step, presented in the next sections, *comp_system* will be refined in a top-down manner.

3.2.2.1. Component Type cs

The following example is a gross specification of user-defined component type *cs*. This component forms a higher layer. It declares its internally specified services *cmd1_processing*, and *cmd2_processing* as externally usable.

```
TYPE cs COMPONENT;

    PROVIDE
        SERVICE
            cmd1_processing;
            cmd2_processing;
    END PROVIDE;


    TYPE cmd1_processing SERVICE;
        USE SERVICE
            compute (m:REAL);
        END USE;
    BEGIN

        AVERAGE 10 TIMES
        LOOP
            compute (negexp(1/0.045));
        END LOOP;

    END TYPE cmd1_processing;


    TYPE cmd2_processing SERVICE;
        USE SERVICE
            compute (m:REAL);
        END USE;
    BEGIN

        AVERAGE 20 TIMES
        LOOP
            compute (negexp(1/0.135));
        END LOOP;

    END TYPE cmd2_processing;


    COMPONENT cpu: server (LET schedule :=  immediate,
                           LET dispatch:=  shared);


    REFER cmd1_processing, cmd2_processing TO cpu
    EQUATING
        cmd1_processing.compute WITH cpu.request;
        cmd2_processing.compute WITH cpu.request;
    END REFER;

END TYPE cs;
```

3.2.2.2. Model Type example2

The model type *example2* is a hierarchical model of a dialog system with two services. The model has two parameters, giving the number of processes for each service in the model.

The load pattern is described by the two services *cmd1* and *cmd2*. They describe the view point of a dialog system user: After thinking and typing a command, the task is executed (run) and the result is returned to the user's terminal. These *think-run*-cycles are modelled using an infinite loop construct.

Note that a component *comp_system* of type *cs* is declared in the following model type.

```
TYPE example2 MODEL (n1,n2:INTEGER);

    TYPE cmd1 SERVICE;
        USE SERVICE
            think (thinktime : REAL);
            run;
        END USE;
    BEGIN
        LOOP
            think (negexp(1/5));
            run;
        END LOOP;
    END TYPE cmd1;


    TYPE cmd2 SERVICE;
        USE SERVICE
            think (thinktime : REAL);
            run;
        END USE;
    BEGIN
        LOOP
            think (negexp(1/10));
            run;
        END LOOP;
    END TYPE cmd2;


    COMPONENT
        terminal_pool:    server
        (   LET accept    :=  always,
            LET schedule :=  immediate,
            LET dispatch :=  equal,
            LET offer :=  all);

    COMPONENT
        comp_system:    cs
        (   LET accept    :=  always,
            LET offer :=  all);


    REFER cmd1, cmd2 TO terminal_pool, comp_system
    EQUATING
         cmd1.think   WITH  terminal_pool.request;
         cmd1.run WITH  comp_system.cmd1_processing;
         cmd2.think   WITH  terminal_pool.request;
         cmd2.run WITH  comp_system.cmd2_processing;
    END REFER;
```

```
BEGIN
    CREATE n1 PROCESS cmd1;
    CREATE n2 PROCESS cmd2;
END TYPE example2;
```

3.2.2.3. The Experiment experiment2

A corresponding experiment is simple. We are interested in population and turnaround time of both kinds of processes in *comp_system* for the case that 20 processes *cmd1* and 2 processes *cmd2* are executed by the user:

```
EXPERIMENT experiment2 METHOD ANALYTICAL"DOQ4";
BEGIN

    EVALUATE MODEL model2 : example2(20, 2);

        EVALUATIONOBJECT
            computer VIA model2.comp_system;

    BEGIN

        MEASURE POPULATION, TURNAROUNDTIME
            AT computer;

    END EVALUATE;

END EXPERIMENT experiment2;
```

3.3. Refinement of a Component Type

In the preceeding section, our aim was to get an overall structure of our model before we get bogged down in too much detail. For simplicity, we ignored some of the more complicated aspects of HI-SLANG. But now it is convenient to consider some of these aspects. As mentioned before, the model *example2* is a hierarchical model because the component *comp_system* is not of the standard component type *server.* It is itself composed of several components.

3.3.1. Horizontal Refinement

It should be noted that HIT provides, in addition to the vertical model structure, a horizontal structuring within a hierarchical layer. In order to fully understand and appreciate this feature, we are going to refine the component *comp_system* horizontally. We model the processing of tasks in *comp_system* in a more detailed fashion, such that computing-and I/0-activities will be distinguished.

It is important to realize that refining the component *comp_system* horizontally only affects the implementation of the layer, whereas the interfaces between the layers remain unchanged!

3.3.1.1. The Refined Component Type cs

The following example gives the refined specification of the user-defined component type *cs*. The *comp_system* component is refined horizontally. The task processing is modelled in a detailed manner.

```
TYPE cs COMPONENT;
    PROVIDE SERVICE
        cmd1_processing;
        cmd2_processing;
    END PROVIDE;

    TYPE cmd1_processing SERVICE;
        USE SERVICE
            compute   (m : REAL);
            access1   (m : REAL);
            access2   (m : REAL);
            access3   (m : REAL);
            access4   (m : REAL);
        END USE;
    BEGIN

        AVERAGE 10 TIMES
        LOOP
            compute (negexp(1/0.045));
            BRANCH
                PROB 0.25   :  access1   (negexp(1/0.035));
                PROB 0.25   :  access2   (negexp(1/0.035));
                PROB 0.25   :  access3   (negexp(1/0.035));
                ELSE        :  access4   (negexp(1/0.035));
            END BRANCH;
        END LOOP;

    END TYPE cmd1_processing;
```

```
TYPE cmd2_processing SERVICE;
    USE SERVICE
        compute    (m : REAL);
        access1    (m : REAL);
        access2    (m : REAL);
        access3    (m : REAL);
        access4    (m : REAL);
    END USE;
BEGIN

    AVERAGE 20 TIMES
    LOOP
        compute (negexp(1/0.135));
        BRANCH
            PROB 0.25 : access1 (negexp(1/0.035));
            PROB 0.25 : access2 (negexp(1/0.035));
            PROB 0.25 : access3 (negexp(1/0.035));
            ELSE      : access4 (negexp(1/0.035));
        END BRANCH;
    END LOOP;

END TYPE cmd2_processing;


COMPONENT
cpu: server (LET schedule    :=  immediate,
                 LET dispatch    :=  shared);

COMPONENT
disk1,
disk2,
disk3,
disk4 : server (LET schedule:=  fcfs,
                    LET dispatch:=  equal);


REFER    cmd1_processing, cmd2_processing
    TOcpu, disk1, disk2, disk3, disk4
EQUATING
      cmd1_processing.compute  WITH    cpu.request;
      cmd1_processing.access1  WITH    disk1.request;
      cmd1_processing.access2  WITH    disk2.request;
      cmd1_processing.access3  WITH    disk3.request;
      cmd1_processing.access4  WITH    disk4.request;

      cmd2_processing.compute  WITH    cpu.request;
      cmd2_processing.access1  WITH    disk1.request;
      cmd2_processing.access2  WITH    disk2.request;
      cmd2_processing.access3  WITH    disk3.request;
      cmd2_processing.access4  WITH    disk4.request;
END REFER;

END TYPE cs;
```

3.3.1.2. Inclusion of cs in example2

To incorporate the refined version of *comp_system* in the source text of *example2* we can of course replace the "gross specification" by the "refined specification" textually. Fortunately HIT provides a better facility. The control statement

```
%COPY "link_name"
```

enables us to insert source text files, e.g., component types or services. The %COPY command requires as parameter a link name. The files can be declared in the control part (i.e., configuration part) by means of:

```
%COMPILER
%BIND "link_name" TO file_name
%END
```

Alternatively the control part can be omitted. In this case the HIT systems responds with

```
%BIND "link_name" TO ?
```

and you have to type the corresponding file name. See also Appendix B.

3.3.2. Vertical Refinement

In the preceeding section we have considered the horizontal refinement of the component type *cs*. Now we intend to vertically refine the same component type *cs*, by refining one of its components with the help of a new component type.

Let us assume that *disk4* will be replaced by a subsystem, which is composed of two disk units and a tape unit. Accordingly, the used service *access4* will not be bound anymore to *disk4.request* but to *io_operation* which is provided by the component of type *io_subsystem*.

For reasons given later we decide that *io_operation* in contrast to *request* should not have any parameters. However, the time consumption of *io_operation* is now completely determined hierarchically and more natural by the pattern of service calls within its body.

3.3.2.1. The Component Type io_subsystem

The vertical refinement of *disk4* within *cs* can be specified as follows:

```
TYPE io_subsystem COMPONENT;

    PROVIDE SERVICE
        io_operation;
    END PROVIDE;


    TYPE io_operation SERVICE;
        USE SERVICE
            write_file_a  (m :REAL);
            read_file_b   (m :REAL);
            save_file     (m :REAL);
        END USE;
    BEGIN

        IF draw (0.5)
        THEN
            write_file_a (negexp (30));

        ELSE
            IF draw (0.1)
            THEN
                save_file  (negexp (0.1));

            ELSE
                AVERAGE 5 TIMES
                LOOP
                    read_file_b (negexp (200));
                END LOOP;
            END IF;

        END IF;

    END TYPE io_operation;


    COMPONENT
        disk_x1,
        disk_x2,
        tape  : server (LET schedule  :=  fcfs,
                        LET dispatch  :=  equal);


    REFER io_operation TO disk_x1, disk_x2, tape
    EQUATING
        io_operation.save_file      WITH   tape.request;
        io_operation.read_file_b    WITH   disk_x2.request;
        io_operation.write_file_a   WITH   disk_x1.request;
    END REFER;

END TYPE io_subsystem;
```

3.3.2.2. Inclusion of io_subsystem in cs

Note that the component type *cs* (refined version) must be changed slightly for this vertical refinement:

- The declaration of component *disk4* must be changed into:

  ```
  COMPONENT  ios : io_subsystem;
  ```

  (We suggest to change the name *disk4*, e.g., into *ios*).

- In the REFER part the binding of *access4* must be changed.

- *access4* is now parameterless, cf. the service *io_operation* above, since the amount of time used for an *io_operation* is now completely determined by the *io_subsystem* itself. Therefore the USE blocks of the services within *cs* and the service calls of *access4* must be changed.

The following points are worth considering:

- Parameterization of service *io_operation* is possible, but service parameters must not be used as a parameter of the procedure *draw* or the BRANCH statement. More general: Service parameters must not occur in the conditions of control statements of service bodies.

- Note also that the vertical refinement does not require any change in the experiment part (if we are not interested in performance indices of the new components).

3.4. HI-SLANG Subset for Hierarchical Models

In this section the syntactical structure of elements of the HI-SLANG model world, which was already presented in terms of examples, is presented in more detail.

Notice, that all structured objects, i.e., components and models, are generated using the type concept.

3.4.1. Components and Component Types

A component type has the following syntactical structure. Note that names and parameters of externally accessable services are given in a PROVIDE part. You can imagine these services as exported from the component.

```
TYPE comp_type_name COMPONENT (...{formal parameters}...);
    PROVIDE
        SERVICEservice1 (...);
                    service2 (...);
                    …
    END PROVIDE;
    …
    {definition of the load, composed of some services}
    {definition of the machine, i.e., some component objects of lower levels}
    …
    REFER service1, service2, ... TO component1, component2,...
    EQUATING
        service1.use1 WITH component1.provide1;
        service2.use2 WITH component2.provide2;
    …
    END REFER;
    …
    {optional static definition of process objects}
    …
BEGIN
    …
    {optional initial statements, like dynamic creation of processes}
    …
END TYPE comp_type_name;
```

In the body of a component type, a load will be referred to a machine. This is done by binding the used services of the load to the provided services of the machine. In a higher layer, objects of type *c_type_name* will be declared as follows:

```
COMPONENT comp_object_name: comp_type_name (...);
```

Even ARRAYs may be used:

```
COMPONENT comp_object_array: ARRAY [1..4] OF comp_type_name (...);
```

Only one-dimensional static component arrays are allowed. At the creation of the component object, the initial statements will be executed exactly once. For more information about component arrays see the HI-SLANG Reference Manual.

3.4.2. Enclosed Components

HI-SLANG allows for using services of the same component in different layers of a model. Such components are called enclosed. Enclosed components will be recognized by using the keyword ENCLOSE instead of the keyword COMPONENT, e.g.,

```
ENCLOSE cpu : server;
```

in place of

```
COMPONENT cpu : server;
```

Additionally there must be a main declaration in the latter style somewhere else in the model. Only here the parameters of the component type may be set. The ENCLOSE declaration can be seen as a reference to that component generated by the main declaration.

The following example which is based on a two-layer model illustrates the use of enclosed components. The model is composed of three components, namely *disks*, *connections* and the control unit *cu*. In this example, the component *cu* plays a special role. It is physically only once available but is actually accessed from several parts of the model.

This shared access is necessary for both the realization of the provided services of the components *disks* and *connections* as well as the realization of the load processes *transactions1* and *transactions2*. This model can be represented graphically, as follows:

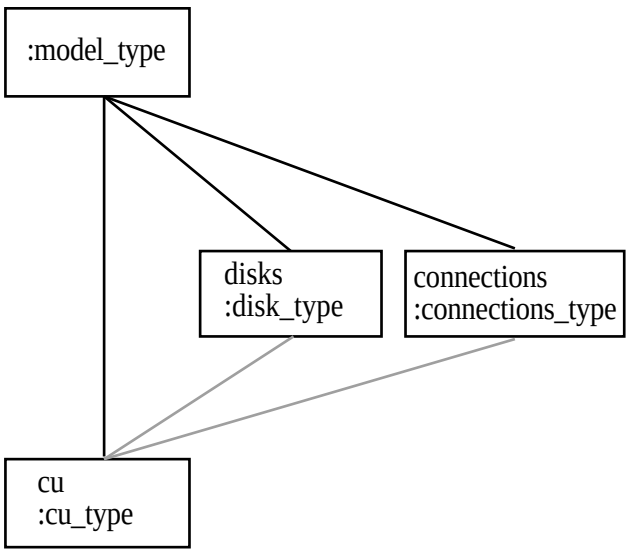

Figure 3.4: A Two-Layer Model

An (incomplete) HI-SLANG description of *model_type* follows, showing only the important parts with respect to the enclosed component *cu.*

```
TYPE cu_type COMPONENT;

    {provides three services, namely: rcu_const, skam_req and skam_access}
    {as externally usable services}

END TYPE cu_type;


TYPE connections_type COMPONENT;

    {the load uses the provided service rcu_const of cu}

    COMPONENT   plc, ltg   :   server;
    COMPONENT   places     :   server;
    ENCLOSE     cu         :   cu_type;

END TYPE connections_type;


TYPE disks_type COMPONENT;

    {the load uses all provided services of cu}

    COMPONENT   dsk1, dsk2 :   server;
    ENCLOSE     cu         :   cu_type;

END TYPE disks_type;


TYPE model_type MODEL;

    {The load is described by the services transactions1 and transactions2}

    COMPONENT   connections :   connections_type;
    COMPONENT   disks       :   disks_type;
    ENCLOSE     cu          :   cu_type;

END TYPE model_type;


COMPONENT cu : cu_type (LET schedule := fcfs);      {the main declaration}
```

An alternative construction is to move the line TYPE *model_type* MODEL to the top of the example. In this case the main declaration of *cu* (the last line) can replace its ENCLOSE declaration within *model_type.*

3.4.3. Load Filtering Hierarchies

The HIT evaluation concept permits detailed specification of desired results for hierarchical models. This is done by identifying a so-called load filtering hierarchy: Service calls in higher layers, which have an effect on components of lower layers, can be distinguished and evaluated separately.

Such a load filtering hierarchy is described by the concatenation of triples, which define a calling hierarchy along the hierarchical structure of the model. The triples consist of:

- the component or model name
- the service name within the component
- the USE name within the service

Note that either the USE name or the service and the USE name may be omitted, with the meaning, that all services of the component or all USE names of the service are concerned. Consider the following concatenation

```
(m.c1, st1, u1).(c2, st2)
```

This notion addresses the effect of *st1*-processes generated (by CREATE statements) in component *c1* of model *m* on the component *c2*, caused by calls of *st2* via USE name *u1* of *st1*.

Note that if the root of the load filtering hierarchy (the first element of the first triple) lies in the uppermost layer of the model, the model name must be specified in place of the component name, otherwise dot notation (starting with the model name) has to be used to identify the load originating component (containing the respective CREATE statements).

By the use of load filtering hierarchies performance indices (streams as, e.g., THROUGHPUT) can be thought of being composed of a set of different performance values, from which only some might be of interest. Consider, e.g., a component *c2* providing two services *s21*, *s22*, which are used by three different services *s11*, *s12*, *s13*, of a component *c1* within the next higher layer, itself being part of a model *m*. The following non-standard graphic may help:

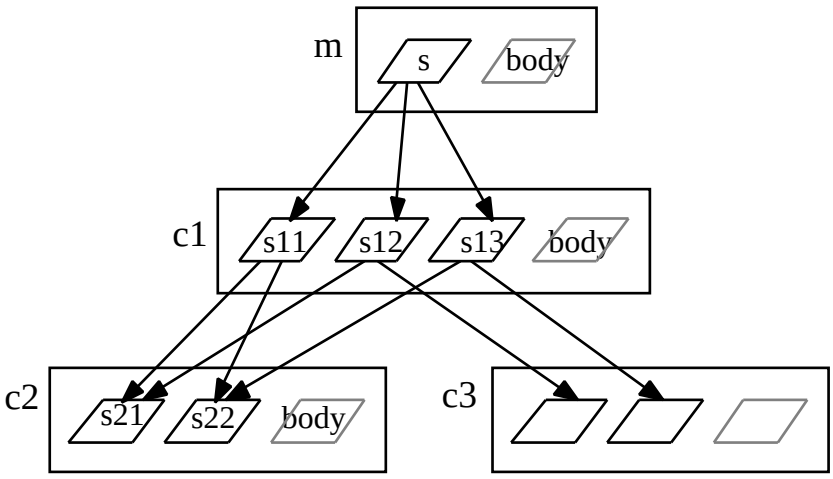

Figure 3.5: Illustration of Load Filtering Hierarchies

Then the throughput for *c2* caused by processes generated in *c1* is composed of, e.g., the following throughput values, which are filtered by the hierarchies given below:

- for all services of c2 (m.c1).(c2)
- for s21 only (m.c1).(c2, s21)
- for s22 only (m.c1).(c2, s22)
- for s21, caused by s11 (m.c1, s11).(c2, s21)
- for s22, caused by s13 (m.c1, s13).(c2, s22)

There are even more possibilities, e.g., those throughput portions caused by processes generated in *m*. These can be filtered by hierarchies with root *m* like

```
(m, s).(c1, s11).(c2, s21)
```

Load filtering hierarchies are specified in HI-SLANG by:

```
HIERARCHY hierarchy_name  DEFAULT  triple_concatenation ;
```

Hierarchies will mainly be used in the MEASURE statement as follows:

```
MEASURE    stream_name1,
           stream_name2, ...
           AT          evaluation_object
           DUE  TO     hierarchies_name_list ;
```

The DUE TO construct specifies the hierarchy (or hierarchies) for which the desired measures are to be evaluated. You can also specify the predefined load filtering hierarchy *all.* In this case no filtering is performed. It is even possible to omit the DUE TO construct at all, because DUE TO *all* is used as default.

The hierarchy declaration is best illustrated by means of an example. Consider the following alternative experiment block of *example2.*

```
EXPERIMENT experiment2 METHOD ANALYTICAL "DOQ4";
BEGIN
    EVALUATE
        MODEL model2 : example2 (20, 2);

        EVALUATIONOBJECT
            computer VIA model2.computer_system;

        HIERARCHY h1 DEFAULT
            (model2, cmd1, run).(computer_system);

        HIERARCHY h2 DEFAULT
            (model2, cmd2, run).(computer_system);

    BEGIN

        MEASURE POPULATION, TURNAROUNDTIME
            AT computer DUE TO h1, h2, all;

    END EVALUATE;
END EXPERIMENT experiment2;
```

Two disjoint load filtering hierarchies, which end in the same component, can be merged to a new hierarchy by:

```
HIERARCHY hierarchy_name MERGE hierarchy_name_1, hierarchy_name_2, ...;
```

The new hierarchy contains the union of load paths of all individual hierarchies listed. The predefined hierarchy *all* exists for any evaluation object. It can be seen as a merge of all possible hierarchies ending in that component.

4. Hierarchical Model Analysis (Aggregation)

4.1. Overview

In this chapter we introduce a HI-SLANG feature which is important for different reasons. The so called technique of pre-analysis or aggregation. It helps to simplify complex models by reducing their size and therefore improves the solution speed.

In short the objective of this chapter is to

- discuss the principle of hierarchical analysis;
- illustrate the aggregation technique by an example and
- explain the associated HI-SLANG constructs.

4.2. Principles of Hierarchical Analysis

In the preceeding chapter, we became acquainted with the concept of hierarchical model construction by means of horizontal and vertical refinement.

We learned that machine and load consist of components and services, respectively. The separation of their specifications strongly supports the goals "division of labour" and "reusability". And it allows therefore the systematical development of complex models.

Structuring a model in components is also greatly advantageous from the analysis point of view. Under certain conditions it is possible to analyse a component type totally separate from its environment and to use the results of this pre-analysis in other models and/or other environments afterwards. This means that HIT supports hierarchical model specification as well as hierarchical model analysis.

We sketch the principle of pre-analysis and finally consider its advantages. Assume a component type providing a number of externally usable services. In performance modelling, we are specially interested in the time needed by the component to process a service. How this service is processed by the component is irrelevant when posing questions of macroscopic nature. E.g., when inquiring about the response time of dialog tasks, the duration of a single IO-CPU-cycle is not of interest.

As a consequence, from the performance modelling point of view, the performance behavior of a component is determined essentially by the processing duration of its provided services. The explicit modelling of numerous details is absolutely not necessary and is also not desirable in the analysis.

HIT provides an option for transforming detailed, deeply structured component types to aggregated component types having a very simple structure. Aggregated component types are substitute representations for the original component types and provide consequently the same services.

Due to the fact that a component is isolated from its environment during the pre-analysis, we also use the pictorial term "off-line analysis".

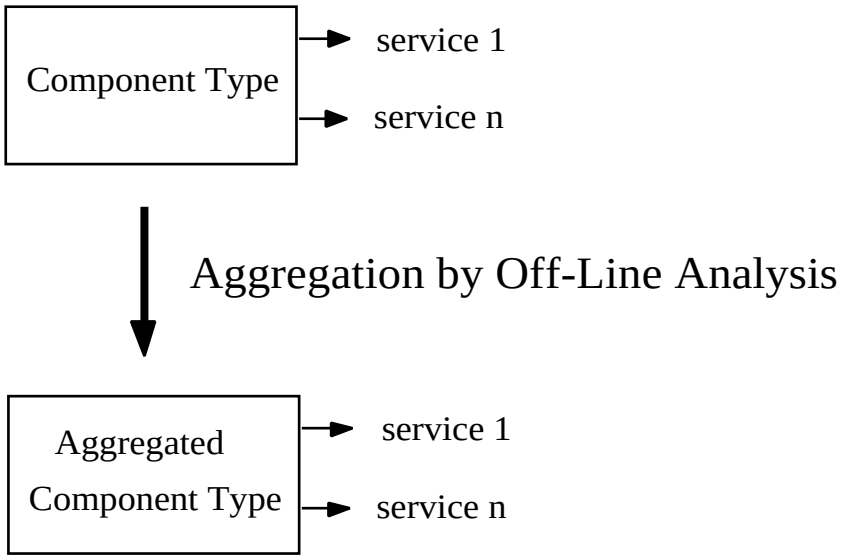

Figure 4.1: Construction of an Aggregated Component Type

The advantage of analyzing isolated component types emerges from the fact that components appearing as identical subsystems in different models have to be analyzed only once (or precisely: pre-analyzed). Afterwards they can always be used as a "pre-fabricated building block" in very different models.

A disadvantage in using aggregated components is that an evaluation of any details inside the component is of course impossible. Consequently you should try to aggregate model parts which are of no interest to your current investigations. In particular if you perform parametric analyses of large models, only those parts should be modelled in detail for which you want to vary an input parameter or for which you want to measure individual quantities. The rest of the model should be aggregated (if possible) to an equivalent substitute representation!

Component types to be aggregated must fulfil the restrictions of the DOQ4 algorithm, i.e., they must be separable, but the extensions listed in Section 5.1. can be used. In the latter case the aggregate is only approximate.

4.3. Applying Aggregation

Now we will explain how to perform a pre-analysis. We intend to aggregate the component type *cs* (i.e., the component *comp_system*) of *example2*, see Section 3.3.1. This is done by pre-analysing a part of this model (i.e., the component type) and creating a substitute representation of it.

The specification of the experiment will merely be composed of a so-called AGGREGATE statement. In it the maximum population of tasks (for each service provided by *cs*) will be specified.

The HI-SLANG source text of the aggregated component type, generated according to this specification, will be saved in a file, which is by default named by the file name generator (suffix *preana*).

The source file for the pre-analysis of *cs* (horizontal refinement) can be specified as follows:

```
TYPE cs COMPONENT;
    {see Section 3.3.1.1.}
END TYPE cs;

EXPERIMENT exp2agg METHOD ANALYTICAL "DOQ4";
BEGIN

    AGGREGATE cs;
        CREATE 20 PROCESS   cmd1_processing;
        CREATE   2 PROCESS   cmd2_processing;
    END AGGREGATE;

END EXPERIMENT exp2agg;
```

It is remarkable that the result of the aggregation is again a HI-SLANG component type with the same name and the same set of provided (parameterless) services. If you are curious, inspect the file containing the aggregated component type.

This means that aggregated component types can be included in HI-SLANG sources. The name of the file containing the aggregated component type is specified by the following control part:

```
%COMMON
%BIND "AGGCS" TO  file_name
%END
```

%COMMON is needed because the aggregate will be read by both the compiler and the used analyzer. The first reads the HI-SLANG interface, while the latter is only interested in the "speeds table".

In the source text the connection to this aggregated component type will be attained through the name of the aggregated component type. Note that the same name is used as component type name and as name of the aggregated component type. The aggregated component type is included by:

```
%COPY "AGGCS"
...
COMPONENT comp_system : cs;
```

The link name should be related to the name of the component type. In the same way an ordinary HI-SLANG component type can be included (see Section 3.3.1.2. and Appendix B.2.). Note that the names of the provided services must not be changed in the main source text. The same holds for the type names in the corresponding object declarations. In general a model can be configured by binding different versions of component types to the link names used.

When inserting aggregated components, one must take into account for which population the corresponding component type was pre-analyzed. In our example, the aggregated component can process a maximum of 20 and 2 processes from *cmd1_processing* and *cmd2_processing*, respectively.

4.4. HI-SLANG Subset for Model Aggregation

We give the syntax of the aggregate statement and list the restrictions which have to be fulfilled to be able to aggregate a component type or to use it.

4.4.1. Aggregate Statement

An AGGREGATE statement is needed for transforming a component type to an aggregated component type. The maximum population for each provided service must be given by a set of corresponding CREATE statements within the AGGREGATE statement. If this population is exceeded when actually using the generated aggregate, a warning will appear.

The AGGREGATE statement is contained in the EXPERIMENT block, in which both the method (in this case ANALYTICAL "DOQ4", which is currently the only one possible) and the experiment name are specified:

```
EXPERIMENT experiment_name METHOD ANALYTICAL "DOQ4"
BEGIN

    AGGREGATE component_type_name [OUTPUT "link name"];

        CREATE max_population_1 PROCESS service_1;
        CREATE max_population_2 PROCESS service_2;
        ...
        CREATE max_population_n PROCESS service_n;

    END AGGREGATE;

END EXPERIMENT experiment_name;
```

If you want to specify a file name for the aggregate (overriding the default name) you can use the OUTPUT "link name" clause. Only when performing an aggregation in this case, the control part must include something like:

```
%ANALYZER
%BIND "link_name" TO  file_name
%END
```

After executing the aggregation, the file with name *file_name* will contain the aggregated component type.

4.4.2. Restrictions in Aggregation

By performing aggregations and by using aggregated component types, the following restrictions must be taken into consideration:

- Component types, which should be aggregated must not have parameters. Also the provided services must have no parameters.
- If the keyword ENCLOSE appears in the source text of the component type, then the corresponding component object must be defined within this component type.
- Aggregation of a component is only feasible for components obeying to the restrictions of separable models and their DOQ4 extensions. But note, the use of an aggregated component is not limited to separable models, i.e., they can also be used in models using other analysis methods.

If you use an aggregated component type, consider the following points:

- Aggregated component types are admitted to be constituents of component types, which should be aggregated, too ("multi-level aggregation").
- Aggregated component types are admitted to be constituents of models, in which permanent and temporary processes exist, but the services of aggregated components may be used only by permanent processes.

5. Extensions and Limits of Separable Models

5.1. Overview

The objective of this chapter is:

- to introduce a class of non-separable models also solvable with DOQ4, but not with LIN2;
- to summarize aspects which cannot be treated by DOQ4 or LIN2, but by METHOD ANALYTICAL "NUMERICAL".

5.2. An Extension of Separable Models

5.2.1. Approximate Solution of a Class of Non-Separable Models

The HIT system offers nearly all relevant features of the class of separable models which have been explored by queueing theoriticians. But if the restrictions from separable models are widened, the class of treatable models will be growing. Of course the price is the loss of formal strength and the exactness of results. Nevertheless, the quality of the resulting quantitative measures is mostly fully sufficient for the needs of practice. The two extensions of separable models described as follows can be treated by the DOQ4 algorithm. Since DOQ4 is used for aggregation, these extensions also apply for the aggregation of component types.

5.2.2. FCFS Scheduling

Note that in case of separable models all requests to the same *fcfs-server* must have the same actual *negexp* parameter. In other words: All service requests to a *server* with *fcfs* scheduling discipline must be specified by the *negexp* function and the parameter of these *negexp* functions must have exactly the same value! If this is not the case an approximate solution technique, which has been integrated in the DOQ4 algorithm, will be applied automatically.

5.2.3. Priorities

Scheduling disciplines like preemptive and non-preemptive priority can not be treated by separable models. An approximate solution technique (again the DOQ4 algorithm) will be applied automatically. We distinguish between priority scheduling with and without preemption.

- **prioprep**: Priority Scheduling with Preemption

Preemption means a "newly arriving" service request interrupts a process of lower priority. The interrupted (preempted) process has to be repeated from the beginning, i.e.,we have a "priority preemptive repeat" discipline.

More precisely we have a "priority preemptive repeat" discipline with "resampling", i.e., the amount of the service to be restarted is determined again.

- **prionp :** Priority Scheduling without Preemption

In the non-preemptive case a newly arriving service request can not cause an interrupt. It has to wait for service until all services of lower priority have finished.

Because the component type *prioserver* is not an intrinsic part of HI-SLANG, but rather a member of the HIT standard modelling base, it must be introduced into the source by %COPY "PRIOSERVER".

A declaration of a component with priority scheduling looks like this:

```
COMPONENT cpu : prioserver (LET schedule := prioprep);
```

Instead of *prioprep* the strategy *prionp* can be used here. A process which is to be executed by a *prioserver* component must be declared in the following manner:

```
TYPE diaproc SERVICE;
    USE SERVICE
        compute (amount: REAL; prio: INTEGER DEFAULT 32767);
        …
    END USE;
BEGIN
    …
    compute (negexp (1/10), 3);
    …
END TYPE diaproc;
```

The service request *compute* has two parameters: the amount specified by a negative expontial distribution with mean value 10 and a priority of 3. Note that the highest priority is 0 and the lowest priority is 32767. Please note, that the REFER part for *compute* does not change comparing to a component of type *server.*

5.3. What Cannot be Treated by DOQ4 or LIN2

We summarize some aspects which can not be treated by METHOD ANALYTICAL "DOQ4" or "LIN2". Nevertheless we recommend to use this method in the early stages of a modelling study, neglecting non-separable aspects. You may include these aspects (if necessary) in later phases of your modelling enterprise; switching from analytical solution to numerical or simulative evaluation is really easy!

5.3.1. Non-Exponential Distributions

METHOD ANALYTICAL "DOQ4", which applies to separable models, considers only the mean value of a distribution. A coefficient of variation different from 1.0 can not be treated. Moreover this holds for general probabilistic distributions, too.

Nevertheless the so-called Coxian distributions can be used in case of the schedule disciplines *immediate* and *lcfspr*! This is not a contradiction to the statement above! It is known from queueing network theory that the resulting performance values are not affected at all by the coefficient of variation. This phenomenon is sometimes called robustness property of separable networks. Consequently you should not try to investigate the sensitivity of separable networks with respect to the coefficient of variation!

5.3.2. General State Dependent Service Speeds

Service speeds depending on the "service mix" at a component are not admitted in complete generality. The most important case for the application of mix-dependent speeds feasible in separable networks concerns the inclusion of aggregated component types.

5.3.3. Multiple Resource Holding

A process can hold more than one resource at a time. The most important examples are passive resources. They do not have time durations associated with them, but they limit the population of jobs that may utilize other devices. Examples of passive resources are main memory or a bus. If multiple resource holding has an essential influence on the performance behavior of the model under study, you should switch to numerical or simulative evaluation.

5.3.4. Blocking and Losses

A device (a component) may be blocked, i.e., prevented from executing processes, when a queue or a buffer elsewhere in the model has reached its full capacity and cannot accept any more tasks. In communication systems a packet attempting to enter a filled buffer may be lost.

Neither blocking nor losses can be treated by separable models. Of course METHOD ANALYTICAL "DOQ4" can be used in case of low blocking probabilities and low loss probabilities, respectively.

5.3.5. Synchronization

The inclusion of synchronization features is not possible in separable models. In particular *semaphores* or *tokenpools* cannot be used within separable models. We refer to the class of Markov models described in later chapters.

Part III

SUBSET FOR MARKOV MODELS

Chapters

6 - 7

6. Introduction to Numerical Evaluation

6.1. Overview

The objective of this chapter is:

- to introduce basic concepts of numerical evaluation and
- to give some hints for the use of METHOD ANALYTICAL "NUMERICAL"

6.2. Basic Concepts of Markov Models

Numerical evaluation in performance modelling is a valuable supplement or even an alternative to other evaluation methods.

By numerical evaluation of a computing system model we mean the computation of the stationary probability distribution of a Markov model by numerically solving the set of so called global balance equations. The coefficients of this equation system are represented in a transition rate matrix. Each entry in this matrix accounts for a transition from one model state to another.

Of course you as HIT user have nothing to do with the complicated process of setting up a large matrix and the subsequent solution of an equation system. As in other evaluation methods HIT automatically transforms models, which are written in HI-SLANG, into executable form and finally provides the desired performance measures.

But you should keep in mind that every single state of your model and every possible transition between the states will be explicitely treated in HIT. Consequently some circumspection is in place.

6.3. Hints and Warnings

Apart from the evident disadvantage of a threatening state space explosion, Markov chains based on numerical analysis have some attractive features. Numerical evaluation allows modelling of features which are not part of the separable models. Evaluation problems are less severe compared to simulation and in many cases the application of the numerical method will be less expensive than simulation.

6.3.1. On Aggregation

To apply the numerical method successfully, it is important to concentrate the modelling efforts on the essential features of your problem. Try to focus on a specific part of your model and aggregate the rest. Note that, e.g., synchronisation features are not part of the world of separable models. In HIT these features can be treated within the class of Markov models, but it would be inefficient or even impossible to evaluate an overall model including all details of the system under study. Consequently all model parts which are not of immediate interest in the given context should be compressed (i.e., aggregated!) as far as possible. Note that the HIT system supports hierarchical analysis by means of automatic aggregation of component types which do not violate the restrictions of separable models. Of course aggregated component types can also be included in models to be evaluated by numerical or simulative methods.

6.3.2. On State Space Explosion

It is well known that even harmlessly looking models can exhibit an enormous number of states resulting in a nearly or completely unsolvable model. You should try to approach the desired level of detail very carefully. A good advice is to restrict the number of processes which can be simultaneously in the system. First restrict your model to one per process type. Probably your model can be evaluated very quickly then. Now increase the process population and observe the behaviour of your model.

6.3.3. Trace Your Models

You should inspect the analyzer listing in case of numerical evaluation to get information about the state space, the matrix size and the cpu time used. In particular you should start with small models, followed by a controlled increase of the process population.

6.3.4. Functional Analysis

The construction of the full state space and of all the transitions between states has the advantage that properties concerning functional aspects can be discovered during model evaluation. For example, the existence of deadlocks can often be recognized. If a deadlock corresponds to an absorbing state, it will even be discovered automatically.

6.3.5. Open Chains

The population of open chains always has to be limited. Add a LIMIT part within your CREATE...EVERY statements, if you come from another solution method:

```
CREATE 1 PROCESS service_name LIMIT n EVERY negexp (arrival_rate);
```

7. HI-SLANG Constructs for Markov Models

7.1. Overview

In addition to the HI-SLANG subset for separable models HIT offers the class of Markov models, which can be solved numerically. They include the following features:

- preemptive and non-preemptive priority scheduling with the help of a standard component type *prioserver;*
- fault tolerant servers which can operate in different degraded modes;
- restricted capacity of *servers;*
- non-exponential distribution functions (Coxian distributions) and
- synchronization with the help of *counters* (including semaphores)

Note that all constructs already introduced also apply for Markov models.

7.2. How to Specify Numerical Evaluation

The solution of Markov models is accomplished by numerical techniques, precisely called analytic-numerical techniques (in contrast to analytical-separable for separable models). Therefore the specification of this method is given by

```
EXPERIMENT experiment_name METHOD ANALYTICAL "NUMERICAL";
```

There are alias names which can be used instead of "numerical", e.g., "markov", see Section 1.3. The execution of the numerical solver can be controlled by, e.g.,

```
CONTROL STOP ACCURACY 1.0 [OR CPUTIME 1000];
```

The option ACCURACY specifies the desired accuracy in percent. Its default value is 1.0. CPUTIME can be used to stop the iterative solution procedure independently from the reached accuracy. If a non-iterative algorithm has been selected by HIT, the CPUTIME stop condition will be ignored.

7.3. Scheduling Disciplines

Markov models admit *schedule* procedures which cannot be treated by separable models. Apart from immediate scheduling (specified by LET schedule:=*immediate*) Markov models admit the disciplines *random*, *prioprep* and *prionp*.

Nevertheless a special standard component type called *prioserver* (an abbreviation of priority server) must be employed if *prioprep* or *prionp* is choosen as scheduling discipline. In case of *random* scheduling, the well-known component type *server* can be used, but *prioserver* is also admitted.

Note that priority scheduling can also be treated approximately with the DOQ4 algorithm, see Section 5.2.

7.3.1. Priority Scheduling

The analytic-numerical technique permits the same type of priority scheduling as described for DOQ4. For more information see Section 5.2.3.

7.3.2. Random Scheduling

Random scheduling plays an important role in modelling, because it is very appropriate for the approximation of *fcfs*. Moreover, in Markov models *random* is used as a tie-break in case of equal priorities.

Applying a *random* discipline means that service requests waiting for execution are selected in a random fashion. For example, if the number of service_1 and service_2 is n_1 and n_2, respectively, scheduling of service_i, i=1,2, will be done with probability $n_i/(n_1+n_2)$.

Random scheduling is, e.g., specified by

```
COMPONENT cpu : server (LET schedule := random);
```

Here instead of *server* the use of a *prioserver* is also possible.

7.4. Servers with Restricted Capacity

The storage capacity of *servers* (and *prioservers*) can be restricted by the *accept* procedure *restrict(n)*, i.e., such a *server* accepts at most n service calls. A rejected task has to stay in its actual status and repeat its last received service phase. This concept of restricted capacity can be used for the modelling of blocking phenomena.

In HI-SLANG this construct is given in connection with a component declaration by:

```
COMPONENT unit : server (LET accept := restrict (5));
```

Note that the *accept* procedure *restrict* cannot be used for simulative evaluations.

7.5. Distribution Functions

For the modelling of time durations like processing time the following probability distribution functions are admitted.

7.5.1. Coxian Distributions

The Cox function is a distribution with an adjustable service time variability. The second parameter of *cox(r,v)* is the so called coefficient of variation v, defined as v:=(standard deviation)/(mean). The case v>1 yields an hyper-exponential distribution (with two exponential phases) whereas in the case of v<1 we obtain a hypo-exponential distribution with two or more(!) exponential phases. We recommend to use coefficient of variations v≥0.5, because for smaller v the number of phases becomes very high and as a consequence the size of the state space can grow to an intractible order. (Indeed in HIT there is a built-in restriction to 10 phases yielding a coefficient of variation of 0.32!)

Note, that the first parameter r is the rate of the distribution, such that r=1/mean.

7.5.2. General Coxian Distributions

Alternatively to *cox (r,v)* we can specify the phases of a Cox distribution explicitly by the *coxg* function (the letter 'g' stands for general). In this case we have to give the rate and the probability to enter the next exponential phase for each phase.

The HI-SLANG notion of *coxg* uses a two-dimensional array; the first row for the rates, the second row for the probabilities. A service call to a standard *server* requesting a processing time according to a coxian distribution with three phases is denoted as follows:

```
compute (coxg ([[0.5, 0.7, 1.3], [0.7, 1/6, 0.0]])).
```

Note that those many brackets are due to the HI-SLANG syntax. The array parameter is specified directly as an array aggregate. The last value of this aggregate must be 0.0, since this is the probability to enter the next phase.

7.5.3. Other Distributions

Of course *negexp(r)* is also admitted, where r is the rate of the distribution. For more information see Section 2.5.4.

Erlang distributions can be introduced via *coxg* functions. For example, the sequential passing through exactly n phases can be achieved by

```
coxg ([[r,r,...,r,r],[1.0,1.0,...,1.0,0.0]]).
```

Deterministic distributions (having a constant value) are of course not possible in the context of analytical techniques, in particular the coefficient of variation must be 0. But a distribution with a rather small coefficient of variation, e.g., 0.5, will usually be a good approximation.

7.6. Synchronization Features

7.6.1. The Concept of Counters

Different from standard servers (*server* or *prioserver*) where a process requests time, a counter is a resource where processes can request to change an integer state vector. This state vector, say [X(1),...,X(n)] is an array owned by the counter. It can be changed according to

```
[X(1),...,X(n)] := [X(1),...,X(n)] + [C(1),...,C(n)]
```

where [C(1),...,C(n)] is an integer array given as an actual parameter of a service request.

The feasible range of the state vector as well as its initial value must be specified for each counter. If a service request cannot be satisfied because the desired change would move the state vector out of range, the requesting service has to wait until another request changes the state vector to a suitable value. Which of several waiting requests, possibly of different types, will be handled first is determined according to a priority discipline or a random discipline. The fulfillment of a request, if possible, happens without any delay. Typical applications of counters are semaphores or memory management schemes.

7.6.2. The Component Type Counter

If you want to use counters in your HIT model you have to introduce the component type *counter* by %COPY "COUNTER". The parameters and the provided service are given in the following type declaration. Of course in HIT this type declaration is not visible for you.

```
TYPE counter COMPONENT (min,max,init : ARRAY OF INTEGER);
   PROVIDE
      SERVICE change (amount  :  ARRAY OF INTEGER;
                      prio    :  INTEGER DEFAULT 32767);

   END PROVIDE;
   …
END TYPE counter;
```

The arrays *min* and *max* give the minimum and maximum values for the state vector such that $min(i) \le X(i) \le max(i)$ for each element X(i) of the state vector. The array *init* specifies the initial value for the state vector.

The parameters of the provided service *change* specify

- the desired amount of change and
- the priority, in case the priority scheduling discipline is used.

Beside the priority scheduling discipline *cprio* the random scheduling discipline *crandom* is also available. In case of *crandom*, all changes of the state variables must have the same absolute value. If you use *cprio* we recommend to choose different values for the priority parameters, otherwise different priorities are assigned automatically by HIT. In the following we consider some examples for the use of counters.

7.6.3. Examples for the Use of Counters

7.6.3.1. A Binary Semaphore

A binary semaphore has two values, 0 and 1, and can be altered by P operations or V operations, which try to decrement or increment the semaphore variable. In HIT we can realize a binary semaphore with the help of a *counter*:

```
COMPONENT bin_semaphore : counter
    (LET min    :=  [0], LET max       := [1],
     LET init   :=  [1], LET schedule := crandom);
```

Note: This semaphore implementation has the property that the V operation may be blocking (in the case that more P than V operations are executed), since P and V are both implemented by *change* and are therefore symmetrical.

HIT also offers a component type *semaphor* with a non-blocking V operation, but it may only be used for simulation, see Chapter 11.

7.6.3.2. Memory Constraints

We give a complete example to demonstrate the use of *counters* for the modelling of memory constraints. Note that an aggregated version of the component type *cs* has been used (see Chapter 3.). In particular for Markov models the HIT features for submodel aggregation should be employed whenever possible. The model's state space is reduced substantially.

If you inspect the solver information written to the listing, you will get some information on the solution process of this example. E.g., the model has 109 states and the direct (non-iterative) numerical solution method has been applied.

Note that the type declaration of component *cs* is not a part of your source text. The aggregated central server system referred by *cs* is included by a %COPY command.

```
%COPY "CSAGG"
%COPY "COUNTER"

TYPE memory_constraint MODEL (m1, m2, no_of_partitions : INTEGER);

    TYPE class1 SERVICE;
        USE SERVICE
            think      (much      :   REAL);
            mem_alloc  (partitions:   ARRAY OF INTEGER;
                        prio       :   INTEGER DEFAULT 32767);
            mem_relea  (partitions:   ARRAY OF INTEGER;
                        prio       :   INTEGER DEFAULT 32767);
            work;
        END USE;
    BEGIN
        LOOP
            think (negexp (1/5000));
            mem_alloc ([+1]);
            work;
            mem_relea ([-1]);
        END LOOP;
    END TYPE class1;
```

```
    TYPE class2 SERVICE;
        USE SERVICE
            think      (much      :   REAL);
            mem_alloc  (partitions:   ARRAY OF INTEGER;
                        prio       :   INTEGER DEFAULT 32767);
            mem_relea  (partitions:   ARRAY OF INTEGER;
                        prio       :   INTEGER DEFAULT 32767);
            work;
        END USE;
        BEGIN
            LOOP
                think (negexp (1/10000));
                mem_alloc ([+1]);
                work;
                mem_relea ([-1]);
            END LOOP;
        END TYPE class2;


    COMPONENT
        term         :  server;
        central_part :  cs;
        memory       :  counter (  LET max  :=  [no_of_partitions],
                                   LET min      :=  [0],
                                   LET init     :=  [0],
                                   LET schedule :=  crandom);


        REFER class1, class2 TO term, central_part, memory
        EQUATING
            class1.think          WITH  term.request;
            class1.work           WITH  central_part.class1_processing;
            class1.mem_alloc      WITH  memory.change
            class1.mem_relea      WITH  memory.change;
            class2.think          WITH  term.request;
            class2.work           WITH  central_part.class2_processing;
            class2.mem_alloc      WITH  memory.change;
            class2.mem_relea      WITH  memory.change;
        END REFER;

    BEGIN
        CREATE m1 PROCESS class1;
        CREATE m2 PROCESS class2;

    END TYPE memory_constraint;


    EXPERIMENT analysis METHOD ANALYTICAL "numerical";
    BEGIN
            EVALUATE MODEL mod1 : memory_constraint (20,2,6);
            EVALUATIONOBJECT
                memory_queue      VIA  mod1.memory,
                terminals         VIA  mod1.term,
                central_server    VIA  mod1.central_part;

            BEGIN
                MEASURE THROUGHPUT, POPULATION    AT memory_queue;
                MEASURE TURNAROUNDTIME            AT terminals;
                MEASURE THROUGHPUT, POPULATION    AT central_server;

                CONTROL TRACEALL STOP CPUTIME 200 OR ACCURACY 0.5;
            END EVALUATE;
    END EXPERIMENT analysis;
```

7.7. Fault Tolerant Servers

HIT provides a component type *ftserver*, which can be used for reliability oriented analyses. A fault tolerant server can be viewed as a homogeneous multiprocessor which is able to operate in different degraded modes, d=0,1,...,*degmax*. *Degmax* denotes the maximum degradation; d=0 is the fault free situation where all processors are operative. Failed processors will be repaired, if one of the repair units is available. The number of repair units is given by the parameter *repair_units*.

A *ftserver* changes its degraded mode according to failure and repair events, occuring with rate *failure_rate* and *repair_rate*, respectively. A dormancy factor determines the failure rate of idle processors: in case of *dormancy*=0.0 an idle processor can not break down. In case of *dormancy*=1.0, an idle processor has the same failure rate as a busy processor. Otherwise the failure rate of an idle processor is given as the product *dormancy*∗*failure_rate*.

The component type declaration is stored in the HIT standard modelling base. Nevertheless we show the interface of the type declaration.

```
TYPE ftserver COMPONENT
        (processors  :   INTEGER;
        degmax       :   INTEGER DEFAULT 1;
        repair_units :   INTEGER DEFAULT 1;
        failure_rate :   REAL;
        repair_rate  :   REAL;
        dormancy     :   REAL DEFAULT 1.0);

    PROVIDE
        SERVICE request (amount  :   REAL;
                         prio    :   INTEGER DEFAULT 32767);
    END PROVIDE;
        …
END TYPE ftserver;
```

The admitted scheduling rules are *random*, *prionp* and *prioprep*. Admitted dispatching disciplines are *equal* and *sdequal*.

As an example consider the following declaration:

```
%COPY "FTSERVER"
    …

COMPONENT triple_processor : ftserver (3, 1, 1, 1E-5, 0.005);
```

This declaration introduces an elementary component with three processors (i.e., three processors can be simultaneously active), at most one processor can break down, there is one repair unit, the failure rate is 0.00001 and the repair rate is 0.005. The dormancy factor has its default value of 1.0.

Please note that the component type *ftserver* cannot be used in simulative models.

Part IV

FEATURES FOR SIMULATIVE MODELS

Chapters

8 - 10

8. On Simulative Evaluation

8.1. Overview

The objective of this chapter is:

- to discuss the inherent problems of simulation techniques;
- to introduce the additional estimators and streams for simulation and
- to discuss the additional features for writing experiments.

Note that all concepts already introduced also apply for simulations, with the exception of component type *ftserver* and the accept procedure *restrict.*

8.2. Inherent Problems in Simulative Evaluations

Simulation is essentially a technique that involves setting up a model of a real or imagined situation and then performing experiments on the model.

To simulate a model is:

- to use a program which behaves like the model;
- to observe the behaviour of this program and
- to measure the performance values of interest.

One of the inherent problems in simulation concerns the measurement of performance values. In particular, it is difficult to estimate the statistical variability and accuracy of simulation results.

In HIT these problems are reduced, but they do not disappear completely. You are still responsible for the control of the simulation, i.e., duration and accuracy of a simulative evaluation are determined by user-supplied control parameters. If you want to obtain a so-called confidence interval (m-w, m+w), where *m* denotes the point estimate mean of the considered performance measure, you have to specify conditions which determine the resulting confidence interval implicitly or explicitly.

Note - and this is very important - that all estimators of performance measures are computed under the assumption that the model would reach a steady-state if the simulation was run during an infinite interval of time. Because we run our simulations finite periods of time, we get approximate estimates, but we can make the error "small" if the simulation run is "long" enough.

In order to sketch some aspects of simulation control, we consider a trajectory of a component's population over time t.

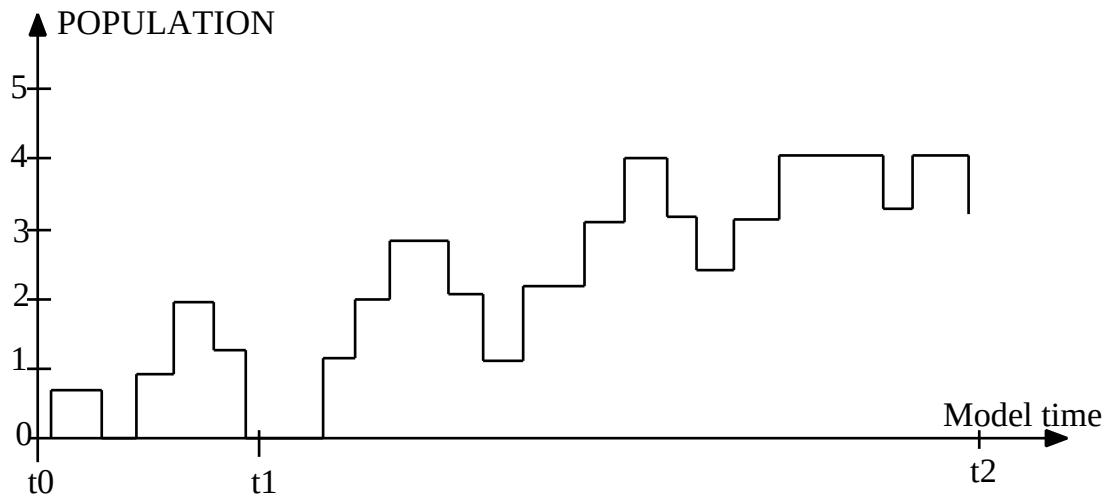

Figure 8.1: A Simple Trajectory

If we wish to obtain a "reliable" estimate of the performance measure POPULATION, there arise some requirements for the simulative method.

- The standard estimator "mean value of population" does not reflect the statistical nature of simulation as displayed in the trajectory. We need estimators, which quantify the goodness of approximation (e.g., confidence level) or which do quantify the variability of a performance measure (e.g., standard deviation). In the next section, we will see how the estimators STANDARDDEVIATION, CONFIDENCE LEVEL and FREQUENCY INTERVAL are used in HIT.

- There must be means to control the durations of the measurements as well as the total length of the simulation. In case of the trajectory given above, the interval *(t0,t1)* seems very inappropriate for measurement purposes and should therefore be neglected. Ideally, the measurement should not start before t1 is reached. The interval *(t0,t1)* is called the "transient phase" of simulation. Unfortunately there is no simple way to determine *t1* in advance.

- An "a priori" determination of a stopping point *t2* is another problem. Of course the simulation can be stopped if a certain amount of model time or cpu time has been spent, but the accuracy of the results, e.g., given by the width of a 95% confidence interval, can be unsatisfactory if the stop condition was too restrictive. For this reason, in HIT the simulation can be controlled by explicitly specifying the desired (relative) accuracy for the performance measures of interest.

8.3. Extensions for Simulation

Being acquainted with the problems of simulative evaluation, we show how they can be handled in HIT. Besides the introduction of estimators for METHOD SIMULATIVE, our main interest will actually be the CONTROL statement, which specifies the global STOP conditions as well as TRACE options for the simulation. We will also conduct different experiments, display some results and discuss their accuracy.

8.3.1. Estimators

We start with a definition of the estimators (addressed by the keyword ESTIMATOR) admitted for METHOD SIMULATIVE:

- **MEAN**

Mean value of the considered performance measure. Note, that in case of simulation we only have an estimation (i.e., an approximation!) of the true mean value.

- **STANDARDDEVIATION**

The standard deviation (square root of the variance) quantifies the variability of a performance measure. A high value for standard deviation indicates, that the observed values are "rather dispersed". On the other hand a standard deviation of (nearly) zero indicates, that the observed values are (nearly) constant.

- **CONFIDENCE LEVEL p**

As a result you obtain a confidence interval including the true mean value with the chosen probability *p*. The probability *p*, expressed in percent, is an integer expression and ranges from 90 to 99. The width of the resulting confidence interval indicates the accuracy of the estimated mean. The width of the confidence interval depends on the chosen confidence level *p*; the higher the confidence level *p* is, the larger the width is. Note, that the true value lies outside the interval with probability (100-p) percent. Also note that the confidence interval is also an estimation!

- **FREQUENCY INTERVAL [interval_list]**

The total number of observations made upon a particular performance measure is grouped into classes according to the specified intervals. Note that the performance measure TURNAROUNDTIME is the only standard performance measure allowed in combination with frequency interval.

8.3.2. Streams

The experiment specification block should precisely describe, which performance indices are to be determined. Simulation result output is initially generated in the form of various data streams. Each stream represents a sequential sample (time series, trajectory) of a particular performance index. Each performance index, i.e., each stream, must be explicitly requested. Streams are bound to components. Each component or the model itself can control one or more of such streams by sequentially generating them.

8.3.2.1. Types of Streams

Depending on the eventual statistical evaluation mode of a sample, the corresponding stream has to be classified as belonging to one of three different types:

- **EVENT** : Event streams store values serially.
  Example: TURNAROUNDTIME.
- **STATE** : State streams comprise the time progress of piecewise constant state variables.
  Examples: POPULATION, OCCUPATION, UTILIZATION
- **COUNT** : Count streams determine rates.
  Examples: THROUGHPUT, SCHEDULE_RATE, PREEMPT_RATE

8.3.2.2.More Predefined Streams

Note that the standard performance indices explained in Section 1.4.1. (THROUGHPUT, TURNAROUNDTIME, POPULATION and UTILIZATION) are predefined standard streams. For simulation there are three further streams available:

- **OCCUPATION** : The probability that a component is not empty (concerning processes).
- **SCHEDULE_RATE** : The transition rate from the *entry area* to the *service area*
- **PREEMPT_RATE** : The transition rate from the *service area* to the *entry area*

8.3.2.3.User-Defined Streams

You can introduce your own performance indices by defining non-standard streams. Defining a non-standard stream is done as follows:

- Declare a stream of a given type (EVENT, STATE, COUNT).
- Provide for the updating of the stream.

For standard streams all this is done automatically. The declaration of a user-defined stream should be made within a component or model type. It is similar in its syntactical structure to the declaration of a variable:

```
STREAM stream_name : stream_type;
```

For updating the stream within a service of the component, the following statement is used:

```
UPDATE stream_name BY observation_value;
```

Different services of a component may update the same stream. As observation value any numerical expression is allowed. If the stream is of type STATE, the observation value will be interpreted as the difference to the previous value. For EVENT streams the observation value is given by evaluating the numerical expression and for COUNT streams the given value is ignored (each update has the same weight of 1.0).

In the experiment the stream is addressed by its name within MEASURE statements. Of course, the corresponding evaluation object must refer to the component where the stream is declared.

8.3.3. A Simulative Experiment

We now present an example of an experiment block and discuss the results obtained. For simplicity, we use the model of *example1*, thus having to change only the experiment block to be adequate for simulation.

Note that the METHOD statement should be changed from METHOD ANALYTICAL "DOQ4" to METHOD SIMULATIVE.

```
EXPERIMENT experiment_sim METHOD SIMULATIVE;
BEGIN

    EVALUATE MODEL model1: example1(0.15);

        EVALUATIONOBJECT
            cpu VIA model1.cpu,
            disk VIA model1.disk_a;

    BEGIN

        MEASURE TURNAROUNDTIME
            AT cpu
            ESTIMATOR
                FREQUENCY INTERVAL
                [0..0.05,0.05..0.1,0.1..0.2,0.2..0.5,0.5..1,1..50];

        MEASURE   POPULATION, UTILIZATION, THROUGHPUT,
                  TURNAROUNDTIME, OCCUPATION
            AT disk
            ESTIMATOR CONFIDENCE LEVEL 95;

        CONTROL
            AT disk STOP MODELTIME 5000;

    END EVALUATE;

END EXPERIMENT experiment_sim;
```

8.3.4. Results from the Simulation

The result of the experiment listed above (Section 8.3.4) are two tables. The first table shows the total number of observations made on TURNAROUNDTIME at different intervals for evaluation object *cpu*:

ESTI	TURNAROUNDTIME	
Freq	1226 0.0	0.05
Freq	1033 0.05	0.1
Freq	1430 0.1	0.2
Freq	2285 0.2	0.5
Freq	1207 0.5	1.0
Freq	905 1.0	50.0

Dependent on the length of the intervals and the distribution of the values of turnaround time, it will often be the case that the highest frequency occurs in the interval including the mean value.

The second table, displayed below, shows a confidence interval for each performance measure specified in the experiment block. The specification of ESTIMATOR CONFIDENCE LEVEL 95 indicates that the resulting confidence interval will include the true value with probability 0.95. Confidence intervals are denoted by mean ± width. The (relative) width of the interval depends on the length of the simulation. In general, longer simulation runs will generate smaller confidence intervals (of course, there are exceptions from this rule!).

ESTI	POPU-LATION	TURNAROUND-TIME	UTILIZATION	OCCUPATION	THROUGHPUT
Mean	0.050813	0.102944	0.048695	0.048695	0.493600
Stdev	0.229224	0.102348	undefined	0.215230	3.966207
Con 95%	0.050813 +- 13.47%	0.102944 +- 4.24%	0.048695 +- 11.95%	0.048695 +- 13.10%	0.493600 +- 9.76%

What, if we are not satisfied with the width of the confidence interval for POPULATION? Fortunately, HIT offers the possibility of specifying the desired width with the aid of the CONTROL statement.

8.3.5. The CONTROL Statement

We now turn our attention to the (for simulation obligatory) CONTROL statement. Apart from the TRACE option, which will be discussed later, the CONTROL statement specifies STOP conditions for the simulation. The simulation will cease if one of the STOP conditions is satisfied. STOP conditions must always be given in connection with an evaluation object, and are tested whenever processes of that component are being simulated.

The syntax of the CONTROL statement is as follows:

```
CONTROL [TRACEALL]
    AT evaluation_object_1 [STOP stop_condition_1] [TRACE];
    …
    AT evaluation_object_n [STOP stop_condition_n] [TRACE];
```

8.3.5.1. Start and Stop Conditions

The *stop_conditions* are special boolean expressions, which may consist of the following operands:

- **CPUTIME** value {e.g., 500 seconds}
- **MODELTIME** value {e.g., 10000 time units}
- **EVENTS** value {e.g., 1000}
  [DUE TO hierarchy] {e.g., all}
- **CONFIDENCE LEVEL** probability {e.g., 98 (%)}
  WIDTH interval_width {e.g., 5 (%)}
  MEASURE stream {e.g., POPULATION}
  [DUE TO hierarchy] {e.g., all}

These operands can be combined with AND and/or OR operations, denoting that the simulation stops when the specified combined condition is satisfied. Note that no brackets are allowed, but AND has higher precedence than OR.

The stop conditions CPUTIME and MODELTIME are intuitively clear. Via EVENTS n the simulation can be stopped when n processes have left the component denoted by the evaluation object.

For CONFIDENCE LEVEL consider the following alternative CONTROL statement for *experiment_sim*:

```
CONTROL AT disk
    STOP
            CONFIDENCE LEVEL 95  WIDTH 5.0  MEASURE POPULATION
        ORCPUTIME 10000;
```

This CONTROL statement ensures that the simulation will not cease until the true value for POPULATION lies with 0.95 probability within the interval [mean-5%, mean+5%]. This could lead the simulation to run for quite a long time until the STOP condition is satisfied. The uncontrolled consumption of large amounts of computing time can be prevented by specifying a maximal simulation time as done above. The simulation will consume at most 10000 seconds of computing time. Of course, it may be the case that a width of ±5.0% can not be achieved under this restriction. But we are

lucky: The results are displayed in the table below. Compare them with the results above.

ESTI	POPU-LATION	TURNAROUND-TIME	UTILIZATION	OCCUPATION	THROUGHPUT
Mean	0.050118	0.103260	0.047668	0.047668	0.485356
Stdev	0.229588	0.103393	undefined	0.213064	4.323507
Con 95%	0.050118 +-4.69%	0.103260 +-1.71%	0.047668 +-3.82%	0.047668 +-4.43%	0.485356 +- 3.91%

The width of the confidence interval for POPULATION is now less than 5%. The other performance values apparently also gained in accuracy.

Summing up, we see that with the aid of the CONTROL statement we are able to specify the required accuracy of the results as well as the duration of the simulation.

8.3.5.2. The TRACE Option

In fact, the CONTROL statement has another important feature, namely the TRACE option. It is a debugging facility to trace the course of an experiment. There are two possibilities:

- **TRACEALL** : traces the event sequences for all components. All area transitions of processes form events.
- **TRACE** : traces only the event sequence for the associated evaluation object every time an event takes place.

We give some hints for using the TRACE option:

- Normally, if you are (only) tracing a simulation, MODELTIME or EVENTS will be the most appropriate STOP condition.
- Do not use TRACEALL and TRACE in combination!
- The trace information will be automatically written into a trace file. Be cautious! Printing trace files may be a waste of paper! For a detailed explanation of the trace files we refer to the Reference Manual.

8.3.6. Measurement Intervals

To specify measurement time intervals for evaluation objects, HIT offers local START and STOP conditions which could be given in the MEASURE statement as follows:

```
MEASURE          stream_list
AT               evaluation_object
ESTIMATOR        estimator_list
OUTPUT TABLE     link_name
START            start_condition
STOP             stop_condition
```

These START and STOP conditions are local in the sense that they affect only the measurement at the associated evaluation object and have no influence on the duration of the whole simulation.

The START condition helps us to ignore, for example, the transient phase of the simulation. The STOP condition indicates that further measurements with respect to the specified evaluation object are of no interest to us.

The operands of the START condition can be either CPUTIME, MODELTIME or EVENTS while the operands of the STOP condition can be either MODELTIME, EVENTS, CONFIDENCE LEVEL or even CPUTIME.

The OUTPUT option may follow the estimator. It serves to specify a link name for the results. In the control file this link name can be bound to a file which can be named explicitely. Moreover you can choose between a formatted table and a dump by substituting the keyword TABLE above by the keyword DUMPFILE. A dump file can be used for further processing, e.g., for generating graphical output, not discussed here.

Alternatively the ESTIMATOR and/or the specifications following it may also be defined in the EVALUATIONOBJECT declaration as a default for all measurements at that evaluation object.

```
EVALUATIONOBJECT
    evaluation_object-name  VIA  component_identification
    DEFAULT
        ESTIMATOR        estimator_list
        OUTPUT TABLE     link_name
        START            start_condition
        STOP             stop_condition
```

Note that all these default specifications can be overwritten in every MEASURE statement. For a complete description please see the HI-SLANG Reference Manual.

8.4. Hints and Warnings

8.4.1. Wide Range of Parameters

If you cannot solve your problem by separable or Markovian models the execution of simulative experiments is in place. But try to explore the relevant parameter space by simplified analytical models. Perform simulations for a few selected parameter sets only. Be economical in the execution of simulative experiment series!

8.4.2. Hierarchical Models

In case of multi-layered models it may be possible that large differences in the frequencies of high-level and low-level events prevent the determination of performance estimations which are statistically significant.

You should consider techniques like off-line analysis of isolated components or the aggregation of detailed lower layers to simplified components. Note that HIT offers automatic aggregation of separable component types (and some extensions)! Tailor your models in a fashion which is amenable to submodel aggregation techniques!

8.4.3. Length of Simulation Runs

Always use confidence level as estimator for streams of interest, but do not choose too small confidence interval widths for the respective STOP conditions. As a rule of thumb you should know that halving a confidence interval (under the assumption of a fixed confidence level) quadruplicates the length of the simulation!

Additionally you may use %PARM=UPDATES to display the number of updates which have occured on each stream.

8.4.4. Tracing Simulations

If you use CONTROL TRACEALL your tracefile will normally become very voluminous. Better use CONTROL AT evaluation_object TRACE and/or the procedures *trace_off* and *trace_on* to restrict the trace to some components and/or some time intervals of interest.

8.4.5. Influence of the SEED Parameter

It is well known that the results of simulation are influenced by the start value of the pseudo-random number generator. For purposes of validation the default value for the *seed* parameter can be altered as follows:

```
EVALUATE MODEL model1 : examples (0.15, LET seed := 5);
```

where the *seed* should be set to an odd integer! The default value for the *seed* parameter is 13 for each evaluation executed. If you want a continuous sequence of seed values you can write

```
EVALUATE MODEL model1 : examples (0.15, LET seed := last_seed);
```

9. The Model World for Simulation

9.1. Overview

The objective of this chapter is

- to introduce all those HI-SLANG features which can mainly be used
  - for simulations (for specifying model types and component types), or
  - within the body of the experiment block (independent from the solution method).

9.2. Basic HI-SLANG Data Structures and Statements

9.2.1. Simple Data Types

The simple data types in HI-SLANG are:

- INTEGER
- REAL
- BOOLEAN
- CHARACTER
- TEXT
- INFILE
- OUTFILE
- POINTER FOR record_type

REAL will be internally represented as LONG REAL, i.e., with the maximal accuracy available. INFILE and OUTFILE enable sequential data processing similar to PASCAL textfiles. POINTERs and RECORDs are described in the HI-SLANG Reference Manual.

Variables and constants may be declared over these basic types:

```
VARIABLE    count        : INTEGER;
            product      : REAL DEFAULT 1.0;
            indata       : INFILE;

CONSTANT    five         : INTEGER DEFAULT 5;
            PI           : REAL DEFAULT 2*arcsin(1);
            E_POWER_PI   : REAL DEFAULT 2.3140692632E01;
```

Variables have an appropriate implicit DEFAULT value (e.g., INTEGER with 0, REAL with 0.0, BOOLEAN with FALSE).

Most simple-typed expressions possible in HI-SLANG can be compared to those in conventional programming languages. In spite of that the constructs AND THEN and OR ELSE need more consideration.

```
i < 5 AND THEN a[i] < > 0;
b(i,j) OR ELSE b(j,i);
```

In the first expression, the part *a[i] < > 0* will only be executed if *i* is less than 5. In the second expression, *b(j,i)* will only be executed if *b(i,j)* is false.

9.2.2. Structured Data Types

Besides records and pointers not described here HI-SLANG provides arrays and especially dynamic arrays.

9.2.2.1 Arrays

Arrays can be declared over all basic types as well as over the structured types COMPONENT and SERVICE. Constants of type ARRAY are also allowed.

The dimension of an array is given by the number of specified index ranges. An index range is given by *l..u*, where *l* and *u* are the lower and upper bound, respectively.

Even for array variables DEFAULT values can be given, e.g.,

```
VARIABLE    arr  : ARRAY    [0..4] OF CHARACTER
                   DEFAULT  ['a', 'b', 'c', 'd'];
                arr1 : ARRAY    [1..2, 1..3] OF INTEGER
                   DEFAULT  [[2,3,5], [3,7,6]];
```

The array *arr1* will contain the following values:

2	3	5
3	7	6

9.2.2.2. Dynamic Arrays

Dynamically sized arrays are possible: The lower or upper bound of an index range at declaration is not given as a constant but as a variable or even as an expression. See the following example:

```
VARIABLE dyn_arr: ARRAY [b1..b1+2*n] OF INTEGER;
```

The variables appearing in the expression must be declared in an outer block (e.g., as parameters). The index range of *dyn_arr* is determined at execution time of the declaration part and will be fixed afterwards.

Note that in HI-SLANG the dimension and index ranges of an array used as formal parameter are not fixed a priori. They are determined by the actual given field.

The statement part of an outer block including the array *field* may contain the following CASE statement in order to manage arrays of different dimensions:

```
CASE field.dimension
    WHEN 1 : ... field [i] …
    WHEN 2 : ... field [i,j] …
    …
END CASE;
```

Note that the dimension of an array is an attribute of each array and can be adressed via dot notation by *array_name.dimension*. Moreover the array attributes *lower_bounds[i]* and *upper_bounds[i]* exist for $1 \leq i \leq dimension$.

9.2.3. Assignments

We distinguish single assignment statements and multiple assignment statements:

```
var                 :=expression;  {single assignment}
var1, var2, var3    :=expression;  {multiple assignment}
```

In multiple assignments all variables get the same value. In contrast multi-value assignments are possible, where the variables within the list normally get different values:

```
(var1, var2, var3) := function_call;  {the function returns three values}
```

There is a type conversion between INTEGER and REAL. If a REAL expression is assigned to INTEGER variables, the value will be rounded and then converted. If an INTEGER expressions is assigned to a to REAL variable, the value will be converted. There are no other type conversions (e.g., between CHARACTER and TEXT) in HI-SLANG.

All elements of an array can be assigned by a single statement:

```
mat   :=  0;
mat1  :=  mat2;
mat   :=  [[0,0,0],[0,0,0]];
```

This is also possible in the DEFAULT part. The first assignment results in setting all elements of the INTEGER (or REAL) array *mat* to zero. The latter assignment is equivalent to the first assignment. By the second assignment, the following conditions must be met:

- *mat1* and *mat2* must be type consistent.
- *mat1* and *mat2* must have the same dimension as well as the same index range within each dimension.

9.3. Handling of Files and Texts

From the performance modelling viewpoint the contents of this section is of minor interest. But for sake of completeness we give a short overview on file and text handling. We suggest to skip this section and use it as reference material.

The assignment statement is also defined for files:

```
f1 := f2;
```

Here *f1* and *f2* must both be either INFILE or OUTFILE variables. The corresponding file will not be copied but one can address the file by *f1* as well as by *f2*.

For TEXT variables assignments may look like this:

```
t       :="This is a text";
t1,t2   :="";
t1      :=t2;
t1      :=t1 & "longer"
```

The operation "&" concatenates the text, "" denotes the empty text. Texts can be lexicographically compared by means of the following operations:

=, <, >, <=, >=, < >, # (both <> and # stand for unequal)

HI-SLANG I/O statements are similar to those of the programming language PASCAL. The following statements are available:

- OPEN, CLOSE
- READLN, WRITELN
- READ, WRITE

The first four statements are only defined for INFILEs and OUTFILEs, while READ and WRITE are also applicable for TEXTs. The LN suffixes only initiate a line feed.

9.3.1. OPEN and CLOSE

Before a file can be accessed it has to be opened. By this the INFILE or OUTFILE variable is connected to an external file via a link name. All file accesses are performed via a buffer of the specified length:

```
OPEN f, "link_name" LENGTH 80;
```

After last access to the file it has to be closed by simply writing

```
CLOSE f;
```

9.3.2. WRITE Statement

The WRITE statement serves to print a series of values to an external file, formatted according to their type (INTEGER, REAL, TEXT, CHARACTER, BOOLEAN). For example

```
WRITE "a(",  n,  ',',  m,  ") :",  a(n,m);
```

displays "a(1, 2) : 16.475" if the variables *n* and *m* and the real array element *a(1, 2)* have these values.

Instead of a file (the default file is SYSOUT) also a TEXT variable can be used:

```
VARIABLE      t1, t2, t3 :  TEXT;
              t4         :  TEXT DEFAULT "example";
CONSTANT      t5         :  TEXT DEFAULT "!";

t2  :=  "This ";
t3  :=  "is " & "an ";
WRITE TEXT t1, t2, t3, t4, t5;
```

After the execution of WRITE statement the TEXT *t1*, which will be newly generated, has the following contents:

```
"This is an example!"
```

To show the formatting of the output, we give some examples.

- BOOLEAN values are printed as 0 and 1; for CHARACTERs their value is printed:

```
WRITE FILE      outf,   TRUE;       ==>1
WRITE FILE      outf,   FALSE;      ==>0
WRITE TEXT      t,      '_';        ==>_
```

- In the case of TEXT, INTEGER and REAL field declarations for output formatting can be used. Otherwise a default field width is used.

```
"OK"              :: 5    ==>    'OK    '            {the field width is five}
"RESULT="         :: 6    ==>    'RESULT'           {better use field width seven or more}
  256  :: 3               ==>    '256'              {exact fit}
  256                     ==>    '          256'    {default width is eleven}
 17.5665 :: 5             ==>    ' 1.7566E+01'      {floating point representation}
 17.5665 :: 3 :: 10       ==>    '     17.566'      {fixed point representation}
 17.5665                  ==>    '1.756650E+01'     {default seven significant digits}
```

9.3.3. READ Statement

The READ statement assigns an input value to a variable according to the types of the given variables. The following types are possible: TEXT, INTEGER, REAL, BOOLEAN and CHARACTER.

Again instead of a file (default SYSIN) a TEXT variable can be used. Consider for example the following READ TEXT statement which is to be executed after the WRITE TEXT statement above. The text *t1* is splitted into *t2* and *t3*:

```
READ TEXT t1,  t2, t3 :: 6;
```

Because the default length of a text is 1, the variables will have the following values, if *t1* has the value assigned by the previous example:

```
t2 : "T"
t3 : "his is".
```

9.3.4. Eof, Lastitem and Eoln

There are three functions to control the reading of files:

- **eof** (f) returns TRUE if no more characters (inclusive blanks!) are encountered in the INFILE *f*.
- **lastitem** (f) will return TRUE, if only blanks are encountered in the rest of INFILE or if the end of the file is reached.
- **eoln** (f) will return TRUE if no more characters (inclusive blank!) in the actually accessed line of the INFILE can be read, i.e., the end of the line is reached.

Note that in these functions the INFILE parameter *f* has the standard input file SYSIN as a default. The following example illustrates how to read all records of a file named *my_file* referenced by the link name DATA:

```
VARIABLE my_file : INFILE;
...
OPEN my_file "DATA" LENGTH 80;
READLN FILE my_file;

WHILE NOT eof (my_file) LOOP
    {reading and processing the records, e.g., by means of READ}
END LOOP;

CLOSE my_file;
```

Note that this is not a perfect example for accessing a file. If only blanks follow the last item read and no next line is in the file, then *eof* will yield false and the next call of READ (for a numerical item) will constitute an error. It is better to use *lastitem* instead of *eof* in this case. *Lastitem* skips blanks. If DATA is bound to a non-existing file a run time error will occur at the OPEN statement.

Moreover don´t use READLN with a list of variables if you are not sure weather the end of the file has already been reached. A previous *eof* query does not suffice, since READLN first skips to the next record and then reads the variables.

9.4. More Control Statements

Compared to METHOD ANALYTICAL more constructs are permitted for METHOD SIMULATIVE, e.g., the CASE statement, the FOR loop and the CONCURRENT statement.

9.4.1. The CASE Statement

The CASE statement is used to select which sequence of statements should be executed next, depending on the value of the expression following the keyword CASE.

```
CASE expression
    WHEN exp1    : {statements}
    WHEN exp2    : {statements}

    WHEN expn    : {statements}
    ELSE         : {statements}
END CASE;
```

The expressions (choices) following WHEN must be of type INTEGER, CHARACTER or TEXT. Lists of expressions separated by commas are also possible. The usual strict type rules apply and the choices must have the same type as the expression following the keyword CASE. The ELSE clause is optional.

As an example consider the following definition of cpu requests depending on the kind of access:

```
CASE access      {access is a CHARACTER variable}

    WHEN 'r' :   cpu_request    ( 5.0); {read}
    WHEN 'w'     :    cpu_request    (10.0);{write}
    WHEN 'u':    cpu_request    (10.0);{update}
    WHEN 'd' :   cpu_request    ( 2.0); {delete}

    ELSE     :   cpu_request    ( 0.5);
                     WRITELN "illegal access ", access, " at model time ", TIME :: 3 :: 10;
END CASE;
```

If the value of *access* is different from 'r', 'w', 'u' or 'd', the service request will be followed by a message. Try to guess the format of the message specified by the WRITELN statement!

9.4.2. The FOR Loop

The FOR loop deals with cases where we go round a loop a certain number of times.

```
FOR var := exp1 STEP exp2 UNTIL exp3 LOOP
    {statements};
END LOOP;
```

exp1, *exp2* and *exp3* must be INTEGER or REAL expressions and *var* must be an INTEGER or REAL variable. Please note, that STEP *exp2* cannot be omitted. Another form of the FOR loop admits a list of expressions. The loop is executed for every member in the list.

```
FOR var := exp1, exp2, ..., expn LOOP
    {statements};
END LOOP;
```

Besides the use of the FOR loop in the specification of model types it can be used in the experiment specification. Note that this possibility is independent from the evaluation method: for specifying experiments all HI-SLANG features can be used independant of the solution method used.

A typical application for this kind of loop is the execution of experiment series. Actual values for the parameterization of models can be assigned in this way:

```
EXPERIMENT model_analysis METHOD SIMULATIVE;

    VARIABLE speed : REAL;
BEGIN
    FOR speed := 1.0, 2.5, 5, 20, 100
    LOOP

        EVALUATE MODEL  mod : mt (speed);
        …
        END EVALUATE;

    END LOOP;
END EXPERIMENT;
```

9.4.3. The CONCURRENT Statement

The CONCURRENT statement is used for the modelling of parallelism. It can be used in services only. An example for the CONCURRENT statement is:

```
CONCURRENT

    proc1_computing (amount1);
TO
    proc2_computing (amount2);
TO
    proc1_computing (amount3);
    proc2_computing (amount4);

END CONCURRENT;
```

The three parts separated by the keyword TO will be executed in parallel (concerning model time). The statement is finished when all of its branches have terminated.

9.5. More on Services

The syntactical structure of services admitted for METHOD ANALYTICAL has already been given. But a few points must be taken into account:

- The only restriction imposed on the parameters of a service is, that call by name is not admitted.
- Declaration of local variables is possible.
- In the body of a service, other statements apart from control statements and service calls can be used. Even CREATE statements may occur here.

9.5.1. The CREATE Statement

The CREATE statement is responsible for generating processes which execute service descriptions dynamically during run time. It can be used in the following four forms:

- CREATE n PROCESS service_name (actual parameters) AT time1;
- CREATE n PROCESS service_name (actual parameters) AFTER time2;
- CREATE n PROCESS service_name (actual parameters) EVERY time3;
- CREATE n PROCESS service_name (actual parameters);

The two latter forms are already known from METHOD ANALYTICAL. Some examples for the generation of processes are:

```
CREATE  10  PROCESS batch_task AT 0;
CREATE  1   PROCESS batch_task EVERY negexp (7.5);
CREATE  2   PROCESS job (x, 13.7) AFTER 217.5;
CREATE  1   PROCESS job (,,) EVERY negexp (1/iat);
CREATE  56  PROCESS dialog_task;
```

The first statement generates an initial filling of the model. If you can make a good guess at the mean population of the model, you can shorten the transient phase of a simulation in this way.

A continous Poisson arrival stream of batch tasks is specified in the second CREATE statement. The third statement creates exactly one process with actual parameters x and 13.7 after 217.5 time units, whereas the following statement creates a Poisson arrival stream of objects having default values. The last statement shows the standard way to generate a fixed number of permanent processes. Note that statements one and three are not allowed for METHOD ANALYTICAL!

The time given in the CREATE statement refers to the model time, not to the CPU time.

9.5.2. The SUBMIT Statement

Another possibility of creating processes dynamically at run time is supported by the SUBMIT statement. When the SUBMIT statement is executed only one process is generated. But this process is named. The name (or names) must be declared by:

```
PROCESS p_name1, p_name2, ...: NAME FOR service_name;
```

or even by a one-dimensional static ARRAY of names:

```
PROCESS p_array_name : ARRAY [1..4] OF NAME FOR service_name;
```

The SUBMIT statement has the following syntax:

```
SUBMIT service_name (actual_parameters) NAME process_name;
```

Additionally the timing specification AT, AFTER and EVERY can be given as in the CREATE statement. For processes like those defined above (which do have a name) service parameters can be accessed by means of the dot notation. In HIT the service parameters are used to model the process state.

In the following example the service parameter, i.e., the state of the process called *p_name* is accessed via dot notation: *p_name.much*.

```
TYPE ct COMPONENT;

    TYPE st SERVICE (much : REAL);
        …
    END TYPE st;

    PROCESS  p_name: NAME FOR st;
    PROCESS  print : state_print;

    TYPE state_print SERVICE;
    BEGIN
        LOOP
            hold (10);
            WRITE time, p_name.much;
        END LOOP
    END TYPE state_print;

BEGIN
    SUBMIT st (17.5) NAME p_name;
END TYPE ct;
```

9.5.3. Static Process Declaration

Processes can also be generated statically by a declaration. It looks quite similar to the declaration of a process name, explained above: A process can be declared and immediately generated by

```
PROCESS p1, p2, ... : service_name (actual_parameters)
```

or even by a one-dimensional static ARRAY of processes:

```
PROCESS p_array : ARRAY [1..4] OF service_name (actual_parameters);
```

9.5.4. Service Arrays

A service may use a number of similar services, called a SERVICE ARRAY. Calling one of the services of a service array is similar to accessing an array element, i.e., by indexing.

Service arrays must be bound to the provided services of a component array. Other ways of binding are not possible. Here is a small example to illustrate the use of a SERVICE ARRAY:

```
TYPE compute SERVICE;
    USE
        SERVICE ARRAY store (...);  {The parameter list is optional}
    END USE;
BEGIN
    …
    store [3] (...);       {The round brackets embrace the actual parameters}
    …
END TYPE compute;

{We assume the existence of a component type ct, which provides a service called file_it}
    …
COMPONENT ca : ARRAY [1...4] OF ct;     {declaration of an array of components of type ct}

REFER compute TO ca EQUATING
    compute.store WITH ca. file_it;
END REFER;
```

As defined by the REFER part, the statement *store [3]* within the service *compute* will cause the execution of *file_it* of the third element of the component array *ca.*

Note that the index of the service array must not exceed the boundaries of the associated component array, otherwise a run time error results. It is often favourable to specify the boundaries of the component array as a parameter of the service, since the attribute *dimension* does not exist for component arrays. Note that the USE declaration of the service array does not contain any bounds (similar to array parameters of procedures).

9.5.5. Services Supplying Results

Services can also supply results! See the following example.

```
TYPE fun_st SERVICE (t: REAL) RESULT REAL;
    USE
        …
    END USE;
BEGIN
    …
    RESULT time - t;  {time is the current model time}
END TYPE fun_st;
```

Just as a procedure with result, results are returned after the creation and execution of a process of service *fun_st*. The call of services should be identical to a function call, otherwise (e.g., in CREATE/SUBMIT statements) the result is lost (although it may sometimes be reasonable to call services and ignore the results).

9.6. Procedures

For simulative models a lot of additional random drawing procedures, predefined procedures and even user-defined procedures are available.

9.6.1. More Random Drawing Procedures

In Section 2.6., the random drawing procedures *negexp* and *draw* have been introduced. Section 7.5. contains the definition of *cox* and *coxg*. In the following we present the most important random drawing procedures, which can be used in simulations only.

- **uniform** (a, b)

  Uniform is a real function with real parameters a and b.

If a ≤ b	the value will be a drawing from a uniform distribution between *a* and *b*, i.e., each value in the interval [a,b] is drawn with the same probability.
If a > b	a run time error will result.

- **erlang** (a, b)

  Erlang is a real function with real parameters a and b.

If a ≥ 0 and b ≥ 0	the value will be a drawing from the Erlang distribution with mean 1/*a* and standard deviation 1/(a∗√b). *b* < 1 results in a small variation, *b*=1 yields the exponential distribution, *b* > 1 results in a large variation.
If a < 0 or b < 0	a run time will error result.

- **normal** (m, s)

  The value given by this function is normally distributed with mean *m* and standard deviation *s*. *Normal* is a real function with real parameters *m* and *s*.

Furthermore, all random drawing procedures known from the host language SIMULA (*discrete, histd, linear, poisson*, *randint*) are available in HIT. See the HI-SLANG Reference Manual, please.

9.6.2. Predefined Procedures

Besides the random drawing procedures there are a lot of other procedures predefined in HI-SLANG:

- The set of arithmetic functions available consists of the standard trigonometric functions (*sin, arccos, tanh, ...*) as well as *abs, sqrt, log* and the like.
- For text and file handling procedures like *digit, letter, eoln* can be used.
- Modelling support procedures are available as, e.g., *time, cpu_time, stop_-evaluation, transfer_results* (intermediate results) and *get_result* (e.g., to control evaluation series depending on earlier results).
- Simulative trace control: Via *trace_state* information about the current location of processes can be added to the trace file (state trace, done automatically when the simulator detects a deadlock). Via *trace_on* and *trace_off* the event trace can temporally be supressed.

For more details see the HI-SLANG Reference Manual.

9.6.3. User-Defined Procedures

Apart from services HIT also provides procedures, whose execution is invoked by a call. A procedure call is a statement, or if it is used like a function it is an expression if exactly one value is returned as result. Multi-valued procedures are a special HIT feature.

If other procedures from lower layers are to be used (called), they must be explicitly imported via a USE declaration part. The HI-SLANG notation of the USE declaration is identical to the USE declaration of services. Note that a procedure cannot use services! Moreover procedures cannot consume model time. Third procedures are not subject to component control, e.g., they cannot be scheduled.

- Procedure with result:

```
PROCEDURE f     (r          :  REAL DEFAULT 0.0;
                 n1, n2     :  INTEGER)
                 RESULT REAL, INTEGER;

{Declaration of local variables, constants and/or procedures}
BEGIN
...
RESULT 27.09, 49;
END PROCEDURE f;
```

This procedure returns a pair (x,n), where x and n are of type REAL and INTEGER, respectively. Here are some possible calls of the procedure *f*, which are completely equivalent.

```
(x, n) := f (0, 8, 15);
(x, n) := f (, 8, 15);
(x, n) := f (, 8, LET n2 := 15);
```

- Procedure with USE part:

```
PROCEDURE proc2 (x:REAL);
    USE PROCEDURE
        proc1 (.....);
        …
    END USE;
BEGIN
    {statements}
    proc1 (.....);
    …
END PROCEDURE proc2;
```

Of course procedures with USE part may also deliver results and vice versa. The following notes applying to all kinds of procedures are advisable:

- Number and type of formal and actual parameters must be compatible.
- Formal parameters without default value must be substituted by an actual parameter.
- Procedures can be called recursively.
- LET parameters (keyword-parameters) must not be followed by other parameters.
- Time consumption, e.g., service calls in procedures is not admitted.

The default parameter transmission mode is "call by value" for all simple types except for POINTER, INFILE and OUTFILE and "call by reference" for all structured types respectively. "call by name" can optionally be used in all cases (but only for parameters of procedures, not for component types and services). As an example consider the following list of formal parameters, where for *n* and *z* the default transmission mechanism has been changed:

```
PROCEDURE   f   (NAME        n  :  INTEGER;
                 VALUE       z  :  ARRAY OF BOOLEAN;
                 REFERENCE   f1 :INFILE);
BEGIN
   …
END PROCEDURE f;
```

9.7. An Extensive Mini Example

To give an impression of HI-SLANG we continue with an example, which shows as many HI-SLANG features as possible on a single page.

```
%ANALYZER
%BIND "FILE"  TO  data.file
%END

VARIABLE   afile          :   INFILE;
CONSTANT file_length      :   INTEGER DEFAULT 80;

TYPE system MODEL (t:REAL DEFAULT 5; n_sim, n_ula:INTEGER);

        TYPE in_out SERVICE (id: TEXT);
            USE SERVICE fetch (t  : REAL);
            END USE;

            VARIABLE i : INTEGER DEFAULT 0;
        BEGIN
            WHILE NOT lastitem(afile)
            LOOP
                READ FILE afile, i;                                  {spends approx. log(i) }
                fetch(normal(1+entier(log(i)),1));                    {sec. to read number i}
                 WRITELN id, " writes", i::8, " at", time::3::10;
            END LOOP;
        END TYPE in_out;

        COMPONENT man : server(LET schedule := immediate);

        REFER in_out TO man EQUATING
            in_out.fetch WITH man.request;
        END REFER;

BEGIN
        CREATE n_sim     PROCESS in_out("SIM");                       {tasks from SIM}
        CREATE n_ula     PROCESS in_out("ULA") EVERY t;               {tasks from ULA}
END TYPE system;


EXPERIMENT analysis METHOD SIMULATIVE;

    VARIABLE s, u : INTEGER;
BEGIN

    FOR s := 0 STEP 10     UNTIL 10 LOOP
        FOR u := 1 , 3, 5 LOOP
            OPEN afile, "FILE" LENGTH file_length;

            EVALUATE
                MODEL i_o : system (, s, LET n_ula := u);
                EVALUATIONOBJECT one_man VIA i_o.man
                    DEFAULT ESTIMATOR MEAN, STANDARDDEVIATION;
            BEGIN
                MEASURE POPULATION, TURNAROUNDTIME AT one_man;
                CONTROL AT one_man STOP CPUTIME 10 OR EVENTS 30;
            END EVALUATE;

            CLOSE afile;
        END LOOP;
    END LOOP;
END EXPERIMENT;
```

The example starts with some percent statements, where the file *data.file* containing some numbers is bound to a link name.

The HI-SLANG source is composed of a model type with name *system* and an experiment specifying an evaluation series of models of that type. The global declarations in the first lines hold for both parts.

The model is simple: There is a man (represented by a component) able to perform the task (service) *in_out* to write the name of his orderer together with integers read from a file until the end of the file is reached. The man needs approximately n seconds (normally distributed with variance 1) to read and write a number with n digits. The connection between *man* and his task is established in the refer part, specifying that the service *fetch* used within *in_out* is to be satisfied by the standard service *request* of the server *man.* The occurence pattern of the tasks is specified by create statements.

The experiment part describes an evaluation series. For each parameter combination (s, u) an evaluation of a model object named *i_o* is performed. Each evaluation is preceeded by opening the file. As a result we obtain a table containing mean value and standard deviation of POPULATION (number of tasks present) and TURNAROUND-TIME (completion time for a task) at the component *man* (adressed by *one_man*). This table is written to a file named by the file name generator (see Appendix A.). The simulation stops after 30 events or if 10 cpu seconds have been spent.

Notice that we presented a flat model, but refining this model can be done by replacing *man* by a more detailed component. On the other hand the model can be transformed to a component providing the service *in_out.* In this way hierarchical models can be built as we have seen in a previous chapter.

10. More Predefined Component Types

In HIT we can store component types in modelling bases or in files and make them available for other HIT users. The component types specially tailored to Markov models (*counter*, *prioserver*, *ftserver*) have been introduced in earlier sections. Of course these types are also admitted for simulative evaluation except *ftserver.*

Now we introduce the rest of the predefined component types which are members of the HIT standard mobase. Please note, that all of them can only be used if you choose simulation.

10.1. Semaphor

An object of type *semaphor* represents a general semaphore. The initial value can be specified via the parameter *sem_init* (>0). The default value of *sem_init* is 1, yielding a binary semaphore.

A semaphore provides the services P and V.

P; If possible, the semaphore variable will be decremented by one, otherwise the requesting process will be passivated.

V; The semaphore variable will be incremented by one and a passivated process may be activated.

Semaphores are known from operating systems to synchronize processes or to protect critical regions. The following example shows how to protect a critical region with the help of a semaphore.

```
TYPE semaphor COMPONENT (sem_init : INTEGER DEFAULT 1);
    PROVIDE SERVICE
        p; v;
    END PROVIDE;
    …
END TYPE semaphor;
```

Note that the default (and only meaningful) *schedule* discipline is *fcfs*-like and that *sem_init* is only the initial value of the semaphore and not an upper bound. By executing only V-operations the semaphor value can infinitely be incremented.

An alternative implementation of a semaphore can be made with the help of the component type *counter*, see the chapters on Markov models. In this case *random* and *priority* scheduling disciplines are possible.

Due to historical reasons a semaphore in HIT indeed spells semaphor, without an "e" at the end!

The following example demonstrates the use of a binary semaphore to protect a critical region within a service:

```
%COPY "SEMAPHOR"
…

COMPONENT  binsem : semaphor (LET sem_init := 1);

TYPE xwrite SERVICE;
    USE SERVICE
        passeer;
        verlaat;    {passeer and verlaat are notions due to Dijkstra}
        …
    END USE;

BEGIN
    …
    passeer;    {if passeer (=p) is successfull, the critical region can be entered}
    …           {critical region, e.g., exclusive file access}

    verlaat;    {verlaat (=v) switches the semaphore variable to its original value}
    …
END TYPE xwrite;

REFER xwrite, ... TO binsem ... EQUATING
    xwrite.passeerWITH binsem.p;
    xwrite.verlaat     WITH binsem.v;
    …
END REFER;
```

10.2. Tokenpool

The component type *tokenpool* models a pool of tokens, which can be allocated, released, destroyed and produced by using the provided services. The following type declaration shows the interface of *tokenpool.*

```
TYPE tokenpool (no_of_tokens : INTEGER) COMPONENT;
    PROVIDE SERVICE
        allocate    (number : INTEGER);
        release     (number : INTEGER);
        destroy     (number : INTEGER);
        produce     (number : INTEGER);
    END PROVIDE;
    …
END TYPE tokenpool;
```

A request to *allocate*, e.g., by *allocate(n)*, demands for a number of tokens and waits until those tokens are allocated. If the number of free tokens is greater than or equal to the number of requested tokens the allocation will happen without delay. Otherwise the requesting process is passivated until the number of free tokens matches. The number of free tokens can be increased by *release*, which frees a number of allocated tokens, or by *produce*, which creates a number of "new" tokens. Free tokens can be removed from the token pool by calling *destroy*.

Please note that *destroy*(n) and *allocate*(n) will result in passivation of the calling process if n exceeds the number of free tokens. Also note that in contrast to the *counter* the limits of available tokens can be manipulated at run time.

One of the most famous applications of a *tokenpool* is the representation of simultaneous resource possession. The simultaneous use of active components, (like cpu and io devices) and passive resources like main storage can be modelled as follows.

```
%COPY "TOKENPOOL"
...

COMPONENT main_storage : tokenpool (LET no_of_tokens := 1024);
```

The default (and only meaningful) scheduling discipline of a *tokenpool* is *fcfs*-like. Also *dispatch* and the other control procedures may not be set. A process can allocate and release n byte of main storage by calling *allocate*(n), and *release*(n), respectively.

For this example calls of *destroy* and *produce* are of minor interest.

10.3. Synchsend

A component object of type *synchsend* enables two processes to communicate with each other in one direction: one process as sender and the other as receiver. If communication takes place in both directions, or if more than two processes exchange messages, then several component objects (or even a component array) must be declared.

The sender and receiver are synchronized when exchanging messages in the sense that some access operations will be delayed by either of them until an appropriate state of the other is reached. *Synchsend* provides the two services *send* and *receive* as given in the type declaration below. The buffer is implemented by a text variable.

```
TYPE synchsend COMPONENT;
    PROVIDE
        SERVICEsend (what : TEXT);
                    receive RESULT TEXT;
    END PROVIDE;
    ....
END TYPE synchsend;
```

The following example demonstrates the use of *synchsend* for a synchronous unidirectional communication between a sender and a receiver process.

```
%COPY "SYNCHSEND"
    ...

COMPONENT commun : synchsend;
```

Two services *sender* and *receiver* (normally belonging to different components) may then communicate over an enclosed *synchsend* component in the following way:

```
TYPE sender SERVICE;

    USE SERVICE
        send(x :TEXT);
    END USE;

    VARIABLE message : TEXT;

BEGIN
    .
    .    {produce message}
    .
    send (message);
    .
    .
    .
END TYPE sender;
```

```
TYPE receiver SERVICE;

    USE SERVICE
        receive RESULT TEXT;
    END USE;

VARIABLE message : TEXT;

    BEGIN
        .
.
        .
    message := receive;
        .
        .    {consume message}
        .
END TYPE receiver;
```

10.4. Nowaitsend

The component type *nowaitsend* enables the communication of processes. In contrary to *synchsend*, the sender, in general, does not have to wait until the receiver receives the messages. It may further produce messages and send them while the buffer is empty. The receiver must obviously wait for the sender in case of an empty buffer.

Note that the buffer is implemented by a TEXT Array. The capacity of the buffer is given by the integer parameter *no_of_buffers* (default=1). The type declaration and application are similar to those of *synchsend*.

```
TYPE nowaitsend COMPONENT (buffer_size : INTEGER);
    PROVIDE
        SERVICEsend (what : TEXT);
                    receive RESULT TEXT;
    END PROVIDE;
    ....
END TYPE nowaitsend;
```

If we want to introduce a component of type *nowaitsend,* e.g., with buffer size 100, we can do this as follows:

```
%COPY "NOWAITSEND"
...

COMPONENT proc_comm: nowaitsend (100);
```

Sending and receiving are done by *send (mess_text1)* and *mess_text2 := receive* respectively. If communication occurs between different components (as usual), one of them (or both) have to enclose the component.

10.5. Observer

The standard component type *observer* can be used to produce intermediate result outputs. The *observer* has no provided services, but internally creates one process which will interactively prompt the user for new time points for the next intermediate results, if the parameter *interactive* is set. An initial observation model time interval can be set by the real parameter *obs_interval*. The interactive *observer* will then produce the results, print the current model time and amount of cpu time used, and query for one of the following alternatives:

q : **q**uit simulation
s : **s**top observing, continue simulation
c : keep **c**urrent model time interval and continue observing
n : as c, but switch to **n**on-interactive mode
<real value n.nnEnn> : set new interval, continue observing

It has the following interface:

```
TYPE observer COMPONENT
    (obs_interval  : REAL;
     interactive   : BOOLEAN DEFAULT FALSE);
    …
END TYPE observer;
```

To use the observer it has to be copied from the standard modelling base. An *observer* component should preferable be declared within the model type or global to the model type.

```
%COPY "OBSERVER"

COMPONENT obs : observer (500, TRUE);
```

Normally the results are directed to a file (the default is OUTPUT TABLE "TABLE") and can in this case not be watched interactively. Thus the *observer* should be used in combination with OUTPUT TABLE "SYSOUT".

Part V

APPENDICES

Appendices

A - F

APPENDIX A. How to Run HIT

In this chapter a brief introduction to the usage of the HIT system in different environments is given.

For every operating system HIT has been ported to there exists an operating system procedure to activate HIT (see the next sections). This procedure calls the HI-SLANG compiler and then the SIMULA compiler and linker to create executable code. After this the compiled and linked module (i.e., the analyzer) will be executed to calculate the desired performance indices. The object manager OMA is always accessible via a similar, operating system procedure.

There are many parameters to control the operation of HIT. All parameters have default values and are therefore optional. For a detailed list please see the corresponding HIT User's Guide. The main parameters which always exist are:

control the name of the control file used by HI-SLANG compiler and analyzer. If you do not use this parameter both the compiler and the analyzer will ask for a control file separately. Suppose you have none, you must enter the file name of your HI-SLANG program then.

task the entry point of the procedure. Possible values are:

- *com*: start with HI-SLANG compilation
- *sim*: start with SIMULA compilation
- *exp*: only perform the experiment (run the analyzer, e.g., with other input data)

option the exit point of the procedure. The value *check* stops HIT after the completion of the HI-SLANG compilation

sizecomp the size of the working storage for the HI-SLANG compiler

sizeexp the size of the working storage for the generated analyzer

If the user does not provide special file bindings in his control file all files generated, e.g., the result files, are named by the HIT file name generator. The file name patterns are given in the next sections. The user may define a different file name pattern by using "%DEFAULT pattern" in his control file. See the Reference Manual.

A.1. Guide for UNIX

For using the HIT system on a workstation (or even a PC) under some UNIX-like operation system, the shellscript *hit* is available. Moreover the graphical interface HITGRAPHIC can be used (on SUN workstations).

HIT may be used by more than one user at the same time. In this case the shellscript *hit* has to be called from different directories because some output files have fixed names.

With *installation-directory* being the name of the directory the HIT system is installed in, starting the HIT system looks like this

*installation-directory/*hit

It is useful to define an alias for this or better to set a path in the .login file by

set path=($path *installation-directory*)

The script may then be called like this:

hit [control [task]]

Control and *task* are positional parameters, while all other parameters are implemented by environment variables. Such parameters can be set typing, e.g.,

env option=check sizecomp=8000 hit *control_file*

Do not use blanks around the '='-characters !

After every run of the HIT system, some new files exist in the current directory. Some files are created by the shellscript *hit* . Their names can only be modified by the parameter *prefix* of hit which has the string "t." as a default value. Normally the following files exist:

t.hitcode	the analyzer to be run
t.hitcode.sim	the generated code of the HI-SLANG compiler
t.compiler	the standard output file of the compiler
t.experiment	the standard output file of the analyzer

More important for the user are the files created by the HIT system itself. Their names may be defined in the control file, or the file name generator of HIT may be used. The latter is automatically used for files having a standard link name which is not bound in the control file. It generates file names t.<c>.<l> where <c> is the name of the control file stripped of the directory prefix and a suffix *.ctl* or *.hit* and <l> are the three leading letters of the standard link name, e.g., *lis* for listing, *tab* for table.

For a control file named *my_dir/example/ex1.ctl*
the standard name of the listing is *t.ex1.lis*
and your results are found in *t.ex1.tab*
within the current directory!

A.2. Guide for BS2000

For using the HIT system in a BS2000 environment the procedure HIT is available. Calling the HIT system looks as follows:

DO $UserId.HIT [, CONTROL = <file name>] [<other parameters>]

UserId stands for the user identification the procedure is situated on your computer system. After a run of the HIT system you find some temporary files created by HIT:

#HIT.CODE.LOAD	the analyzer to be run
#HIT.CODE	the generated code of the HI-SLANG compiler
#HIT.SYSLST.COMPILER	the standard output file of the compiler
#HIT.SYSLST.EXPERIMENT	the standard output file of the analyzer

More important for the user are the files which are by default named by the HIT file name generator. It creates temporary files #<c>.<l>, where <c> is the name of the control file and <l> is the link name. So by default your results will be written to #<control file name>.TABLE.

A.3. Guide for VM/CMS

For using the HIT system in a VM/CMS environment the REXX procedure HIT EXEC is available. Starting the HIT system look as follwos:

HIT FNAME FTYPE [FMODE] [(<other parameters>)]

Here the usual parameter *control* is split into three parameters FNAME, FTYPE and FMODE due to file name conventions. The other parameters may follow.

Special parameters are

SIZE	the memory size for one analyzer run (default 4096K) (SIZE is the *sizeexp* parameter, currently there is no *sizecomp* parameter)	
OUTPUT	PRINT	the output is send to the line printer
	TERM	the output is displayed on the terminal only

After a run of the HIT system by default you find your results in files named by the file name generator. It generates file names HIT <l>, where <l> is the link name. So by default your results will be written to HIT TABLE.

APPENDIX B.Handling of the HIT System

The HIT system is integrated with the so-called HIT File Access Network (HIT-FAN). HIT-FAN supports the development and configuration of HIT models from modules like component types, services, procedures or arbitrary pieces of HI-SLANG code. These modules can be either files or members in a modelling base (called mobase) and are accessed or created via HIT-FAN. Every logical file which is used (or created) during the processing of a HIT model is linked by HIT-FAN to a physical file or to a member of a mobase. Moreover the HI-SLANG compiler is controlled by FAN.

B.1.Some Compiler Control Statements

The following compiler control statements may appear at any place in the HI-SLANG source text. They are used to invoke various listing options of the compiler. Note that all of then start with a '%'-character, which must appear in column 1.

%NOSOURCE

The formatted HI-SLANG listing will be usually written into a file. This option suppresses the HI-SLANG listing until a **%SOURCE** statement is encountered.

%PAGE

A form-feed will be inserted.

%TITLE This is a title

Similar to the PAGE statement, a form-feed will be inserted. Additionally the text written in the control statement will be printed as a title at the head of the page. This title will be preserved on the following pages until it is overwritten by a new title.

% arbitrary comment

Source text lines which contain a "%" in the first column and a blank in the second column are considered to be comments:

```
% This is a comment
%No comment; error!
```

The second example is wrong formatted and will lead to an error! Another way to include comments in your source text is the use of braces {...}. We do use both possibilities. Please note, that such comments must be terminated in the same line.

%COPY "link name"

The file bound to the link name (in the control file) is textually inserted at this position. This facility can be used to access logical units of text (e.g., component types and experiments) from separate files or separate "design objects" within a modelling base.

B.2. The Control/Configuration File

The linking of physical objects to logical objects is defined by control records, which can be placed either at the beginning of a HI-SLANG source file or in a separate control file. Calling HIT you have to supply the name of the file containing the control records. If your control file is incomplete, HIT requires for the resolution of unresolved references by

%BIND "link_name" TO ?

As a consequence you can use HIT even with an empty control part. The control part has a structure as follows:

%COMMON
{control records common to the HI-SLANG compiler and the analyzer}

%COMPILER
{control records for the HI-SLANG compiler}

%ANALYZER
{control records for the analyzer}

%END
{last statement of the control part, now the HI-SLANG source can follow}

All parts are optional. Comments can also be included, starting by '%', followed by at least one blank. There are several control records; the most important are %PARM and %BIND.

B.2.1. %PARM. Compilation and Analyzer Options

The admitted parameters in a %PARM record are either concerned with the compilation of HI-SLANG sources or with the formatting of the HI-SLANG listing or with analyzing models. The parameters are given by

%PARM = parameter [,...]

For a complete list see the HI-SLANG Reference Manual. Most important are the following options:

CHECK

The HI-SLANG source is only checked for syntactical and semantical correctness. No generation of SIMULA code (neither of executable code) is performed.

NOSOURCE

Normally a listing of the HI-SLANG source (including the control part) is generated. NOSOURCE supresses the HI-SLANG listing.

XREF

A cross reference listing is generated and appended to the listing. Test it!

NOWARN

Additional to error messages, HIT normally provides warnings. NOWARN suppresses warnings.

INDENT =[character] number

The HI-SLANG listing is indented (i.e., shifted right) to show the block structure of the program. The *number* determines the number of indented shifts per block level. You can optionally specify a *character* which is used for threading between block-begin and block-end. (We suggest a blank or '|'.)

UPDATES

Additional to the mean value, the number of updates to a stream will be displayed in tables resulting from a simulation. By %PARM=MINMAX even minimal and maximal values which have occured in the observation interval can additionally be displayed within the mean value table fields.

B.2.2. %BIND. Binding and Linking

The %BIND record is used to bind logical link names to physical files or to members of a modelling base. Besides the link names you define, e.g., by %COPY or OPEN statements there are a lot of link names predefined, e.g.,

"TABLE"	for the tabular result output
"LISTING"	for the compiler source listing (including messages and XREF)
"TRACE"	for the Markovian or simulative trace output
"PREANA"	for the aggregation output.

You may bind these link names to define your own file names, disabling the HIT file name generator. Moreover you can alternatively or additionally bind the link names to members in a modelling base. A %BIND statement has the structure

```
%BIND "link name" TO file_object
```

As *file_object* you can specify either the name of a physical file or you can specify a member of a modelling base. We shortly explain the second case by some examples, where we presuppose the existence of a HIT-specific modelling base. See the OMA User's Guide for more informations. The general structure of the %BIND statement in that case is

```
%BIND "link_name" TO mobase_name (parameters)
```

The *mobase_name* specifies the name of your (private) modelling base. The list of parameters contains up to four entries, which specify the object to be included as follows:

1. The representation of the *module* to be included. You can choose between HISLANG, PRECOM, PREANA, CONTROL, SIMULA and DATA. PRECOM is a pre-compiled intermediate representation of the object, PREANA is associated with a pre-analyzed (i.e., aggregated) component type.

2. The *type* of the object, e.g., COMPONENT, PROCEDURE etc.; notice that this information may be omitted.

3. The *name* of the object. This name must begin with a letter followed by letters, digits, dots or underscores. The first 12 characters of the name are significant. If this parameter is omitted, it is assumed to be identical with the *link_name*.

4. Specifies whether the object is *protected* (P) or unprotected (U). If an object is protected it can only be overwritten if this access is also specified with P. Reading access is always possible. The default value is U.

Some examples shall demonstrate the usage:

```
%BIND "installation" TO mylib (HISLANG)
```

The link name "*installation*" is bound to an object with the same name (name omitted) and module HI-SLANG. By, e.g., %COPY "installation" this HI-SLANG source can be read out of the modelling base named *mylib*.

```
%BIND "preana" TO mylib (,,cpu,P)
```

The predefined link name "*preana*" (you can use lower- or upper-case letters) is bound to the modelling base *mylib*. The execution of the corresponding AGGREGATE statement stores the aggregate named *cpu* as a protected member of that data base. Module and type are automatically set.

APPENDIX C.HIT Experiment Syntax Sketch

This appendix sketches the most important parts of the experiment block of the HI-SLANG syntax in a BNF-like form, being the most complicated part of the HI-SLANG syntax. The syntax for describing models is quite similar to that of high level programming languages. For a complete syntax and even HI-SLANG syntax diagrams see the Reference Manual.

```
experiment ::=
   EXPERIMENT experiment-name METHOD method;
      [ VARIABLE
         { object-name [, …] : simple_type [ DEFAULT expression]; } […] ]
   BEGIN
         statement […]
   END EXPERIMENT [ experiment-name];


   method::=
      ANALYTICAL "method-name"
   |  SIMULATIVE


   simple_type::= …
   |  INTEGER
   |  REAL


   statement::= …
   |  for_loop
   |  aggregate_statement
   |  evaluate_statement


   for_loop::= ...
      FOR variable-identifier := expression [, ...]
         LOOP
            statement […]
         END LOOP;


   aggregate_statement::=
      AGGREGATE componenttype-name;
         { CREATE expression PROCESS service-name; } […]
      END AGGREGATE;


   evaluate_statement::=
      EVALUATE
         MODEL model-name : model_type-name
            [ ( { [ LET parameter-name := ] expression] } [, …] } ] ;

         EVALUATIONOBJECT
            {{ evaluationobject-name VIA component-identifier } [, …]
               [ DEFAULT  estimator_part]
         HIERARCHY
            { hierarchy-name [, …] default_or_merge ; } […]
```

```
   BEGIN

      { MEASURE    stream [,…]
          AT       evaluationobject-name
        [ DUE  TO  hierarchy-name [, …]]
        [ estimator_part]

      [ CONTROL [ TRACEALL ]
        {[ AT      evaluationobject-name]
         [ STOP    start_or_stop_condition]
         [ TRACE ] } [… ] ; ]

   END EVALUATE;


estimator_part::=
   [ ESTIMATOR estimator [, …]]
   [ OUTPUT        TABLE "linkname" [, DUMPFILE "linkname"]]
   [ START         start_or_stop_condition]
   [ STOP          start_or_stop_condition] ; } […]


default_or_merge::=
   DEFAULT  ( component-name [, service-name [, use-name]] ) [. ...]
|  MERGE    hierarchy-name [, …]


start_or_stop_condition::=
{  CPUTIME          expression
|  MODELTIME        expression
|  EVENTS           expression
      [ DUE TO      hierachy-name]
|  CONFIDENCE LEVEL expression
      WIDTH         expression
      MEASURE       stream
      [ DUE TO      hierarchy-name]
|  ACCURACY         expression
} [ AND | OR …]


stream::=
   THROUGHPUT
|  TURNAROUNDTIME
|  POPULATION
|  OCCUPATION
|  UTILIZATION
|  SCHEDULE_RATE
|  PREEMPT_RATE
|  stream-name


estimator::=
   MEAN
|  BOUNDS
|  STANDARDDEVIATION
|  CONFIDENCE LEVEL          expression
|  FREQUENCY INTERVAL [ { expression .. expression } [, …]]
```

APPENDIX D. More HI-SLANG Features

This appendix sketches some more features, which are not handled in this Introduction, but in the HI-SLANG Reference Manual. Moreover it sketches the recent changes to HIT.

- **User-defined component control procedures**.
  Components in HIT are dynamic and autonomous systems (up to a certain degree), the progress of processes is goverend by predefined rules for *accept, schedule, dispatch* and *offer*. The HIT user can write his own component control procedures in HI-SLANG (or even in SIMULA).

- **Predefined Procedures.**
  There are much more procedures predefined than listed in Section 9.6.2. E.g., every component provides several procedures to determine its population, and for every service its state can be determined by predefined procedures.

- **Pre-compilation**
  Procedures, services, component types and total experiments can be transformed from HI-SLANG to PRE-SLANG (pre-compiled HI-SLANG).

- **Graphical output**.
  There are features to produce graphs and histograms (on a line printer).

- **Records and pointers**.
  In HIT there exists a concept for records and pointers similar to PASCAL.

- **Solver information.**
  The analyzer listing is extended by solver information, which, e.g., gives the reasons, why a certain algorithm within the desired solver was selected. For MARKOV it contains detailed state information.

Recent additions to HIT 3.1.000 are: the CHAIN statements, more efficient *synchsend* and *nowaitsend* components, the *observer*, trace control procedures, some %parm options, and the declaration of user-defined streams in components. For a more complete list see version 1.1.00 of the Reference Manual, Chapter 0.

APPENDIX E.References

/Beil85/ Beilner, H.:
Workload Characterization and Performance Modelling Tools, Proc. of the International Workshop "Workload Characterization of Computing Systems",
Pavia, Italy, 1985 (North Holland)

/BeMW88/ Beilner, H.; Mäter, J.; Weißenberg, N.:
Towards a Performance Modelling Environment: News on HIT,
Proc. of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation,
Palma de Mallorca, 1988 (Plenum Publishing Corporation)

/BeSt87/ Beilner, H.; Stewing, F.J.:
Concepts and Techniques of the Performance Modelling Tool HIT,
Proc. of the "European Simulation Multiconference, ESM '87",
Vienna, Austria, 1987

/Beil89/ Beilner, H.:
Structured Modelling - Heterogeneous Modelling
Proc. of the 1989 European Simulation Multiconference, Rome, 1989

/BüPS88/ Büser, M.; Pape, D.; Stewing, F.J.:
Simulation of Integrated Information and Material Flow in Logistics Systems: An Application of the Modelling Tool, HIT;
Proc. of European Simulation Multiconference, Nizza, 1988

/Heck91/ Heck, E. (ed.):
HITGRAPHIC User's Guide,
Universität Dortmund, Informatik IV, 1991

/LiSS89/ Litzba, D., Szczittnick, M., Stewing, F.J.:
Yet another simulation output analysis algorithm: the autoregressive, online-update evaluation technique of the modelling tool, HIT
Proc. of the 3rd European Simulation Congress, Edinburgh, 1989

/Weis92/ Weißenberg, N. (ed.):
HI-SLANG Reference Manual,
Universität Dortmund, Informatik IV, 1992

/Weis91/ Weißenberg, N.:
HIT-OMA User's Guide,
Universität Dortmund, Informatik IV, 1991

/Wolf86/ Wolf, H.:
Outil de Modelization et d'Evaluation HIT, Proc. of the "Workshop on Computer Performance Evaluation",
Sophia Antipolis, France, 1986, (in French)

All papers and documents referenced above are available on request.

APPENDIX F.Index

R

S

T

U

V

W

X