HI-SLANG

REFERENCE MANUAL

Document Version 3.6.00

for the
Hierarchical Evaluation Tool

HIT

Version 3.6.000

HI-SLANG REFERENCE MANUAL

FOR THE HIERARCHICAL EVALUATION TOOL HIT

Christel Wysocki (Editor)
Achim Wilde (Editor)

Martin Büttner
Beate Fricke
Othmar Klaaßen
Siegfried Nolte
Michael Szczittnick
Harald Stahl
Norbert Weißenberg

Abstract:

The Hierarchical Evaluation Tool HIT is a software tool for performance evaluation of computing systems during all phases of their life cycle. The hierarchical model description language HI-SLANG allows to build deeply structured models in a very modular way. Quantitative model evaluation can be performed by simulative or analytical methods.

HIT has been developed at the chair of Prof. Dr. H. Beilner, Department Informatik IV, Universität Dortmund. The project, HIT, has been partially supported by the Nixdorf Computer AG and the BMFT (German Federal Ministery of Research and Technology).

This document is released for internal and external use. Corrections, comments, criticism and suggestions for improvement of this document are welcome.

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

0. Foreword

This reference manual describes syntax and semantics of HI-SLANG and contains all the information important for running and using (the language interface of) the HIT system. It refers to the HIT system version as indicated on the front page.

0.1. References and Acknowledgement

The manual is made for advanced users of HIT/HI-SLANG. For a general and informal overview (e.g., for beginners) see /Weis92a/. Related documents describing HIT environments are the HITGRAPHIC User's Guide (/Sczi93/) and the OMA User's Guide (/Weis92b/).

Complex applications and examples of modelling with HIT can be found in /BeSt87/ (centering on simulation techniques), /BeMW88/, /Beil85/ and /Beil89/ (centering on analytical solvers, aggregation and the modelling environment) as well as /Beil88/ (in German), /BuPS88/ and /PaSt89/ (modelling of logistic systems). Many diploma thesises written at the chair Informatik IV, Universität Dortmund, contain detailled examples, but are written in German.

Currently there are installation guides for HIT for the operating systems BS2000 (/WaHo92/) and most UNIX like systems (/Weis92c/). For UNIX systems there is additionally a HIT User's Guide (/LeWe92/).

This reference manual is based on the German version /Stew89/, written by H.Scholten, R.Speckmann, F.-J.Stewing, N.Weißenberg and L.Wiggershaus. While translating it, many improvements and additions have been made, e.g., by the referees J.Mäter, F.Beilner and H.Deike-Glindemann. Most typing has been done by I.Koch and N. Weißenberg. For manual version 1.0.00 significant additions have been made by M.Sczittnick, V.Strell, R.Schelleter, L.Wiggershaus and N.Weißenberg. Most people of Universität Dortmund, Informatik IV have given valuable hints to improve the manual.

0.2. Warranty

The information contained within this document or the tool, HIT, is subject to change without notice. The Universität Dortmund, Informatik IV, makes no warranty of any kind with regard to this material or the tool, HIT, including, but not limited to, the implied warranties of merchantabilty and fitness for a particular purpose. The Universität Dortmund shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

The tool, HIT, is furnished under a license and may be used only in accordance with the terms of that license. It may not be provided or otherwise made available to any other person. No title to and ownership of the tool, HIT, is hereby transferred. It is provided "as is" without warranty of any kind.

0.3. What is new?

This is the sixth edition of the English HI-SLANG Reference Manual, which has been reworked and re-edited to improve clearness. This is the temporary last version of the Reference Manual.

Compared to version 3.4.00 of this manual and HIT version 3.4.000 only a few changes are fundamental. The following main changes apply:

- **Local Record Types**

  The restriction that record types cannot be declared within model types or component types is eliminated.

- **UTILIZATION for *prioserver***

  The stream UTILIZATION is usable for components of type *server* and *prioserver* from now on.

- **FREQUENCY INTERVAL for STATE streams**

  The estimator FREQUENCY INTERVAL is also eligible for STATE streams (beside EVENT streams).
  For a STATE stream, the amount of time the state is within the interval is accumulated. (For an EVENT stream, the number of occurrences of values in an interval is counted.)

- **Addition of some control file parameters (%PARM):**

  - **WARNACCESS:** Switch to produce warnings if used objects should be declared locally.

  - **FREQUENCYFORMAT:** This option controls the table output of the estimator FREQUENCY.

  - **TRACEFORMAT**: This option affects the format of the event trace of a simulation.

- **Implementation of *counter* and *synchsend* modified**

  The implementation of the standard component types *counter* and *synchsend* has been modified because of problems with arrays as formal parameters in LUND Simula.

1. Introduction

Aspects of performance play an important role in the whole life cycle of a computer system beginning in the design stage. Apart from this, other requirements like functional correctness and reliability have to be obeyed.

The system's fulfillment of the requirements with respect to performance has to be taken into consideration by the designer in the phase of conception already, according to the underlying specifications. At this stage, apart from his own experience and intuition, which are not reconstructable for others, the designer is aided by mathematical models and measurements taken from simulation models and (at a later stage) from prototypes of the computing system. Often the use of prototypes is, on account of the large expenses, only possible in a very restricted way, because the necessary iterations of design steps may result in the product becoming obsolete before it becomes marketable.

Aspects of performance are also essential for the choice and configuration of a system for one specific application. Examples are the choice and configuration of an operating system for a large computer in a computer centre or the introduction or expansion of a local area computer network in an office environment.

In the following phase of operation, the real performance behaviour is controlled by monitors (hardware/software). The evaluation of these measurements serves as a base for the optimization (tuning) of the system and as support in adapting it to new applications.

Since the effects of changes do not become evident before observation in further real operation, there is the risk of incorrect decisions, which then may appear directly as negative performance behaviour to the user.

The interpretation of the measurements from a simulation model or a prototype without corresponding support of specially developed tools as well as the design of mathematical models requires mathematical knowledge, which the designer does usually not have. Conversely, a mathematician called on supporting performance evaluation, first would have to laboriously acquire all details of the computer system to be analyzed.

1.1. Design Objectives of HI-SLANG/HIT

The outlined situation calls for a supporting tool for systematic performance evaluation securing all the demands (performance, efficiency) a computer system has to fulfill. In particular, such a tool should be flexible to use in the whole life cycle of the system, in the phase of design and realization as well as at the point of choosing a computer system for a specific application and also in the following phase of operation. The usage of this tool should be possible for anyone who has to judge the performance behaviour of a computer system at any phase of its life cycle, without further mathematical knowledge.

Furthermore, such a tool should support team-work, particularly in the phase of design. Performance evaluation, even of complex systems, should be possible at reasonable expense.

HIT (**HI**erarchical Evaluation **T**ool) was conceived as a tool for performance evaluation to perform most of the precedingly motivated requirements:

- As a modelling tool it can be applied during all phases of the life cycle of a computer system.

- The specification language HI-SLANG (**HI**T **S**ystem **LANG**uage) supports the structural description of models of computer systems, in which among other things accepted methods from the area of software engineering are applied:

  - A vertical model structuring based upon the concept of hierarchical, virtual machines.

  - A horizontal structuring in form of mutually excluded modules.

  - Support of the methods of divided labour in the specification of models, with the possibility of isolated specification of single vertical and horizontal model elements. A modelling base provides managing versions and configurations of model elements as well as of the model as a whole. Additionally, the HIT system controls resulting consistency problems.

  - The functionally oriented specification of models by transfer of the widespread functional hierarchy of calls into the area of quantitative models.

- The evaluation of the models is done automatically according to the specification of the user, by use of analytical (algebraical or numerical) and simulative techniques and requires only few special mathematical knowledge of the user.

- Even complex models can be evaluated with tenable expense due to the efficient analysis techniques and the possiblility of pre-evaluation of model elements.

Another important design goal of HIT was portability. HIT is completely implemented in Standard SIMULA (see /SIMULA87/ or /Pool87/), a high level programming language which has become wide-spread recently.

Thus HIT does not only run on most main frames but as well on UNIX workstations (Sun, DEC) and UNIX-PC's (PC'386, ...).

Summary:

HIT is a universal tool for performance evaluation operational on a wide range of hardware and operating systems, and is applicable particularly for the performance evaluation of computer systems in all phases of their life cycles. The specification language HI-SLANG, based upon the concepts of modern, high level programming languages, enables everyone to independently produce performance analyses with support of the tool HIT. Team-work for module design is supported by a modelling base.

1.2. Language Summary

A HI-SLANG program (*hit_unit*) is a complete, independent unit which can be compiled, evaluated or pre-evaluated by itself. As a rule, such a progam is composed of two parts, which can be specified separately: the model description and the description of the experiment, by which the model is to be analyzed.

HI-SLANG provides a number of data types, operations, and control statements for the algorithmic description of model behaviour, which are known from other high level programming languages (the progamming language kernel of HI-SLANG). In addition, the language covers special elements for the description and analysis of models.

Syntactically, a HI-SLANG program consists of an experiment block with its declaration part representing the model description and its statement part representing the model analysis. The model description or parts of it can be placed in front of the experiment block resulting in a better survey and increased flexibility for modelling. At the beginning of the experiment block the method which is to be used for performance analysis should be specified. The options are an analytic-algebraical solver (an exact method for separable networks, approximate methods for: "large" separable networks, special non-separable networks and performance bounds), an analytic-numerical solver for models of computer systems which can be represented by Markov chains, and simulation for more general models. The realization of series of evaluations can be achieved by an arrangement of evaluation statements in control structures.

Model description covers the description of one or more models (model types) for their complete performance analysis and/or the description of one or more model parts (component types) for pre-evaluation (creation of a substitute server).

The static structure of a model consists of a model type, which itself contains hierarchical, nested components (objects) described by component types. Using a virtual declaration (ENCLOSE), a component can simultaneously be part of various other components. Although arbitrary model structures may be specified in this way, only models or component types having the structure of a directed, cycle-free graph with a root can be evaluated.

Behaviour patterns are modelled as processes, their types are described by services. Within the model type and within every component type, processes can be created and started as local processes using the CREATE statements or by static declarations of processes. CREATE, for example, enables the time-controlled generation of one or more local processes at a fixed point of model time or in fixed intervals. Within services, local processes can also be initiated event-driven by CREATE. In this system of parallel processes, the execution of a process can be sequential or by itself concurrent using a dedicated control structure (CONCURRENT statement).

With a PROVIDE-declaration (ADT-property of components) components can make services available to the environment. These services can be used by services of other components declaring such services with USE, followed by a local identifier. The binding of used services to provided services of components takes place in the REFER-part of the surrounding component type. By the (nested) call of services, a process can propagate hierarchically down the model structure.

The local processes and service executions are controlled by the components they run in (monitor property of components). In this regard, a process can be in one of three coarse states in this component, related to its possible progress:

- It can momentarily be excluded from all progress.
- It may have a right to progress or it is progressing.
- It may have terminated his activities.

In HIT terminology, the process is always in one of three component areas, corresponding to those states (*entry area, service area, exit area*). The transition of states respectively the transport between those areas results from predefined procedures or component control procedures, formulated by the user.

The local processes of the model type define the load of the model. By a service call a machine consisting of all the components declared in the model type is loaded. This combination of a calling load and a called machine repeats itself within all components of the model. An amount of (local) processes, the load, is opposed to a machine, consisting of the locally declared components. The machine is loaded by calls of the services provided by the machine. A model, described with HI-SLANG, therefore consists of a hierachy of load-machine pairs, its lowest layer usually built of components of a standard type *server.* This component type provides the basic service *request* (*amount: REAL*), in which the parameter expresses the amount of , e.g., the work to be done resulting in a time the component is occupied. Applying the operations *spend* and *hold* it is possible to "consume" model time without explicit usage of a component of type *server.*

In analysis there is a distinction between the complete analysis of a model and the pre-analysis of model parts. The pre-analysis is specified by an AGGREGATE statement and, under observance of certain conditions, causes the creation of a state-dependent substitute server for a model part, described by a possibly complex component type.

The EVALUATE statement in the statement part of the EXPERIMENT block initiates the creation and analysis of the model. It is possible to analyze the so-called evaluation objects, which are attached to the components (or component areas). The specific evaluation for certain identified load portions (service calls) is achieved by the declaration of so-called load filtering hierarchies. Local processes within components of higher layers or the current component may cause the load.

For this purpose, evaluation objects and load filtering hierarchies are declared in the declaration part of the EVALUATE statement and have to be specified completely in terms of the model hierarchy or the calling hierarchy respectively.

The actual description of measurements and evaluations occurs in MEASURE statements. Streams to be measured are local to components. Beside standard streams as, e.g., THROUGHPUT and POPULATION, it is possible to evaluate user-defined streams of the component.

Results can be presented in form of tables, machine-readable files or graphical output.

The complete specification of a measurement consists of the stream(s) to be evaluated, and the evaluation object attached to the component in which the stream is declared. Furthermore, load filtering hierarchies, estimators, the form of result representation and perhaps a measuring time interval may be specified. The last four specifications, the so-called evaluation attributes, can already get defaults when declaring the evaluation object.

In the case of simulation, some interaction possibilities for observation and control of the simulation runs are available. A CONTROL statement within the EVALUATE

statement serves as a specification of the end of the simulation by criteria like elapsed CPU- or model time, occurance of a fixed number of events or by reaching a predefined confidence interval related to a measurement. In the case of numerical analysis it is only possible to specify stop conditions relative to the consumed CPU-time and/or to the demanded accuracy of the results within such a CONTROL statement.

The programming kernel of HI-SLANG provides variables and constants of simple type (INTEGER, REAL, BOOLEAN, CHARACTER, TEXT), of structured type (ARRAY) as well as terms built from elements of such types. Furthermore records and pointers can be used, which can be created and accessed by special operations or statements (partly known from, e.g., PASCAL or SIMULA).

Further statements are the assignment, the BLOCK statement with a local declaration part, the condition statement, especially one with several possible cases (CASE) or with branching according to probabilities. Additionally, a number of LOOP statements, one of them specifying an endless loop, that is necessary for modelling permanent activities, are available. Even open and closed chains (known from queueing network terminology) can be specified directly.

There are procedures with very flexible mechanisms of parameter transmission modes: apart from three different transmission modes, known for instance from SIMULA (call by name, call by value, call by reference), it is possible to define actual parameters as positional parameters and/or as keyword parameters. Formal parameters with a default may be omitted when the actual parameters are defined. Procedures or services with one result value may be used within terms, while those with multiple result values may only be used within assignments.

Language constructs for file handling enable sequential file accessing similar for example to PASCAL or SIMULA. As file objects either operating system files or objects within a modelling base are accessible.

There is a standard modelling base providing a great amount of useful component types as building blocks. The user may create own collections according to his needs.

1.3. Syntax Notation

The formal description of the syntax is written in a modified Backus-Naur form. The following conventions are valid:

- Capitalized identifiers of extra bold print are terminals (reserved words) of HI-SLANG, e.g.:

  PROCEDURE

- Identifiers printed in small letters are non-terminals (syntactical categories), e.g.:

  formal_parameters

- All non-terminals, ending with *-name* or *-identifier*, belong to the syntactical category *name* or *identifier* respectively. The prefix is regarded as a semantic comment:

  procedure-name

- The derivation symbol '::=' attaches the possible derivations to every non-terminal. Several derivations for one non-terminal are separated by line feed and a vertical line:

  mode ::=
  VALUE
  | **NAME**
  | **REFERENCE**

- If not all possible derivations of a non-terminal are listed, the omission is marked by three points:

  simple_type ::= ...
  | **INTEGER**
  | **REAL**

  The omitted derivations are listed and explained in other places:

  simple_type ::= ...
  | **BOOLEAN**
  | **CHARACTER**
  | **TEXT**

  simple_type ::= ...
  | **INFILE**
  | **OUTFILE**

  simple_type ::= ...
  | **POINTER FOR** recordtype-name

- Optional parts are specified in square (meta-)brackets:

END PROCEDURE [procedure-name] ;

- Possible repetitions are marked by [*separator* ...], the terminal *separator* connecting the repeated units. The *separator* may be a blank:

  formal_parameters ::=
  (parameter_declaration [; ...])

  sequence_of_statements ::=
  statement [...]

  If the repetition does not only refer to the last non-terminal in front of the '['-meta-character, the repeatable unit has to be placed between braces {}:

  array_bounds ::=
  ⟦{simple_real_expression .. simple_real_expression} [, ...]⟧

- Meta characters, (e.g., ⟦, ⟧, ⦃, ⦄) are printed in contour style if they are not used as meta characters but as terminals, as in the example above.

1.4. Structure of the Reference Manual

This reference manual consists of nine chapters and ten appendixes (A up to J) including an index. The chapters two up to five present a complete description of syntax and semantics of HI-SLANG. Chapters which in turn consist of several subchapters begin with a short outline of these subchapters. All subchapters concerning the representation of single language constructs are structured in a uniform way.

After a short description of the purpose of the concerned language construct, the corresponding syntax is defined. The notation for representation of the syntax is explained in Section 1.3. Subsequently syntactical particularities and restrictions not obvious from the context-free representation of the syntax are explained. The detailled description of the semantics, which calls attention to potential particularities and typical mistakes, follows next. Examples illustrate possible applications of the described language construct. Within all chapters identifiers used in examples or predefined identifiers of HI-SLANG are referred in *italics* style, whereas the identifiers of component control procedures are written upper-case.

The index at the end of the reference manual provides references to the pages, on which the concerned language construct also gets additional (perhaps more detailed) mentioning. It also refers to sections where language constucts are described which may be fitted into the described construct, corresponding to the syntactical structure. Furthermore, the index covers references to central terms of the HIT-model view and of modelling in general.

The language is described bottom-up, which expresses that, beginning with the lexical elements (Chapter 2.), more and more complex constructs are represented step by step up to complete programs and models, respectively (Chapter 5.). The description is devided in the programming language kernel of HI-SLANG (Chapter 3.), covering the language constructs which are known from high level programming languages, and the elements of model description (Chapter 4.) and model analysis (Chapter 5.).

The chapters six to nine explain the language environment of HI-SLANG and contain all the information necessary for the application of HIT: program structure and precompilation (Chapter 6.), installation dependent properties and restrictions (Chapter

7.), and the control of the HIT-system (Chapter 8.). A survey of structure and operation of the compiler is already given in Section 1.5. These chapters are not uniformly structured, their structure depends on the treated contents. The references to literature (Chapter 9.) conclude the main part of the reference manual.

The appendices present different information relevant to the application of HIT in a compact form. This includes, as a summary and extension of the description of HI-SLANG, a survey of the complete syntax (Appendix A. and B.), a listing of reserved words and symbols (Appendix C.), and a listing of operators and precedence rules, predefined procedures and constants (Appendix D.).

Appendix E. describes all restrictions on the use of the different solvers and on the statistical evaluation in the case of simulation. This includes, as a special case, the restrictions on the (analytic-algebraical) pre-analysis of component types. The contents of the HIT standard modelling base is illustrated in Appendix F. Next come the description of output formats (Appendix G.), and some advices on error identification (Appendix H.).

A comprehensive example in Appendix I., which uses most language constructs of HI-SLANG, and an index (Appendix J.) of all central terms of the HIT model world and of all language elements of HI-SLANG conclude the reference manual.

1.5. Structure and Operation of HIT

A major aim in designing HIT was portability. Therefore, HIT was implemented in such a way that HI-SLANG programs first are translated by a precompiler (HI-SLANG compiler) to SIMULA and then by a SIMULA compiler to executable code. The HI-SLANG compiler itself is implemented in SIMULA.

The higher programming language SIMULA was chosen, because a lower expense of implementation could be expected on account of the powerful class concept of SIMULA; in particular, it was possible to use the standard SIMULA class "simulation" for the implementation of the simulator. Furthermore all predefined functions and control structures of SIMULA could be integrated into HI-SLANG. An efficient implementation of the HIT system additionally required the availability of dynamic arrays.

In the following, a rough survey of the structure and operation of the HI-SLANG compiler will be given. For more details concerning the techniques used to construct the compiler and the implementation language SIMULA see /Wolf86/ and /MuWe87/.

The modular structure of HIT is illustrated by the diagram on the next page. An exceptional position is held by the FAN module (**F**ile **A**ccess **N**etwork): it controls all input and output interfaces of the compiler and of the (generated) analyzers. Every file object read or created can be bound to a file or to an object in a modelling base, shortly called mobase. This is specified in the control file. Thereby the HIT system is able to read objects directly from a mobase and write as many objects as desired into one or more of such data bases. The object access procedures are separated in a special modul to be able to support other object management systems. The FAN system is described in more detail in Chapter 8.

A further program also using the FAN system is HIT-OMA. It serves for all operations on file objects which do not occur through the HIT system or which cannot be carried out with HIT. There is a standard mobase for HIT which contains a number of SIMULA and HI-SLANG objects required as standards in modelling and which therefore are predefined.

Details of HIT-OMA can be looked up within the HIT-OMA User's Guide (see /Weis92b/), while the elements of the standard mobase are described in Appendix F.

HI-SLANG programs are translated to SIMULA in four steps (passes) and one prolog (pass 0):

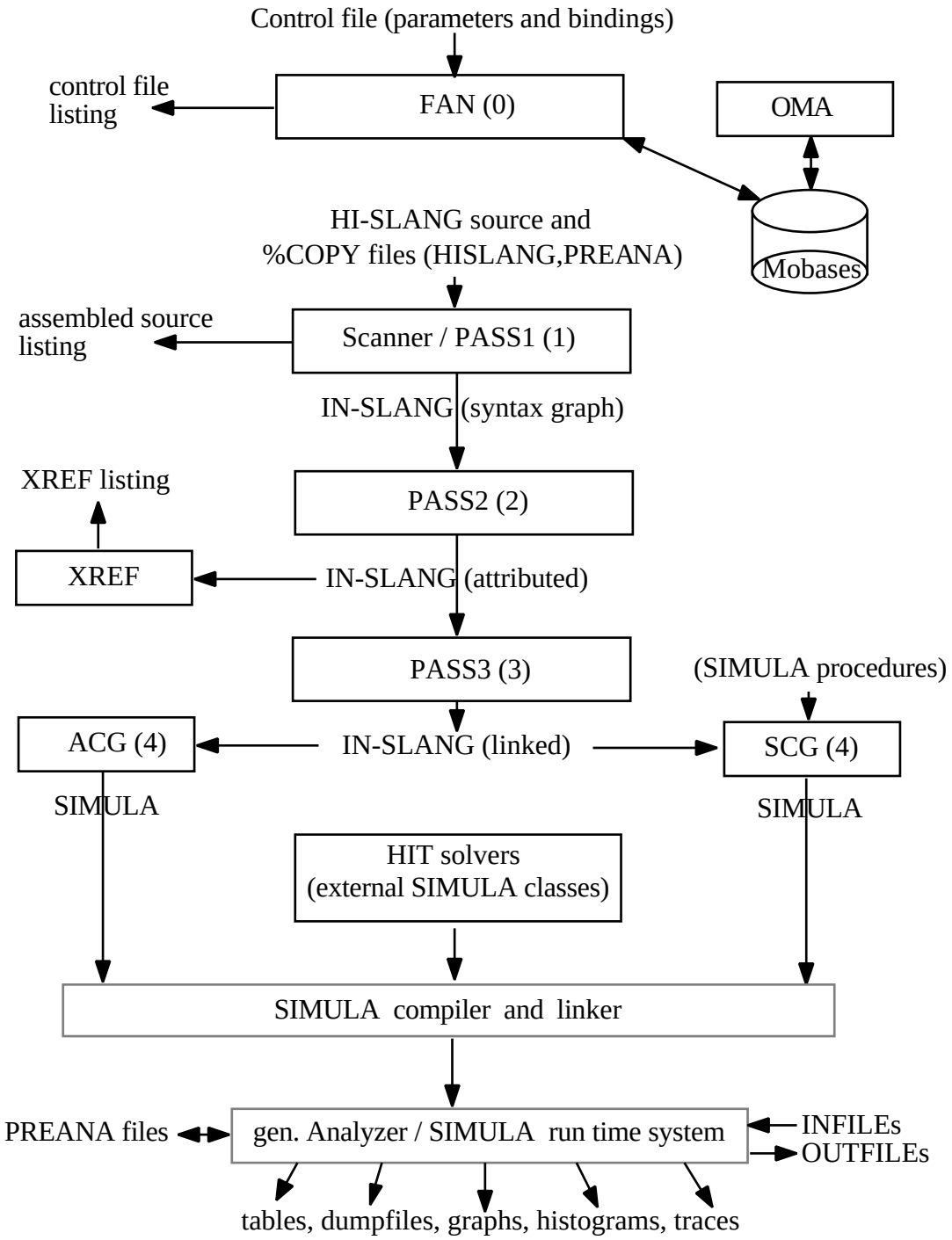
(0) Initially the FAN system may interpret a control file, which may contain several parameters for the translation of a HI-SLANG source (e.g., INDENT, XREF) and bindings of file objects. Next the main program of the compiler is executed which activates all other moduls and which may collect and emit (error-)messages for the single passes and for the whole process of translation.

This last task is also supported by the FAN system therefore controlling an error message library. This library is produced by OMA from an editable file. This increases portability and maintainability, since single messages can be changed or all error messages can be translated, for example, from English to German.

(1) The first pass is realized by a scanner and a parser working together, the latter being contained in Module PASS 1. These modules can read several HI-SLANG sources and other representations of model elements from files or mobases with support of the FAN-system. Except for the first HI-SLANG source, these sources are included by %COPY. Besides source texts there exist:

- aggregated component types, which are created by a so-called pre-analysis. These types consist of a HI-SLANG component interface for the parser and a component body (which contain a table of REAL values) not read until a later stage, when evaluation runs are carried out.

The scanner provides the PASS 1 with lexical HI-SLANG units, which are transformed to an IN-SLANG representation. IN-SLANG (**IN**ternal **S**ystem **LANG**uage) consists of instances of SIMULA classes, which represent a table of symbols and an attributed syntax graph. IN-SLANG is defined by a large number of SIMULA classes, which serve as a local communication interface for all compiler modules. A generator was used to simplify the implementation of the HI-SLANG compiler. It has generated the scanner and the (recursive-descent-) parser based on approximately 180 HI-SLANG keywords and symbols and a HI-SLANG grammar consisting of about 450 productions.



Legend:

PREANA: contains an aggregated component type
(n): precompiler pass number (see below)

Overview of Structure and Operation of the HIT system

(2) In the second pass the semantical correctness of the generated syntax graph is checked by the modul PASS 2. At the same time its attributes are filled with values.

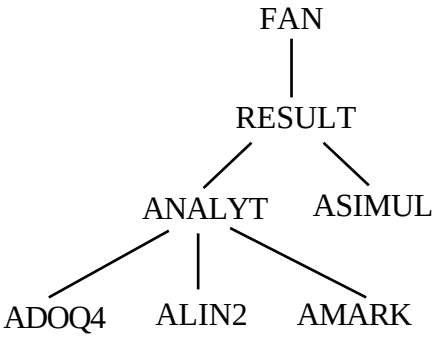
The attributed IN-SLANG graph can furthermore be used for the production of a cross reference listing of all different types and objects occuring in the assembled HI-SLANG source. This listing is created by the modul XREF after completion of PASS 2.

(3) In the third pass, the modul PASS 3 checks some consistency regarding hierarchy and EVALUATE statement. Additionally some attributes of IN-SLANG are filled with values.

(4) In the last pass a code generator translates the attributed syntax graph into the intended SIMULA code. Two different code generators exist, one for simulation and another for the analytical methods, corresponding to the different methods of performance evaluation of computer systems.

Both code generators are realized as subclasses of the same superior class, which contains the basic functions for code generation. The code generators construct a main program, which takes access to different precompiled external classes. The **S**imulative **C**ode **G**enerator SCG creates the main program for a simulator, whereas the **A**nalytical **C**ode **G**enerator ACG produces code for transferring the model parameters to data structures and for the call of an (exact or approximate) solver.

(5) The generated programs use precompiled, external classes (called HIT solvers), which are the leaves of the following class hierarchy:



Within this FAN class hierarchy a SIMULATION class hierarchy is nested, so that any analyzer is based on the concepts of both hierarchies (multiple inheritance).

The following analyzer classes are at disposal:

- The class ASIMUL contains the basic functions for the simulator. The employed data structures are predefined by SIMULA.

- The class ADOQ4 (**DO**rtmund **Q**ueueing Analyzer **4**) contains solvers for the exact and approximate analysis and aggregation of separable or non-separable networks (for the model class solvable with DOQ4 see /MuNS85/ and /KLMN88/).

- The class ALIN2 (**LIN**earizer 2) contains an approximate solver for (large) separable networks (for the solvable model class see /Knau88/). Furthermore, LIN2 also allows the calculation of exact upper and lower bounds (performance bounds) for the mean values of the desired performance measures (see /KnND89/).

- The class AMARK implements a numerical solver for markov chains (for the solvable model class see /MuNS85/ and /MuRo87/).

The analytical methods have the class ANALYT in common. This class consists mainly of the modul USEFAA (**U**niver**S**al **E**nvironment **F**or **A**nalytic **A**lgorithms), which contains data structures and operations on these data structures used by all algorithms.

All solver classes coincide by using the class RESULT, which provides functions mainly serving the formatted representation of results. Furthermore, RESULT contains basic functions which are not provided by SIMULA, but are of common importance (e.g., cox-distribution, log, eoln, eof).

(6) In the last step (no part of the HIT system, indicated by broken lines in the diagram above) the SIMULA code is translated and linked to executable code by the SIMULA compiler. This object code is called analyzer and executed under control of the SIMULA run time system for calculation and representation of the desired performance values in form of tables, dump files or even simple graphs or histograms. In case of a so-called pre-analysis, an aggregated component type is created which is represented in HI-SLANG and which can be used as a substitute for the original component type.

In case of simulation, an event trace can be produced in additon to the execution. Furthermore, methods for file handling are at disposal.

2. Lexical Elements

This chapter defines the lexical elements of the language HI-SLANG. By this we think of the set of characters the language is based on, and of the lexical symbols that are composed of them. Lexical symbols are considered to be names, reserved words, numbers, characters, character strings, comments and compiler directives. The syntax of lexical symbols will be perceived by the scanner.

Please note that the length of an input line to read must not be longer than 132 characters.

2.1. Set of Characters

The set of characters to construct the elements of the language consists of

- letters:

  letter ::=
  A | **B** | ... | **Z** | **a** | **b** | ... | **z**

- digits:

  digit ::=
  0 | **1** | **2** | ... | **9** |

- special characters:

  special_character ::=
  " | **#** | **%** | **&** | **'** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | < | = | > | **[** | **]** | **{** | **}** | **_**

- and the blank.

We do not distinguish between capital and small letters, except for characters within character strings or within comments. Moreover, any other ASCII or EBCDIC character, depending on the installation of the HI-SLANG compiler, may be used. For some of the special characters shown below, a substitution is provided (for instance for IBM VM/CMS installations):

'[' ⇔ '(.'
']' ⇔ '.)'
'{' ⇔ '(*'
'}' ⇔ '*)'

2.2. Lexical Symbols and Separators

A HI-SLANG source is a sequence of lexical elements that are arranged from the left to the right within a line and from the top to the bottom through the lines. The end of a line as well as blanks and comments are used as separators.

Names, numbers and symbols may not contain blanks. A symbol is any special character except '_' as well as one of the following character combinations:

.. | **<>** | **<=** | ****** | **//** | **>=** | **::** | **:=** | **(.** | **.)** | **(*** | ***)**

A lexical element must be terminated before the end of a line.

Compiler directives (see Chapter 8.) differ from the other lexical elements in the following manner: they start with a percent character as the first character different from blank or tab in the line, and they are limited by the end of the line. Compiler directives permit only blanks and tabs as separators. They will not be scanned by the scanner but by a preprocessor included in the HI-SLANG compiler.

2.3. Names

Names are used for constructing identifiers. Underscores '_', also several in succession, are allowed. All characters within a name are significant.

```
name ::=
        letter [letter_or_digit_or_underscore [...]]


letter_or_digit_or_underscore ::=
        letter
|       digit
|       _
```

Names begin with a letter. As just mentioned before, we do not distinguish between capital and small letters, for example the names *IO_TIME* and *io_time* are identical. Just like symbols names are terminated by the end of a line. Moreover they must not be longer than 70 characters.

Examples:

```
Franz_Josef
A1
aBC_
this_is_a_very_long_but_correct_hi_slang_name
```

2.4. Numbers

There are two categories of numbers: INTEGER numbers and REAL numbers. INTEGER numbers are of the simple data type INTEGER. REAL numbers are of the simple data type REAL.

number ::=
digit [...] [.digit [...]] [**E** [unary_operator] digit [...]]

unary_operator ::=
+ | **-**

(In this case the symbol […] means an iteration of previous element without separator.)

A number always starts with a digit. An INTEGER number consists only of digits whereas a REAL number consists of digits and contains a decimal point '.' and/or the exponential character 'E'. A decimal point must be followed by at least one digit. An exponential character may be followed by a digit or a *unary_operator*, which itself must be followed by a digit. Exponent parts are based on 10. The representable scope of numbers and the precision depends on the installation.

Examples:

1234	{INTEGER number}
0.1234E4	{REAL number}
1234.0	"
123400E-2	"
12340.0E-1	"

2.5. Characters

Depending on the installation of the HI-SLANG compiler, permitted characters are all ASCII or EBCDIC characters between quotes. Quotes are the character delimiters.

character ::=
'ascii_or_ebcdic_character'

ascii_or_ebcdic_character ::=
<one of the ASCII- or EBCDIC-characters>

A character, limited by quotes, must be terminated before the end of the line.

Examples:

'A'	
'+'	
' '	{blank}
''''	{character delimiter as character}

2.6. Character Strings

Character strings, also called TEXTs in HI-SLANG, consist of no, one or more than one characters, the character set (ASCII or EBCDIC) depends on the installation. They are embraced by double quotes as string delimiters. If the string delimiter is to be part of the string itself, it must be specified twice.

```
string ::=
    "[ascii_or_ebcdic_character [...]]"
```

A string may not contain more than 70 characters and may not exceed line bounderies. For creating longer strings one uses the concatenation operator '&', which concatenates two strings to a new one. The length of a string is determined by the number of characters between two string delimiters. The length of the empty string is 0.

Examples:

```
""                                          {empty string, length 0}
" ", "A", """"                              {strings, length 1}
```

Note:

"A" is a character string, 'A' is a character.

2.7. Comments

Comments are permitted at any place within a HI-SLANG-program. Like blanks, they are skipped by the scanner. The comment text is located between the comment delimiters '{' and '}'. It may consist of any ASCII and EBCDIC characters with the exception of '{' and '}'. '{' and '}' may be substituted by '(*' and '*)', for instance in HIT installations on VM/CMS machines.

A comment must be terminated before the end of a line. If you want to comment out a set of lines you can use compiler directives (see Section 2.8 or 8.3.)!

```
comment ::= ...
|       { [ascii_or_ebcdic_character [...]] }
```

Examples:

```
{This is a comment.}
{}
% This is a comment.

%RESET comment_out
%IF comment_out THEN
arbitrary_lines
%FI
```

2.8. Compiler Directives

Compiler directives serve for controlling the HI-SLANG compiler. They start with a '%' character, which should be the first character of a line different from blank or tab, directly followed by the name of the directive. Directives with a blank following the '%' sign are considered to be comments.

```
comment ::=
|       % ' ' [ascii_or_ebcdic_character [...]]
```

Compiler directives are limited by the end of the line. They may appear at any place in the HI-SLANG source. Fore more details see Section 8.3.

Examples:

```
%COPY "SEMAPHOR"
% This is a comment
% COPY "SEMAPHOR" is also a comment, due to the blank before COPY
```

2.9. Reserved Words

Reserved words as, e.g., BEGIN and END are written in capital letters in this reference manual, but can also be written lower-case. User-defined names may not be identical to them. A complete list of reserved words of HI-SLANG is contained in Appendix C.1.

3. Programming Kernel of HI-SLANG

This chapter describes the programming kernel of HI-SLANG which is used as a base for modelling computing systems and for the analysis of the models.

3.1Declarations of Variables and Constants

HI-SLANG models are objects composed of other objects. Every object has a type and is associated with a name. This chapter describes the rules for the declaration of variable and constant objects.

HI-SLANG offers different kinds of objects, for instance constants and variables of simple or structured types and procedures. In addition, there are some special object types necessary for describing and analyzing models.

An object is created by a declaration. A declaration defines one or more objects and a name which can be used to access the object. A name for an object may be used textually before its declaration. An implicit declaration of names is not allowed in HI-SLANG programs.

```
common_declaration  ::= ...
|       variable_or_constant_declaration
```

Declaring constants or variables of a simple or structured data type causes them to be initialized by default values.

The use of an object's name preceding its declaration will create a node of the syntax tree with the name being its only attribute. The missing attributes are added to the node when the declaration of the object is found in the program source during following passes. If a declaration cannot be found, the compiler yields an error message.

The block concept of HI-SLANG, also known from other higher programming languages, solves conflicts concerning the use of names and their scopes.

Expressions included in declarations of constants of simple data types are evaluated while compiling the HI-SLANG program. Expressions within declarations of objects of other types are not evaluated but passed to the SIMULA compiler following the HI-SLANG compiler.

3.1.1. Variables and Constants of Simple Data Types

Constants and variables are created by a declaration. Correspondingly their names are notified to the program. When a variable or constant object is declared, a simple or structured data type is assigned to it. In the following, we will consider only simple data types, such as INTEGER, REAL, BOOLEAN, CHARACTER and TEXT.

variable_or_constant_declaration ::= ...
| variable_or_constant simple_object_declaration [...]

simple_object_declaration ::=
object-name [, ...] : simple_type [**DEFAULT** expression] ;

variable_or_constant ::=
VARIABLE
| **CONSTANT**

simple_type ::= ...
| **INTEGER**
| **REAL**
| **BOOLEAN**
| **CHARACTER**
| **TEXT**

The object specification (VARIABLE, CONSTANT) is followed by the name of the object or a list of names for several objects which are separated by commas. A colon ':' separates the name list from the type specification. So for declaring several objects using the same object specification, the latter needs not to be repeated.

Constant objects have to receive a default value, variable objects may, but don't necessarily have to receive one. An object is initialized by a default value using the keyword DEFAULT that must be followed by an expression.The expression must yield a value of the same type as the declared object. If not, type conversion must be possible.

An expression which is a default of a constant is evaluated by the HI-SLANG compiler and therefore it may not consist of variables, formal parameters, procedure calls, except arithmetic functions or standard functions for handling characters (regard Appendix D.2.1. resp. D.2.2.). Constants used within these expressions must be declared in a surrounding (outer) block or textually before in the same block.

For example declarations as

```
CONSTANT    a    :    INTEGER DEFAULT  -5;
CONSTANT    b    :    INTEGER DEFAULT  -a;
```

are allowed, but only in this order.

An expression being the default of a variable is evaluated by the SIMULA system. It must not contain any objects declared in the same block except constants declared textually before. So if CONSTANT is replaced by VARIABLE the example above constitutes an error.

List of names within variable declaration are initialized by the same default value because the DEFAULT expression is evaluated exactly once whenever the program reaches the block of the declaration.

The following table gives the range of values and standard defaults for simple typed objects:

type	default	range of values
INTEGER	0	installation-dependent
REAL	0.0	installation-dependent
BOOLEAN	FALSE	FALSE, TRUE
CHARACTER	char (0)	installation-dependent
TEXT	NOTEXT	NOTEXT (i.e., empty string) or a character string, restriction of length depends on the installation

char is a standard procedure. *char (0)* delivers the first ASCII or EBCDIC character.

Examples:

```
VARIABLE   popul1, popul2  :  INTEGER;        {initialized with 0}
VARIABLE   amount          :  REAL           DEFAULT 10.0;
           flag            :  BOOLEAN        DEFAULT TRUE;

CONSTANT   a               :  CHARACTER      DEFAULT 'a';
           string          :  TEXT           DEFAULT "CPU TIME:";
```

3.1.2. Variables and Constants of Type ARRAY

An array is a homogeneously structured data type. It connects several elements of the same type. Array elements may be of simple data types, record types, component types, procedures and services. In the following we introduce the declaration of array variables and array constants, where the array elements are of simple data type.

```
variable_or_constant_declaration  ::= ...
|       variable_or_constant    array_object_declaration [ ...]


array_object_declaration  ::=
        object-name [, ...] :
        ARRAY ⟦ array_bounds [, ...] ⟧ OF simple_type
        [DEFAULT expression_or_aggregate] ;


array_bounds  ::=
        simple_real_expression .. simple_real_expression

expression_or_aggregate  ::=
        expression
|       aggregate

aggregate  ::=
        ⟦ expression_or_aggregate [, ...] ⟧
```

An array is characterized by the number of indices (the dimension of the array), the lower and upper bounds of every index and by the type of the array elements. At run time lower bound ≤ upper bound must hold.

Indices must be of type INTEGER (REAL expressions will be converted to INTEGER indices by rounding).

Index bounds are defined by arithmetic expressions. Variables appearing in these expressions must be declared in a surrounding block. Constants appearing in these expressions must be declared in a surrounding block or textually before in the same block. If there are names of variables and procedures within these expressions the array is called a **dynamic array**. This is an array that has a variable number of elements which depends upon the results of the expressions. Arithmetic expressions used as bounds of arrays directly within an experiment block may not contain variables or procedure calls. Some predefined procedures as especially marked in the overview in Appendix D.2. may not be used for array bounds definition within the hit unit and experiment block.
Array bounds expressions are evaluated while running the program, when the run time system reaches the block or scope of the array declaration. All array bounds expressions in the declaration part (of the block) are evaluated prior to any default expressions in the declaration part.

Please note that a list of names within a dynamic array declaration means a declaration of arrays with same bounds.

Each array of simple data type or record type has the attribute *dimension* and the attributes *lower_bounds* and *upper_bounds*, all of which can only be read. These attributes support the access to dynamic arrays and to arrays being passed as parameters. For instance, if arrays are defined as formal parameters, their dimension is not specified. Specification occurs when the current array object is handed over. Array attributes are accessed by a dot notation, the attribute following the array name.

Example: **{accesses to array attributes}**

```
VARIABLE      turn:     ARRAY [1..2, 1..3] OF REAL;

    turn.dimension              {=2}
    turn.lower_bounds [1]       {=1}
    turn.upper_bounds [1]       {=2}
    turn.lower_bounds [2]       {=1}
    turn.upper_bounds [2]       {=3}
```

Array elements of variable arrays may have a default value. Those of constant arrays must have a default. The default values for an array may be given as an *aggregate*. It is also possible to specify a simple *expression* here, the value of this expression will initialize all array elements.

The expressions of an aggregate must be of the same type as the array, or they must be convertable to it.

Expressions in aggregates to initialize a constant array are evaluated during compilation of the HI-SLANG source. Therefore they may not consist of variables, formal parameters and no other procedures than arithmetic functions or standard procedures for character handling (see Appendix D.2.1., D.2.2.).

These procedures and operators may operate on other constants if they have been declared before (in surrounding outer block or textually before in the same block). For example the following declarations are allowed, but only in this order:

```
CONSTANT    a    :    INTEGER DEFAULT -5;
CONSTANT    b    :    ARRAY [1..5] OF INTEGER DEFAULT -a;
```

Aggregate expressions for initializing variable arrays are passed to the SIMULA system. They must not contain any objects declared in the same block. So if CONSTANT is replaced by VARIABLE the example above constitutes an error.

Furthermore, the following must coincide with:

- the dimension of the aggregate expression and the dimension of the array
- the number of expressions within a "sub-aggregate" (between '[' and ']') and the number of the array indices in the corresponding index range.

For dynamic arrays, it is not possible to check this while compiling a program.

Aggregate assignments (i.e., the assignment of default values to array elements) of an n-dimensional array must be specified as a one-dimensional aggregate of elements of a (n-1)-dimensional array, n>1.

Example:

```
variable A [1..3, 1..4] of integer default [[11, 12, 13, 14],
                                            [21, 22, 23, 24],
                                            [31, 32, 33, 34] ];
```

The aggregate assignment for this example array is handled in the following way:
the 2-dimensional array A is specified as a one-dimensional aggregate of 3 elements of a one-dimensional array (each with 4 elements); thus the first element of the first "sub-aggregate" ([11, 12, 13, 14]) is assigned to the first "sub-array" (A[1, .]) (A[1, 1] := 11; A[1, 2] := 12; …) and so on.

If the elements of a variable array are not initialized explicitly, they acquire the standard default values of their type (see Section 3.1.1.).

The access to an array element is gained by specifying its name followed by the indices of the array element. These may be arithmetic expressions embraced in square brackets ('[', ']'; see *identifier*). Square brackets can be substituted by '(.' and '.)', for example on HIT installations in VM/CMS.

Examples:

```
CONSTANT n : INTEGER DEFAULT 10;      {constant from the same or the surrounding block}

    {array variables}

    VARIABLE  table       :    ARRAY [1..10]      OF INTEGER;
              matrix      :    ARRAY [1..n, 1..n] OF REAL;
              bit_vector  :    ARRAY [-1..n+1]    OF BOOLEAN;
```

{record array; *complex* : see Section 3.7.1.}

```
RECORD    c_matrix    :    ARRAY [1..4, 1..4] OF complex (1);
```

{array of constants with aggregates}

```
CONSTANT  days        :    ARRAY [1..7] OF TEXT DEFAULT
                           ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"];
```

{array of variables with aggregates}

```
VARIABLE  alpha       :    REAL DEFAULT 90.0; {declared in an outermost block}

VARIABLE  turn        :    ARRAY [1..2, 1..2] OF REAL DEFAULT
                                                  [ [cos (alpha), -sin (alpha)],
                                                    [sin (alpha),  cos (alpha)] ];
                                                    {e.g., turn [2, 1] = sin (alpha)}

          tab         :    ARRAY [1..10] OF INTEGER DEFAULT 0;

          square      :    ARRAY [1..3, 1..1, 1..3] OF REAL DEFAULT
                                                  [ [[1.0, 0.0, 0.0]],
                                                    [[0.1, 1.1, 0.1]],
                                                    [[0.0, 0.0, 1.0]] ];
```

{dynamic array}

```
VARIABLE  dyn_lim     :    INTEGER; {variable from the surrounding block}

VARIABLE  dyn_matrix  :    ARRAY [1..dyn_lim, 1..dyn_lim] OF REAL;
```

3.2. Expressions

An expression is a formula that describes the computation of a value. The value has a (standard) data type which depends on the types of the operands and the operators occuring in the expression. We do not regard expressions and operations on records and pointers here (they are treated in Section 3.7.4.). The OF operator within *primary* below will be introduced in Section 4.2.2.3. All other operators are known from other programming languages.

3.2.1. Arithmetic Expressions

An arithmetic expression results in an INTEGER or a REAL value. The nonterminal *simple_real_expression* denotes arithmetic expressions in the following syntax:

```
simple_real_expression  ::= ...
|       [unary_operator] term [adding_operator ...]

adding_operator  ::=
        + | -

term  ::=
        factor [multiplying_operator ...]

multiplying_operator  ::=
        * | / | // | MOD

factor  ::=
        primary [** ...]

primary  ::=
        identifier [OF identifier]
|       number
|       (simple_real_expression)

identifier  ::=
        {name [ ⟦ simple_real_expression [, ...] ⟧ ] [actual_parameters] } [. ...]
```

Examples:

```
  2.0
   a            {a is REAL or INTEGER variable}
(b [3])         {b is REAL or INTEGER array}
 f (3)          {f is INTEGER procedure}
```

Numbers, variables and constants, calls of procedures and services returning one result value, accesses to elements of structured objects (arrays, records) and INTEGER or REAL expressions in parentheses can appear as operands. There are two unary operators for arithmetic expressions: '+' and '-':

unary operator	type of operand	type of result
+	INTEGER	INTEGER
	REAL	REAL
-	INTEGER	INTEGER
	REAL	REAL

Examples:

```
-a
+2.0                    {equivalent to 2.0}
-(a+b)
-(5*matrix [6, 1])
-rank ('a')             {rank is a standard procedure}
```

Binary operators in arithmetic expressions may be addition operators ('+', '-') and multiplication operators ('*', '/', '//', '**', MOD). Arithmetic expressions can appear as their operands. The following tables show the admissible operators and the resulting types of the expressions, which depend on the used operand types:

Addition:

+	INTEGER	REAL
INTEGER	INTEGER	REAL
REAL	REAL	REAL

Subtraction:

-	INTEGER	REAL
INTEGER	INTEGER	REAL
REAL	REAL	REAL

Multiplication:

*	INTEGER	REAL
INTEGER	INTEGER	REAL
REAL	REAL	REAL

Division:

/	INTEGER	REAL
INTEGER	REAL	REAL
REAL	REAL	REAL

Integer division:

//	INTEGER	REAL
INTEGER	INTEGER	INTEGER
REAL	INTEGER	INTEGER

Modulo:

MOD	INTEGER	REAL
INTEGER	INTEGER	INTEGER
REAL	INTEGER	INTEGER

Exponentiation:

**	INTEGER	REAL
INTEGER	REAL	REAL
REAL	REAL	REAL

Expressions, when used as operands, need only be parenthesized if the desired sequence of computation is not achieved by precedence rules. Parentheses are permitted for means of clearness, too. Expressions with operators of the same precedence will be evaluated from left to right.

For normal division, integer division and the modulo operation one has to observe that the operand on the right hand side may not be equal to 0. Integer division and modulo demand operands of type INTEGER. Here REAL values are rounded and converted to INTEGER values.

Examples:

```
-(a+b)
a*c+b                   {will be evaluated just like (a*c)+b     }

e**f*g                  {will be evaluated just like (e**f)*g    }
e**f**g                 {will be evaluated just like (e**f)**g   }

-3+4+5-1                {yields the same result as               }
(((-3)+4)+5)-1

3.4 MOD 1.5             {will be evaluated like ...              }
3 MOD 2                 {and yields 1                            }

2.5 MOD 2.4             {will be evaluated like ...              }
3 MOD 2                 {and yields 1                            }
```

Exponentiation with negative values in both arguments is not checked by the HI-SLANG compiler, but depends on the run time system of the SIMULA compiler.

3.2.2. CHARACTER Expressions

CHARACTER expressions yield a value of type CHARACTER. Operands of them may be CHARACTER constants or CHARACTER variables, CHARACTER elements of structured objects, the standard procedure *char*, or calls of procedures and services which result in exactly one CHARACTER value. The operands of a CHARACTER expression may be parenthesized. Of course INTEGER expressions may occur within CHARACTER array indices and expressions of the corresponding type may serve as actual parameters of CHARACTER procedures (see syntax of *identifier*).

```
simple_expression  ::= ...
|       character
|       identifier
|       (expression)
```

Examples:

```
'a'
char (10)               {the standard procedure char yields the 10th        }
                        {ASCII or EBCDIC character in this case }
(char_variable)

char (rank (f (x))      {rank is the invers of char;  (f is a CHARACTER function) }

char_array [x+y]
```

3.2.3. TEXT Expressions

TEXT expressions yield a value of type TEXT. Operands of them may be string constants (with a maximal length of 70 characters) or string variables, TEXT elements of structured objects, and calls of procedures and services which result in exactly one TEXT value. The operands of a TEXT expression may be parenthesized. The only operator in TEXT expression is the binary concatenation operator '&'.

```
simple_text_expression  ::=
    simple_text [& ...]


simple_text  ::=
    string
|   identifier
|   (simple_text_expression)
```

Examples:

```
"texts"
"tex" & "ts"                {="texts"}
("texts")
(("tex") & ("ts"))
days [1] & "DAY"            {="SUNDAY"; see also examples in Section 3.1.2.}
"H" & "A" & "L" & "T"       {="HALT"}
```

3.2.4. BOOLEAN Expressions

Boolean expressions result in TRUE or FALSE. They may contain operator NOT or binary operators, especially relational operators:

```
boolean_expression ::=
        TRUE
|       FALSE
|       disjunction [EQV disjunction]

disjunction ::=
        conjunction [or_else ...]

or_else ::=
        OR [ELSE]

conjunction ::=
        {[NOT] relation } [and_then ...]

and_then ::=
        AND [THEN]

relation ::=
        simple_expression [relational_operator  simple_expression]

relational_operator ::=
        = | < | > | <= | >= | <> | #

simple_expression ::= ...
|       identifier
|       ( boolean_expression )
```

Operands of boolean expressions are constants or variables of type BOOLEAN, boolean elements of structured objects, relations and calls of procedures and services that yield exactly one boolean value. The keywords TRUE, FALSE are also allowed as boolean constants.

Examples:

```
VARIABLE bit      :  BOOLEAN;
   bit_vector     :  ARRAY [1..8] OF BOOLEAN;

   TRUE
   bit
   bit_vector [1]
```

The only applicable unary operator in boolean expressions is the NOT operator. It negates the value of a boolean operand, which must be a boolean expression that may be parenthesized.

NOT	TRUE	FALSE
	FALSE	TRUE

Examples:

```
NOT   bit_vector [1]
NOT   (a=b)
NOT   (TRUE)
```

Note:

The keywords TRUE and FALSE, when used as operands must be parenthesized.

Binary operands in boolean expressions may either be relational operators ('=', '<', '>', '<=', '>=', '<>', and '#') or logical operators ('EQV', 'AND', 'OR'). Logical operators allow only boolean expressions as operands . Relational operators allow arithmetic expressions, CHARACTER or TEXT expressions; '=', '<>' and '#' also allow boolean expressions. Terms consisting of relational operators and their operands are boolean expressions.

EQV / =	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	TRUE

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

<>, #	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	FALSE

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

The operators EQV or '=' compare two boolean expressions for equivalence and yield either TRUE or FALSE. The operators '<>' and '#' (which may be read as XOR) compare two boolean expressions for inequality and yield either TRUE or FALSE. The AND operator delivers TRUE, if all operands have the value TRUE. The OR operator delivers TRUE, if at least one operand has the value TRUE. All operands of a boolean expression are evaluated. The sequence of evaluation depends on the HIT installation. This may cause side-effects.

To avoid these side-effects the operators AND THEN and OR ELSE are provided. Their semantics correspond to those of AND and OR, respectively, but only the first operand is evaluated if it defines the result of the expression clearly. Using AND THEN leads to the next operand not being evaluated unless the previous one results in TRUE. OR ELSE leads to the next operand not being evaluated unless the previous one results in FALSE. Boolean expressions with the operators AND THEN and OR ELSE are called conditional boolean expressions.

Examples:

```
VARIABLE  flag1, flag2 : BOOLEAN;

flag1         EQV   flag2
                                  {= TRUE, because flag1 and flag2 are initialized by FALSE}

flag1         AND   (TRUE)                                              {= FALSE}
flag1         OR    flag2     OR    (TRUE)                              {= TRUE}
( flag1       AND   flag2 )   OR    (TRUE)                              {= TRUE}

flag1         AND   THEN      flag2          {= FALSE; 2. operand will not be evaluated}
NOT flag1     OR    ELSE      NOT   flag2    {= TRUE  ; 2. operand will not be evaluated}
```

Comparing two operands requires them to be of the same type, except for the comparison of INTEGER and REAL numbers. In this case, the INTEGER number is converted to REAL and the comparison deals with REAL values. The result of a comparison always is of type BOOLEAN.

Operator	Meaning		
=	TRUE	:	both operands are equal
	FALSE	:	else
<	TRUE	:	the left operand is less than the right one
	FALSE	:	else
>	TRUE	:	the left operand is greater than the right one
	FALSE	:	else
<=	TRUE	:	the left operand is less than or equal to the right one
	FALSE	:	else
>=	TRUE	:	the left operand is greater than or equal to the right one
	FALSE	:	else
<>, #	TRUE	:	both operands are not equal
	FALSE	:	else

Two real expressions should not be compared using '=' or '<> (resp. '#') (because of possible inaccuracies in REAL representations).

The comparison of TEXT expressions depends on the internal character coding. Two text elements are equal if they coincide in their length and in all of their characters. Text comparison means that all characters of the text are compared, regarding their respective positions. Text elements are compared from left to right. The first different character determines the decision which of the text strings is smaller or greater.

The text with the smaller character with regard to character coding, or the text with less characters, is the smaller text.

Examples:

```
        "TEXTS"          >     "TEXT"
         "text"          <>    "TEXT"
         "texts"         >     "texti"
          "abc"          <>    "abc "      {note the blank}
       "Attention"       <     "C"
       "attention"       <     "C"         {the EBCDIC case    }
       "attention"       >     "C"         {the ASCII case     }
"T"&"E"&"X"&"T"&"S"      =     "TEXTS"
          "A"            >     ""
```

All of the above examples result TRUE. CHARACTER expressions can also be compared with each other based on the internal coding:

Examples:

```
CONSTANT a, c    :     INTEGER DEFAULT 2;
CONSTANT b       :     BOOLEAN DEFAULT TRUE;

    'a'          =     'b'                          {= FALSE}
     a           <      c                           {= FALSE}
    a+1          >=    c+1                          {= TRUE}
     a           >      c          AND    b         {= FALSE}
b OR   (c+3      <      5)                          {= TRUE}
  "text"&"s"     <>   "text"                        {= TRUE}
   "text"        <    "texts"      AND    b         {= TRUE}
```

Examples:

```
VARIABLE      i  :  INTEGER    DEFAULT 2;
              j  :  INTEGER    DEFAULT 4;
              k  :  REAL       DEFAULT 0.5;

j <> 0  AND THEN  i/j <> k                  {= FALSE; 2. operand will be evaluated}
```

Just as in arithmetic expressions, operands in boolean expressions can be parenthesized if the desired sequence of evaluation differs from that specified by precedence rules. Parentheses for the purpose of clearness are allowed. Boolean expressions with operators of the same precedence are evaluated from left to right.

3.2.5. Precedence Rules for Evaluating Expressions

Precedence rules determine the process of computing an expression. The syntax of HI-SLANG determines the following priorities for the operators of HI-SLANG:

(1)	=	lowest priority	EQV
(2)			OR, OR ELSE
(3)			AND, AND THEN
(4)			NOT
(5)			=, <, >, <=, >=, <>, #
(6)			+, -, &
(7)			*, /, //, MOD
(8)			**
(9)	=	highest priority	(expression), OF

Operators of higher priority are applied prior to operators of lower priority. Operators of the same precedence are applied from left to right. Expressions embraced in parentheses are of highest priority. Expressions within parentheses are also computed according to the precedence order described above. Using parentheses, the programmer is able to define the desired order of subexpression evaluation.

Examples:

```
VARIABLE        a, b, c:INTEGER DEFAULT 3;

a+b*c  {yields the value 12}
(a+b)*c        {yields the value 18}
a*b*c          {yields the value 27}
(a*b)**c       {yields the value 729}
```

3.3. Statements

This chapter describes the application of assignment statements, conditional statements, loop statements and block statements. Other statements, such as procedure calls, result assignments, input and output statements etc., will be described in following chapters.

```
sequence_of_statements ::=
        statement [ ...]


statement ::= ...
|       simple_statement
|       compound_statement


simple_statement ::= ...
|       assignment_statement
|       empty_statement


compound_statement ::= ...
|       conditional_statement
|       loop_statement
|       block_statement


empty_statement ::=
        ;
```

The execution of an empty statement entails no action. The statements in a sequence of statements are executed one after the other.

3.3.1. Assignment and Type Conversion

An assignment replaces the actual value of a variable or of an element of a structured object by a new value described by an expression or an aggregate. By using a multiple assignment, one value can be assigned to several variables or elements of structured objects simultaneously. Assignments involving records or pointers are not treated in this section. They are dealt with in Section 3.7.4.

The variables or elements of a structured object on the left hand side of an assignment and the expression or aggregates on its right hand side must be of the same type or, otherwise they must be convertible.

Assignments made to elements of a structured object require the object to be defined as a variable. Assignments to array variables demand consideration of the array´s dimension and its index bounds.

```
assignment_statement  ::= ...
|       common_assignment

common_assignment  ::=
        identifier [, ...] := expression_or_aggregate ;
```

An assignment will be executed by computing the expression on the right hand side first. Next, the computed value is given to the variable, unless the variable denotation contains expressions like array indices or procedure calls. These expressions determining the indices of the array element are calculated first (i.e., before computing the right hand side expression and performing the assignment). Of course this calculation has to yield values within the array bounds.

Multiple assignment is accomplished in the following manner: the first variable on the left side of the assignment operator ':=' acquires the value of the right hand side. Now, from right to left, the variables on the left hand side are successively assigned the new value, the latest updated variable serving each time as the right hand side of the next assignment. The same is valid for assigning a value to array elements.

Type conversion is allowed for INTEGER and REAL variables only. The conversion from INTEGER to REAL will not change the interpretation of the value as real number. The conversion from REAL to INTEGER occurs by rounding the REAL value to an INTEGER value and requires the REAL value to be representable as INTEGER.

Note:

If there are array variables or expressions in the list on the left and right hand side of a multiple assignment, then all indices and expressions will be computed first (left to right) before any assignment is performed (as seen in the examples).

Examples:

```
VARIABLE          i:      INTEGER;
                  x, y:   REAL;
                  a, b, c:  ARRAY [1..6] OF INTEGER;

x, i, y   :=  1.2;                      {corresponds to:}

    y     :=  1.2;
    i     :=  y;                        {i = 1,    conversion REAL → INTEGER}
    x     :=  i;                        {x = 1.0, conversion INTEGER → REAL}

a[i], i   :=  a[i] + 4;                 {corresponds to:}

    i     :=  a[i] + 4;                 {     i   =  4, since a[1] = 0 }
 a[1]     :=  i;                        { a[1]   =  4, since index is evaluated first }

    c     :=  [1, 2, 3, 4, 5, 6];
    a     :=  0;                        {all array elements of a are set to 0}
 a, b     :=  c;
```

3.3.2. Conditional Statements

Conditional statements offer the possibility of controlling the execution of the program with regard to one or more conditions.

conditional_statement ::=
    if_statement
| case_statement
| branch_statement

3.3.2.1. IF Statement

In an IF statement, either one sequence of statements or no statements at all are selected for execution. The selection depends on the value of a boolean expression.

if_statement ::=
    IF boolean_expression
    THEN sequence_of_statements
    [**ELSE** sequence_of_statements]
    END IF ;

An IF statement is executed by computing the boolean expression following the keyword IF first. If the evaluation of the expression results in TRUE, the statements of the THEN branch are carried out. If the result is FALSE and an ELSE branch exists, the statements of the ELSE branch are executed. The IF statement is terminated by END IF. Both the THEN branch and the ELSE branch may in turn contain IF statements themselves. The IF-END IF pairs clearly define the nesting structure.

Note:

Since every statement is terminated by a semicolon, there is always a semicolon before ELSE or END IF.

Examples:

```
    VARIABLE a, b, c : INTEGER;

IF a < b THEN                                   {1}
    a := a + 1;
    IF a = b THEN                                   {2}
        a := 0;
    ELSE                                            {2}
        IF  a < b AND c > 0 THEN                        {3}
            c := 0;
        END IF;                                     {3}
        c := c + 1;
    END IF;                                         {2}
ELSE                                            {1}
    a := a - 1;
END IF;                                         {1}
```

3.3.2.2. CASE Statement

A CASE statement selects either one or no statements from a choice of several alternative (sequences of) statements. The selection depends on the value of an INTEGER, CHARACTER or TEXT expression. The expression may not be of the type REAL or BOOLEAN. In the sequence of statements to chose from, statements of any kind except EVALUATE are allowed.

```
case_statement ::=
 CASE      simple_expression
 {WHEN     simple_expression [, ...] :
           sequence_of_statements } [ ...]
 [ELSE :   sequence_of_statements]
 END CASE ;
```

A list of expressions (or one expression) following WHEN precedes each eligible sequence of statements. The values of these expressions are compared with the value of the expression behind the keyword CASE. If the first of these comparisons become TRUE only the corresponding sequence of statements will be executed. A list of expressions following a WHEN is implicitly connected with OR-operations.

All of these expressions must deliver a value of type INTEGER, CHARACTER or TEXT. The type of the expressions behind the keyword WHEN must correlate to the type of the expression behind the keyword CASE, the latter being computed first. As a consequence an error message is yielded (also in the case of INTEGER and REAL type clash, where usually a type conversion takes place).

If the evaluation of the expression behind the keyword CASE yields a value that does not exist in any of the WHEN lists, the ELSE branch is executed (supposing it exists, otherwise the empty statement will be executed). The ELSE branch necessarily forms the last part of the CASE statement.

Examples:

```
VARIABLE        i1, i2  :  INTEGER;
                   t, c  :  TEXT;

CASE  (0.4 * (i1 + i2)) // 5
    WHEN      1, 3  :  i1, i2  :=   0;
    WHEN      2, 4  :  i1, i2  :=   1;
    WHEN         0  :  i1, i2  :=  -1;
    END CASE;

    CASE t & c
    WHEN "text"           :  t  :=  "text" & "s";
    WHEN "text"&"s"       :  c  :=  "";
    ELSE                  :  t  :=  "tex"; c  :=  "t";
 END CASE;
```

3.3.2.3. BRANCH Statement

The BRANCH statement serves to select either one or no sequence of statements out of a choice of several alternatives. The selection is based on probabilities.

```
branch_statement ::=
        BRANCH
        {PROB simple_expression   : sequence_of_statements } [ ...]
        [ ELSE                    : sequence_of_statements ]
        END BRANCH;
```

Each alternative sequence of statements within the PROB statement is preceded by a REAL expression. Its value determines the probability of selecting the alternative. Consequently the value of the expression must be within the interval [0..1]. The sum of all probabilities has to be equal to or less than 1.0, otherwise an error message occurs.

The ELSE branch is optional. If there is one, it covers the difference between the sum of probabilities and 1.0. This difference may be 0.0, but not negative. The statements of the ELSE branch are selected depending on this difference in probability. If there is no ELSE branch, no statement will be selected in this case.

Please note that the expressions (simple_expression) are evaluated in accordance to the order of appearance but only until the sum of the expressions is greater than the drawn random number.

Examples:

```
VARIABLE    a : REAL DEFAULT 0.7;
            b : REAL DEFAULT 1.0;

   BRANCH
   PROB    0.3 :   a  :=  0.5;
   PROB      a :   a  :=  0.6;
END BRANCH;

BRANCH
   PROB    (b-a)/2+0.1    :  a  :=  0.4;
   ELSE                   :  a  :=  0.3;
END BRANCH;

a  := 0;

BRANCH
   PROB    a  :  b  :=  1;    a  :=  0;
   PROB    b  :  b  :=  0;    a  :=  1;
END BRANCH;
```

3.3.3. LOOP Statements

The LOOP statement determines, whether the sequence of statements within a so called basic loop is not performed at all, performed exactly once, or performed several times.

```
loop_statement  ::=
        infinite_loop
|       while_loop
|       until_loop
|       for_loop
|       times_loop
```

3.3.3.1. Infinite Loop

The infinite loop combines statements that are to be repeated without termination. Such an infinite loop can only terminate due to external conditions as for example the end of requested cpu time or model time in case of simulative experiments.

```
infinite_loop  ::=
        basic_loop ;
```

```
basic_loop  ::=
        LOOP
                sequence_of_statements
        END LOOP
```

The infinite loop is nothing else than the *basic_loop*, which is part of any other loop statement. The infinite loop is important for modelling reasons, and it should be used mainly to call time consuming services.

Examples:

```
VARIABLE sum, run : INTEGER;

LOOP
    sum    :=  sum + run;
    run    :=  run + 1;
    spend  (sum / run);
END LOOP;
```

3.3.3.2. WHILE Loop

The WHILE loop causes the repeated execution of a sequence of statements as long as the boolean expression following the WHILE keyword results in TRUE. This expression represents the termination criterion of the WHILE loop.

```
while_loop ::=
    WHILE boolean_expression basic_loop;
```

The termination criterion defined by the boolean expression is evaluated and checked before each repetition. If the result is TRUE, the sequence of statements will be executed. If the result is FALSE, the execution of the WHILE loop terminates. The END LOOP of the *basic_loop* syntactically terminates the WHILE loop. If the first evaluation of the termination criterion already results in FALSE, the WHILE loop is not executed at all.

A WHILE loop can cause an infinite loop if the termination criterion is not altered within the loop, or it is altered in such way that the value of the terminating condition is never FALSE. In case of simulative model evaluation, such a statement may lead to a program that runs infinitely, unless the WHILE loop is terminated by an external condition.

Example:

```
VARIABLE sum, run : INTEGER;

run := 1;

WHILE sum < 6 * run
LOOP
    sum  :=   sum + run;
    run  :=   run + 1;
END LOOP;
```

3.3.3.3. UNTIL Loop

The UNTIL loop initiates the repeated execution of a sequence of statements as long as the boolean expression following the UNTIL keyword results in TRUE. This expression represents the termination criterion of the UNTIL loop.

```
until_loop ::=
    basic_loop UNTIL boolean_expression;
```

The sequence of statements within the basic loop is executed at least once. After every repetition of the basic loop the termination criterion is checked by evaluating the boolean expression. If the test results in FALSE, the *basic_loop* is run once more. If it results in TRUE, the loop terminates.

Similar to WHILE loops, one can generate infinite loops using UNTIL loops. In case of simulative model evaluation this may cause programs running infinitely.

Example:

```
VARIABLE sum, run : INTEGER;

LOOP
   sum  :=    sum + run;
    run:=      run + 1;
END LOOP UNTIL sum >= 6 * run;
```

3.3.3.4. FOR Loop

The FOR loop initiates a number of executions of the *basic_loop* specified in the head of the FOR loop.

```
for_loop  ::=
        FOR variable-identifier := loop_value_list  basic_loop;


loop_value_list  ::= ...
|   simple_real_expression      STEP     simple_real_expression
                                UNTIL    simple_real_expression
```

First of all, a so-called loop variable is assigned an initial value by using the expression that follows the assignment operator. The loop variable has to be declared in the same block to avoid modification of the loop variable by concurrently executing processes. Next, the arithmetic expressions behind the keywords STEP (the increment value of the FOR loop) and UNTIL (termination value) are evaluated. The *basic_loop* will be gone through only if the increment value is positive (≥ 0) and the initial value $\leq$ termination value or if the increment value is negative (≤ 0) and the initial value $\geq$ termination value. In all other cases, the *basic_loop* will not be executed.

The loop variable is incremented or decremented based on the increment value after every cycle of the loop. The loop terminates when the value of the loop variable is equal to the termination value or exceeds it. All arithmetic expressions to determine the increment value and the termination value are newly computed after every execution of the *basic_loop*. These expressions should contain no function calls that could cause side effects.

The loop variable as well as all three expressions must be of type INTEGER or REAL. Any conversion is based on the type of the loop variable, which must be declared before in the same block. An assignment to the loop variable should occur in the head of the FOR loop only, because assignments to this variable within the *basic_loop* would violate the idea of the FOR loop and may lead to infinite loops. Infinite loops can also occur if variables used within the increment value expression or termination value expression are reassigned within the *basic_loop*. Termination of the FOR loop implies the loop variable remaining at its last received value that caused the loop termination.

Example:

```
VARIABLE sum, run : INTEGER;

FOR    run := -1 STEP -1 UNTIL -13
LOOP
        sum := sum - run;
END LOOP;
```

Using a FOR loop with a value list one can, e.g., easily control evaluation series with non-equidistant or non-arithmetic parameter values:

```
loop_value_list  ::= ...
|        expression [, ...]
```

The statements of the *basic_loop* are run through as long as there are expressions in the list. The respective expression is always evaluated before execution and its value is handed over to the loop variable. This variable has to be declared in the same block and needs to be of a simple data type. All expressions within the expression list either have to be of the same type or they have to be convertible.

Example:

```
EXPERIMENT exp;
        VARIABLE  r_run   :        REAL;
                                   c_run  :        CHARACTER;
        …
BEGIN
        FOR r_run := 0, 0.7, 0.8, 5.7, 5.8
        LOOP

                FOR c_run := 'r', 'w'
                LOOP

                        EVALUATE MODEL m : mod_type (c_run, r_run);
                                …
                        END EVALUATE;

                END LOOP;
        END LOOP;
END EXPERIMENT exp;
```

3.3.3.5. TIMES Loop

A TIMES loop initiates an average number of executions of a *basic_loop*, the average being determined by an arithmetic expression in the head of the loop.

```
times_loop ::=
        AVERAGE simple_real_expression TIMES basic_loop;
```

A TIMES loop is performed as follows: first the arithmetic expression, having to yield an INTEGER or REAL value, is evaluated. The result of the expression determines the average number of repetitions of the *basic_loop* which must be ≥ 0.

The distribution behind AVERAGE is geometric; it is interpreted as

```
x := simple_real_expression; WHILE draw(x/(x+1)) basic_loop;
```

Example:

```
VARIABLE sum, run : INTEGER;

AVERAGE 13 TIMES
LOOP
sum    :=  sum + run;
run    :=  run + 1;
END LOOP;
```

3.3.4. BLOCK Statement

A BLOCK statement opens a new scope for names. It contains declarations and statements.

```
block_statement ::=
        BLOCK
                common_declaration [ ...]
        BEGIN
                sequence_of_statements
        END BLOCK;
```

A BLOCK statement is executed by dealing with the declaration part first and then running the sequence of statements.

The declaration part of a BLOCK statement only allows declarations of variables or constants, records, pointers and procedures. The validity of these objects is limited to the extent of the block indicated by the BLOCK-END BLOCK keywords. Outside of this block the declared names and objects are no longer valid.

BLOCK statements may be nested. This facilitates multiple usage of the same name in several different blocks. Name conflicts can be solved in the same manner as in other higher level languages: the scope of a name is determined by the block the name is declared in, with the exception of all inner blocks in which this name is also declared.

Any kind of statement, except an EVALUATE statement and an AGGREGATE statement, is allowed within a block.

Example:

```
BLOCK                                                          {block 1}
   VARIABLE        a, b  :  REAL;
                   i, j, k  :  INTEGER;
BEGIN
   i   := 1;                                                   {i of block 1}
   k   := 2;                                                   {k of block 1}

   BLOCK                                                       {block 2a}
      VARIABLE i, l : INTEGER;

   BEGIN
      j   :=   10;                                             {j of block 1}
      i   :=    3;                                             {i of block 2a}

      IF i > k THEN                                {i of block 2a, k of block 1}

         BLOCK                                                 {block 3}
            VARIABLE b : BOOLEAN;

         BEGIN
            b := i = j;                  {b of block 3, i of block 2a, j of block 1}
         END BLOCK;                                            {end of block 3}

      END IF;

   END BLOCK;                                                  {end of block 2a}


   BLOCK
      VARIABLE k : INTEGER;                                    {block 2b}

   BEGIN
      i := k + 2;                                {i of block 1, k of block 2b}

                        {i := l + 2  leads to an error, because l is only known within block 2a }

   END BLOCK;                                                  {end of block 2b}

   a := b + i * j;                                {a, b, i and j of block 1}

END BLOCK;                                                     {end of  block 1}
```

3.4. Procedures

Procedures are separated, executable subprograms. They are triggered by procedure calls. They support a structuring of the program because repeatedly occuring statement parts can be combined to one part.

Procedures consist of a declaration part and a statement part. They are therefore regarded to be a block and local elements are consequently not available outside the block. A proper communication with surrounding program parts is only possible by using parameters or USE declarations.

3.4.1. Procedure Declarations

In case of simulative experiments, declarations of procedures are allowed anywhere, provided that declarations are generally permitted. In case of analytical experiments, declarations of procedures are not allowed in the model description.

```
common_declaration ::= ...
|       procedure_declaration


procedure_declaration ::=
        PROCEDURE procedure-name [formal_parameters]
        [RESULT simple_type [, ...]];
                [use_declaration_part]
                [common_declaration [ ...]]
        [BEGIN sequence_of_statements]
        END PROCEDURE [procedure-name];
```

A procedure is divided into the head of the procedure and the body of the procedure. The head contains conventions for calling the procedure (the interface). These are the name of the procedure and its formal parameters which make communication with the environment of a procedure call possible. Calling the procedure causes the formal parameters to be substituted by a corresponding set of actual parameters. All formal parameters can be used in the body of the procedure.

A procedure may contain definitions of results. Procedures returning results may be employed on the right hand side of a special kind of multiple assignment (result value assignment). If they have exactly one result, they may also appear in simple expressions.

The body of a procedure may contain a USE declaration (*use_declaration_part*), a *common_declaration* part and a sequence of statements following the keyword BEGIN. Within the declaration part, only the definition of constants, variables, records and other procedures is allowed. The USE declaration part is not discussed in this section (it is treated in Section 4.1.1.4.).

The running of a procedure does not consume any time under the aspect of modelling. Therefore, only statements which do not consist of time-consuming operations (this means they do not contain service calls) are allowed in a procedure body. Furthermore, special statements for model evaluation (Section 5.) are not permitted, excepting the UPDATE statement.

Within the body of a procedure that returns results, an assignment statement for the result variables should exist. This is the result assignment. If such an assignment is missing or cannot be reached, the default values are handed over to the calling environment.

Examples:

```
    PROCEDURE count_1 (x, y : INTEGER);
        VARIABLE r1, r2 : REAL;
    BEGIN
    r1   :=  1;
    r2   :=  2;

    x    :=  x + r1;                    {These two assignments have no effects to outer parts of }
    y    :=  y + r2;                    {this block; see also transmission modes, Section 3.4.3.  }
END PROCEDURE count_1;


PROCEDURE count_2 (x, y : INTEGER) RESULT INTEGER, INTEGER;
    VARIABLE r1, r2 : REAL;
BEGIN
    r1   :=  1;
    r2   :=  2;

    RESULT x + r1, y1 + r2;
END PROCEDURE count_2;
```

3.4.2. Procedure Calls

Procedures with one result (functions) can be called within expressions and assignments, while calls of procedures without result are statements (simple_statements). Procedures with more than one result can only occur in assignment statements.

```
procedure_or_service_call  ::=
    procedure_or_service-identifier;


    identifier ::=
        {name [ ⟦ simple_real_expression [, …] ⟧ ]   [actual_parameters] } [. …]
```

Please note that the identifier only consists of the name and possibly actual parameters.

3.4.2.1. Procedures without Results

Procedures without results are called by using their names followed by a list of actual parameters, provided that the procedure definition contains a formal parameter part.

```
simple_statement  ::= ...
|   procedure_or_service_call
```

In case of simulative model evaluation (self-defined) procedures may be called wherever statements are admissible. Corresponding to the use of variables, the procedure employed has to be declared within the calling block or in a previous surrounding block.

A procedure call first initiates the transmission of an actual parameter set to the formal parameters in obedience to the parameter transmission modes specified in the head of the procedure. Subsequently, the declaration part is worked off and instances of all local objects are created. Then the statement part is executed. After finishing the last statement the procedure object is removed including its local objects.

Procedures may be activated (either directly or indirectly) recursively. Each recursive procedure call causes a local instantiation of all objects. It is important to terminate the recursive procedure in time, so that memory overflow and run time errors can be avoided.

Example:

```
PROCEDURE count_3 (NAME count : INTEGER; x, y : INTEGER);
BEGIN
        IF x > y THEN
                count := count + 1;
                count_3 (count, x+1, y+2);
        END IF;
 END PROCEDURE count_3;

{Call of count_3  in a statement part somewhere within the program. }

IF a > b THEN
        count_3 (a, b, c);
END IF;
```

3.4.2.2. Procedures Returning Results

Procedures returning results are called in the same way as procedures without results, but in addition procedures with exactly one result may be used as operands in expressions. The type of the procedure has to coincide with or has to be convertable to the type of the expression in which it is called (*primary*, see Section 3.2.1.). It is also possible to call procedures with results on the right hand side of assignments:

```
assignment_statement  ::= ...
|       result_assignment


result_assignment  ::= ...
|       ( identifier [, ...] )  := procedure_or_service_call;
```

Number, type and type sequence of the variables on the left hand side of a result assignment must relate to number, type and type sequence of the variables of the result definition in the head of the procedure (these conditions also apply for the result assignments within the body of the procedure). If the types do not coincide, they have to be convertible.

Results should be defined properly by result statements within the body of the procedure (see Section 3.5.). Otherwise, default result values are put to the disposition of the calling environment.

If the procedure has exactly one result, the variable on the left hand side of an assignment need not be parenthesized. Such an assignment can be considered just like any other "normal" assignment. Here all identifiers (if there is a list) are assigned to the same result value (multiple assignment).

```
result_assignment  ::= ...
|      identifier [, ...]  := procedure_or_service_call;
```

Following the block concept, the called procedure has to be declared within the block it is used in or in a surrounding one.

The call of a procedure with results causes the actual parameters to be transferred to the formal parameters of the procedure, utilizing the parameter transmission mode specified in the head of the procedure. Subsequently the declaration part is worked off. All local objects are instantiated. Now the body of the procedure is executed. Following this execution, the result values (determined by the last executed RESULT statement) are supplied to the environment and the local object instances are eliminated.

Note:

The results of a multi result procedure are implemented as name parameters (see Section 3.4.3.1.2.). Thus the following equivalence holds:

```
PROCEDURE proc ( r1:P1; ...; rm:Pm ) RESULT  T1, ...,Tn;
BEGIN
        …
        RESULT v1, ..., vn;
END PROCEDURE proc;
…

(x1, ..., xn) := proc (y1, ..., ym);
```

:<=>

```
PROCEDURE proc ( r1:P1; ...; rm:Pm; NAME  t1:T1; ...; tn:Tn );
BEGIN
        …
        t1:=v1; ...; tn:=vn;
END PROCEDURE proc;
…

proc (y1, ..., ym, x1, ..., xn);
```

where P1,…,Pm and T1,…,Tn are symbols for type names.

Calls of single result procedures cause an execution of the procedure, the resulting value is used for further evaluation.

Procedures with results can be called (directly or indirectly) recursively. Each recursive procedure call causes the instantiation of all local objects. It is important to terminate the recursive procedures in time to prevent memory overflow and run time errors.

Examples:

```
VARIABLE      a, b  :   INTEGER;
                  c  :   ARRAY [1..10] OF INTEGER;

PROCEDURE count_4 (x, y : INTEGER) RESULT INTEGER;
BEGIN
    IF x > y THEN
            RESULT count_4 (x+1, y+2);
    ELSE
            RESULT 2 * y;
    END IF;
END PROCEDURE count_4;

{ Call of count_2  (example of Section 3.4.1.) }
{ and count_4 in a statement part with a=0,b=0 and c[4]=6 }

(a, b) := count_2 (a, b);                                                    {a = 1, b = 2}

IF b > a THEN
    c [count_4 (b, a)] := c [count_4 (b, a) - 2] * count_4 (b, a);     {count_4 (b, a) = 6, c [6] = 3}
END IF;
```

3.4.3. Parameters and Transmission Modes

In HI-SLANG, procedures as well as modelling objects like services, model types and component types can communicate with the calling environment by the way of parameters.

In the head of a procedure declaration or of one of the mentioned modelling objects, this communication interface is defined by specifying a list of formal parameters. In calling such a parameterized procedure or modelling object, a list of actual parameters can be handed over which substitutes the list of formal parameters. These actual parameters must comply with the conventions specified in the formal parameter definition (number and type of the variables in the parameter list, transmission modes). Please note that integer and real arrays are not compatible.

It must be secured that every formal parameter receives a value either by its default value or by the value of its corresponding actual parameter. For the application of records or pointers as parameters refer to Section 3.7.7.

3.4.3.1. Parameter Transmission Modes

Every parameter requires a description of its possible employment. This is achieved by specifying one of the three available transmission modes.

```
formal_parameter ::=
    ({[mode] parameter_declaration} [; ...])


parameter_declaration ::=
    [VARIABLE]  parameter-name [, ...] :
    [ARRAY OF]  simple_type
    [DEFAULT    expression_or_aggregate]
|   RECORD      parameter-name [, ...] :
    [ARRAY OF]  recordtype-name


mode ::=
    VALUE
|   NAME
|   REFERENCE
```

The different transmission modes depend upon the data types of the corresponding parameters. Using a transmission mode for a parameter which does not comply with the data type of the latter leads to a compiler error message. Specific transmission modes are either preset (**D**EFAULT), optional (**O**PTIONAL) or not eligible (**I**LLEGAL) for the different data types. The following table illustrates this:

data type	transmission mode		
	by value	by name	by reference
INTEGER, REAL, BOOLEAN, CHARACTER, TEXT	D	O	I
ARRAY OF INTEGER, REAL, BOOLEAN, CHARACTER, TEXT	O	O	D
(ARRAY OF) INFILE, OUTFILE, POINTER, RECORD	I	O	D

If the default mode is employed for defining the transmission mode of a parameter, the specification of the keyword NAME, VALUE or REFERENCE becomes obsolete.

3.4.3.1.1. Call by Value

The formal parameter specified by the mode VALUE (*call by value*) receives a duplicate value of the actual parameter, which may be an expression. The expression is evaluated when the procedure is called resp. the modelling object is instantiated.

The actual value of a VALUE parameter is not altered by using the formal parameter in the statement part within the body of the procedure (resp. the modelling object). Consequently the value, being assigned to a parameter value, is accessible only within the body of the procedure or the modelling object .

Parameters of the standard type INTEGER, REAL, BOOLEAN, CHARACTER and TEXT have the transmission mode call by value as default. The keyword VALUE needs not be specified within the parameter definition. The mode VALUE is also allowed for array parameters. However, it should be applied for "small" arrays since, as mentioned before, value parameters are copied entirely. Record and pointer parameters do not allow the transmission mode VALUE.

Example:

```
PROCEDURE count_1a (VALUE x, y : INTEGER);
BEGIN
        x := x + 1;
        y := y + 2;
END PROCEDURE count_1a;
```

The procedure declaration *count_1a* is equivalent to *count_1* (see the example in Section 3.4.1.). The value assignments to *x* and *y* within the body of the procedure do not have any effect outside of the procedure.

3.4.3.1.2. Call by Name

If the parameter variable is declared using the keyword NAME (*call by name*), the value it receives within the statement part of the procedure is also valid outside of the procedure.

Actual parameters passed to NAME-declared formal parameters substitute them textually. If the formal parameter variable is changed somewhere within the procedure, we have to ensure that its corresponding actual parameter is not a constant or expression. This is also true for DEFAULT value expressions declared for formal parameters. If the actual parameter is an expression, it will always be evaluated when the corresponding formal parameter is used within the procedure.

Example:

```
PROCEDURE count_1b (NAME x, y : INTEGER);
BEGIN
        x := x + 1;
        y := y + 2;
END PROCEDURE count_1b;
```

After having executed the procedure *count_1b (a, b), a* will be incremented by 1 and *b* will be incremented by 2. A call just like *count_1b (a+1, b)* is erroneous and it effects a SIMULA run time error because it causes the substitution a+1 := a+1+1, which is not permitted.

The transmission mode *call by name* is allowed for parameters of any type available in HI-SLANG. It may only be used for formal parameters of procedures.

3.4.3.1.3. Call by Reference

The mechanism *call by reference*, indicated by the keyword REFERENCE, is allowed only for arrays, records and pointers. In this case, when the procedure is called or the modelling object is instantiated, the formal parameter variable receives a reference or a pointer to the actual parameter. Not the reference to the object but its contents are altered within the body of the procedure (resp. modelling object).

3.4.3.2. Arrays as Formal Parameters

Formal array parameters of procedures (or modelling objects) are declared without bounds and dimension. The first access (textual appearance (!)) of this array parameter (assignment via actual parameter or default) determines bounds and dimension. Therefore, one has to avoid an inconsistent access for multiple procedure calls resp. modelling object instantiations.

And also an access with incorrect dimension or indices to such an array within the body of the procedure (resp. modelling object) should be avoided (leading to a run time error). Checking the array bounds and dimension is possible via the predefined array attributes (see Section D.3.1.).

3.4.3.3. Default Values for Parameters

The declaration of formal parameters is comparable to the declaration of variables, since they, too, can be used within the body of the procedure or the parameterized modelling object they belong to. Just like variables, they can receive default values.

The expressions used to define the default (*expression_or_aggregate*) may not contain variables declared in the same block or formal parameters of the procedure (or modelling object). A DEFAULT value is computed when the procedure is called (or modelling object is instantiated) and no actual parameter is given for the formal parameter. All elements of a list of formal parameters with one default expression receive the same default value, the default expression is evaluated once. The evaluation of the actual parameters resp. the corresponding default expression is performed in the order of the formal parameters. DEFAULT values for records are not allowed. Formal parameter arrays may not be used as DEFAULT for a formal parameter, because the dimension of the DEFAULT must be fixed at compile time.

When a formal array parameter receives default values by an aggregate expression and no actual parameter is given, the array attributes of dimension and maximal length of the index range are specified by those defaults (If those defaults do not imply the specification of dimension and bounds an error occurs.). The index range is defined from 1 to number of array elements in this case. A list of formal array parameters is handled equally.

Example:

```
VARIABLE    a     :   ARRAY [1..2] OF INTEGER;

PROCEDURE f (b    :   ARRAY OF INTEGER DEFAULT [1, 2, 3]);
...
END PROCEDURE f;
```

The call of *f(a)* effects the substitution: b [1] ":=" a [1] and b [2] ":=" a [2]. In the statement part of the procedure, *b [3]* is not accessible.

3.4.3.4. Specification of Actual Parameters

The actual parameter list may consist of positional parameters, in correspondence to other higher level programming languages, or keyword parameters, often to be found in command languages. But multiple assignments of a formal parameter (via positional and (one or more appearance of one) keyword parameter) is not allowed.

actual_parameters ::=
({ [[**LET** parameter-name :=] expression_or_aggregate] } [, ...])

If a formal parameter does not have a default value, it must be assigned an actual parameter. If it does, an actual parameter need not necessarily be assigned. An empty positional parameter is only possible if a default value does exist.
Positional parameters may not appear following keyword parameters. Keyword parameters have the advantage that the actual parameters are commented (by the formal parameter name), and that they can be given in any order. The evaluation of the actual parameters is performed in the order of the formal parameters, not the textual order of the keyword parameter.

Examples:

```
PROCEDURE p1    (i, j, k, speed : INTEGER   DEFAULT 5;
                 m, token, r: TEXT          DEFAULT "HELLO" );
```

```
{possible procedure calls:}
    p1;                                                    {All defaults will be used}
    p1 (6, 3+s, a*b, 0);                          {The defaults of the TEXT parameters will be used}
                                                  {The INTEGER parameters will be assigned actual values}
    p1 (,,,8, "TEXT",, "END");                    {Substitution of the defaults of speed, m and r }
    p1 (   LET speed := 6, LET token := "TEXT"); {speed and token get an actual value; equivalent to:}
    p1 (,,,LET speed := 6, LET token := "TEXT");
    p1 (8, a*b, LET token := "END");                       {i, j and token get an actual value}
```

3.5. RESULT Statement

In a RESULT statement, the result variables of procedures or services receive values that are assigned to them by simple expressions. These values are handed over to the environment that has activated the procedure (resp. service).

```
simple_statement  ::= ...
|       result_statement

result_statement  ::=
        RESULT expression [, ...];
```

RESULT statements may only appear in the body of a procedure or service. If more than one RESULT statements are actually executed, only the last one determines the result values.

If no RESULT statement appears in the procedure (resp. service) body or no RESULT statement is executed, the value which will be handed over to the environment is the default value belonging to the type of the result. The expressions determining the results must comply with the number, type and type sequence of the results specified in the head of the procedure (or service). All simple data types are allowed for results.

If the procedure (or service) call is not on the right hand side of an assignment or within an expression, the result values are lost.

Example:

```
VARIABLE      a, b  :  INTEGER;
              flag  :  BOOLEAN;

PROCEDURE f (x, y : INTEGER) RESULT INTEGER, BOOLEAN;
BEGIN

        IF x <> y THEN
              RESULT 2*y, TRUE;
        ELSE
              RESULT 2-y, FALSE;
        END IF;

    END PROCEDURE f;
    ...

    f (a, b);                   {results are lost}
    (a, flag) := f (a, b);      {results assigned to a and flag}
```

3.6. Text Processing and I/O

Internal texts are objects of data type TEXT. They are admitted as operands in TEXT expressions which already have been introduced.

Files are objects of the type INFILE or OUTFILE. They serve as sequential, line structured external data stores accessible in HI-SLANG programs. They may be connected to external files or to objects within a modelling base (see Section 8.).

```
simple_type  ::= ...
|       INFILE
|       OUTFILE
```

Like other HI-SLANG objects, files must be defined as variables. It is possible to declare several file variables at a time. Furthermore, array variables can be declared of type INFILE or OUTFILE. Just like variables of other types, files can be used as parameters.

Files can either be of read-only mode or of the write-only mode. Read-only files have to be of type INFILE. These are input files. Write-only files are required to be of type OUTFILE. These are output files. Files have to be opened before access to them is possible. They must be closed after the last access.

```
simple_statement  ::= ...
|       io_statement


io_statement  ::=
        read_statement
|       write_statement
|       open_or_close_statement
```

3.6.1. Structure of Files

HI-SLANG files are structured in lines. They can be accessed sequentially. The maximum length of a line is declared at the point of opening the file. A line supposed to be processed in the buffer may not be longer than this line length, otherwise a FAN error could occur. After having opened a file, an internal buffer of the size of the line length is prepared. The maximally possible value for this length depends on the installation.

Every file has an internal position pointer for the internal buffer. Regarding INFILE files, the position pointer is always directed at the next character not yet been read within the line currently in the buffer. Regarding OUTFILE files, the position pointer refers behind the last written character in the buffer. The position pointer is set by READ and WRITE statements following this convention. When the position pointer reaches the end of the buffer, the next INFILE line is read or the OUTFILE line is written, setting the position pointer to 1.

3.6.2. File State Queries

Working on files of type INFILE, the boolean standard procedures *eof(f), lastitem (f)* or *eoln(f)* can be applied. *eof(f)* tests whether the end of the file *f* has already been read. *eoln(f)* tests whether the end of the current buffer line has already been reached. If the file variable *f* is missing, these procedures will refer to the standard file *sysin*.

- eof: The standard procedure *eof*(f) results in TRUE if the last character of *f* has been read from the stream, or if the file *f* has not yet been opened or already been closed. Please note that *eof* is FALSE immediately after opening the file, even if the file is empty.

- l**astitem:** The procedure *lastitem(f)* causes all blanks, TABs and end-of-line signs between the position of the pointer and the next non-blank or end-of-file sign to be overread. The position pointer stops at the next non-blank character or at the end of the file. If the position pointer stops at a non-blank character, *lastitem(f)* delivers FALSE, otherwise TRUE. The result is also TRUE if the file *f* has not been opened yet or has already been closed again. For controlling the reading of numerical items please use *lastitem* instead of *eof*.

- **eoln:** The procedure *eoln(f)* results in TRUE if no more characters of the current line of *f* can be read from the buffer. Procedure *eoln(f)* also delivers TRUE if the file *f* has not been opened yet or has already been closed again.

3.6.3. Opening and Closing Files

A file has to be opened before the first and closed after the last access taken.

```
open_or_close_statement ::=
    OPEN file-identifier, simple_text_expression
        LENGTH simple_real_expression;
|   CLOSE file-identifier;
```

An OPEN statement must contain a file identifier and a link name, both separated by a comma. The file identifier must be a variable of type INFILE or OUTFILE. The link name, specified by a simple text expression, forms the connection between a logical HI-SLANG file and a physical file or an object within a modelling base (see Section 8.2.2.).

The expression to determine the line length must be of type INTEGER or REAL (which is converted to INTEGER).

If more than one INFILE objects are opened that all refer to one physical file or object by using the *%BIND* statement (see the CONTROL part in Chapter 8.), all INFILE variables can be used for access to that file simultaneously and independently. Every single INFILE stream, though, can only be read sequentially.

The same mechanism applies for OUTFILE variables. All OUTFILE objects write their lines into the one physical file or object they are bound to. This may cause merging, the FAN system will warn the user in that case.

If no OUTFILE or INFILE object relating to such a file of joint access is open any longer, the latter can be opened and manipulated again using the same link name, even changing from INFILE to OUTFILE or vice versa.

Executing an OPEN statement causes the creation of an internal buffer with the specified size. Regarding INFILEs, the position pointer points to the end of the buffer (*eoln(f)* results in TRUE). Regarding OUTFILEs, the position pointer points to the beginning of the buffer. Opening an OUTFILE object will delete an already existing physical file first, unless the EXTEND mode (see the control part of HIT in Chapter 8.) is used. Linking a file by using the EXTEND mode causes the new output to be appended.

If a line of a physical file is longer than the buffer of the linked logical one, the run time system detects an error, since it then tries to read a not fitting physical line into the buffer. Consider, for example, a file that has formerly been opened and written with lines of 132 characters and is now opened as an INFILE object with line length 80. Reading a line from this file will lead to an error.
If the OUTFILE object is bound to a mobase object the size of length should not be greater than 133.

The standard files *sysin* (standard input; e.g., the keyboard), *sysout* (standard output; e.g., the screen) and *tracefile* (trace output) are opened and closed automatically by the system.

Examples:

```
CONSTANT    link_name : TEXT DEFAULT "LINKNAME";

VARIABLE    f1 :   OUTFILE;
            f2 :   INFILE;
            f3 :   INFILE;

OPEN f1,    "LINKNAME1"      LENGTH   80;
OPEN f2,    "LINKNAME2"      LENGTH  132;
OPEN f3,    link_name & "3"  LENGTH   80;

...                                          { Working on files f1, f2, f3 }

CLOSE f1;
CLOSE f2;
CLOSE f3;
```

3.6.4. Reading from Texts and Files

Texts and files of type INFILE can be read using the READ or READLN statement. READLN is allowed for files only. A file must be opened before it can be accessed.

```
read statement ::=
    READ     [TEXT  text-identifier ,  ]      input_list;
|   READ     [FILE  file-identifier ,  ]      input_list;
|   READLN FILE     file-identifier      [,   input_list];
|   READLN [input_list];


input_list ::=
    {identifier [:: simple_real_expression] } [, ...]
```

READLN statements initiate a line feed. The next line is read into the buffer and the position pointer points at the first character of this line. Next, the *input_list* is worked off. If there is no next line and the attempt is made to read one, a FAN error will occur. Several successive READ statements may also cause a line feed if the end-of-line sign of the current buffer line has meanwhile been read. This is known as implicite READLN.

Note:

Although it is allowed, READLN should not be followed by an *input_list*, especially in the following loop or similar cases:

```
WHILE NOT eof
LOOP
        READLN n;
END LOOP;
```

since in between executing READLN n (which is equivalent to READLN, READ n;) the end-of-file can be reached. Prefer one of the following:

```
WHILE NOT lastitem
LOOP
    READ n;
END LOOP;
```

```
WHILE NOT eof
LOOP
    READ n;
    IF NOT eof THEN
      READLN;
    END IF;
END LOOP;
```

READ and READLN can be used to read character (strings) from a file or a text. Reading from a file requires the keyword FILE to appear behind one of the keywords READ or READLN. The *file-identifier* has to follow the keyword FILE. If the reading operation inflicts the standard input file *sysin*, this specification is not essential.

Reading from a text requires the keyword TEXT behind the keyword READ. The *text-identifier* has to follow the keyword TEXT.

Character strings read from a file can only be assigned to variables of simple data type . In the process of reading character strings into variables of one of the types INTEGER, REAL or BOOLEAN, leading blanks will be overread. If it is not possible to assign a character string to a variable of an appropriate type, a run time error is yielded.

When character strings are read from a stream and assigned to a variable of type TEXT, the number of characters to be read can be specified by an INTEGER or REAL expression. REAL expressions will be converted to INTEGER. If no number is specified, only a single character is read from the stream.

Reading into boolean variables gives them the value TRUE if the position pointer points to '1' (after having ignored leading blanks). Otherwise, the boolean variable receives the value FALSE.

The *input_list* is worked off sequentially until all specified variables have received a value from the stream. If this is not possible, a FAN error occurs (for example when the end of the file has been reached without the input list having been worked off). The position pointer is moved forward after every read access. Each READ TEXT statement causes the position pointer to point to the first character of the text.

Examples:

{Reading of texts}

```
VARIABLE        i :  INTEGER;
              c1, c2 :  CHARACTER;
                  t1 :  TEXT DEFAULT "TEXT 1.";
                  t2 :  TEXT;

READ TEXT t1, t2::4, i, c1;    {t2 = "TEXT", i = 1, c1 = '.'}
READ TEXT t1, c1, c2;          {c1 = 'T' and c2 = 'E'}
```

{Reading of files}

```
VARIABLE        f1 :  INFILE;
                t3 :  ARRAY [1..8] OF TEXT;
              j, k :  INTEGER;

OPEN f1, "LINKNAME1" LENGTH 80;

{Contents of f1: "This is a file with 1 line and 2 numbers."}

READ FILE f1,      t3 [1]::5,   t3 [2]::3,  t3 [3]::2,
                   t3 [4]::5,   t3 [5]::5,  j,
                   t3 [6]::6,   t3 [7]::4,  k,
                   t3 [8]::9;

{t3 [1]  = "This ",   t3 [2]  = "is ",          t3 [3]  = "a ",       }
{t3 [4]  = "file ",   t3 [5]  = "with ",        t3 [6]  = " line ",   }
{t3 [7]  = "and ",    t3 [8]  = " numbers.",    j=1, k=2}
```

{The same result is reached with:}

```
READ FILE f1,      t3 [1]::5,   t3 [2]::3,
                   t3 [3]::2,   t3 [4]::5,
                   t3 [5]::5;

READ FILE f1,      j;
READ FILE f1,      t3 [6]::6,   t3 [7]::4;
READ FILE f1,      k;
READ FILE f1,      t3 [8]::9;
```

{Reading of files using READLN:}

```
VARIABLE      f2 :  INFILE;
               a :  ARRAY [1..8] OF REAL;

OPEN f2, "LINKNAME2" LENGTH 80;

{Contents of f2: }
{line1  :   1      2.1     3      4.7  }
{line2  :   5.0    6       7      8    }
{line3  :   9.4    10.0    11     12   }

READ FILE f2,      a [1],   a [2],   a [3];     READLN FILE f2;
READ FILE f2,      a [5],   a [6],   a [7];     READLN FILE f2;

{a [1]  = 1.0,   a [2]  = 2.1,   a [3]  = 3.0,   a [4]  = 0.0}
{a [5]  = 5.0,   a [6]  = 6.0,   a [7]  = 7.0,   a [8]  = 0.0}

CLOSE f2;
```

```
{Contrasting the above, the following statements yield}

OPEN f2, "LINKNAME2" LENGTH 80;

READLN FILE f2;
READLN FILE f2,      a [1],   a [2],   a [3];   a [4];
READLN FILE f2,      a [5],   a [6],   a [7];   a [8];

{a [1]  =  5.0,   a [2]  =   6.0,   a [3]  =   7.0,   a [4]  =   8.0}
{a [5]  =  9.4,   a [6]  =  10.0,   a [7]  =  11.0,   a [8]  =  12.0}
```

3.6.5. Writing to Texts and Files

OUTFILE files can be written by using WRITE or WRITELN statements. WRITELN statements are permitted for files only. A file must be opened before a WRITE or WRITELN statement can be performed.

```
write statement ::=
    WRITE      [TEXT text-identifier , ]   output_list;
|   WRITE      [FILE file-identifier , ]   output_list;
|   WRITELN FILE    file-identifier [,    output_list];
|   WRITELN  [output_list];


output_list ::=
    {expression [:: simple_real_expression [:: simple_real_expression] ] }  [, ...]
```

WRITELN statements effect a line feed. The current buffer line is written into the file after having worked off the *output_list* and a new buffer line of the line length defined in the OPEN statement is opened. The position pointer is set to 1. The run time system implicitely performs WRITELN statements if the WRITE operation exceeds the current buffer line.

WRITE or WRITELN can be used to write character strings into a file. WRITE, but not WRITELN, can also be used to write character strings into a text. Writing into a file requires the keyword FILE to follow WRITE or WRITELN, and a *file-identifier* has to succeed the keyword FILE. This also applies when writing into text variables, the keyword FILE being replaced by TEXT. These specifications are dispensable if the WRITE statements are used to write into the standard output file *sysout*.

The *output_list* is worked off sequentially. Each expression preceding the first '::'-symbol defines those character strings that are to be written into the file or the text. The expressions must result in values of simple data type. Expressions of type TEXT may not contain the concatenation operator '&' since concatenation is done implicitely. Expressions of type INTEGER, REAL and TEXT may be followed by specifications of their length (behind the first '::'-symbol). Length specifications are expressions of type INTEGER (or REAL) and define the number of characters to be written into the file.

Only if REAL values are written into a file, a further length specification is allowed following a second '::'-symbol. The first specification defines the number of digits behind the decimal point, the second describes the total length of the REAL number. The value of the REAL expression is therefore represented within the character string by a fix-point notation. If the second expression is not given, the REAL value is represented

using the floating-point notation with as many significant positions as described in the first specification.

In the absence of length specifications, the texts are written in full length. INTEGER values in this case are represented with 11 characters, REAL values with 13 characters plus 7 characters for the mantissa. If the length of an INTEGER value is less than 11 characters, it is preceded by blanks. The length of INTEGER and REAL values should not exceed the respective length specifications, because otherwise the run time system produces warnings ("edit overflows"). When writing texts, the length specification is interpreted as a limiter. If the text is shorter than its specification length, it is printed adjusted left and the vacant space is filled by following blanks. If it is longer, it is shortened to the specified length.

A boolean expression with the result value TRUE is written into the file or text as "1". If the value is FALSE, it is written as "0".

Examples:

```
VARIABLE     t1, t2  :  TEXT;
                 f1  :  OUTFILE;
                  r  :  REAL;
CONSTANT         tc  :  TEXT DEFAULT "Number";

OPEN f1, "LINKNAME" LENGTH 80;

t1  :=  "Such a" & " " & "statement writes the";
t2  :=  "Look! ";
 r  :=  17.777;

WRITE FILE f1, t2, t1, tc, ":   ", r::2::8;
{Contents of the 1. line of f1:}
{"Look! Such a statement writes the Number:         17.77"}

WRITE TEXT t1,       "OK"              ::5;          {       "OK    "                  }
WRITE TEXT t1,       "      "          ::6;          {       "           "             }
WRITE TEXT t1,       ""                ::2;          {       "   "                     }
WRITE TEXT t1,       "TOO LONG"        ::3;          {       "TOO"                     }

WRITE TEXT t1,                   13::4:              {       "    13"                  }
WRITE TEXT t1,                  123::3;              {       "123"                     }
WRITE TEXT t1,            1234567890;                {       " 1234567890"             }
WRITE TEXT t1,           -1234567890;                {       "-1234567890"             }

WRITE TEXT t1,            r - 0.2105::5;             {       " 1.7566E+01"             }
WRITE TEXT t1,          17.5665::3::10;              {       "          17.566"        }
WRITE TEXT t1,                17.5665;               {       " 1.756650E+01"           }
```

Example:

```
VARIABLE     t   :  TEXT DEFAULT "abcd";
             u   :  TEXT;

 READ TEXT    t, u::4;
WRITE TEXT    u, t;          {both statements have the effect u := t here}
```

3.7. Records and Pointers

Apart from arrays, HI-SLANG provides another structured data type, the record.

common_declaration ::= ...
| type_declaration
| record_declaration

type_declaration ::= ...
| recordtype_declaration

A record combines several objects of different types. Differing from array objects, these objects need not be of simple data types, and don´t have to be of the same type.

Records may consist of constants, variables, arrays, pointers, records and procedures. They therefore support the creation of linked structures (lists, trees, general graphs). They can furthermore be used to define abstract data types.

Because of their complex structure, records are defined with the help of record types.

3.7.1. Record Types

Record types are patterns for the statical or dynamical generation (by NEW) of records.

recordtype_declaration ::=
TYPE recordtype-name **RECORD** [formal_parameters] ;
[common_declaration [...]]
[**BEGIN** sequence_of_statements]
END TYPE [recordtype-name] ;

A record type declaration consists of a headline with its name and, possibly, its formal parameters. The headline is followed by a declaration part to define the local objects of the record type. The declaration of a record within a record type must not result in a (direct or indirect) recursion of record declarations. Upon the keyword BEGIN, statements can follow that describe operations to be performed in each new incarnation of a record object, as for example initializing record objects.

Record types declared within model types or component types cannot use or access objects declared outside this record type.

The keywords END TYPE conclude the type declaration. The name of the type should be repeated. If a different name than that of the just declared type is used here, a warning occurs.

Example:

```
TYPE complex RECORD (r, i : REAL DEFAULT 0.0);

        ...

        PROCEDURE mult (re, im : REAL) RESULT POINTER FOR complex;
                VARIABLE p : POINTER FOR complex;
        BEGIN
                NEW complex (r*re - i*im, r*im + i*re)   POINTER p;
                RESULT p;
        END PROCEDURE mult;

        PROCEDURE exp   (n : INTEGER);
                VARIABLE  hr, hi : REAL;
        BEGIN
                hr, hi := 1;
                WHILE n > 0
                LOOPn := n-1;
                        (hr, hi) := mult (hr, hi);
                END LOOP;
                r := hr; i := hi;
        END PROCEDURE exp;

BEGIN
        WRITELN "complex just generated";
END TYPE complex;
```

3.7.2. Declaration of Record Objects and Pointers

Record objects can be generated statically or dynamically in a way that facilitates individual access to them by using a pointer. In case of dynamical generation using the NEW statement, the pointer must previously be declared. In simulative experiments, records and pointers can be declared in any declaration part of the program. In analytical experiments, records and pointers are not allowed within the model.

record_declaration ::=
RECORD
{record-name [, ...] :
[**ARRAY** ⟦ array_bounds [, ...] ⟧ **OF**]
recordtype-name [actual_parameters] ; } [...]

Beside record objects, arrays of record objects that have any desired dimension can be defined following this pattern.

Declared record objects are statically generated when the program is executed. Subsequently, the initialization statements, providing their existence within the record type, are run. The so-generated record object is of the type *recordtype-name*. If the record type has formal parameters and if they do not have default values, actual parameters must be provided when a record object is declared. The transmission technique equals that of procedure parameters, though the mode "call by name" is not allowed.

If variables appear within the expressions to compute *array_bounds* or *actual_parameters*, these must have been declared in surrounding blocks. Thus records defined

in the EXPERIMENT block do not allow variables as actual parameters, since the EXPERIMENT block does not form a new scope.

Pointers serve for access to dynamically generated record objects. The association between a pointer and its (dynamically generated) record object is achieved with the NEW statement or by a pointer assignment (taken over the same association). Pointers are declared similar to variables, for a pointer type is a simple data type. Please note that dynamically and statically generated record objects even of the same type are not convertible.

simple_type ::= ...
| **POINTER FOR** recordtype-name

Examples:

```
RECORD      c1, c2  : complex (1, 0);
          c_matrix  : ARRAY [1..4, 1..4] OF complex (1);

VARIABLE        p   : POINTER FOR complex;
          pa1, pa2  : ARRAY [0..4] OF POINTER FOR complex;
```

There is a predefined pointer. It is called NONE and it is a universal pointer constant that refers to "nothing". NONE may be used for any self-defined pointer variable. Please also note that NONE is the default value of a self-defined pointer variable.

simple_expression ::= ...
| **NONE**
| identifier

An expression yielding a pointer may be very complex, since procedure calls yielding a pointer and pointer arrays may be contained.

Example:

```
pa1 [c1.r].mult (1, 2);
```

3.7.3. Dynamical Generation of Records

Apart from the statical declaration of record objects, there is the possibility of dynamical generation by using the NEW statement. In this case, the record objects are reversibly bound to a pointer. The initializing operations of the record type definition are executed whenever an object is generated. In modelling, the NEW statement is only applicable for simulation.

simple_statement ::= ...
| new_statement

new_statement ::= ...
NEW recordtype-name [actual_parameters] **POINTER** pointer-identifier [, ...]
;

Example:

```
NEW complex (-1.5, 2.7) POINTER p, pa1 [0], pa2 [0];

{sets 3 different pointers on one dynamically generated record of type complex.}
{The pointer list is worked off from the left to the right.}
```

3.7.4. Operations on Records and Pointers

Pointers referencing dynamically generated record objects (via the NEW statement) can be modified by pointer assignments. Each record object can be addressed by an arbitrary number of pointer variables.

Pointer assignment allows to set a pointer onto another record object. If the last pointer on a record object is reassigned, the original record object cannot be addressed any longer.

All pointer variables on the left hand side of the assignment are set onto the address to which the pointer on the right hand side refers to throughout the operation. All pointers must be declared as pointer variables of the same record type.

Example:

```
pa1 [0], pa2 [0]    :=NONE;
p                   :=pa1 [1];
```

Pointer comparison allows the investigation of whether two pointers refer to the same dynamically generated record object. Tests exist for referential equality ('=') and for referential inequality ('<>', '#').

Example:

```
IF p # NONE AND pa1 [0] = pa2 [0] THEN ... END IF;
```

Record objects as a whole cannot be compared to one another. However, there is a record assignment for statically generated records. In the course of this assignment, all record elements of all records on the left side are replaced by the contents of the corresponding elements of the record on the right hand side of the assignment. Of course, all records involved in such an assignment have to be of the same type.

Example:

```
RECORD c, d : complex (1, 2);
...
c := d;
```

The most important operation on records and pointers is the access to elements of an indicated record object. Two possibilities exist for carrying out this operation: the dot notation and the WITH statement.

3.7.5. Access to Record Elements via Dot Notation

The elements of a record object can be addressed via dot notation, in a way that corresponds to the indexing of arrays. No distinction is made between statically or dynamically generated record objects in this case. On the left of the dot, either the name of a statical record object or the name of a pointer referring to a dynamic record object is situated. The name of the element to which access shall be taken locates at the right of the dot. If the name is a pointer variable or a statical record object, this access mechanism can be applied iteratively .

Examples:

```
TYPE complex_tree RECORD (n : INTEGER DEFAULT 0);

  VARIABLE node :  POINTER FOR complex;              {see Section 3.7.1.}
         left, right :  POINTER FOR complex_tree;

BEGIN

      IF n > 0 THEN
            NEW complex_tree (n-1) POINTER left;
            NEW complex_tree (n-1) POINTER right;
      END IF;

END TYPE complex_tree;

…

      VARIABLE    wood    :   ARRAY [0..5] OF POINTER FOR complex_tree;
                  root    :   POINTER FOR complex_tree;
      RECORD      leaf    :   complex_tree;

…

{After the dynamical creation of the structures, their elements can be addressed as follows: }

  leaf.node                leaf.node.i          leaf.node.exp  (3)
  root.node                root.node.r          root.node.mult  (leaf.node.i, 2)

  wood [1].left.right.node.exp (3)
```

Note:

If you try to access an element of a dynamic record object which has not yet been generated a run time error occurs and the analysis is stopped.

3.7.6. Access to Record Elements using the WITH Statement

If access to several elements of one record object or repeated access to the same element is intended, the dot notation proves to be rather inconvenient. In this case, the WITH statement is more favorable. By employment of the WITH statement, the record object can be selected and access to its elements can be taken directly, without dot notation. Furthermore conforming to the block concept conventions, all names remain usable in the statement part, unless there is a name equality with the name of the selected record object.

```
compound_statement  ::=  ...
|       with_statement

with_statement  ::=
        WITH record_or_pointer-identifier
        DO
                sequence_of_statements
        END WITH ;
```

In this manner, not only statical record objects can be selected by their name, but additionally dynamical record objects can be selected by the pointer referring to them. If referring to a record using a NONE-valued pointer is attempted, the sequence of statements is not executed.

Nesting of WITH statements does not improve the clarity of a program and should therefore be avoided. With the exception of the EVALUATE and AGGREGATE statement, all statements are permitted within a WITH statement.

Example:

```
NEW complex_tree POINTER root;                                  {dot notation}
root.node.i, root.node.r := 1;

WITH root.node DO                                       {identical WITH statement}
        i, r := 1;
END WITH;
```

Moreover, the WITH statement can be used correspondingly to make the access to process states easier (see Section 4.2.2.4.). A process state is represented by the set of parameters of a service. These can also be addressed by using the dot notation.

Examples:

```
TYPE st SERVICE (my_state : INTEGER);
...
END TYPE st;

PROCESS p : st (1);

IF p.my_state = 1 THEN ...
END IF;

WITH p DO IF  my_state = 2 THEN ... END IF;
END WITH;
```

3.7.7. Records and Pointers as Parameters

Records and pointers can be defined as formal parameters of procedures and all types of modelling objects (in case of simulation). The definition of a record as formal parameter requires the keyword RECORD afore the name of the parameter. The keyword VARIABLE may appear preceding all other parameter names, including array parameters, in order to distinguish between record and other parameters. Neither array dimension and bounds nor actual parameters of a record type are specified.

Records as well as pointers cannot be handed over by VALUE. They are passed by NAME or REFERENCE, the latter being the default mode. A DEFAULT part for records is not allowed.

Example:

```
PROCEDURE calculate  (RECORD c1, c2  :  ARRAY   OF     complex;
                      NAME VARIABLE p_c  :  POINTER FOR  complex;
                                   VARIABLE  :  REAL DEFAULT 1.0);
        …
END PROCEDURE calculate;
```

The employment of records and pointers as actual parameters corresponds to the use of variables as actual parameters:

Example:

```
calculate (c_array_1, c_array_2, c_pointer);
```

Procedures and services can yield values of type POINTER, but not of type RECORD. Therefore, the result specification of procedures and services consists of a *simple_type* list:

Example:

```
PROCEDURE multi (link : TEXT) RESULT  POINTER FOR complex,
                                      BOOLEAN, INFILE;
   VARIABLE     f : INFILE;
                p : POINTER FOR complex;
BEGIN
   OPEN f, link LENGTH 80;

   NEW complex POINTER p;
   READ FILE f, p.r, p.i;
   RESULT p, eof (f), f;
END PROCEDURE multi;
```

4. Model Description

This section deals with HI-SLANG elements used for a structured model description. The description of modelling objects relies on the programming language notations developed in Chapter 3. Those parts of the language which are used to describe model generation and model analysis are the topic of Chapter 5.

From a global point of view, a model, as described by a model type (Section 4.4) is a hierarchical arrangement of load-machine pairs. A load, given by a set of processes (Section 4.1), is bound to a machine, given by a set of components (Section 4.2). In this section, every modelling object (process, component, model) is an instance of some object type. The type is referred to in the declaration part of a modelling object.

```
type_declaration  :: =  ...
|       service_declaration
|       componenttype_declaration
|       modeltype_declaration


modelling_declaration  :: = ...
|       component-declaration
|       process_declaration
|       enclose_declaration
```

4.1. Services and Process Generation

The load imposed on a model is represented by processes acting as time and space consuming entities. They are always described by, and instances of, a so-called service, which could therefore also be named a process type. Due to historical reasons, services can also be formulated as service types in HI-SLANG.

In general, a service refers to (uses, calls) external services which are equated to services provided by a lower layer component (machine). These provided services in turn are descriptions of processes (services) running in the lower layer component.

Once a process is generated in a component or, at the topmost level, in a model, it is termed a local process of that component (model). A service call in the behaviour pattern followed by the process leads to the descent of this process to a lower layer. In this case, the process has a hierarchical structure which mirrors the hierarchy of layers in the model. Processes can be generated statically via declaration (Section 4.1.2) or dynamically via CREATE or SUBMIT statements.

Process organization is handled by control procedures (Section 4.2.2) within the component for which a process is local. If, by a service call, a process descends to a subcomponent (Section 4.1.4), the subcomponent takes control of the process until the service call has finished and control is transferred back to the higher layer component.

4.1.1. Services

Services describing the behaviour patterns followed by processes of that type are used for process generation. Within analytical models services correspond to so-called chains.

The enclosing component type can declare a service to be a provided service of the component. Services may only be declared in the declaration part of component or model types.

```
service_declaration :: =
        TYPE service-name SERVICE [formal_parameters]
                [RESULT simple_type [, ...]] ;
                [use_declaration_part]
                [declaration [ ...]]
        [BEGIN sequence_of_statements]
        END TYPE [service-name] ;

|       SERVICE service-name [formal_parameters]
                [RESULT simple_type [, ...]] ;
                [use_declaration_part]
                [declaration [ ...]]
        [BEGIN sequence_of_statements]
        END SERVICE [service-name] ;



declaration :: =
        common_declaration
|       modelling_declaration
```

A service declaration starts with a header comprising the service name, optional formal parameters and an optional specification of results. The header is followed by a declaration of the external services and procedures used (*use_declaration_part*), a local declaration part and a statement part.

The *service-name* is any unique name for the behaviour pattern. It is subject to the rules governing the scope and validity of identifiers. It is used to determine the service of a process to be generated and to identify a service which is provided (PROVIDE) by a component.
The optional service name repetition following the END TYPE resp. END SERVICE must match the name given in the header. If not, the compiler will produce a warning.

Objects local to the service can be declared in the *declaration* part and subsequently accessed in the statement part (*sequence_of_statements*). Constants, variables, records, pointers, procedures, and possibly streams may be declared. Scope rules for identifiers must be observed.

The statement part of the optional service body describes a process behaviour pattern. Except for AGGREGATE and EVALUATE statements, any statement may occur here. A RESULT statement is valid only if it is in accordance with the result specification given in the service header (services returning results, Section 4.1.1.2). In addition to its parameters and objects local to the service, external services and procedures may be used if they are listed in the USE-declaration part. Global constants, variables, record

types, records, pointers, and procedures may also be accessed, as well as the streams declared in the surrounding component type.

Recursive service calls are not allowed within services.

Note:

Be very careful when accessing global objects. Access to global variables including records and pointers is highly dangerous, especially due to model maintainability as well as due to the potential parallelism (of the processes writing to such variables), and will cause a warning in a future version of HIT.

Example:

```
TYPE dialog SERVICE (thinktime, computetime : REAL);
   USE
        SERVICE         think       (thinktime          : REAL);
                        transfer    (transferdelay      : REAL);
                        compute     (cputime, iotime    : REAL);
        PROCEDURE       short_delay RESULT BOOLEAN;
   END USE;

BEGIN

   LOOP
        think (negexp (1.0/thinktime));

        IF short_delay  THEN transfer (0.5);
                        ELSE  transfer (5.0);
        END IF;

        compute (computetime, computetime/2.0);
   END LOOP;

END TYPE dialog;
```

4.1.1.1. Services with Parameters

Similar to a procedure, a service may have formal parameters. When a process is generated or a service is called, its formal parameters are substituted by actual parameters. Formal parameters can be accessed in the statement part of a service. The rules for the specification of formal parameters and their substitution by actual parameters are the same as for procedures, with the restriction that a "call by name" is not allowed for service parameters.

If local objects of a process which was generated statically or dynamically by a SUBMIT statement are to be accessed via dot notation, these objects must be declared as formal parameters. The formal parameters of a service define the explicite state of corresponding processes. Accordingly, actual parameters supplied to a process define the initial state. Only state variables (i.e., service parameters) of a process can be accessed.

Example:

```
TYPE shopping SERVICE (shopping_list    : ARRAY OF TEXT;
                                  money: REAL DEFAULT 10000;
                                  by_car: BOOLEAN DEFAULT TRUE);
...
END TYPE shopping;


CREATE 1 PROCESS
    shopping (letter_to_santa_claus, 0.99, FALSE);

CREATE 1 PROCESS
    shopping (list_of_things_desired, ,) AFTER 100.0;
```

4.1.1.2. Services Returning Results

Processes and services can, just as procedures, return one or several results to the caller. The data types of the results, separated by commas, are specified after the RESULT keyword in the header of a service definition. There should be at least one reachable RESULT statement in the statement part of the service.

The service returns its results if it is a provided service of a component which is used (called) by a higher level service. In this case, the service call is syntactically identical to a procedure call.

If a process of a service returning results is generated statically ("PROCESS p : service-name") or dynamically (e.g., "CREATE 1 PROCESS service-name" or "SUBMIT service-name NAME p"), the results will be lost as soon as the process terminates. The results are also lost if the service call does not occur on the right hand side of an assignment or in an expression.

Processes which return results are not allowed to be used in a model which is to be analyzed by an analytical solver.

Example:

```
{use of a service returning results}

TYPE st SERVICE;
    USE
        SERVICE find_employee (name : TEXT)
            RESULT TEXT, INTEGER, REAL, BOOLEAN;
    END USE;
    …
BEGIN
    …
    (address,age,salary,married)  := find_employee ("miller");
END TYPE  st;
```

4.1.1.3. Services with USE Declarations

The USE declaration part of a service lists procedures and/or services provided by some component and used (called) by this service. Services/procedures provided by component arrays may be used in total as service/procedure arrays.

use_declaration_part ::=
    USE
        use_declaration [...]
    END USE ;

use_declaration ::=
    procedure_or_service [**ARRAY**]
    {procedure_or_service-name [formal_parameters]
     [**RESULT** simple_type [, ...]] ; } [...]

procedure_or_service ::=
    PROCEDURE
| **SERVICE**

The binding between services and procedures used by a service to those provided by components is done in the REFER part of the surrounding component type.

Names in a USE declaration may be chosen freely but must be different from names of local variables and formal parameters of the service. The scope rules for identifiers are valid, i.e., names in a USE declaration can hide global names.

If a used service or procedure has parameters, the number, order, and types of parameters in the USE declaration must be identical to those of the service or procedure to which it is bound (no type conversion is done). DEFAULT values can be defined in the USE declaration, overriding any other DEFAULT values. Names of parameters and names occurring in expressions for DEFAULT values can be chosen freely. The same restrictions as for the initialization of variables with DEFAULT values apply to them.

If a used service or procedure has results, the number, order, and types of results in the USE declaration must also be identical to those of the service or procedure to which it is bound (no type conversion is done).

A service or procedure array denotes a set of services and procedures, respectively. In a REFER part of an enclosing component (model), service and procedure arrays must always be bound to services and procedures provided by component arrays. A single service or a single procedure of an array is called by an indexing mechanism similar to that of an ordinary array access. Errors will occur if indices are outside the range of the component array. Service and procedure arrays do not have standard array attributes. The dimension is always one.

Example :

```
TYPE pt SERVICE;
   USE
         SERVICE              compute;
         SERVICE ARRAY        disk_access (time : REAL);
         PROCEDURE            short_delay;
   END USE;
   …
   disk_access [2] (11.5);
   …

END TYPE pt;
```

Any service may call the predefined services *spend* and *hold* (cf. Section 4.1.5.2) without explicitely using them.

4.1.1.4. Procedures with USE Declarations

Similar to services, a USE declaration part of a procedure lists procedures provided by some component and used (called) in this procedure. The syntax is the same as in Section 4.1.1.3 with the exception that access to ("external") services is not allowed here.

The binding between procedures used by a procedure and procedures provided by components is done in the REFER part of the surrounding component type. Names in a USE declaration can be chosen freely but must not clash with names of local variables or formal parameters.The scope rules for identifiers are valid, i.e., names in a USE declaration can hide global names.

If a used procedure has parameters, the numbers, order, and types of parameters in the USE declaration must be identical to those of the procedure to which it is bound (no type conversion is done). Names of parameters and names occurring in expression for DEFAULT values can be chosen freely. If a used procedure has results, the number, order, and types of the results in the USE declaration must also be identical to those of the procedure to which it is bound (no type conversion is done).

From a modelling point of view, the execution of a procedure is not consuming any time. Therefore, procedures listed in a USE declaration are not subjected to a component control mechanism of the (sub-)component to which the procedure is bound in the REFER part of the surrounding component type.

Example:

```
PROCEDURE status_test;
   USE
         PROCEDURE      printer_statusRESULT BOOLEAN;
                        net_status     RESULT BOOLEAN;

   END USE;
...
END PROCEDURE status_test;
```

Procedures are not allowed to be used in a model which is to be solved analytically.

4.1.2. Declaration of Processes and Process Names

Processes can be generated statically (process declaration) or dynamically in a way that they can be accessed individually by name or an additional index, if a one-dimensional process array is declared. If a process is generated dynamically by a SUBMIT statement (see Section 4.1.3.2.), its name must be declared.

process_declaration ::=
PROCESS
{process-name [, ...]: [**ARRAY** ⟦array_bounds⟧ **OF**]
process_name_or_object_declaration ; } [...]

Static generation of processes and declaration of process names is valid only in declaration parts of component or model types.

Speaking in simulation terminology, process generation happens without model time consumption and is implemented by simply inserting additional events into the eventlist. Global variables, e.g., parameters supplied to "EVERY negexp (...)" in CREATE statements, are accessed when this event becomes "current" and not at process generation time. Due to the serialization of events done by a simulation program, even processes generated at the same model time (e.g., static process generation) may see different values of global variables.

4.1.2.1. Declaration of Processes

Process object declarations lead to a static generation of one or several processes, or process arrays, which are started immediately as local processes.

process_name_or_object_declaration ::= ...
| service-name [actual_parameters]

The process is accessed via its *process-name* (Section 4.1.2.2). The name, like any other name, can be chosen freely under the restrictions governing validity and scope of identifiers. Process arrays must be one-dimensional. Furthermore, the array bounds expressions are evaluated at compile time. As for other arrays, array bounds expressions are built over simple (standard) types. Process arrays do not have array attributes.

A generated process is of type *service_name*. This is the reason why services are formulated as a type in HI-SLANG. This type must be declared within the same component type. If a service has parameters, the same rules as for procedures apply to parameter substitution. For a list of generated processes the actual parameter expressions, if existing, are evaluated once.

Process parameters can be accessed via process name and dot notation. But the access to process parameters does not make sense after the process has left the component.

If access to local service variables is necessary, they must be declared as formal parameters. DEFAULT values can be supplied, so these extra parameters need not be considered when the service is called.

Example:

```
PROCESS     watch_me            : pt (1, 2);
            parallel_process    : ARRAY [1..10] OF pt (5,10.0);

{declaration of pt:}

TYPE pt SERVICE (form1 : INTEGER; form2 : REAL);
…
END TYPE pt;

{access to formal parameters via dot notation: (r is of type REAL, i is of type INTEGER)}

r := watch_me.form2;
i := parallel_process [8].form1;
```

4.1.2.2. Declaration of Process Names

Process names are used to reference dynamically generated processes. They are assigned to newly generated processes via a SUBMIT statement.

process_name_or_object_declaration ::= ...
| **NAME FOR** service-name

service-name is a name chosen freely under the restrictions governing validity and scope of identifiers.

In a SUBMIT statement, process names declared in this way can only be assigned to processes of type *service-name*. According to the scope rules, the *service-name* must be known at the time of process name declaration. A process name variable can be reassigned by an assignment statement or by another SUBMIT statement, i.e., at run time a process name variable can sequentially reference different processes. A process name variable cannot reference a process generated statically, however.

The state of a process (e.g., actual parameters) can be accessed via process names and dot notation. But the access to the implicit state does not make sense after the process has left the component. If a process name does not reference any process, an access results in a run time error.

Process names are not allowed to be used in a model which is to be analyzed by an analytical solver.

Example:

```
PROCESS     watch_me_too : NAME FOR pt;
            {pt: see above, no actual parameters given here}

SUBMIT      pt (10, 5.0) NAME watch_me_too;          {process generation}

r := watch_me_too.form2;                        {access via dot notation}
```

4.1.3. Dynamic Process Generation

Apart from being generated statically by declaration, processs can also be generated and started dynamically by CREATE and SUBMIT statements. These statements are described in detail in the following subsections. In both statements a so-called timing condition determines either a process generation time or a duration between process generations.

```
simple_statement  ::= ...
|       create_or_submit_statement

timing_condition  ::=
        time_specification simple_expression

time_specification  ::=
        AT
|       AFTER
|       EVERY
```

The *simple_expression* must have a REAL or INTEGER value and denotes either a point in model time or a duration. Values less than zero are forced to zero. AT, AFTER and EVERY keywords have the following meaning:

AT: *simple_expression* determines the point in (model) time at which a process (or the given number of processes) is started. If this point of time is before the current model time, it is interpreted as "now".

AFTER: *simple_expression* determines the duration (in model time units) between "now" and the point in time at which a process (or the given number of processes) is started. This point in time can be computed by adding *simple expression* to the current model time, i.e., the time at which the statement is executed.

EVERY: *simple_expression* determines an interval in model time units which is the time between successive process generations. The first process is generated immediately when the statement is executed. The generation sequence stops at the end of the simulation run.

Service variables and parameters as well as BLOCK variables may not be used in the EVERY-*simple_expression* in a CREATE statement ocurring in a service statement part. Global variables and variables of the enclosing component type may be used, however.

Only the *simple_expression* following EVERY is evaluated each time a process is generated to determine the next process generation time. See the note below.

In contrast to service calls, CREATE and SUBMIT statements yield a new local process. The execution of these statements is not consuming any model time, even if a timing condition is specified. The process generation is thus performed by a separate dedicated process.

Note:

Every expression occuring in a CREATE/SUBMIT statement is evaluated exactly at that time the statement is executed, and not at the time of process generation. The only exception is the expression following EVERY. This exception has the consequence that in

```
FOR i := 1 STEP 1 UNTIL 4 LOOP
        CREATE 2 PROCESS service-name EVERY negexp (i);
END LOOP;
```

the variable *i* has the value 5 (the value of *i* after termination of the loop) each time the expression *negexp (i)* is executed!

Note:

If no timing condition is given, the process surely exists after execution of the CREATE statement and after a progress in modeltime. Since executing CREATE does not consume model time, the process may not yet exist when the statement following the CREATE statement is executed.

4.1.3.1. CREATE Statement

Processes without names (anonymous processes) are generated dynamically by a CREATE statement:

```
create_or_submit_statement ::= ...
|  CREATE        simple_real_expression
   PROCESS       service-name [actual_parameters]
   [     [LIMIT  simple_real_expression] timing_condition]  ;
```

Both *simple_real_expressions* must yield an INTEGER or REAL value. REAL values are converted to INTEGER values according to type conversion rules. *service-name* must be a known service name according to the rules governing the scope of identifiers. Actual parameters must be supplied according to the service declaration and parameter transmission rules.

The expression following CREATE specifies the number of processes generated simultaneously. These processes follow the behaviour pattern given by service s*ervice-name* and the actual parameters. No process generation will be done if the expression yields a value less than or equal to zero.

Processes are generated immediately, i.e., they exist at the next point of model time (see Note above), if *timing_condition* is omitted. If not, they will be generated according to the *timing_condition*. CREATE statements do not consume (model) time, i.e., the next statement is executed at the same point of model time as a CREATE statement, whatever the *timing_condition* may be.

If a model is analyzed by an analytical solver, the timing condition must either be an EVERY clause (to model open chains; the EVERY clause denotes the arrival rate here) or be omitted (to model closed chains). The corresponding processes must be either temporary (no endless loop) or permanent (endless loop), respectively.

Furthermore, a LIMIT clause is not allowed to be used if a model is analyzed by an analytic-algebraical solver. In queueing network terminology the LIMIT constitutes an upper bound for the number of customers in an open chain (numbers of processes in progress). LIMIT may only be specified in conjunction with EVERY. The expression must yield an INTEGER or REAL value > 0, a REAL value being converted to an INTEGER. A LIMIT is an upper bound for the number of processes in progress only with respect to the particular CREATE statement and not on the (total) number of processes of the service.

The actual parameter *time* in a CREATE statement

```
CREATE 1 PROCESS pt (time) EVERY 10;
```

is evaluated just once and at the time when the CREATE statement is executed, because the actual parameters are evaluated only once even if EVERY is present (see Note above). For example, at the start of a simulation *time* would be zero and a process of type *pt* and actual parameter zero would be generated every ten model time units.

Examples:

```
CREATE 1 PROCESS pt (5, 10.0)      AT 100;
CREATE 2 PROCESS pt (a, b);
CREATE n PROCESS pt (a, f)         EVERY a + b;
```

4.1.3.2. SUBMIT Statement

Named processes are generated dynamically by a SUBMIT statement. Their name is stored in variables of type "NAME FOR service-name":

```
create_or_submit_statement ::= ...
|     SUBMIT  service-name              [actual_parameters]
      NAME    process_name-identifier  [timing_condition] ;
```

The *service-name* must be a known service of a surrounding component or model type. Actual parameters must be supplied according to the service declaration and parameter transmission rules. *process-identifier* is a process name declared in a process name declaration. The name must be a known identifier and be declared as a name for service *service-name*.

Processes generated dynamically by a SUBMIT statement follow the behaviour pattern given by service *service-name* and the actual parameters. Formal parameters of the service (i.e., the process state in HI-SLANG convention) can be accessed via dot notation. Access is possible as long as the name references the process, even after the process has finished its activities. But the access to the implicit state does not make sense after the process has left the component.

Processes are generated immediately, i.e., whenever the SUBMIT statement is executed, if *timing_condition* is omitted (see Note above). If not, they will be generated according to the *timing_condition*. SUBMIT statements are not (model) time consuming, i.e., the next statement is executed at the same point of model time as the SUBMIT statement, whatever the *timing_condition* may be. Run time errors will occur if a process is accessed immediately after a SUBMIT statement with an AFTER timing condition (see the example below).

SUBMIT statements may only be used in models analyzed by the simulative solver.

Examples:

```
PROCESS process1, process2 : NAME FOR pt;

SUBMIT    pt    (5, 10.0)    NAME process1 AFTER 5;
SUBMIT    pt    (a, b)       NAME process2;
SUBMIT    pt    (5, 10.0)    NAME process1 AT 2000;

n := process1.form1;
```

Note:

The last statement above will produce a run time error if *process1* has not yet been activated: The process and its formal parameters (as *form1*) do not exist before activation.

4.1.4. Service Calls

Service calls represent a way of describing a hierarchically structured load, the load being a part of a hierarchical model made up of layers of load-machine pairs. Processes and service calls are running under the control of the component which CREATEs and/or PROVIDEs them. Processes and services can, in turn, USE services provided by a subcomponent.

In contrast to a process generation via CREATE or SUBMIT statements, a service call will not result in a new local process (with respect to a component). Rather, the calling process descends to the next lower layer of the model. It starts executing the statements of the called service under the control of the component which provides the called service.

In queueing network terminology, service calls represent a way to describe class changes within a customer chain in a hierarchical fashion.

Services used by other services have to be listed in the USE declaration part of the calling service, with the exception of services *spend* and *hold*. The binding between used services and services provided by some components is done in the REFER part of the surrounding component type. Services provided by a component must be listed in the PROVIDE part of the component type declaration.

Syntactically, service calls within the statement part of the calling service are identical to procedure calls, i.e., services can be called

- in a statement comprising the service name given in the USE declaration and its (optional) parameters;

- in an expression, if the service yields a single result of the type expected in the expression. Again, the service name and its (optional) parameter as described in the USE declaration must be given;

- on the right hand side of an assignment or RESULT statement. Again, the service name and its (optional) parameters as described in the USE declaration must be given. In this case, the called service is allowed to have more than just one result.

If a service which is to return results is called in the first way mentioned above, the results of the service call are lost.

In contrast to procedure calls, the progress of service calls is controlled by component control procedures of the component providing the service. A user can choose from a set of predefined standard procedures or write his own control procedures in HI-SLANG.

4.1.5. Special Statements within Services

These are statements only to be used in the statement part of a service declaration. They are important for modelling purposes.

4.1.5.1. CONCURRENT Statement

CONCURRENT statements are used for simulations to describe the fact that certain actions of a service can be executed concurrently. This refers to model time. Concerning cpu time, the statements are executed in an interleaved fashion respecting the order based on the model time.

```
compound_statement  ::= ...
|       concurrent_statement


concurrent_statement  ::=
        CONCURRENT
                sequence_of_statements
        {TO
                sequence_of_statements } [ ...]
        END CONCURRENT ;
```

With the exception of AGGREGATE, EVALUATE, CHAIN and RESULT, any kind of statement may occur in a CONCURRENT statement. Sequences of statements separated by TO are executed concurrently. In order to be useful, time or space consumption should occur in the concurrent statements. This can be expressed by *spend* or *hold* or service calls which ultimately require a service of, say, *servers* or *counters*.

With regard to component control, a CONCURRENT statement is treated as any other statement of a service, but it may cause the execution of multiple used services in parallel. The implication is that if the service issuing the CONCURRENT statement in a component is preempted (returns from *service area* to *entry area*, see Section 4.2.2) by a scheduling in the component, all of its concurrent activities are suspended in the same way as a non-parallel activity.

Service calls in different TO parts of a CONCURRENT statement are executed concurrently and, in case they share some component, they are competing with each other. The next statement following a CONCURRENT statement is executed after each concurrent subactivity has finished. CONCURRENT statements may be nested.

Access to objects and types local to an enclosing block is somewhat restricted in TO parts of a CONCURRENT statement. Parameters of the enclosing service and objects and types declared in a block enclosing the service (e.g., variables of an surrounding component type) can be accessed. Variables declared in the declaration part of the enclosing service or in a block "between" the service and the CONCURRENT statement cannot be accessed. Of course, variables declared in a block statement within the CONCURRENT statement can be accessed.

Examples:

```
LOOP
        CONCURRENT

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

        END CONCURRENT;

        proc3_computing (amount3);
END LOOP;


TYPE ct COMPONENT (c_para : REAL);
        PROVIDE
                SERVICE s (s_para : REAL);
        END PROVIDE;

        VARIABLE c_var : REAL;

        TYPE s SERVICE (s_para : REAL);
                USE
                        SERVICE     s1 (m : REAL);
                                    s2 (m : REAL);
                END USE;

                VARIABLE s_var : REAL;
        BEGIN

                BLOCK
                        VARIABLE block_var1 : REAL;
                BEGIN

                        CONCURRENT

                                s1 (c_para);                    {1}
                                s2 (c_var);                     {2}
                        TO
                                s1 (s_para) ;                   {3}
                                s2 (s_var);                     {4}
                        TO

                                BLOCK
                                        VARIABLE block_var2 : REAL;
                                BEGIN
                                        s1 (block_var1);        {5}
                                        s2 (block_var2);        {6}
                                END BLOCK;

                        END CONCURRENT;
                END BLOCK;
        END TYPE s;
END TYPE ct;
```

Applying the rules above, you can see that {4} and {5} are not valid, whereas {1}, {2}, {3} and {6} are.

It has already been mentioned that TO parts of a CONCURRENT statement are executed concurrently in model time. Internally, a new subprocess is generated for each TO part, similar to process generation by CREATE statements. Its type is generated by the HIT system and it executes the statements of the TO part. A process issuing a CONCURRENT statement continues its activities after all subprocesses (one for each TO part) have finished. While waiting for its subprocesses, the process and its subprocesses are still being controlled by the component which provides the service declaration for the statements currently being executed by the process.

Two situations are conceivable:

- With regard to the service declaration, no services have been called prior to the CONCURRENT statement, i.e., the very first service is called in the CONCURRENT statement:

  While its subprocesses are executed, the process stays in the *service area* (*entry area*, if it is preempted by a rescheduling) of the component. If the process encounters another service call after all subprocesses have finished, it descends into a corresponding lower layer component.

- Another service has already been called prior to the execution of a sequence of CONCURRENT statements without any call of a service between them:

  In this case, the process also remains under the control of the lower layer component providing the previously called service. More precisely: the process stays in the lower layer component's *exit area* until all concurrent subprocesses of the sequence of CONCURRENT statements have terminated. If the process encounters a service call after the CONCURRENT statements, the accept-offer-mechanism is performed between the lower layer components of the service calls prior to and after the execution of the CONCURRENT statements.

  Subprocesses of TO parts are regarded as separate processes in this context, so there is no service called prior to them. Therefore, also the operational semantics of nested CONCURRENT statements is defined by this rules.

  An example was given above: Since CONCURRENT is called in a LOOP, all but the first execution of the CONCURRENT statement is preceded by the execution of *proc3_computing*.

CONCURRENT statements are not allowed in models to be analyzed analytically.

4.1.5.2. Spend and Hold

The predefined services *spend* and *hold* can be used in the statement part of a service to model time consumption without having to declare a component of type *server* and using its provided service *request*. The obligatory actual parameter of *hold* and *spend* specifies the model time to be spent. The difference between *spend* and *hold* is that *spend* does respect the *dispatch* control procedure of the component type enclosing the service, whereas *hold* does not. *Hold* only expresses a delay, while *spend* requests service. The actual time a process spends in a *spend* statement is a function of the actual parameter supplied to *spend* and the service speeds (as determined by the *dispatch* procedure) of the component.

Note that *spend* calls can only be preempted by another *spend* call, if a *schedule* procedure allows the preemption of processes.

The *spend* call can effectively be used to replace *request* calls on *servers* of the "Infinite Server" or "Processor Sharing" type. The main advantage is that time consumption in a service does not have to be realized by a declaration of a *server* and a call on the service of that *server*. All *spend* calls in a service declared at the topmost layer of a model are controlled by the default *dispatch* procedure as it is not possible to specify control procedures at the topmost layer of a model.

A process calling *hold* is disabled for exactly the period (in model time) given as the actual parameters to *hold*, not influenced by a preemption or a change of service speeds by component control procedures. It resumes its activities after this period is exhausted.

Parameters of *spend* and *hold* must be of type INTEGER or REAL. Actual parameters less than zero are forced to zero.

Examples:

```
TYPE polling SERVICE (poll_time : REAL);
        USE
                SERVICE initiate_work;
        END USE;
BEGIN
        LOOP
                initiate_work;
                hold (poll_time);
        END LOOP;
END TYPE polling;
```

{within a lower level component}

```
TYPE initiate_work SERVICE;
BEGIN
        spend (2.5);            {expressed, e.g., in milliseconds}
END TYPE initiate_work;
```

4.1.5.3. CHAIN Statements

Besides the algorithmic description of load patterns in services HI-SLANG offers two CHAIN statements to describe "graphically" specified load patterns. The nodes in the graph represent the service calls and the arcs describe the selection of the next service call or the termination of the process. The arcs are weighted by selection probabilities. The station-class pairs of a chain in a queueing network correspond to service calls in a CHAIN statement. This allows an easy transformation of queueing network models to HI-SLANG models. For example, for the graphical queueing network analyzer SIQUEUE-PET also developed in Dortmund (see /Deik89/) this transformation has been implemented. An integration of graphically specified services to HITGRAPHIC is in preparation.

There are two CHAIN statements, which can be used independently from the analysis method: one for open chains and the other for closed chains:

```
compound_statement ::= ...
        open_chain_statement
|       closed_chain_statement


open_chain_statement ::=
        OPEN_CHAIN  [ arrival-prob_part ]
                qnode [  ...]
        END OPEN_CHAIN ;


closed_chain_statement ::=
        CLOSED_CHAIN
                qnode [  ...]
        END CLOSED_CHAIN ;


qnode ::=
        QNODE  qnode-identifier  [prob_part]


prob_part ::=
        {       PROB simple_real_expression     : qnode-identifier ; } [  ...]
        [       ELSE                            : qnode-identifier ; ]
```

One of these two statements may only occur within the body of a service and then the body may only consist of this statement. Several nodes (QNODE) describe the service calls, and the PROB part of each node describes the selection probabilities for the successor nodes and their names.

The names of the nodes are those names which are specified in the USE part of the service and the actual parameters of the service calls have to be given once for all calls as default value in the USE declaration.

Every name of a node may only occur once in the QNODE parts of one CHAIN statement and every node which is used as a destination has to be defined as QNODE. Even service arrays can be used. In this case one array index has to be supplied for each node name (confirm syntax for *identifier*). In this case all nodes of an array have the same actual parameters specified as default.

But all array indices for the qnode-identifier as well as in the expressions of PROB (simple_real_expression) must be constant values.

In the optional ELSE part the destination node is specified which is reached with the probability computed as the difference of 1.0 and the sum of the specified probabilities in the CHAIN statement.

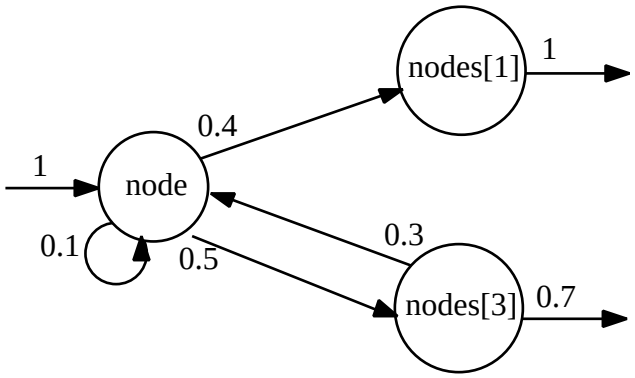
Specifying closed chains the sum of the denoted probabilities has to be equal to 1.0, otherwise the analyzer will report an error message.

Describing open chains the sum of the probabilities in a PROB part has to be smaller than or equal to 1.0, otherwise the analyzer will report an error message. If the sum of the probabilities in a PROB part of a QNODE is smaller than 1.0, the complementary probability is taken as exit probability for the OPEN_CHAIN statement. If the PROB part of a QNODE is missing, the execution of the OPEN_CHAIN statement finishes deterministically with this node.

The arrival PROB part of the OPEN_CHAIN statement defines a probability distribution for the selection of the first node to be executed. If the sum of probabitlities is smaller than 1.0, with the complementary probability no node of this OPEN_CHAIN is executed at all. In contrast to the QNODE description a missing arrival PROB part means that the syntactically first QNODE is selected deterministically as the first node of execution.

Please note that the evaluation of all probabilities of one QNODE is performed before any decision about the destination node is done; i.e., after executing the qnode statement all expressions (simple_real_expression) of corresponding prob_parts are evaluated and the sum is checked (equal to 1.0; greater/smaller than 1.0). After this check the prob part (node) is chosen for which the sum of the expressions (simple_real_expression) is greater than the drawn random number for the first time.

Example:



The open chain as described above can be described by the following service:

```
TYPE net SERVICE;
   USE   SERVICE node (duration: REAL DEFAULT negexp(3));
                 SERVICE
                 ARRAY nodes (duration: REAL DEFAULT negexp(6));
   END USE;

BEGIN
   OPEN_CHAIN
         QNODE node                          {entry node, since there is no arrival-
prob_part}
                 PROB 0.1     : node;
                 PROB 0.4     : nodes [1];
                 ELSE         : nodes [3];
         QNODE nodes [1]                     {exit node}
         QNODE nodes [3]
                 PROB 0.3     : node;        {else exit}
   END OPEN_CHAIN;
   END TYPE net;
```

4.2. Components and Component Types

A HI-SLANG component is an autonomous dynamic system. Components are the most important building blocks of a model, since they are self-contained and have a well-defined interface. Moreover components control all processes which have been created within them. As a rule components are generated statically either by component declarations of component types or by declaring component objects directly.

The usage of component types is meaningful if many components of the same kind exist in the model. They can, of course, differ concerning their parameter values. On the other hand the usage of a direct component declaration is meaningful if a component of this kind is used only once within the model.

This chapter describes the declaration and structure of component types (see Section 4.2.1.), the operation of components (Section 4.2.2.), the declaration of component control procedures (Section 4.2.3.) and the declaration of component objects (Section 4.2.4.). The next section (4.3.) presents all standard component types of HI-SLANG.

4.2.1. Component Types

Components can be generated as instances of component types. Component type declarations can be given in declaration parts of model or component types, of an EXPERIMENT block, and on the same level as model types, textually preceding the EXPERIMENT block.

componenttype_declaration ::=
TYPE componenttype-name **COMPONENT** [formal_parameters] ;
[provide_declaration_part]
[collect_block]
[control_declaration_part]
[declaration [...]]
[refer_part]
[**BEGIN**
sequence_of_statements]
END TYPE [componenttype-name] ;

A component type declaration starts with a header comprising a type name and (optional) formal parameters. The header is normally followed by the declaration of those services and procedures provided for external use by the component type (*provide_declaration_part*). Next a collect_block may follow for upward compatibility reasons, but is no longer necessary, since all user-defined streams declared within *declarations* are collected automatically. An optional *control_declaration_part* describes how the execution of provided services is controlled by component control procedures.

Local declarations, a REFER part and a statement part may follow. Component type declarations are bracketed by TYPE - END TYPE pairs with an optional type name repetition. The *componenttype-name* is a unique identifier to be used in component declarations. It can be chosen freely under the rules governing scope and validity of identifiers.

Objects local to the component type and accessible in the *sequence_of_statements* part can be declared in the *declaration* part. Constants, variables, records, pointers, and procedures can be declared as well as services, component types, and corresponding objects. Virtual (enclosed) components can also be declared here.

The optional statement part starting with BEGIN is executed at declaration time of a corresponding object and is used for initialization purposes, e.g., local process generation. With the exception of RESULT, EVALUATE, AGGREGATE and UPDATE, any statement which does not consume model time may occur here. Any referenced object must be known according to the scope rules.

Component types are basic building blocks from which models are constructed. They have a properly defined interface to an environment which is given by

- the services and procedures provided to the environment (PROVIDE declaration),
- the streams used for performance evaluation purposes (former COLLECT block, now a set of stream declarations),
- its parameters,
- its ENCLOSED components (virtual declarations),
- the access to global objects, depending on scope rules.

Principles of structured modelling promote the opinion that only very restricted use should be made of the last facility.

Example:

```
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;
```

4.2.1.1. Component Types with Parameters

Component types can be parameterized to facilitate the generation of similar, yet individually different component objects. Its parameters can be accessed within the component type. Formal parameters are substituted by actual parameters in a component object declaration.

The rules for parameter transmission modes and parameter types are the same as for services with parameters, i.e., a "call by name" is not allowed here. The rules for formal parameter specification and parameter substitution by actual values are the same as for procedures. The scope rules for identifiers have to be observed, e.g., parameter names must be different from names of objects or data declared locally in the component type.

Example:

```
{declaration of a component type with formal parameters}

TYPE os_type COMPONENT    (virtual_storage          :    INTEGER DEFAULT 2;
                           mean_disk_access_time    :    REAL;
                           number_of_cpus           :    INTEGER DEFAULT 1);
    ...
END TYPE os_type;

{declaration of corresponding component objects}

COMPONENT    operatingsystem    : os_type (  , 0.005, 2);
             os3                : os_type (4, 0.001, 3);
```

4.2.1.2. PROVIDE Declaration

A PROVIDE declaration lists those services and procedures local to a component type which are intended to be used (called) by the environment. Used (called) services or procedures have to be bound to services or procedures respectively provided by a component in the REFER part of the surrounding component type.

```
provide_declaration_part  ::=
    PROVIDE
        provide_declaration  [ ...]
    END PROVIDE ;


provide_declaration  ::=
    procedure_or_service {procedure_or_service-name [formal_parameters]
        [RESULT simple_type  [, ...] ; } [ ...]
```

Of course, a service or procedure listed in a PROVIDE declaration must be declared in the declaration part of the component type. Furthermore, the specifications given in a PROVIDE declaration must correspond to those in the declaration part of the component type. *procedure_or_service-name* must be identical to the name given in the declaration part. Syntactically, a PROVIDE declaration equals a "regular" declaration and its constituent parts have the same meaning.

The predefined procedures *popul*, *popul_announce*, *popul_entry*, *popul_service* and *popul_exit* of each component type are automatically provided and may not be listed in a PROVIDE declaration.

Example:

```
PROVIDE
      SERVICE    compute      (ctime      :     REAL);
                 disk_access  (disk_no    :     INTEGER)  RESULT INTEGER;
END PROVIDE;
```

4.2.1.3. COLLECT Block

The set of collected streams (cf. Section 5.1) forms the connection between the model and its evaluation.

The COLLECT block currently is only present in HI-SLANG for reasons of upward compatibility. Since streams declared in component or model types are collected implicitely, the COLLECT block is no longer needed for that purpose.

Only streams declared in services still have to be collected in the surrounding component type, but such stream declarations should be avoided, i.e., the stream declaration should be moved to that component type. This has the additional advantage, that such streams can be updated from different services.

If a user-defined stream declared in a service is required, then the stream must be made visible to the environment. A corresponding specification is given in the COLLECT block of the surrounding component type.

User-defined streams and correspondingly COLLECT blocks may only occur in models to be analyzed by the simulative solver.

collect_block :=
COLLECT
{ [service-name .] stream-name [**AS** external_stream-name] ; } [...]
END COLLECT ;

The stream *stream-name* must be declared in a service named *service-name*, or in the component type. The stream can be accessed in a model evaluation by its *external_stream-name*. The *external_stream-name* may be any unique identifier within the component type. If the optional AS part is omitted, the stream can only be accessed by *service-name*. *stream-name* in the experiment description.

In general, streams declared in services and not listed in a COLLECT block cannot be accessed during model evaluation. However, streams declared in component and model types are all implicitely collected. This especially holds for the standard streams, which can always be accessed (if they make sense for that component).

Example:

```
COLLECT
        read.access   AS readdisk;
        write.access AS writedisk;
        write.fault;
END COLLECT;
```

4.2.1.4. REFER Part

A load is referred to a machine in the REFER part of a component type declaration. Services and procedures used by services declared in the component type (the load) are referred to services and procedures provided by subcomponents (the machine).

refer_part ::=
REFER procedure_or_service-name [, ...] **TO** component-name [, ...]
EQUATING
{use-identifier **WITH** provide-identifier [**OF** service-name] ; } [...]
END REFER ;

The *procedure_or_service-name* is the name of a procedure or service (declared locally in the component type) with a USE declaration, i.e., the procedure or service uses some other services or procedures. *component-name* is the name of a subcomponent or of a subcomponent array declared in the component type and providing services or procedures via a PROVIDE declaration.

The *use-identifier* denotes an external reference made by a service or procedure. It consists of a service or procedure name, a separating dot (.) and the name of a service, service array, procedure or procedure array used by the service or procedure preceding the dot. This service or procedure name must be a member of the list of names following the REFER keyword. The name following the dot must be listed in the USE declaration part of the service or procedure preceding the dot. The *use-identifiers* are made up of service names, service array names, procedure names and procedure array names, i.e., index ranges, parameter and result specifications are omitted.

The *provide-identifier* denotes a service or procedure provided by a subcomponent. It consists of a subcomponent name or a subcomponent array name, a separating dot (.) and the name of a provided service or procedure. The subcomponent (array) name must be a member of the list of names following the TO keyword. The provided service or procedure name must either be listed in the PROVIDE declaration part of the subcomponent or be one of the predefined procedures *popul, popul_announce, popul_entry, popul_service* or *popul_exit.* Component arrays are specified by their name, i.e., parameters, array dimensions or index ranges are omitted.

Used procedures can be bound to the predefined procedures *popul, popul_announce, popul_entry, popul_service* or *popul_exit.* In this case an optional *OF service-name* clause indicates a service of which a procedure call returns population numbers. The *service-name* must be a name of a service declared in the subcomponent specified by the *provide-identifier*. It restricts the function value to the population of that service.

Every service or procedure listed in a USE declaration part of a service has to be bound to some service or procedure PROVIDEd by a subcomponent in the REFER part of the surrounding component type. Services must be bound to services and procedures must be bound to procedures, of course.

Corresponding USE and PROVIDE declarations must have corresponding formal parameters and result specifications. The types INTEGER and REAL are not compatible here. Even if a parameter of a provided service has a default value, it must be given (with no default or a possibly different default value) in the corresponding USE declaration. Service (procedure) arrays can only be referred to component arrays.

The load, represented by services, is REFERred to the machine, represented by components, at the service level. This implies that every process of a given type USEs the same procedures and services. The binding of USEd to PROVIDEd services, for

example, determines the service and component to which a process descends when a used service is called. Similarly, the call of a USEd procedure results in the call of the corresponding PROVIDEd procedure.

Different services can effectively use the same service or procedure provided by a component. This is the case whenever a *provide-identifier* occurs more than once in a REFER part. On the other hand, the same *use-identifier* may occur only once in a REFER part. Similarly, different procedures can effectively call the same procedure provided by a component, but a procedure may not call a service.

Example:

```
REFER      read, write, supervisor          TO          disk_controller
EQUATING
      read.diskaccess          WITH^   disk_controller.access;
      write.diskaccess         WITH    disk_controller.access;
      supervisor.diskdemands   WITH    disk_controller.popul OF access;
END REFER;
```

4.2.2. Component Control

Every component is a highly autonomous dynamic system, its task being the organization of the progress of interacting processes. We can distinguish between two kinds of processes running in a component: A *local process* is generated inside the component by a process declaration or by CREATE/SUBMIT statements. It is an instance of a service declared in the component type. On the other hand, processes local to some higher layer component descend into a component when they call a service provided by that component.

With respect to its progress in a component, a process can be in one out of three possible states: progress can be denied; progress can be granted, or, thirdly, a process can have finished its activities in the component.

4.2.2.1 Component Areas

The component reflects the process status by being subdivided into three component areas (*entry area*, *service area* and *exit area*), each area containing processes of the corresponding state. Yet another queue (*announce queue*) maintains processes waiting to enter the entry area of a component:

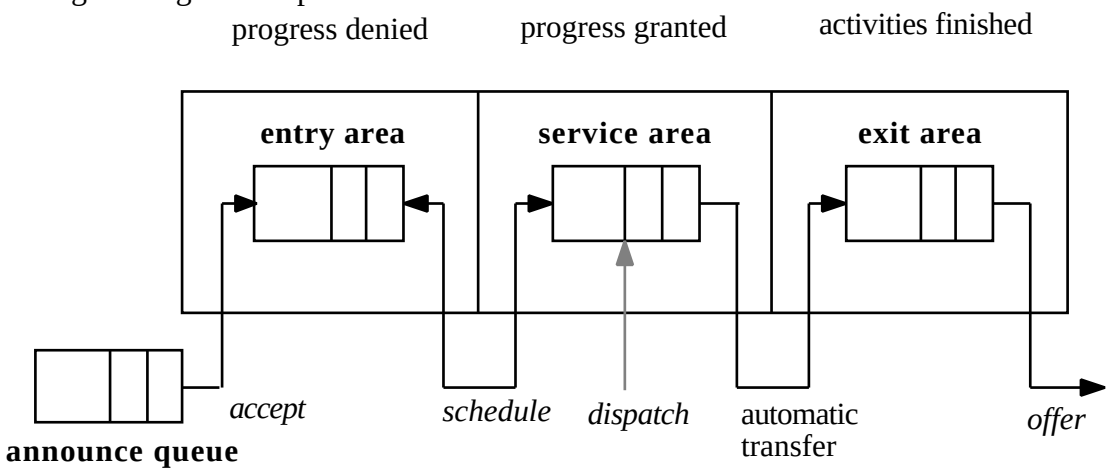
- The **entry area** of a component contains processes whose request of a service provided by the component has been accepted, but whose activities have not yet commenced or have been suspended.

- The **service area** contains processes currently in progress.

- The **exit area** contains processes whose request of a service provided by a component has been fulfilled, but they have not yet been offered by the component or they have not yet been accepted by the component of their next service call (e.g., because of capacity restraints).

- Every component maintains an **announce queue** containing references to processes requesting a service provided by the component but their request has not yet been accepted by the component or they are currently not offered by the component of the previous service call.

A diagram might be helpful:



4.2.2.2. Component Control Procedures

Four control procedures are responsible for the assignment of processes to areas, i.e., they control the acceptance and offering behaviour of the areas: *accept, schedule, dispatch* and *offer*. A user can write his own HI-SLANG control procedures, choose from predefined standard procedures, or fall back to the defaults. All predefined standard procedures are members of the HIT standard mobase, and are written directly in Standard Simula to increase efficiency.

In more detail:

- **accept:** An *accept* procedure controls the transfer of processes from the *announce queue* to the *entry area* of a component.

- **schedule:** A *schedule* procedure controls the transfer of processes between *entry* and *service area.* Processes in the *entry area* can be transferred to the *service area* (progress is granted) and vice versa (further progress is denied).

  In case of a preemption (progress denied), a process remembers its remaining service demand so that upon a rescheduling into the *service area* the process resumes its activities where they were interrupted. Processes running in components not of type *server* can be preempted only between *spend* calls or service calls listed in the USE declaration and provided by some lower layer component. As an exception to the rule, *spend* calls can be interrupted by another *spend* call, and then the process is preempted immediately.

- **offer:** A process is automatically transferred from the *service area* to the *exit area* of a component when the service execution in the component is finished. Furthermore, the process may progress in the higher layer up to the next call of a service. Meanwhile, the process remains in the *exit area* waiting to be accepted by the next component (providing the next requested service), but only those processes selected by an *offer* procedure are actually allowed to leave the component. Processes offered but not accepted are blocked. If the service call is the last in the higher layer, the process leaves the *exit area* with the transfer from *service* to *exit area* in the higher layer.

- **dispatch:** Processes in the *service area* are further controlled by a *dispatch* procedure. The procedure assigns service speeds to processes.

  Service speeds determine the progress of calls to the predefined service *spend*, by which also the service *request* of *server*s is implemented. Their influence is limited to the current model layer, i.e., *spend* calls of a process in some lower layer subcomponent are not influenced. An example: A service speed of 2.0 implies that from now on a service request of 6.0 in a spend call has an effective duration of just 3.0 model time units.

  Examples:

spend call	service speed during the spend call	effective model time spent by the spend call
spend (2.5)	0.25	10.000
spend (2.5)	0.50	5.000
spend (2.5)	1.00	2.500
spend (2.5)	2.00	1.250
spend (2.5)	4.00	0.625

It has already been mentioned that service speeds as determined by the *dispatch* procedure are only relevant for processes in the service area calling a *spend*. A component's service speeds are immaterial for processes in the component's *service area* calling a service of a lower layer component, since they are under control of that component at that time.

Every component type has formal parameters *accept*, *schedule*, *dispatch* and *offer* which can have corresponding control procedures as actual parameters. Control procedures are always the textually last formal parameters in a component type declaration (but they are not explicitly listed). Actual parameters can be given as positional parameters (in the order *accept, schedule, dispatch, offer*) or as keyword parameters. Default values are:

accept	ALWAYS
schedule	IMMEDIATE
dispatch	EQUAL (1.0)
offer	ALL

Some of the predefined component types (see Appendix F.1.) have a different schedule default if IMMEDIATE is not meaningful for that type.

4.2.2.3. The State of a Component

The implicit state of a component (versus its explicit state which is given by the values of its parameters) is given by the vector of current component area population numbers, i.e., number of processes in a component area. Population numbers can be retrieved by predefined parameterless INTEGER procedures, which every component automatically provides.

- **popul_announce** retrieves the number of process references in the *announce queue*
- **popul_entry** retrieves the number of processes in the *entry area*
- **popul_service** retrieves the number of processes in the *service area*
- **popul_exit** retrieves the number of processes in the *exit area*
- **popul** retrieves the total number of processes (=*popul_entry* + *popul_service* + *popul_exit*)

A population change only occurs after the execution of a control procedure because a process is transferred automatically to another area only after having been selected by a control procedure. Thus, during the execution of a control procedure population numbers will remain constant.

If a state retrieval procedures is called and subcomponent populations are to be retrieved (populations of components of the next-lower level), a USE declaration in the service and a binding to the subcomponent in the REFER part of an surrounding component type is required (see next example).

The OF operator provides a mean to retrieve population numbers of a specific service only. It takes a *popul* procedure and a *service-name* as arguments and yields a new restricted *popul* procedure:

OF:	popul_announce popul_entry popul_service popul_exit popul	x	service-name	|→	INTEGER procedure

The OF operator can be used directly in statement parts of services to retrieve population numbers of a specific service of the surrounding component type. The OF operator has the highest priority of all operators. It is not allowed to be used in models which are to be analyzed by an analytical solver. It is also not allowed to be used inside a *dispatch* procedure. The OF operator can also occur in REFER parts in order to retrieve subcomponent population numbers of a specific provided service only.

Example:

```
TYPE ct COMPONENT (...) ;

    TYPE s SERVICE (...) ;
        USE
            SERVICE         compute (...);
            PROCEDURE       filling RESULT INTEGER;
        END USE;

    BEGIN
        …
        IF filling < 5 AND POPUL OF s < 10 THEN compute (...);
        END IF;
        …
    END TYPE s;

    COMPONENT cpu : cpu_type; {provides add, mult, ...}

    …
    REFER s, ...        TO    cpu, ... EQUATING
        s.compute       WITH cpu.add;
        s.filling       WITH cpu.popul OF add;
        …
    END REFER;
    …
END TYPE ct;
```

4.2.2.4. The State of a Process

The **implicit state of a process** (versus its explicit state which is given by the values of its parameters) can be retrieved by the following parameterless procedures:

- **arrival_announce** returns the arrival time at the *announce queue*
- **arrival_entry** returns the arrival time at the *entry area* (this value is also updated when a process is preempted, i.e., transferred from *service area* to *entry area*)
- **arrival_service** returns the arrival time at the *service area* (this value is also updated when a process resumes its activities after a preemption)
- **arrival_exit** returns the arrival time at the *exit area*
- **arrival** returns the arrival time at the component, i.e., the time of the first arrival at the entry area.

These REAL procedures return "-1" if a process has not yet been in the area implicitly given by the procedure name or has left the component completely. There are two further parameterless procedures predefined for all services:

- **preempted** returns a BOOLEAN and is TRUE if the process has been preempted (transferred from *service area* to *entry area* by a rescheduling) at least once. If the process has left the component this procedure returns FALSE.
- **speed** returns a REAL value which is the service speed most recently assigned to the process by a SETSPEED statement in a *dispatch* procedure. Returns "-1" if this has not yet happened or the process has left the component completely.

If these procedures are applied to named processes, they are related to the component the process is local in. Otherwise, they are related to the component they are called in.

The (explicit or implicit) state of a process can be accessed in the body of a corresponding service and in WHEN parts of an INSPECT statement. Of course, the WHEN part must specify the service of the process whenever service parameter values are accessed. The implicit state retrieval procedures listed above may also be called in INSPECT statements without WHEN parts and in ELSE parts of INSPECT statements because every process has the same implicit state retrieval procedures. The state of a process can also be accessed via dot notation or WITH statements and process names.

4.2.3. HI-SLANG Control Procedures

The user can choose component control procedures from a set of predefined standard control procedures or, in a simulation model, write his own component control procedures in SIMULA (which is no longer supported, but still possible to be upward compatible) or, more conveniently, in HI-SLANG. In the latter case, one or more control procedure declarations are given in a CONTROL block of a component type declaration.

control_declaration_part ::=
    CONTROL
        control_procedure_declaration [...]
    END CONTROL ;

control_procedure_declaration ::=
    PROCEDURE control_procedure-name ;
        [common_declaration [...]]
    [**BEGIN** sequence_of_statements]
    END PROCEDURE [control_procedure-name] ;

The *control_procedure-name* must be *offer*, *accept*, *schedule* or *dispatch*. As indicated by the syntactical description, control procedures may not have formal parameters.

HI-SLANG component control procedures may contain any declaration or statement of the programming kernel of HI-SLANG. Additionally, several special purpose statements are available, e.g., the INSPECT statement (see the next section). Control procedures may contain local procedure declarations. Local or global procedures can be called in the *sequence_of_statements*. Global procedures are declared outside a CONTROL block and must not contain INSPECT statements. The OF operator can be used to retrieve the state of the component.

Example:

```
CONTROL
    PROCEDURE schedule;   …   END PROCEDURE schedule;
    PROCEDURE offer;      …   END PROCEDURE offer;
END CONTROL;
```

Control procedures cannot be called within HI-SLANG source code, as indicated by the CONTROL block. Rather, they are called by the HIT simulation system itself. They are called whenever certain events occur in a component. The following events could activate the following control procedures:

Control Procedure	activated by the following events
Accept	• Announcement of a process or • the procedure *Offer* is called in the preceding component (of an announced process) or • a service is finished or • a process in a higher layer component is resumed
Schedule	• Transition of accepted service into *entry area* or • Transition into the *exit area*
Offer	• Transitions into or from the *exit area*
Dispatch	• Transitions between *entry area* and *service area* or • transitions into *exit area*

The (simple) order of calls is:

offer → accept → schedule → dispatch

Note that only the activated control procedures are called. One control procedure may cause the activation of another. This leads to a repetition of the sequence until no control procedure is activated again.

A control procedures is also called when a time slice, as determined by its TIMESLICE statements, is exhausted.

HI-SLANG control procedures override the respective default control procedures (ALWAYS, IMMEDIATE, EQUAL (1.0), ALL). They in turn can be overridden by predefined standard control procedures whenever standard control procedures are given as actual parameters in a corresponding component declaration.

4.2.3.1. The Spend Server

Note that *dispatch* procedures affect only those processes in a *service area* of a component which are currently performing a *spend* call (see Section 4.1.5.2). Such *spend* calls in services are bound to the service *request* provided by a so-called Spend Server. There is a Spend Server in every component containing services with *spend* calls.

The *dispatch* procedure of a component is used only by the Spend Server in order to determine service speeds for processes calling *spend* in the component. Thus, it is not used by the component itself. Consequently, INSPECT statements, component state retrieval procedures such as *popul*, etc. in a HI-SLANG *dispatch* procedure refer to the Spend Server and not to the component itself.

Please note that the OF operator is not allowed to be used inside a *dispatch* procedure.

Nevertheless, the component state can be retrieved in a *dispatch* procedure by calling another procedure which is declared locally in the component.

Example:

```
TYPE ct COMPONENT;
        …
        CONTROL
                PROCEDURE dispatch;
                BEGIN
                        IF filling <10 THEN
                                INSPECT SERVICE_AREA
                                LOOP
                                        SETSPEED 1/filling;
                                END LOOP;
                        END IF;
                END PROCEDURE dispatch;
        END CONTROL;
        …

        PROCEDURE filling RESULT INTEGER;
        BEGIN
                RESULT popul;
        END PROCEDURE filling;
        …
END TYPE ct;
```

The Spend Server uses a *schedule* procedure which allows the preemption of *spend* calls. The call is put into the *entry area* if the calling process is in the *entry area*, it is put into the *service area* if the calling process is in the *service area*. Note that this *schedule* procedure is only executed if there is a new *spend* call at the component or a *spend* at the component has finished. Preempted *spend* calls have no progress concerning their service *request*.

4.2.3.2. INSPECT Statement

Component areas can be investigated in INSPECT statements. Processes in corresponding component areas can be manipulated in the statement part of the INSPECT statement. INSPECT statements may only occur in component control procedures or in services.

```
compound_statement ::= ...
|       inspect_statement


inspect_statement ::=
        INSPECT area [WHILE boolean_expression]
        LOOP [REVERSE]
                when_or_sequence
        END LOOP;


when_or_sequence ::=
        sequence_of_statements
|       {WHEN service-identifier   :       sequence_of_statements } [ ...]
        [ ELSE                     :       sequence_of_statements ]


area ::=
        ANNOUNCE_QUEUE
|       ENTRY_AREA
|       SERVICE_AREA
|       EXIT_AREA
```

In an INSPECT statement without WHEN/ELSE, the *sequence_of_statements* is executed once for every process in the specified *area.* If the area is empty, the *sequence_of_statements* is skipped. If not, the *sequence_of_statements* is executed for every process in the area in the order given by the time of arrival at the area. LOOP REVERSE reverses the order, i.e., the latest arrivals are treated first.

The optional WHILE part can be used to abort the execution of the process inspection loop. The controlling *expression* must be of type BOOLEAN.

The *boolean_expression* is reevaluated each time before the execution of the statements in *when_or_sequence.* If it returns TRUE, the *when_or_sequence* is executed for the current process. If it returns FALSE, the INSPECT loop is aborted. The loop is skipped if the *boolean_expression* returns FALSE at its first evaluation.

No distinction is made between processes of different types in an INSPECT statement without WHEN/ELSE. Therefore, only the implicit process state, e.g., time of arrival at the area, can be accessed in the statement part.

In order to access other process state variables, e.g., process parameters, the INSPECT statement has to be used in conjunction with WHEN. The identifier following WHEN must be the name of a service declared in the enclosing component type. In the corresponding statement part the state variables of a process of this type currently under investigation in the INSPECT loop can be accessed without having to use the dot notation. Different WHEN parts can be used to distinguish between different services. The optional ELSE part is executed for processes of a type not mentioned in the WHEN

part. Therefore, only the implicit process state, e.g., time of arrival at the area, can be accessed in the ELSE part.

Under the usual rules governing the scope of identifiers, all kinds of objects and types can be referenced in statement parts of INSPECT statements. One exception is that in a WHEN part parameter names of the service *service-identifier* can hide global identifiers. Of course, objects declared locally in the INSPECT statement can also be accessed. Scope rules must be observed. The call of services inside the statement part of the INSPECT statement is not allowed because time consumption can possibly disturb the order of the traversed area.

Only those processes in an *announce queue* which are actually allowed to be accepted by a component are inspected in an INSPECT statement in an *accept* procedure. A process is allowed to be accepted by a component if it is offered (as determined by the *offer* procedure) by the component whose *exit area* the process is in (if any) and if it is in the *service area* of the next higher layer component.

Examples:

```
INSPECT ANNOUNCE_QUEUE WHILE nr < 10 LOOP REVERSE
        SELECT;
        nr  := nr + 1;
END LOOP;

{nr is a local variable of this accept procedure }
{SELECT statements are explained below }

INSPECT ENTRY_AREA LOOP
        WHEN allocate       :       IF sched + number <= tokens THEN
                                                SELECT;
                                                sched  := sched + number;
                                                END IF;
        WHEN release        :       SELECT;
        ELSE                :       WRITELN "unknown service";
END LOOP;

{ sched is a local variable of  this schedule procedure}
{ tokens is a variable of  the enclosing component}
{ number is a parameter of service allocate}
```

4.2.3.3. SELECT, SETSPEED and TIMESLICE

SELECT, SETSPEED and TIMESLICE statements may only be used in component control procedures.

```
simple statement ::= ...
|       control_procedure_statement


control_procedure_statement ::=
        SELECT ;
|       SETSPEED     simple_real_expression ;
|       TIMESLICE    simple_real_expression ;
```

SELECT statements may only occur at very specific locations within a control procedure. They

- may occur within an INSPECT statement of an ***accept*** procedure. The inspected area must be the *announce queue.*
- may occur within an INSPECT statement of a ***schedule*** procedure. The inspected area must be the *entry area* or the *service area.*
- may occur within an INSPECT statement of an ***offer*** procedure. The inspected area must be the *exit area.*
- may not be used in ***dispatch*** procedures.

A currently INSPECTed process is marked for further treatment if it is SELECTed in a control procedure. The type of control procedure determines what is to be done with the selected process:

- **accept**: After the *accept* procedure has exited, the *announce queue* is searched for selected processes. Every selected and offered process is transferred from the *announce queue* to the *entry area,* i.e., the selected processes are accepted by the component.
- **schedule**: After the *schedule* procedure has exited, the *entry area* is searched for selected processes. Every selected process is transferred from the *entry area* to the *service area,* i.e., the process is granted progress in its service request. Likewise, the *service area* is searched for selected processes. Every selected process is transferred from the *service area* to the *entry area,* i.e., the process is preempted. Processes resume their activities at the point of preemption when they are rescheduled to the *service area* ("preemptive-resume")
- **offer**: After the *offer* procedure has exited, the *exit area* is searched for selected processes. Every selected process is offered by the component, i.e., may be accepted by some other component.

Process transfers, changes of component states (e.g., changes in population counts in component areas), process states, and other necessary actions are executed automatically after a control procedure has exited.

SETSPEED statements may only occur in INSPECT statements within a *dispatch* procedure. The inspected area must be the *service area*. SETSPEED determines the service speeds of *spend* calls of the process currently under investigation. The *expression* denotes the new service speed assigned to the process and must result in an INTEGER or REAL greater than or equal to zero.

Time slicing disciplines can be modelled by TIMESLICE statements. The *expression* denotes a time slice (duration) and is a positive REAL or INTEGER. The control procedure containing the TIMESLICE statement is reinvoked automatically each time after the specified time slice is exhausted. If a TIMESLICE statement is executed more than once for the same control procedure the time sequence to reinvoke this procedure is redefined, i.e., the old sequence is removed and the sequence defined by the last executed TIMESLICE statement is valid from now on.

TIMESLICE statements should not be used within INSPECT statements.

Example:

```
{HI-SLANG declaration of default component control procedures, see also Section 4.2.2.2.}

CONTROL

    PROCEDURE accept;                          {ACCEPT ALWAYS}
    BEGIN
        INSPECT ANNOUNCE_QUEUE LOOP
            SELECT;
        END LOOP;
    END PROCEDURE accept;

    PROCEDURE schedule;                        {SCHEDULE IMMEDIATE}
    BEGIN
        INSPECT ENTRY_AREA LOOP
            SELECT;
        END LOOP;
    END PROCEDURE schedule;

    PROCEDURE dispatch;                        {DISPATCH EQUAL (1.0)}
    BEGIN
        INSPECT SERVICE_AREA LOOP
            SETSPEED 1.0;
        END LOOP;
    END PROCEDURE dispatch;

    PROCEDURE offer;                           {OFFER ALL}
    BEGIN
        INSPECT EXIT_AREA LOOP
            SELECT;
        END LOOP;
    END PROCEDURE offer;

END CONTROL;
```

Example:

{Implementation of a last-come-first-scheduled preemptive resume discipline in a *schedule* procedure}

```
PROCEDURE schedule;
   VARIABLE found : BOOLEAN DEFAULT  FALSE;
BEGIN
   INSPECT ENTRY_AREA WHILE NOT found LOOP REVERSE
         SELECT;
         found := TRUE;
   END LOOP;

   IF found THEN
         INSPECT SERVICE_AREA LOOP
               SELECT;
         END LOOP;
   END IF;
END PROCEDURE schedule;
```

4.2.4. Component Declarations

Component objects are either

- instances of component types (useful if more than one component of the same kind exist)

or

- directly declared components (useful if a component of this kind is unique).

Component object declarations may occur in declaration parts of model or component types, in the declaration part of an EXPERIMENT block (which is not recommeded to clearly separate model and experiment descriptions), and anywhere textually preceding the EXPERIMENT block.

component_declaration ::=
COMPONENT
{ component-name [**,** …] **:** [**ARRAY** ⟦ array_bounds ⟧ **OF**]
componenttype-name [actual_parameters] **;** } […]

| **COMPONENT** component-name [**,** ...] [actual_parameters] **;**
[provide_declaration_part]
[collect_block]
[control_declaration_part]
[declaration [...]]
[refer_part]
[**BEGIN** sequence_of_statements]
END COMPONENT [component-name] **;**

A direct component object declaration starts with the component name and (optional) *actual_parameters* of predefined formal parameters, e.g. for some component control procedures. Normally, this is followed by the declaration of those services and procedures provided for external use by the component (*provide_declaration_part*). Next a collect_block may follow for upward compatibility reasons, but is no more necessary, since all user-defined streams declared within *declarations* are collected automatically. An optional *control_declaration_part* describes how the execution of provided services is controlled by component control procedures. Local declarations, a REFER part and a statement part may follow. Component declarations are bracketed by COMPONENT … BEGIN - END COMPONENT pairs with an optional name repetition.

A lot of characteristics of directly declared component objects are analoguous to the characteristics of the *component types* as described in Chapter 4.2.1 . For more information a look on this and the following chapters is recommendable.

If a component type is instantiated one or more components or component arrays are generated by a component declaration. In case of a direct component object declaration no arrays are allowed. The *component-name* is a unique identifier which can be chosen freely under the rules governing the validity and scope of identifiers.

Component arrays must be one-dimensional and the bounds on the index range are evaluated at compile time, i.e., expressions over constants and constant values can be used as array bounds. The rules for array bound specifications are just the same as for arrays of simple types. Component arrays do not have any array attributes.

The *componenttype-name* must be a known component type name according to the scope rules. It may be an aggregated component type as well, but in this case no *schedule* or *dispatch* control procedures can be specified. Some *actual_parameters* corresponding to formal parameters of the component type and component control procedures can be supplied here as well. The statement part of a component type body is executed when a component of that type is generated.

Examples:

```
COMPONENT
        terminals                :       server;
        peripheral_processor     :       server          (LET schedule := fcfs);
        main_processor           :       mp_type         (slicetime, scheduletime);

COMPONENT
        multiprocessor           :       ARRAY [1..10] OF processorelement;

COMPONENT  special_device (let schedule := fcfs);
…
BEGIN
…
END COMPONENT special_device;
```

4.3. Standard Component Types

The HIT standard mobase contains several component and service declarations which are intended to be the basic building blocks from which more complex objects can be constructed. This section gives a brief description of their functionality, interfaces and parameters. A detailed description is given in Appendix F.

4.3.1. Server

The component type *server* provides a basic service *request* having a single REAL parameter *amount*. *amount* denotes the service amount requested by a caller from the *server*. Thus, a *server* and its provided service *request* can be used to model time consumption. HI-SLANG notation for the component type *server* is:

```
TYPE server COMPONENT;
        PROVIDE
                SERVICE request (amount : REAL);
        END PROVIDE;
        ...
END TYPE server;
```

4.3.2. Counter

The component type *counter* is, in a way, a counterpart of the component type *server*, in that the first is used to model space consumption whereas the latter is used to model time consumption. Important application areas for *counters* are, for example, the modelling of semaphores, tokenpools, and synchronization mechanisms. Note that for some of these tasks more efficient (in terms of simulation run time) standard component types have been developed (see below).

A component of type *counter* provides the alteration of its internal state vector as a service called *change*. The service *change* has formal parameters *prio*, denoting a priority of service calls in progress, and *amount*. The *amount* is used to compute the new state vector by state vector := state vector + *amount* upon service call completion. Zero is the highest service call priority.

```
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;
```

min, *max* and *init* denote lower bound, upper bound and initialization vectors for the internal state vector. Blocking can occur if a *change* operation would exceed some lower or upper bound. Predefined control procedures are available to resolve arising conflicts between, for example, competing *change* operations.

4.3.3. Semaphor

If semaphores are to be modelled, it is more efficient (in terms of simulation run time) to use a component of type *semaphor* than to use a counter. A *semaphor* component (note the missing `e` at the end) provides parameterless services *p* and *v* which operate on the internal state variable initialized by the formal parameter *sem_init.* DEFAULT value of *sem_init* is "1", thus a binary semaphore is the default.

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

4.3.4. Tokenpool

Another variant to model space consumption is the component type *tokenpool.* Components of this type can only be declared in models to be analyzed by simulation.

Token pools (tokens plus operations) are a modelling paradigm which is also known from RESQ (see /SaMN84/): A token pool consists of a certain number of objects (tokens) and processes can access tokens via predefined functions (services). Tokens can be *allocated* for exclusive usage by a process and *released* again. New tokens can be *produced* and existing tokens can be *destroyed.* Tokens cannot be be accessed individually, i.e., there are just numbers of tokens.

Accordingly, services provided by the component type *tokenpool* have an INTEGER parameter *number* denoting the number of tokens to be allocated, released, produced, or destroyed. The parameter *no_of_tokens* gives the number of tokens initially available.

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

4.3.5. Synchsend

The parameterless component type *synchsend* is a facility to model communication (message exchange) between processes. A single component of this type can be used to implement unidirectional communication between two processes, i.e., one process acts as a sender, another as a receiver. Bidirectional communication and communication between more than two processes can be modelled by declaring several *synchsend* components or component arrays.

The message exchange between sender and receiver requires a synchronization of the two communication participants: If *send* is called before *receive*, the sending process has to wait for a *receive* call of another process, and vice versa.

The component type *synchsend* provides the services *send* and *receive*. A sending process calls *send* whenever it intends to send a message, the message text being the actual parameter supplied to *send* (formal parameter *what*). A receiver calls *receive* which returns the message text as a RESULT.

Text processing capabilities implemented in HI-SLANG (see Section 3.6) facilitate an easy construction of rather complex messages.

```
TYPE synchsend COMPONENT;
        PROVIDE
                SERVICE     send (what : TEXT);
                            receive RESULT TEXT;
        END PROVIDE;
        …
END TYPE synchsend;
```

4.3.6. Nowaitsend

Similar to *synchsend* components, components of type *nowaitsend* facilitate process communication via message exchange.

The main difference between these component types is that *nowaitsend* components can store messages in a ring buffer implemented by a TEXT array. The maximum number of messages is a parameter of nowaitsend components (*no_of_buffers* DEFAULT 1).

The process acting as a sender calls the provided service *send* with a message text as actual parameter. The process does not have to wait for a receiver if a buffer place is available ("no-wait-send"). If the buffer is full, the sender has to wait until a buffer place becomes free. The process acting as a receiver calls the provided service *receive* which returns the oldest message of the ring buffer as a RESULT and frees its buffer place. The receiver has to wait for a *send* if the buffer is empty.

```
TYPE nowaitsend COMPONENT (no_of_buffers : INTEGER DEFAULT 1);
        PROVIDE
                SERVICE     send (what : TEXT);
                            receive RESULT TEXT;
        END PROVIDE;
        …
END TYPE nowaitsend;
```

4.3.7. Ftserver

Components of type *ftserver* implement fault tolerant servers with a fixed number of *processors* ≥ 1, each processor being subject to failures described by a Poisson process. The failure rate of a single processor is given by *failure_rate* in its active state and by *dormancy*failure_rate* in its dormant state, 0 ≤ *dormancy* ≤ 1. *dormancy* is also called dormancy factor. A defective processor is repaired with rate *repair_rate* if one of the *repair units* is available.

The service call to be reactivated after a repair is selected randomly, each with the same probability. A maximum degradation 0 ≤ *degmax* ≤ *processors* gives the maximum number of processors simultaneously in the failed state.

An *ftserver* component provides a service *request* with service demand *amount* and a priority *prio* as formal parameters. Zero is the highest priority. The default schedule strategy is PRIONP, which equals RANDOM when parameter *prio* of *request* calls is not set or all priorities are equal.

```
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;
```

4.3.8. Prioserver

Components of type *prioserver* can be used to model servers with priority scheduling disciplines. The default schedule strategy is PRIONP. A *prioserver* component provides a service *request* with formal parameters *amount* denoting the service *amount* requested from the server and *prio* denoting the priority of service calls in progress. Zero is the highest service call priority.

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

4.3.9. Observer

For the component type *observer*, which is useful to produce intermediate result outputs, see Appendix F.1.9.

4.4. Model Types

Model type declarations should textually precede an EXPERIMENT block. They may as well be given in the declaration part of an EXPERIMENT block.

modeltype_declaration ::=
TYPE modeltype-name **MODEL** [formal_parameters];
[collect_block]
[declaration [...]]
[refer_part]
[**BEGIN** sequence_of_statements]
END TYPE [modeltype-name] ;

The model type constitutes the highest layer in the description of a model which is to be analyzed. Model types are syntactically and semantically similar to component types, with the following exceptions:

- The COMPONENT key word is substituted by MODEL.

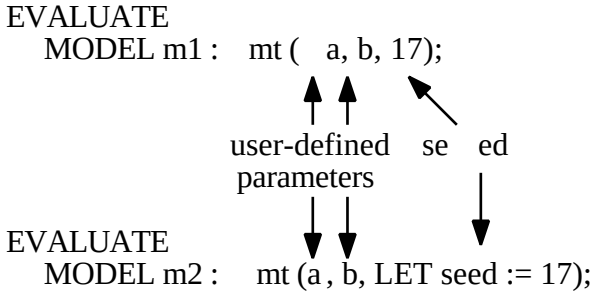
- Models are self-contained, i.e., they do not PROVIDE procedures or services. Thus, PROVIDE blocks are not valid in model type declarations.

- Models, as instances of model types, are analyzed in an experiment. Thus, models can only be declared in an EVALUATE statement.

- Every model type by standard has the parameter *seed*, the starting value of the generators of pseudo-random numbers. *Seed* is of type INTEGER and has a default value of 13. If another starting value is desired, the parameter has to be specified when creating a model within the EVALUATE statements (the theory recommends always to use odd positive integers as starting values for random-number generators used here).

  In principle, *seed* is the last formal parameter of a model type.

Examples:



- In contrast to a component type the default values of the component control procedures cannot be changed, neither by giving actual model parameters nor by a CONTROL procedure block.

- For *arrival* procedures within services of a model (see Section 4.2.2.4.) the following is valid: *arrival_announce* = *arrival_entry* and *arrival_exit* = -1.

Example:

```
TYPE computing_center MODEL (number_of_terminals      :     INTEGER;
                             mean_think, mean_io, mean_compute  :     REAL);

STREAM     cycletime      :      EVENT;


TYPE dialog SERVICE (thinktime, io_time, computetime :     REAL);
       USE
              SERVICE     compute      (t      :      REAL);
                          think        (t      :      REAL);
                          in_out       (t      :      REAL);
       END USE;

       VARIABLE   starttime      :      REAL;

BEGIN
       LOOP
              think          (negexp (1.0/thinktime));
              starttime      := time;
              in_out         (negexp (1.0/io_time));
              compute        (negexp (1.0/computetime));
              in_out         (negexp (1.0/io_time));
              UPDATE         cycletime BY time - starttime;
       END LOOP;
END TYPE dialog;


COMPONENT      terminals      :      server;
               channel        :      server (LET schedule  := fcfs);
               cpu            :      server (LET schedule  := fcfs);


REFER dialog                  TO          terminals, channel, cpu      EQUATING
       dialog.compute         WITH        cpu.request
       dialog.think           WITH        terminals.request
       dialog.in_out WITH channel.request
END REFER;

BEGIN
CREATE number_of_terminals PROCESS dialog (mean_think, mean_io, mean_compute);
END TYPE computing_center;
```

4.5. Model Structure and Virtual Declarations

A model consists of several hierarchically arranged layers, each layer referring a load (a set of processes, services) to a machine (a set of components). At each layer, the services and procedures used by the load are equated to services and procedures provided by the machine in an explicit binding operation (the REFER part of a component or model type declaration). Flat or single layer models are obtained, if the machine of a model type consists of components of standard types only. If *spend* is used within the model, there might even be no explicit component declaration at all.

Model types can be converted to component types by adding a PROVIDE declaration to the model type declaration, i.e., providing some local services or procedures for external usage. Components of this type can then be used as subcomponents in a higher layer model or component type (bottom up design). On the other hand, a component can be replaced by another possibly highly structured component providing the same services and procedures (top-down design).

In this way multi-layered models can be constructed, the topmost layer being a load/machine pair without provided services or procedures, the bottom layer made up of standard components. Every machine in such a load/machine pair arrangement consists of a set of components, thus we have a tree structure: A component containing a machine is the root of a tree of lower layer components.

We do not have a pure tree structure if services (procedures) provided by a component are used by services (procedures) of more than just one higher layer component. In this case, there has to be a single "regular" declaration of the component and a virtual ENCLOSE declaration of the component in every component using services (procedures) provided by the component.

enclose_declaration ::=
ENCLOSE
{component-name [, ...] [: [**ARRAY OF**] componenttype-name] ; } [...]

The main point of virtual ENCLOSE declaration is that no component object is generated. Rather, a reference is made to the (global) declaration of the component to be enclosed.

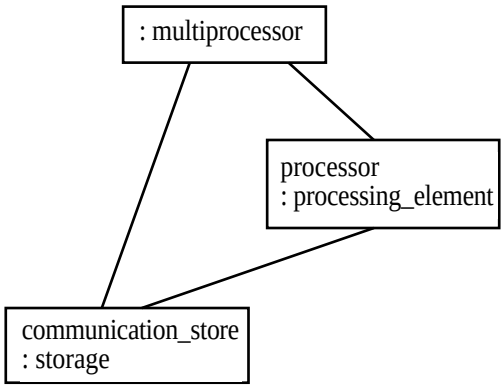
Therefore, array bounds or actual parameters must not be specified in a virtual declaration. There has to be exactly one "regular" component declaration for an enclosed component. The component declaration must be given in a declaration block which encloses every block with a virtual declaration of that component. Any number of virtual declarations can be given. The *component-name* and, if used, the *componenttype-name* have to be identical to that of the "regular" component declaration. Please note that a usage of *componenttype-name* is recommended if *component-name* conflicts could exist.

Thus, a hierarchical model containing virtual component declarations normally has the structure of a cycle free graph with a common root, the root being a model type, the leaves being standard component objects. Arbitrary graph structures can be specified, but only cycle-free graphs can be analyzed with HIT.

Example:

```
TYPE multiprocessor MODEL;

    TYPE storage COMPONENT;
        …
    END TYPE storage;

    TYPE processing_element COMPONENT;
        …
        ENCLOSE communication_store   :     storage;
    END TYPE processing_element;

    COMPONENT     communication_store     :     storage;
                  processor               :     processing_element:

END TYPE multiprocessor;
```

This model consists of two components, *communication_store* and *processor*, the latter using the same *communication_store* component as the *multiprocessor* model. This means that all services declared in *processing_element* as well as all services in *multiprocessor* may use services of the same *communication_store*.



5. Model Analysis

This section deals with the elements of HI-SLANG needed to analyze complete models or to pre-analyze submodels. For evaluating a (sub)model one has to:

- describe the model as a model type (Chapter 4.)

- declare and update streams, if user-defined streams are needed within a simulation (Section 5.1.)

- specify the preparation and representation of results (Section 5.2.)

- choose a solver (simulative or a special analytical solver) for the EXPERIMENT block. This decision is valid for all the evaluations in the EXPERIMENT block (Section 5.10.).

A very important part of the EXPERIMENT block is the evaluation statement (Section 5.9.). There may be multiple EVALUATE and/or AGGREGATE statements in one EXPERIMENT block. The AGGREGATE statement causes a component type to be pre-analyzed rendering a component type comparable to a state-dependent *server* as its result (Section 5.9.2.). The EVALUATE statement generates a model (-object) of the given model type and causes its analysis.

In the simulative case the analysis is based on the (steady state) estimation of the parameters of autoregressive models (see /LiSS89/ or /Litz85/). This is done with online update technique. So it is not necessary to save the "history" of a simulation. In the analytical case several exact or approximate algorithms are used. In both cases the results can be shown in tabular or simple graphical form.

The EVALUATE statement consists of a declaration and a body part. So-called evaluation objects (Section 5.5.), which are defined in the declaration part according to the hierarchical structure of the model, are attached to exactly one component of the model (or the model itself) or to just one area of a component. Results are computed only for the evaluation objects, so the analysis can be restricted to the relevant parts of the model. Here one can specify which estimators for which time interval (start and end of the measurement) are to be computed and how the results are to be presented.

Whereas the declaration of evaluation objects reflects the static structure of a model, the declaration of load filtering hierarchies (Section 5.6.) reflects the dynamic structure of the service calls. Load filtering hierarchies describe the sequence of service calls down to the relevant model part. Thus it is possible to evaluate a component with respect to different parts of the load.

The MEASURE statement (Section 5.7.) specifies

- which streams of an evaluation object should be analyzed,
- which part of the load should be considered,
- which estimators should be computed, and
- which (time) interval should be evaluated.

In a simulative evaluation there is a CONTROL statement to specify the end of the simulation and a possible trace of the service calls (Section 5.8.). The lexical elements for defining this stop condition are the same as those for defining evaluation attributes (for the whole evaluation object or for a single MEASURE statement - Section 5.4.). This is especially true for the start and stop conditions of the interval to be evaluated (Section 5.3.). For an analytic-numerical evaluation there is a CONTROL statement, too, for controlling the accuracy and switching on a trace of the evaluation.

5.1.Streams

In HIT, so-called streams form the most important interface between a model and its evaluation.

From the analytical point of view a stream represents a kind of stochastic process described by the model. From the statistical point of view a stream is a multivariable time series, from the simulative point of view it is a list of pairs, generated during the simulation. Each pair (t_i, x_i) consists of a time stamp (or a counter) t_i and a numerical value x_i. The interpretation of x_i (and therefore its evaluation) depends on the type of the stream. There are three stream types, COUNT, EVENT and STATE, which are described below.

For any solver a standard set of streams for analyzing the model is available (see Section 5.1.2.). By the simulative solver additionally user-defined streams can be measured. In contrast to standard streams, which can simply be evaluated by MEASURE statements, for user-defined streams the user additionally has to

- declare a stream of the appropriate type (Section 5.1.1.), and to
- update the stream, i.e., generate a new pair (UPDATE statement; Section 5.1.3.)

in his model description.

5.1.1. Declaration of Streams

Streams may be declared in the declaration part of component and model types. Such streams are called user-defined streams in contrast to predefined standard streams, which may not be declared. The declaration of user-defined streams is only allowed when using simulation.

```
modelling_declaration ::= …
|   stream_declaration

stream_declaration ::=
    STREAM
    {stream-name [, …] : stream_type ; } [ …]

stream_type ::=
    COUNT
|   EVENT
|   STATE
```

The *stream-name* is the name of the stream (any identifier allowed by the rules for constructing names and according to the scope of validity for identifiers), by which this stream can be accessed in the body of all services.

Examples:

```
STREAMcreek      :  STATE;
          s1, s2 :  COUNT;

STREAM cycletime :  EVENT;
```

Note:

Due to historical reasons streams may as well be declared in services, but for such streams a COLLECT block has additionally to be provided in the surrounding component type. Declaring a stream in a service will produce a warning.

5.1.1.1. Type EVENT

The type EVENT covers event streams for serially collecting values. The generated pairs (either explicitly by an UPDATE statement or implicitly for standard streams) consist of a number, which is interpreted as an observed value and a consecutive counter starting at 1.

The numerical value must be a REAL. The counter value is generated by the system. The estimator MEAN for streams of type EVENT is defined by

$$x = \frac{1}{N} \sum_{i=1}^{N} x_i$$

where N is the number of pairs and x_i (i = 1, …, N) the numerical value. Thus the estimator MEAN is the mean of the observed values, if no start condition is specified. An estimator for the standard deviation is computed with respect to this mean value, an estimator for the confidence interval is computed with respect to this mean estimator.

If %parm=minmax is used within the control file (see Chapter 8), the minimum and maximum values shown in the table file are the smallest and largest ones of the observed values.

Example:

```
STREAM  MY_TURNAROUNDTIME : EVENT;
```

5.1.1.2. Type STATE

The type STATE covers streams of states which may take different values but are constant while no update occurs. Continuously changing states cannot be described. The pairs generated (either explicitly by UPDATE statements or implicitly for standard streams) consist of a numerical value and a time stamp. For user-defined streams the numerical value is computed by evaluating the expression given in the UPDATE statement. The time stamp is generated by the system.

The numerical value is interpreted as the difference to the previously observed state value, i.e., the new state is computed by adding the value just calculated to the previous

state. The range of values of the numerical value is REAL. Streams of type STATE are initialized by 0.0 at model time 0.0.

The estimator MEAN for streams of type STATE is defined by

$$x = \frac{1}{T}\left[\left(\sum_{i=1}^{n}(t_i - t_{i-1})\ \ x_{i-1}\right) + (T - t_n)\ \ x_n\right]$$

where T is the observation time starting at t_0=0 with x_0=0, n is the number of state changes until T, t_i the time of a state change and x_i the new state value (not the difference between new and old value, which is given in the UPDATE statement). Thus the estimator for the mean is the mean of the observed values, weighted by the length of the intervals the value has been observed, if no special start condition is specified. An estimator for the standard deviation is computed with respect to this mean value, an estimator for the confidence interval is computed with respect to this mean estimator.

If %parm=minmax is used within the control file (see Chapter 8), the minimum and maximum values shown in the table file are the lowest and highest values of the state trajectory.

Example:

```
STREAMMY_POPULATION,
        MY_OCCUPATION,
        MY_UTILIZATION : STATE;

{
```

5.1.1.3. Type COUNT

Streams of type COUNT count events for estimating rates. The pairs generated (either explicitly by UPDATE statements or implicitly for standard streams) always consist of the value 1 and a time stamp. If the expression given in the UPDATE statement does not yield 1, it is set to 1 by the system. If the counter value shall be increased by more than 1, the appropriate number of UPDATE statements have to be executed. The time stamp is generated by the system. Streams of type COUNT are initialized by 0 at model time 0.0.

The estimator MEAN for streams of type COUNT is defined by

$$x = \frac{N}{T}$$

where T is the observation time (start t = 0) and N the number of the pairs generated. Thus, the estimator for the mean is the number of events observed, divided by the length of the observation interval, if no special start condition is specified. An estimator for the standard deviation is computed with respect to the inter-event time. Confidence intervals are given for mean rates.

If %parm=minmax is used within the control file (see Chapter 8), the minimum and maximum values shown in the table file are the smallest and largest interevent time observed.

Example:

```
STREAM MY_THROUGHPUT : COUNT;
```

5.1.2. Standard Streams

The standard streams THROUGHPUT, TURNAROUNDTIME, POPULATION, UTILIZATION, OCCUPATION, SCHEDULE_RATE and PREEMPT_RATE are predefined for every component (area) and the model itself. They must neither be declared nor updated.

THROUGHPUT, TURNAROUNDTIME, POPULATION and UTILIZATION may be used in analytical and simulative evaluations, whereas OCCUPATION, SCHEDULE_RATE and PREEMPT_RATE are allowed for simulation only. UTILIZATION is permissible for components of type *server* only. As models have no control procedures, it does not make sense to measure SCHEDULE_RATE or PREEMPT_RATE for them. Furthermore both streams are not allowed for component areas.

These streams can be obtained for every model and component as well as for special component areas. The latter is only possible for simulations, and the word component has to be replaced by the special component area (and the control procedure names have to be omitted) in the following explanations of these standard streams:

- **THROUGHPUT** (type COUNT)

  The stream is updated whenever a process is leaving the component

  The MEAN estimator yields the number of processes leaving the component in one time unit. The STANDARDDEVIATION estimator renders the standard deviation of the interdeparture times, not of the throughput.

- **TURNAROUNDTIME** (type EVENT)

  The stream is updated whenever a process is leaving the component. TURNAROUNDTIME measures the time this process has spent in the component, i.e., the difference between departure time and arrival time.

  The MEAN estimator yields the mean time a process is spending in the component.

- **POPULATION** (type STATE)

  The stream is updated whenever a process is entering the component or leaving it by +1 or -1 resp.

  The MEAN estimator yields the mean number of processes resident in the component.

- **UTILIZATION** (type STATE)

  This stream is allowed (and does only make sense) for components of type *server* and *prioserver* only, component areas are not permitted. UTILIZATION measures the service speed given to the processes. It is updated whenever the speed of a process changes.

  The parameter *speed* of the standard *dispatch* control procedures *shared*, *sdshared equal* and *sdequal* (see also Appendix F.3.4.) defines the standard speed of a component. For this component UTILIZATION is normalized to this standard speed by dividing the actual speeds by the standard speed.

  For *shared* and *equal*, the standard speed corresponds to the actual speed of the component. For *sdshared* and *sdequal*, the standard speed is multiplied by the appropriate entry of the *speeds*-array, yielding the actual speed.

  The MEAN estimator gives the utilization of the component (type *server* or *prioserver*). Values greater than 1 are posssible, due to actual speeds being greater than standard speed and/or multiple processes in service.

  In the case of simulation the evaluation of UTILIZATION is more CPU-time consuming compared to the other standard measures. In some cases, there are alternative streams resulting the same measures. For *dispatch* control procedure *shared* UTILIZATION is identical to OCCUPATION of the service area. For the *dispatch* procedure *equal* the alternative measure depends on the *schedule* procedure. POPULATION of the server can be used with *immediate*, POPULATION of SERVICE_AREA with *fcfs* and *lcfs* and OCCUPATION with *lcfspr*, *fcfs(1)* and *lcfs(1).*

- **OCCUPATION** (type STATE)

  This stream is allowed in case of simulation only. It is updated, if a component, which was previously empty, takes a process into itself or if the last process in the component is leaving. Arrivals (and departures) that do not find the component empty (or do not leave it empty, resp.) are ignored. The stream only knows about the two states "component is occupied" and "component is empty".

  The MEAN estimator yields the probability of finding the component occupied.

- **SCHEDULE_RATE** (type COUNT)

  This stream is only allowed when using simulation. It is updated whenever the control procedure *schedule* puts a process from the *entry area* to the *service area.*

  The MEAN estimator yields the transition rate from the *entry area* to the *service area*, thus this stream must not be applied for evaluation objects referring to an area.

- **PREEMPT_RATE** (type COUNT)

  This stream is only allowed when using simulation. It is updated whenever the control procedure *schedule* preempts a process (from the *service area* to the *entry area*).

The MEAN estimator yields the transition rate from the *service area* to the *entry area*, thus this stream must not be applied for evaluation objects referring to an area.

5.1.3. Update Statement

The UPDATE statement is applied to generate a new value for user-defined streams. It is allowed only in the body of the service which declares the stream (old concept) or which is contained in the component type declaring the stream.

simple_statement ::= …
| update_statement

update_statement ::=
UPDATE stream-name **BY** simple_real_expression ;

The *stream-name* is the name of the stream updated. The expression must be of type INTEGER or REAL. The interpretation of this value (and as a consequence the evaluation of the stream) depends on the type of the stream. For streams of type STATE, the value is the difference between new and previous state. For streams of type EVENT, it takes the value which was actually observed. For streams of type COUNT the value is ignored and the internal counter is incremented (by 1). Nevertheless the expression may not be omitted.

The corresponding time stamps or counters are supplied by the system automatically. User-defined streams are updated at the time the UPDATE statement is executed, standard streams are updated implicitly. The number of updates which have occured for a stream can be displayed (in a table) for all kinds of streams by using *%parm=updates* in the control file.

Examples:

```
UPDATE my_stream    BY time - started_at;
UPDATE s1           BY 5;
```

5.1.4. Undefined Results of Streams

The evaluation of streams may lead to undefined results. They are marked by the keyword "Undefined" in the corresponding fields of the result table.

Normally not enough simulation time has been spent in such cases: The most common reasons for undefined MEAN values are, in case of EVENT streams, that the evaluated stream is not updated during simulation. In case of COUNT or STATE streams, an undefined result value is produced if the evaluation time and start time of the measurement are identical. For estimator STANDARDDEVIATION additionally a very small (COUNT) or large (EVENT, STATE) mean value may be the reason. For estimator CONFIDENCE several reasons are possible and will therefore be given on the analyzer listing.

Undefined results for self-defined streams can as well have the following reasons:

- The declared user-defined stream is not updated within the evaluated service. The UPDATE statement is missing.

- The service does contain an UPDATE statement, but this is not executed during the simulation. It may be nested, e.g., in IF clauses or in alternatives of a branch statement being executed with a low probability.

- User-defined streams are often evaluated restricted to load filtering hierarchies. Another possible reason for undefined result values is, that the service containing the UPDATE statement is not catched by the load filtering hierarchy given in the corresponding MEASURE statement, i.e., that a service other than that or those which do update the stream has been specified by that hierarchy.

Missing or unexecuted UPDATE statements can be identified by counting the corresponding events which occur during simulation (using the compiler directive %PARM=UPDATES in the control file).

5.2.Representation of Results

HIT can present the results either as a table, as a machine-readable dump file, or as a simple graph or histogram. The presentation of results can be specified for evaluation objects (i.e., for all measurements at it) or for single measure statements. The default output format is one table for all performance values.

Tables and dump files can be obtained quite simply. For syntax and semantics of the table selection, see Section 5.4., for the format of the tables Appendix G.3.1. Further information can be found in Section 8.

If the results shall be presented as graphs or histograms, they have to be saved in so-called dump files (for syntax and semantics see sections 5.4. and 8., for the format of a dump file see Appendix G.3.2.).

By a GRAPH or HISTOGRAM statement the saved results can be transformed to a (semi-) graphical representation. The generated graph is a text file containing "ASCII-graphic". These statements (*plot_statements*) may take their input (i.e., results of experiments) from multiple dump files. The results may be from the current or from a previous experiment, but of course the files must exist at the time the plot statement is executed. For every dump file used in the plot statements there must exist a binding in the control part, if the standard link name "DUMP" is not used.

plot_statement ::=
    graph_statement
| histogram_statement

PLOT statements must be the last statements of the body of an experiment. Furthermore, they must not be used within loops or blocks.

5.2.1. GRAPH Statement

A GRAPH sta tement specifies to plot the results as a curve in a cartesian coordinate system in a format described in Appendix G.3.4.

graph_statement ::=
    GRAPH [inscription]
    {**PLOT** simple_text_expression plot_specification_graph
        INPUT simple_text_expression [, …] } […] ;

plot_specification_graph ::=
    MEASURE simple_text_expression
    ESTIMATOR simple_text_expression
    EVALUATIONOBJECT simple_text_expression
    HIERARCHY simple_text_expression

inscription ::=
[simple_text_expression]
ABSCISSA simple_text_expression
ORDINATE simple_text_expression
[**OUTPUT** simple_text_expression]

A title for the diagram may be given (*simple_text_expression* before ABSCISSA in *inscription*), as well as names for both axes (after ABSCISSA and ORDINATE). All expressions must be of type TEXT. The *inscription* part may be omitted. In this case defaults are used. By OUTPUT "link name" a link name of a file to write the resulting graph into may be specified. If multiple GRAPH or HISTOGRAM statements use the same OUTPUT file, this file should be used in EXTEND mode to avoid overwriting (see Section 8.2.2.). There is a default in case OUTPUT *simple_text_expression* is omitted: The graph is then written via standard link name "GRAPH" having a default file binding.

For each curve to be drawn in the coordinate system there must be one PLOT part in the GRAPH statement. The expression (of type TEXT) after PLOT specifies the name of the curve. Each curve is drawn with a unique symbol (character) as follows:

A curve is a number of points, each point representing a measured value (i.e., the result of one measurement in one evaluation). The independent value (x-, abscissa-value) was given in the MEASURE statement as the expression after ABSCISSA. This expression must be of type REAL or INTEGER. The dependent value (y-, ordinate-value) is the observed value. In the example below the ABSCISSA value represents the sequence number of an evaluation in a series of evaluations.

After INPUT the link name (type TEXT) of the dump file that holds the data is specified. Within this file, the *plot_specification_graph* (all expressions must be of type TEXT and must be identical to some MEASURE statements of the experiment) identifies the measurements to be drawn.

Note:

After ESTIMATOR the text "CONFIDENCE" has to be given for the confidence interval, and "UPPERBOUND" and "LOWERBOUND" (separately) for ESTIMATOR BOUNDS. Lower-case letters can be used.

For every link name used in the GRAPH statement, there must exist a binding in the control part, except for the standard link name "DUMP".

For estimator FREQUENCY, the GRAPH statement is not allowed, a HISTOGRAM statement is more appropriate.

Example:

```
EXPERIMENT testseries METHOD SIMULATIVE;

   VARIABLE  no_of_evaluation:      INTEGER;
             processor_speed    :   REAL DEFAULT 1.0;

BEGIN

   FOR no_of_evaluation  := 1 STEP 1 UNTIL 5
   LOOP

      EVALUATE MODEL computer_system : computer_type (processor_speed);

         EVALUATIONOBJECT processor VIA … ;
      BEGIN
      …
         MEASURE THROUGHPUT
         AT processor  ABSCISSA no_of_evaluation ESTIMATOR MEAN
                              OUTPUT DUMPFILE "DUMP";
         …
      END EVALUATE;

      processor_speed := processor_speed + 1.0;

   END LOOP;

   GRAPH   "CPU-throughput"        ABSCISSA          "Number of Evaluation"
                                   ORDINATE          "Throughput of CPU"

   PLOT    "overall throughput"    MEASURE           "THROUGHPUT"
                                   ESTIMATOR         "MEAN"
                                   EVALUATIONOBJECT  "PROCESSOR"
                                   HIERARCHY         "ALL"
                                   INPUT             "DUMP";

END EXPERIMENT    testseries;
```

Please note the double quotes enclosing THROUGHPUT in the *plot_specification_graph*: THROUGHPUT is not the HI-SLANG keyword, but the text to look for in the dump file. Since the standard link name "DUMP" is used for the intermediate dump file output there need not be any binding in the control file. The dump file name is determined by the HIT file name generator in this case (see Section 8.2.6.).

5.2.2. HISTOGRAM Statement

The results of estimator FREQUENCY can be presented as histograms in a format described in Appendix G.3.3. The HISTOGRAM statement takes its input from dump files, so the dump files must exist at the time the HISTOGRAM statement is executed and they must contain all the information necessary.

histogram_statement ::=
HISTOGRAM [inscription]
PLOT plot_specification_histo
INPUT simple_text_expression ;

plot_specification_histo ::=
MEASURE simple_text_expression
EVALUATIONOBJECT simple_text_expression
HIERARCHY simple_text_expression

Syntax and semantics of the HISTOGRAM statement are similar to those of the GRAPH statement (Section 5.2.1.), but:

- The ESTIMATOR cannot be specified, since estimator FREQUENCY is always concerned.
- Please note that there is no *simple_text_expression* after PLOT (name of one curve).
- After INPUT exactly one link name must be supplied. There must exist a binding in the control part for this link name, except for the standard link name "DUMP".

The scaling of the abscissa (x-axis) is derived from the intervals given in the MEASURE statement, the scaling of the ordinate (y-axis) is generated by the system (dependent on the observed frequencies).

Example:

```
HISTOGRAM   "Processor Times"
            ABSCISSA          "Intervals of Turnaroundtimes"
            ORDINATE          "Frequencies"

PLOT        MEASURE           "TURNAROUNDTIME"
            EVALUATIONOBJECT  PROCESSOR"
            HIERARCHY         "HIER1"
            INPUT             "DUMP";
```

5.3. Specification of Start and Stop Conditions

Start and stop conditions are used to control the execution of evaluations and measurements. They may be used in simulative experiments and (with restrictions) in analytical ones. The conditions are specified by logical expressions, connected by the operators AND and/or OR.

start_or_stop_condition ::=
    basic_condition [and_or …]

and_or ::=
    AND
| **OR**

The conditions are evaluated according to the usual rules. AND has higher precedence than OR and parentheses are not allowed. The time of evaluation depends on the context of the condition. Start and stop conditions are not evaluated each time their (potential) value changes.

The basic conditions yield TRUE or FALSE, dependent on the actual state. This state always refers to the actual evaluation and includes:

- the CPU time used,
- the model time reached,
- the number of events at one evaluation object,
- the length of confidence intervals,
- the number of updates on a stream (GLOBALSTOP), and
- the accuracy of the solution (for MARKOV solver and LIN2 solver (performance bounds)).

The formal description and the semantics of the corresponding basic conditions are the topics of the following subsections.

For simulative evaluations start conditions can be specified, so that collecting data starts after the transient phase of the system under investigation. Here some caution is necessary, for the end of the transient phase may be difficult to determine (if it is reached at all!) and the loss of information may be significant, especially for rare events.

Start conditions may consist of basic conditions about the reached model time and the number of events. They may be given for an evaluation object (i.e., for all measurements at it) or for one single MEASURE statement.

In simulative experiments all kinds of basic conditions for stop conditions except accuracy of the solution may be used. Stop conditions can set the end of the evaluation of one evaluation object or a single measurement as well as the end of the complete evaluation (CONTROL statement (Section 5.8.), GLOBALSTOP statement (Section 5.4.)).

When using the MARKOV solver or else the solver LIN2 together with estimator BOUNDS, a stop condition in the CONTROL statement determines the end of the

evaluation. Here only basic conditions about the accuracy of the solution are allowed, in MARKOVian experiments the used CPU time may be limited as well.

Examples:

```
START EVENTS 1000
STOP    CPUTIME 5000 OR MODELTIME 50000 AND EVENTS 100000 OR
        CONFIDENCE LEVEL 95 WIDTH 10 MEASURE THROUGHPUT
```

For user convenience a message is given during simulation on the analyzer listing whenever any basic start or stop condition is reached, which is either specified within a CONTROL statement, a MEASURE statement or an EVALUATIONOBJECT declaration. This special kind of event trace helps the user in understanding unexpected system behavior.

5.3.1. CPU Time

The specification of the used CPU time consists of the keyword CPUTIME and an expression of type REAL or INTEGER, whose evaluation yields the limit of the CPU time.

basic_condition ::= …
| **CPUTIME** simple_real_expression

This basic condition may only be used within CONTROL statements and results TRUE, if the used CPU time for the current evaluation is greater than or equal to the value of *simple_real_expression*, FALSE otherwise.

Please note that CPUTIME condition is only checked if model time is also consumed, i.e., endless loops without modeltime consumption cannot reach the CPUTIME limit.

Examples:

```
CPUTIME 500
CPUTIME gettime (a)
CPUTIME a + b
```

5.3.2. Model Time

The specification of the reached model time consists of the keyword MODELTIME and an expression of type REAL or INTEGER, whose evaluation yields the limit of the model time.

basic_condition ::= …
| **MODELTIME** simple_real_expression

This basic condition results in TRUE, if the reached model time is greater than or equal to the (INTEGER) value of *simple_real_expression* and at least one event has occurred, FALSE otherwise. The model time is managed by the system. At the start of each simulation its value is 0.0, and it is incremented by time consumption, i.e., calls of

service *request* of component type *server* or the statements *spend* and *hold*. The definition of future events (process generations by CREATE or SUBMIT with AFTER/EVERY/AT, TIMESLICE in control procedures) may also result in a progress of time. All other statements do not affect the model time directly.

Examples:

```
MODELTIME a + b/c
MODELTIME mtime (5.0, b)
MODELTIME 100000
```

5.3.3. Number of Events

The specification of the number of observed events consists of the keyword EVENTS and an expression of type REAL or INTEGER, whose evaluation (REAL converted to INTEGER) yields the limit of the number of events. A load filtering hierarchy may be given to specify the origin of the events to be counted.

basic_condition ::= …
| **EVENTS** simple_real_expression [**DUE TO** hierarchy-name]

An event occurs, when a process is leaving the component or a component area. Which component (or component area) is to be observed is specified in the context of the condition (*estimator_part, control_statement*). If a load filtering hierarchy is supplied (after DUE TO), only the events of processes selected by that load filtering hierarchy are counted. The load filtering hierarchy must be declared in the declaration part of the EVALUATE statement and must end at the given component. It is not possible to give multiple hierarchies; the default is *all*.

This basic condition results in TRUE, if the number of relevant events is greater than or equal to the (INTEGER) value of *simple_real_expression*, FALSE otherwise. The events are counted from the start of the simulation, so START EVENTS 100 STOP EVENTS 200 stops after 200 events (not after 300, as might be supposed).

Examples:

```
EVENTS  5000
EVENTS  10000    DUE TO hier1   AND   EVENTS  7000   DUE TO hier2
EVENTS  a + b    DUE TO hier1   OR    EVENTS  no_of_ev (c)
```

5.3.4. Confidence Interval

The specification of the length of confidence intervals consists of the probability of confidence, the desired length of the interval (in percent) and the name of the stream. Optionally a load filtering hierarchy and the maximum degree for the autoregressive model may be specified.

basic_condition ::= …
| **CONFIDENCE LEVEL** simple_real_expression
WIDTH simple_real_expression
MEASURE stream
[**DEGREE** simple_real_expression]
[**DUE TO** hierarchy-name]

The expression after CONFIDENCE LEVEL must be of type REAL or INTEGER and render a value between 90 and 99. Otherwise it is set to 95 and a warning is given. A value of type REAL is converted to INTEGER. This value defines the probability that the real mean lies within the given interval around the estimated mean (90 indicating probability 0.9, 99 indicating 0.99).

The length of the interval is specified by the expression after WIDTH, which must be of type REAL or INTEGER. It is interpreted as one half of the interval, given in percent of the estimated mean. E.g., the value 10 for an estimated mean of 200 specifies the interval (180, 220).

After MEASURE the stream the confidence interval is to be computed for must be specified. This is either one of the standard streams THROUGHPUT, TURNAROUNDTIME, POPULATION, OCCUPATION, UTILIZATION, SCHEDULE_RATE and PREEMPT_RATE or a user-defined stream (*stream-identifier*).

stream ::=
THROUGHPUT
| **TURNAROUNDTIME**
| **POPULATION**
| **OCCUPATION**
| **UTILIZATION**
| **SCHEDULE_RATE**
| **PREEMPT_RATE**
| stream-identifier

There are the following restrictions:

- The standard stream UTILIZATION is allowed for components of type *server* only.
- SCHEDULE_RATE and PREEMPT_RATE are only allowed for components, not for component areas.
- *stream_identifier* must be a valid stream for a service of this component.

After DEGREE the maximum degree of the autoregressive model may be given. The expression must render a value between 1 and 20. Otherwise a warning is given and it is set to 10 (the default value). If the DEGREE part is omitted also the default value 10 is used. See the analyzer listing for more information.

After DUE TO a load filtering hierarchy (default *all*) may be given. Only updates passing this filter are considered. The load filtering hierarchy must be declared in the enclosing EVALUATE statement and end in the observed component.

This basic condition yields TRUE, if the length of the computed confidence interval is equal to or smaller than the specified interval length.

Examples:

```
CONFIDENCE LEVEL  95 WIDTH 10.0 MEASURE THROUGHPUT OR
CONFIDENCE LEVEL  95 WIDTH 10.0 MEASURE TURNAROUNDTIME

CONFIDENCE LEVEL  92 WIDTH   5.0 MEASURE UTILIZATION

CONFIDENCE LEVEL  a+b WIDTH getwidth (5.0) MEASURE cs DEGREE 15 DUE TO hier1
```

5.3.5. Accuracy

The specification of an accuracy stop is allowed for MARKOV and LIN2, but with different interpretations. The accuracy specification is only useful within CONTROL statements.

The specification of the accuracy of an analytical solution consists of the keyword ACCURACY and an expression of type REAL or INTEGER.

```
basic_condition  ::= …
|   ACCURACY simple_real_expression
```

The meaning of ACCURACY depends on the solver used.

5.3.5.1. Accuracy Stop for the MARKOV Solver

Here the *simple_real_expression* is interpreted as the relative error (in percent) of the required solution. Please note that the accuracy of the solution is only an estimation and that it is recommended to use accuracies (or better: relative errors) less than 1 %, although greater values may be supplied. The example below specifies a relative error of 0.1 %, that is 0.001.

This basic condition yields TRUE, if the estimated relative error of the solution is equal to or smaller than the value of *simple_real_expression.* Besides as well as in combination with ACCURACY the basic condition CPUTIME is allowed for MARKOV.

Example:

```
ACCURACY 0.1
```

5.3.5.2. Accuracy Stop for the LIN2 Solver

If the LIN2 solver together with estimator BOUNDS is used, the quality of the bounds for the desired mean values may be influenced by the ACCURACY condition. The value of *simple_real_expression* is rounded to give an INTEGER value. The result must be less than 5. With increasing values, more accurate results are computed (for 0, i.e., accuracy < 0.5, no bounds are computed at all ("Undefined")), but more computation is necessary, too. For very large models LIN2 reduces this value (to keep the computation in limits) and gives a warning.

Example:

```
ACCURACY 2
```

5.3.6. Specification of GLOBALSTOP conditions

The facilities for the specification of GLOBALSTOP conditions differ from the other kind of stop condition specification. Please see Section 5.4.4. for a detailed description.

This section presents the basic conditions which can be used especially for GLOBALSTOP specification.

5.3.6.1 Width of Confidence Interval

The specification of the desired width of a confidence interval only consists of the keyword WIDTH and an expression which must be of type REAL or INTEGER. It is interpreted in the same way as the WIDTH specification for CONFIDENCE LEVEL condition (see Section 5.3.4.) namely as one half of the interval, given in percent of the estimated mean.

stop_expression ::= …
| **WIDTH** simple_real_expression

The specification of WIDTH is only allowed in combination with estimator CONFIDENCE LEVEL.

5.3.6.2 Number of Updates

The specification of the desired updates only consists of the keyword UPDATES and an expression which must be of type REAL or INTEGER. It is interpreted as the number of updates of the corresponding stream (also taking into account load filtering and start conditions). The condition becomes true as soon as at least one update has occured and the number of updates on the stream is equal or greater than the specified number.

stop_expression ::= …
| **UPDATES** simple_real_expression

Please note that the number of updates is not the same as the number of events as described in Section 5.3.3. .

5.4.Specification of Evaluation Attributes

The required *estimator* as well as the form of result output (*output_link*) may be given for the specification of evaluation attributes. These specifications can occur in MEASURE statements and/or EVALUATIONOBJECT declarations. Moreover start and stop conditions (*start_or_stop_condition*) valid for all measurements at the evaluation object or only for a single measurement (depending on the context) can be stated for the simulative case.

```
estimator_part ::=
   [ESTIMATOR estimator [, …]]
   [OUTPUT   output_link [, …]]
   [START    start_or_stop_condition]
   [STOP     start_or_stop_condition
     | GLOBALSTOPstop_expression ]

estimator ::=
   MEAN
|  BOUNDS
|  STANDARDDEVIATION
|  CONFIDENCE LEVEL        simple_real_expression
|  FREQUENCY INTERVAL ⟦ array_bounds [, …] ⟧

output_link ::=
   TABLE      simple_text_expression
|  DUMPFILE   simple_text_expression

stop_expression ::=
   WIDTH      simple_real_expression
|  UPDATES    simple_real_expression
|  WIDTH      simple_real_expression  and_or
   UPDATES    simple_real_expression
|  UPDATES    simple_real_expression  and_or
   WIDTH      simple_real_expression
```

5.4.1. Estimator Specification

Some notes about the estimators:

- **MEAN**
  MEAN (the mean value) is permitted in analytic and simulative evaluations. See Section 5.1.1. for how it is defined for different types of streams.

- **BOUNDS**
  BOUNDS is only eligible for the solver LIN2. Exact upper and lower bounds for the mean values will be computed (performance bounds). If the ACCURACY stop condition in the CONTROL statement is omitted or if an ACCURACY value smaller than 0.5 is given in the stop condition, an estimated mean but no bounds are computed and a warning is submitted.

- **STANDARDDEVIATION**
  If STANDARDDEVIATION is specified, an estimator for MEAN is also computed. The standard deviation of the stream THROUGHPUT is related to the mean of the interdeparture times and not to the mean of the throughput.

- **CONFIDENCE LEVEL**
  The *simple_real_expression* defines the confidence probability. It must be of type REAL or INTEGER (REAL is implicitly converted to INTEGER) and its evaluation must yield a value between 90 and 99, defining a probability of 0.90 and 0.99, respectively. If these limits are exceeded, the value is set to 95 and a warning is given. When the estimator CONFIDENCE LEVEL is specified, MEAN and STANDARDDEVIATION are also computed.

- **FREQUENCY INTERVAL**
  The observed stream must be of type EVENT or STATE. Intervals have to be specified for this estimator. For an EVENT stream, the number of occurrences of values in an interval is counted. For a STATE stream, the amount of time the state is within the interval is accumulated.The intervals need not to be disjoint, but may be. Intervals are specified by their lower and upper limits (similar to array bounds) and separated by commas. The lower limit is included in the interval, the upper limit is excluded. An interval is ignored if its upper limit is smaller or equal its lower limit. The sequence of the intervals is arbitrary. By default, tables and dumpfiles contain the absolute number of occurrences for an EVENT stream and the relative amount of time (time in interval divided by total observation time) for a STATE stream. For tables, this behaviour can be modified by the analyzer option FREQUENCYFORMAT (see 8.2.1.3). FREQUENCY INTERVAL is eligible for simulation only.

5.4.2. OUTPUT Specification

Every *output_link* within the OUTPUT specification denotes whether the results are to be presented as a table or to be saved in a dump file (see GRAPH and HISTOGRAM statement). The *simple_expression* (of type TEXT) after TABLE or DUMPFILE resembles the link name of the file into which the data are to be written. If OUTPUT is omitted, OUTPUT TABLE "TABLE" is used (the default file is written in the EXTEND mode, allowing the results of series of evaluations to be saved). A binding in the control part must exist for every user-defined link name.

Note:

If OUTPUT is given and a non-standard link name is specified, the given link name should be supplied with an EXTEND binding in the control file, or else only the results of the last evaluation (of a series) performed can be found in that file. The same holds when a standard link name is used and rebound.

5.4.3. START and STOP Specifications

In the case of simulation a time interval for the measurements for this evaluation object can be specified. If there is a start condition all updates occuring before the condition has become true for the first time are ignored for measurement; in case of state streams they are only used to actualize the state value. In case of a stop condition measurement is stopped as soon as the stop condition has become true for the first time, ignoring all updates after this point in time. If there are no start or stop conditions, all events occurring in the evaluation object are taken into account.

Evaluation attributes may be specified for an entire evaluation object (see *evaluate_declaration*) or only for a single MEASURE statement (see *measure_statement*). In a set of evaluation attributes, some attributes may be omitted. The value of an evaluation attribute for one measurement is determined as follows:

1. If the given attribute is specified in the MEASURE statement, the supplied value is used. Else:

2. If the given attribute is specified for the evaluation object, the given value is used (values for attributes already found by 1. are ignored). Else:

3. The following default is used:

```
ESTIMATOR    MEAN
OUTPUT       TABLE "TABLE"
```

Examples:

```
ESTIMATOR    CONFIDENCE LEVEL 95
OUTPUT       TABLE "out1", TABLE "out2"
START        MODELTIME early
STOP         MODELTIME late;

ESTIMATOR    STANDARDDEVIATION,
             FREQUENCY INTERVAL [0..2.5,2.5..5,5..15,15..30,30..50,50..1000]
OUTPUT       TABLE "noncumulative", DUMPFILE "dump1";

ESTIMATOR    MEAN,
             FREQUENCY INTERVAL [0..5, 0..15, 0..30, 0..50]
OUTPUT       TABLE "cumulative";
```

5.4.4. GLOBALSTOP Specification

The specification of GLOBALSTOP condition is a facility to define conditions for stopping an evaluation also in the ESTIMATOR part of a MEASURE statement - beside the specification within a CONTROL statement (see Section 5.8.).

The GLOBALSTOP conditions or normal STOP conditions (see Section 5.4.3.) can only be used alternatively. Normal STOP conditions control the end of a measurement, but GLOBALSTOP conditions influence the end of total evaluation. A measurement with a GLOBALSTOP condition can only stop at the end of the evaluation.

An evaluation terminates if

- either **all** GLOBALSTOP conditions in all MEASURE statements are fulfilled (implicit AND-operation)
- or one condition in the CONTROL statement becomes true (implicit OR-operation).

Thus an implicit OR-operation exists between both possibilities (Specification via GLOBALSTOP and CONTROL statement).

If a MEASURE statement contains a list of streams and a GLOBALSTOP condition it is internally expanded in a way that a separate GLOBALSTOP condition of the same kind is associated with each stream.

The evaluation of GLOBALSTOP conditions starts after fulfilling corresponding START condition resp. after the beginning of corresponding measurement.

Four possibilities exist to specify a GLOBALSTOP condition:

- one single WIDTH condition (see Section 5.3.6.1.),
- one single UPDATES condition (see Section 5.3.6.2.),
- one WIDTH condition and one UPDATES condition connected by an AND-operator, or
- one WIDTH condition and one UPDATES condition connected by an OR-operator.

Example:

```
ESTIMATOR    CONFIDENCE LEVEL 95
OUTPUT       TABLE "out1"
GLOBALSTOP   WIDTH 5 AND UPDATES 5000;
```

Because usually a user is interested in a certain accuracy of some results, the kind of specification by GLOBALSTOP conditions is recommended; the usage of a CONFIDENCE condition in the CONTROL part is only recommended if absolutely necessary.

5.5.Declaration of Evaluation Objects

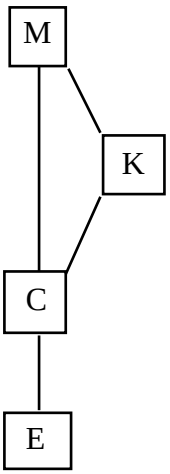
The relevant units in evaluating a model are components (including the model itself) and component areas. For a clearer separation of model and evaluation description, evaluation objects need to be declared and attached to a component (area). Results are computed for these evaluation objects only. For an evaluation object, evaluation attributes (*estimator_part*) may be specified as defaults for the corresponding MEASURE statements.

evaluationobject_declaration ::=
EVALUATIONOBJECT
{ {evaluationobject-name **VIA** [area **OF**] component-identifier} [, …]
[**DEFAULT** estimator_part] ; } […]

Declarations of evaluation objects are permissible in the declaration part of EVALUATE statements only. The *evaluationobject-name* is any identifier satisfying the rules concerning the construction of names and the scope rules for identifiers. This name will also be used in the MEASURE statement when referring to the component to be evaluated.

A complete path from the model (object) down to the component according to the component hierarchy (*component-identifier*) must be stated after VIA to specify the component to be evaluated. The elements are seperated by dots and each element of the sequence must be a subcomponent of its predecessor, beginning with the model and ending with the component considered. If component arrays arise in the path, the array index of the component must be stated. If enclosed components are part of the model, several paths might lead down to the component. In this case it does not matter which path is specified.

Example:




```
EVALUATIONOBJECT    e1 VIA M.C.E,
                    e2 VIA M.K.C.E;
                    e3 VIA SERVICE_AREA OF M.K.E;
```

There is no difference between the evaluation objects *e1* and *e2*, except that there will be seperated in result tables. All measurement of *e3* is restricted to the service area of *e*.

The hierarchical specification of the component is necessary since there may be several components of the same name in a model.

When evaluating the model object itself, only the name of the model occurs for *component-identifier*. Evaluation of model areas is not possible.

The names of the evaluation objects must be distinctive within one EVALUATE statement (and must be distinctive from all load filtering hierarchy names declared). Several evaluation objects may be attached to one component or component area, each one producing a separate result table.

The (optional) DEFAULT part allows specification of defaults for the evaluation attributes of all measurements for this evaluation object (MEASURE ... AT *evaluation-_object_name* ...). Of course the evaluation attributes may be set anew in the MEASURE statements.

Note:

The DEFAULTs are ignored in CONTROL statements, they are only taken into account for MEASURE statements.

Examples:

```
EVALUATIONOBJECT
    ev_ob1              VIA mod.ct.cpu
        DEFAULT
        ESTIMATOR       FREQUENCY INTERVAL [0..5, 0..50, 0..500];

    ev_ob2              VIA SERVICE_AREA OF mod.ct.cpu
        DEFAULT
        ESTIMATOR       MEAN, STANDARDDEVIATION
        OUTPUT          TABLE "out1", TABLE "out2";

EVALUATIONOBJECT
    ev_ob               VIA mod1;

EVALUATIONOBJECT
    processor1          VIA central.system[1].cpu,
    processor2          VIA central.system[1].iop
        DEFAULT
        ESTIMATOR MEAN;
```

5.6.Load Filtering Hierarchies

Whenever streams to be measured or events to be observed (e.g., MEASURE statements or start / stop conditions) are specified, the events to be regarded may be restricted by load filtering hierarchies. They enable the user to filter the effects of special events occurring on higher model levels for a measurement at the evaluation object. The load to be regarded is filtered with both respect to the component it has been originated in (that is the component the calling process is local to, resp. has been created in) and to the hierarchy of service calls, from the originating component down to the component the evaluation object is attached to. Only those events are considered that are results of local processes within the originating component and that are calling services along the load filtering hierarchy down to (the component of) the evaluation object.

Load filtering hierarchies consist of a set of load paths, leading from originating components down to the evaluation object. Each load path consists of REFER bindings, each one connecting one used with one provided service of two components.

For using load filtering hierarchies the concept of streams (cf. Section 5.1.) is extended: Streams can be seen as lists of triples (t_i, x_i, l_i), where t_i is the time stamp, x_i is a numerical value, and l_i is a load path. At the time when the update of a stream occurs (automatically or via UPDATE statement), the load path l_i can be uniquely determined.

If a measurement is restricted by a load filtering hierarchy *h*, then only load portions initiated in the originating component and propagating down on a load path contained in *h* are considered for measurement, simply by considering only those triplets of the stream for which l_i is contained in *h*.

Please take care that (user-defined) streams are really updated to avoid undefined result values. I.e., the service containing the UPDATE statement should be catched by the load filtering hierarchy given in the corresponding MEASURE statement.

Hierarchies may be merged to define a new hierarchy, containing the union of the load paths of all merged hierarchies. In that case all events are considered that are passing any of the merged hierarchies.

Please note that for a simulation all hierarchies to be merged need not be disjoint, i.e., the intersections of all load path sets need not be empty.

A load filtering hierarchy may be empty (i.e., there is no sequence of service calls compliant to it) for different reasons:

- There is no appropriate CREATE / SUBMIT statement or local process within the originating component or one of its services.
- One of the required REFER bindings does not exist (see Section 5.6.1.).
- There is no fitting used service call in the body of a relevant service down the declared hierarchy.

In those cases the compiler issues a warning. Although a hierarchy is not empty due to the first reason the results may be meaningless if no CREATE / SUBMIT statement is actually executed.

5.6.1. Declaration of Load Filtering Hierarchies

The declaration of a load filtering hierarchy is allowed in the declaration part of an EVALUATE statement only.

```
hierarchy_declaration ::=
    HIERARCHY
    {hierarchy-name [, …] default_or_merge ; } [ …]

default_or_merge ::=
    DEFAULT   hierarchy_part [. …]
|   MERGE     hierarchy-name   [, …]

hierarchy_part ::=
    hierarchy-name
|   triplet

triplet ::=
    ( component-identifier [, service-name [, use-name] ] )
```

The name of the hierarchy follows the keyword HIERARCHY. It is arbitrary but must follow the rules on the construction of names and for the scope of validity of identifiers. All hierarchy names must be distinct from each other and all names of evaluation objects declared.

There are three possibilities of declaring load filtering hierarchies (*default_or_merge*) :

1. by explicitely specifying the hierarchy of service calls from the originating component down to the evaluation object (via DEFAULT)

2. by specifying exactly one hierarchy, referring, however, to other declared hierarchies (again via DEFAULT)

3. by merging (joining) existing hierarchies (via MERGE).

These possibilities are explained in the next three subsections.

5.6.2. Load Filtering Hierarchies Defined by Triplets

Explicitely specified hierarchies are described by a sequence of triplets, separated by dots, each of the form (c, s, u), with the following meaning:

- c : component-identifier (mandatory)
- s : service-name (optional)
- u : use-name (optional)

These elements of triplets are separated by commas. Don´t confuse these triplets to describe a hierarchy with those used to explain the concept of streams.

Example: (m.c.c1, s1, u1) . (c2, s2) . (c3) . (c4, s4)

- The **first triplet** specifies the originating component (*component-identifier*) and, (as an option) the originating service (*service-name*) and the originating used service (*use-name*). Just like evaluation objects (see Section 5.5.), the originating component must be unambiguously identified by a component path (in dot-notation) down from the model. If several paths are possible, the choice between them is arbitrary. Only those events that result from CREATE / SUBMIT statements in the body of the originating component, or from process objects declared in this component are considered. If a *service-name* is given, the events are furthermore restricted to local processes of the stated type (not necessarily provided), and, submitting also a *use-name*, to calls of the given USE name within that service.

- The **triplets that follow** define the sequence of service calls down through the static structure of the model. Some (obvious) restrictions apply:

  - *component-identifier* must be a single name (instead of a whole path), but it must specify a subcomponent of the component of the previous triplet.

  - If a *service-name* is supplied, it must belong to a provided service of the given component, and

  - if a *use-name* was (additionally) given in the previous triplet, a corresponding REFER binding to the given *service-name* must appear.

  - If a *use-name* is given, it must be that of a used service of the service regarded.

  Both, *service-name* and *use-name* may be omitted. In that case, no further restrictions to the load are implied (beyond the named component). Note that omitting of the *service-name* means that all services of the component are selected. Thus more than one load path is contained in such a hierarchy (implicit merge). The same holds when omitting the *use-name*.

- The **last triplet** defines the end of the load filtering hierarchy, i.e., the component the evaluation object of interest is attached to. For obvious reasons a *use-name* does not make sense here and is ignored if given. A *service-name* may be supplied selecting only such kind of processes, otherwise all kinds of processes within the component are measured.

- If a load filtering hierarchy consists of **only one triplet**, the *component-identifier* is interpreted as described for the first triplet, the *service-name* as described for the last triplet. A *use-name* is ignored if given.

Examples:

```
HIERARCHY

h0      DEFAULT    (m.c1.c2, s2, u2) . (c3, s3) . (c4) . (ev_ob)
h1, h2DEFAULT      (model.system, batch, cpuaccess) . (cpu, compute);
```

5.6.3. Load Filtering Hierarchies Defined by Hierarchies

Instead of a triplet as described above, the name of an appropriate hierarchy may be given (not enclosed in parentheses). In this case, the named hierarchy is inserted. This is very useful if only slightly different complex hierarchies have to be defined, using "the same way for a long distance". The examples below only demonstrate the technical aspect: The starting point of the inserted hierarchy must be a local component of the component denoted in the previous triplet or hierachy.

Note:

Hierarchies declared by MERGE must not be given within such dot notations.

Examples:

```
HIERARCHY

h1 DEFAULT    (model.system, batch, cpuaccess) . (cpu, compute);
h2 DEFAULT    (model);
h3 DEFAULT    (model.system.cpu.processor1);
h4 DEFAULT    (model) . h1 . (processor1);
h5 DEFAULT    h2.h1.h3;
```

Both *h4* and *h5* denote the same hierarchies.

5.6.4. Load Filtering Hierarchies Defined by MERGE

Load filtering hierarchies may be merged, i.e., the union of all load paths of the merged hierarchies is built.

For every evaluation object one such merged hierarchy is predefined:

```
HIERARCHY all MERGE ...
```

The dots are standing for a MERGE of all load filtering hierarchies ending at that evaluation object. Omitting the DUE TO-part in MEASURE statements means DUE TO all.

Please note the following restrictions when merging hierarchies:

- All merged hierarchies must end in the same component (last triplet), although they may start in different components.
- The hierarchy *all* may not be used within MERGE, since it already represents a maximal merge of hierarchies.
- There must be at least two hierarchies given on the right hand side of MERGE.

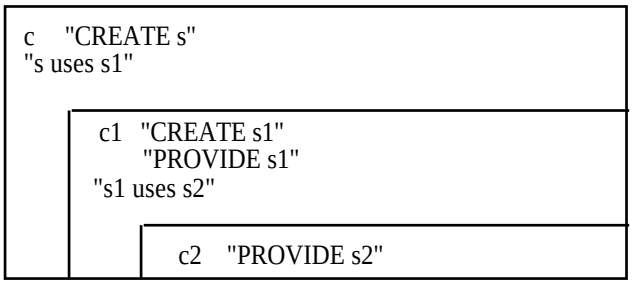
Examples:

```
HIERARCHY

h1 DEFAULT  (model.system, dialog)              .  (cpu);
h2 DEFAULT  (model.system, batch, cpuaccess)    .  (cpu, compute);
h3 DEFAULT  (model). (system). (cpu);
h4 MERGE    h1, h2, h3;
```

The role of the originating component is illustrated by the following example.

Example:

May *c* be a model, CREATEing a local process of type *s*,
s uses the provided service *s1* of the subcomponent *c1,*
c1 CREATEs a local process of type *s1,*
s1 uses service *s2* of the subcomponent *c2* of *c1:*




```
HIERARCHY
   at_c          DEFAULT (c) . (c1) . (c2);
   at_c1         DEFAULT (c.c1) . (c2);
   at_c_or_c1    MERGE at_c, at_c1;
```

The hierarchies *at_c* and *at_c1* are disjoint:

at_c describes all events that are results of CREATE *s* (in *c!*) and end in *c2* (at *s2*). The hierarchy *at_c1* describes all events that are results of CREATE *s1* (in *c1*!) and end in *c2* (at *s2*). If all events ending at *s2* are to be described then *at_c* and *at_c1* must be merged (to *at_c_or_c1*)

Next, an artificial, but relatively short example illustrating disjoint and empty hierarchies:

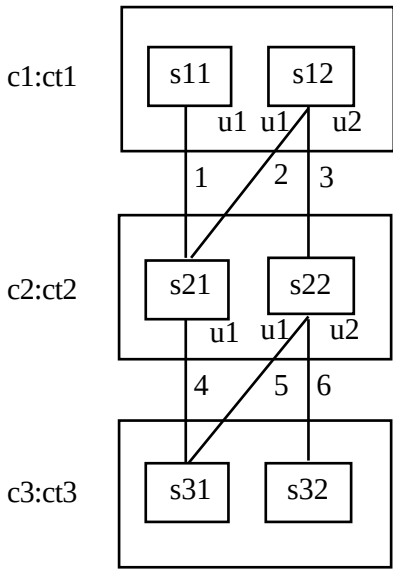
Example:

```
TYPE ct3 COMPONENT;
   PROVIDE
      SERVICE    s31; s32;
   END PROVIDE;

   TYPE     s31    SERVICE;
   BEGIN ...
   END TYPE s31;

   TYPE     s32    SERVICE;
   BEGIN ...
   END TYPE s32;
END TYPE ct3;
```

```
TYPE ct2 COMPONENT;
   PROVIDE
      SERVICE    s21; s22;
   END PROVIDE;

   TYPE     s21    SERVICE;
      USE SERVICE  u1;
      END USE;
   BEGIN
      u1;
   END TYPE s21;

   TYPE     s22    SERVICE;
      USE SERVICE  u1; u2;
      END USE;
   BEGIN
      u1; u2;
   END TYPE s22;

   COMPONENT  c3  :   ct3;

   REFER   s21,   s22    TO    c3 EQUATING
      s21.u1    WITH     c3.s31;
      s22.u1    WITH     c3.s31;
      s22.u2    WITH     c3.s32;
   END REFER;
END TYPE ct2;


TYPE ct1 COMPONENT;
   PROVIDE
      SERVICE    s11; s12;
   END PROVIDE;

   TYPE     s11    SERVICE;
      USE SERVICE  u1;
      END USE;
   BEGIN
      u1;
   END TYPE s11;


   TYPE     s12    SERVICE;
      USE SERVICE  u1; u2;
      END USE;
   BEGIN
      u1; u2;
   END TYPE s12;

   COMPONENT  c2  :   ct2;

   REFER   s11,   s12    TO    c2 EQUATING
      s11.u1    WITH     c2.s21;
      s12.u1    WITH     c2.s21;
      s12.u2    WITH     c2.s22;
   END REFER;

BEGIN
   CREATE     1  PROCESS   s11;
   CREATE     1  PROCESS   s12;
END TYPE ct1;
```

```
TYPE mt MODEL;
   COMPONENT   c1 :   ct1;
END TYPE mt;


EXPERIMENT exp METHOD SIMULATIVE;
BEGIN
   EVALUATE      MODEL   m :   mt;

      EVALUATIONOBJECT    eva     VIA    m.c1.c2.c3;

      HIERARCHY        h1a    DEFAULT   (m.c1)         . (c2);
                       h1b    DEFAULT   (m.c1, s11)    . (c2);

                       h2a    DEFAULT   (m.c1.c2)      . (c3, s32);
                       h2b    DEFAULT   (m.c1.c2)      . (c3);

                       h1     MERGE     h1a,    h1b;
                  { not disjoint, note the common path s11 -> s21 }

                       h2     MERGE     h2a,    h2b;
                  { not disjoint, note the common path s22 -> s32 }

                       ha     DEFAULT   h1a . (c3, s32);
                       hb     DEFAULT   h1b . (c3);

                       h      MERGE     ha, hb;
                  { disjoint, this MERGE is allowed }

                     empty  DEFAULT   (m.c1, s11, u1) . (c2, s22)
                  { empty, there is no REFER binding u1 <-> s22 }
   BEGIN

      MEASURE POPULATION AT eva DUE TO ha, hb, h;

      CONTROL AT eva STOP CPUTIME 200;

   END EVALUATE;
END EXPERIMENT exp;
```

The example can best be illustrated by the following (non-standard) graphics:



The paths which are constituted by the REFER parts defined in *ct1* and *ct2* are numbered here (1..6).

The following set of paths is contained in the load filtering hierarchies defined above:

h1a : {1,2,3}
h1b : {1}

h2a : {6} empty hierarchy, for c2 has no local processes
h2b : {4,5,6} empty hierarchy, for c2 has no local processes

h1, h2 : invalid, since the hierarchies above are not disjoint.

ha : {3.6} concatenationh1a · h2a
hb : {1.4} concatenationh1b · h2b
h : {3.6,1.4}

empty : { }

5.7.MEASURE Statements

In HIT only those measurements are performed in which the user is really interested in. MEASURE statements define the desired results of an analysis of a model and cause the measurement. They are permissible in the body of an EVALUATE statement only.

measure_statement ::=
MEASURE {stream [**DEGREE** simple_real_expression] } [**,** …]
AT evaluationobject-name
[**DUE TO** hierarchy-name [**,** …]]
[**ABSCISSA** simple_real_expression]
[estimator_part]**;**

The *evaluationobject-name* is the name of the evaluation object to be analyzed. It must be declared in the enclosing EVALUATE statement, and the streams listed must be standard streams or be declared in the component it is attached to.

The *stream* is either one of the standard streams (THROUGHPUT, TURNAROUND-TIME, POPULATION, UTILIZATION, OCCUPATION, SCHEDULE_RATE or PREEMPT_RATE) or a user-defined stream. User-defined streams must be declared in the component of interest (or one of its services). Each stream listed here will be measured according to the given specification.

After DEGREE the maximum degree of the autoregressive model may be given (for simulation only). The expression must be of type REAL or INTEGER (REAL is converted to INTEGER) and its evaluation must yield a value between 1 and 20. If these bounds are exceeded or DEGREE is omitted at all, 10 is used (and a warning is given in the first case).

The *hierarchy-name* after DUE TO restricts the measured updates to the given load filtering hierarchy. The hierarchy must be declared in the enclosing EVALUATE statement and end in the component of interest. If multiple hierarchies are supplied, one measurement is performed for each.

If the DUE TO part is omitted or *all* is specified, a measurement concerning all possible events is executed. The hierarchy *all* is predefined for all evaluation objects, and comprises (MERGEs) all possible hierarchies that end in the component of interest. Hierarchy *all* may not be redefined.

If it is planned to represent the results as a graph (GRAPH statement, see Section 5.2.1), one should supply an ABSCISSA part, that is the keyword ABSCISSA and an expression of type REAL or INTEGER, which values will be used for the scaling of the abscissa (x-axis).

In the *estimator_part*, values for the evaluation attributes (ESTIMATOR, OUTPUT, START / STOP condition, see Section 5.4.) may be specified. There is a default mechanism. The value of one evaluation attribute for one measurement is determined as follows:

1. If the given attribute is specified in the MEASURE statement, the supplied value is used. Else:

2. If the given attribute is specified for the evaluation object, the given value is used (values for attributes already found by 1. are ignored). Else:

3. The following default is used: ESTIMATOR MEAN OUTPUT TABLE "TABLE" and the measuring interval is not restricted.

If multiple *streams*, *hierarchy-names* and estimators are given in the *estimator_part*, one measurement is taken for each possible and meaningful combination.

Examples:

```
MEASURE       THROUGHPUT, TURNAROUNDTIME, my_own_stream { 3 }
   AT         eva_object
   DUE TO     h1, h2, h3 { 3 }
   ESTIMATOR MEAN, STANDARDDEVIATION { 2 };
   {3 * 3 * 2 = 18 measurements!}

MEASURE UTILIZATION AT server_object;

MEASURE   stream1,
          stream2 DEGREE 5,
          stream3 AT ev_ob;
          { Note: only stream2 is measured with DEGREE 5! }
```

Notes:

In the tabular output the results generated by different MEASURE statements for the same evaluation object are presented in the same table. To prevent this use different evaluation objects for the same component.

If multiple measurements with identical component, stream, load filtering hierarchy and estimator are desired (e.g., for different start or stop conditions), different evaluation objects should be attached to the same component.

A similar restriction applies if different FREQUENCY estimators are used together, differing only in the intervals specified. Here only the last one is found in the table, but all are written to the dump file if desired.

Examples:

```
EVALUATIONOBJECT
   cpu_1          VIA sys.cpu,
   cpu_2          VIA sys.cpu;

MEASURE           POPULATION
   AT             cpu_1
   DUE TO         all
   ESTIMATOR      MEAN
   START          EVENTS 100
   STOP           EVENTS 500;

MEASURE           POPULATION
   AT             cpu_2
   DUE TO         all
   ESTIMATOR      STANDARDDEVIATION
   STOP           CONFIDENCE LEVEL 95 WIDTH 10
                  MEASURE TURNAROUNDTIME;
```

5.8.CONTROL Statement

The CONTROL statement has two different purposes:

- A trace of the simulation can be demanded and
- the end of the solution can be specified by stop conditions. This is meaningful only for the following solvers: simulative, analytic-numerical (MARKOV) and LIN2. For a stationary analysis with the simulative solver it is more recommendable to specify the end of evaluation via GLOBALSTOP in the estimator part (see Section 5.4.) and to specify a simple stop condition here, e.g., CPUTIME.

The CONTROL statement is permissible in the body of an EVALUATE statement only.

control_statement ::=
CONTROL [**TRACEALL**]
{[**AT** evaluationobject-name]
[**STOP** start_or_stop_condition]
[**TRACE**] } […] ;

5.8.1. STOP Condition

To determine the end of the simulation (or evaluation for other solvers) stop conditions (logical expressions about the state of the evaluation) may be specified for evaluation objects. The actual evaluation terminates, when one stop condition given in the CONTROL statement becomes TRUE (OR-operation). The condition has the same syntax as that of MEASURE statements and evaluation object defaults (see Section 5.3.). It is only evaluated when activities take place at the selected evaluation object. There must not be an empty load filtering hierarchy specified within *start_or_-stop_condition* to avoid infinite simulations.

Expressions about EVENTS or CONFIDENCE LEVEL refer to the evaluation object given (of course). The expression is evaluated whenever a process is entering or leaving the component (area), in case of confidence interval width or CPU time conditions more rarely to reduce the computational overhead.

Note:

The DEFAULT specifications of the evaluation object are only valid in MEASURE statements; they are not used for CONTROL STOP conditions.

Examples:

```
EVALUATIONOBJECT e VIA m.c DEFAULT ESTIMATOR CONFIDENCE LEVEL 90
    START MODELTIME 500;
...
MEASURE POPULATION AT e;        {here the defaults are used}
...
CONTROL AT e STOP CONFIDENCE LEVEL 90 MEASURE POPULATION;
{caution: here the measurement is started at model time 0, i.e., the start default of e is ignored}

CONTROL AT system STOP CPUTIME 1000;
```

For the MARKOV solver, only expressions concerning accuracy and the used CPU time are permissible. For the LIN2 solver, only expressions concerning accuracy are permissible. The DOQ4 solver ignores STOP conditions.

5.8.2. Trace Specifications

There are to forms of trace specifications:

- **TRACE**
  A trace of a simulation consists of all process transitions from one area of a component to another area (not necessarily of the same component). The model time of the transition, the name of the process, the actual service call and the names of the source- and sink-component are saved. The exact format is described in Appendix G.5. The trace is written via link name "TRACE" which has a default EXTEND binding to a file.

- **TRACEALL** If a complete trace of the simulation is desired, TRACEALL can be specified after CONTROL. If the trace shall be restricted to a special evaluation object only (or to some evaluation objects), the CONTROL AT *evaluationobject-name* TRACE construct must be used. In this case, only the transitions concerning the component (area) the evaluation object is attached to are saved, the format of the output remains the same. Of course the evaluation object must be declared in the enclosing EVALUATE statement.

TRACE and STOP may be specified for the same evaluation object, but TRACE should not be specified when TRACEALL is given.

Examples:

```
CONTROL   TRACEALL
                AT disk1  STOP MODELTIME 20000
                AT cpu    STOP EVENTS 10000;

CONTROL
                AT disk1  STOP MODELTIME 20000
                AT cpu    TRACE;
```

Note:

Traces produced via TRACEALL often are very voluminous. Via AT evaobj TRACE the trace can be restricted to components of interest. Moreover it is possible to temporarily switch the trace off and on again via the predefined procedures *trace_off* and *trace_on* (see Apendix D.2.4), hereby restricting the trace to time intervals of interest.

5.9.Evaluation Statements

HI-SLANG offers two different kinds of evaluation:

- The analysis of a model by the EVALUATE statements. It renders statistical performance values, either as a table or in a dump file.

- The pre-analysis of a component type by the AGGREGATE statement. It generates a state dependent *server* as an aggregate of the analyzed component. The result is written into a file (see Section 8. and Appendix G.4.) and represents a (special kind of) component type.

These statements are permissible in the body of an EXPERIMENT block only. One EXPERIMENT block may contain multiple EVALUATE and/or AGGREGATE statements.

```
statement ::= …
|  evaluate_statement
|  aggregate_statement
```

5.9.1. EVALUATE Statements

The EVALUATE statement causes a statistical analysis of a model. To accomplish that, it must perform the following tasks:

- Create a model object of a model type to be analyzed including parameter passing. This includes the incarnation of all component objects referenced (declared or enclosed) directly or indirectly within the model. Components declared outside the model type but not referenced within the model are not incarnated.

- Define precisely which performance indices are of interest and how to obtain them.

One EVALUATE statement can cause a series of evaluations by, e.g., placing it into the body of a loop. The EVALUATE statement consists of a declaration part and a body.

```
evaluate_statement ::=
   EVALUATE
      MODEL model-name : modeltype-name [actual_parameters] ;
         evaluate_declaration [ …]
   BEGIN
      measure_statement [ …]
      [control_statement]
   END EVALUATE ;


evaluate_declaration ::=
   evaluationobject_declaration
|  hierarchy_declaration
```

The statement causes the generation of a model with the name *model-name*. This name is arbitrary, but must obey the rules for constructing names and the scope of validity for identifiers. The model is of type *modeltype-name*, which must be declared in the

environment of the EVALUATE statement. Model types may have parameters, so actual parameters may be necessary. The rules for parameter passing for procedures apply.

The desired results are specified by *measure_statements* in the body of the EVALUATE statement. Load filtering hierarchies and evaluation objects used here must be declared in the declaration part (*evaluation_declarations*, consisting of *evaluationobject_declaration* and *hierarchy_declaration*).

For a simulation a *control_statement* must be given indicating when to stop the evaluation. When using solver DOQ4 a *control_statement* is ignored. For MARKOV and LIN2 there may be a *control_statement*.

The EVALUATE statement is only permissible in the body of an EXPERIMENT block. For the simulative method it must not be used within a BLOCK, CASE or WITH statement.

An EVALUATE statement may not contain calls of self-defined procedures or random drawing procedures (see Apendix D.2).

Example:

```
EVALUATE  MODEL computer_system : computer_type (processor_speed, 10.0);
    EVALUATIONOBJECT …
    HIERARCHY …
BEGIN
    MEASURE …
    CONTROL …
END EVALUATE;
```

5.9.2. AGGREGATE Statements

The AGGREGATE statement specifies to pre-analyze model parts by aggregating complex component types to a state dependent *server* (a component type as well, a so-called substitute component type). The AGGREGATE statement is permissible in the analytic-algebraical case (DOQ4) only; furthermore, it is only permissible in the body of an EXPERIMENT block.

Aggregation is on the one hand a convenient means for reducing the state space, particulary for numerical solvers, and on the other hand for reducing the simulative complexity in nearly complete decomposable (NCD) networks.

Within the AGGREGATE statement the maximum population of each provided service of the component type has to be defined by CREATE statements.

```
aggregate_statement  ::=
    AGGREGATE componenttype-name [OUTPUT simple_text_expression] ;
        create_or_submit_statement  [ ...]
    END AGGREGATE ;
```

The component type to be aggregated (*componenttype-name*) must be declared in one of the blocks enclosing the AGGREGATE statement. Neither the component type nor one of its provided services may be parameterized. For further restrictions see Appendix E.

The CREATE statements (*create_or_submit_statements*) specify the maximum population for each provided service of the component type. The statements must be of the form

```
CREATE max_population PROCESS service-name;
```

The generated aggregated component type is written via the link name which may be specified after OUTPUT (default is "PREANA"). The expression must be either a constant TEXT or a constant or variable of type TEXT. The compiler control statements for using an aggregated component type are described in Section 8. For the format of an aggregated component type see Appendix G.4. The aggregated component type has the same name as the component type to be aggregated, so that there are no changes in the environment necessary (e.g., when declaring components).

Example:

```
AGGREGATE  ct;
   CREATE    10  PROCESS  pt1;
   CREATE     5  PROCESS  pt2;
END AGGREGATE;
```

All solvers are able to use aggregated component types, hereby enabling hierarchical and heterogeneous analyses.

Note:

When you use the produced aggregated component type, don't forget to include a %BIND statement on the produced file in the %COMMON part, since it must be read by both, compiler and analyzer.

5.10. EXPERIMENT Block

The "evaluation program" (that is the sequence of evaluations or aggregations desired) is defined by the EXPERIMENT block. The EXPERIMENT block consists of the specification of a solver, the declaration part and the body.

```
experiment_block ::=
    EXPERIMENT experiment-name METHOD method ;
        [declaration [ …]]
    BEGIN
        [sequence_of_statements]
        [plot_statement [ …]]
    END EXPERIMENT [experiment-name] ;


method ::=
    ANALYTICAL simple_text_expression
|   SIMULATIVE


declaration ::=
    common_declaration
|   modelling_declaration
```

For user convenience each experiment has a name (which will be used as a title when presenting results, etc.). This name is arbitrary (but must obey the rules for constructing names and the scope of validity for identifiers). This name may be repeated at the end of the EXPERIMENT block. A warning is submitted if it is repeated incorrectly.

After METHOD the solver (*method*) for this experiment is selected. This decision is valid for the whole EXPERIMENT block. The simulative solver allows to use (nearly) the complete set of HI-SLANG constructs, for the very few restrictions compared with the analytical solvers see Appendix E.2.

If one of the analytical solvers is chosen, only a subset of HI-SLANG is permissible, these restrictions are described in Appendix E.1.

The *simple_text_expression* after ANALYTICAL must always be of type TEXT. The following values are permissible (case-insensitive):

- "**DOQ4**", "DOQ3", "SEPARABLE", "BCMP", "PRODUCTFORM", "NONSE-PARABLE-APPROXIMATE"

  These strings all select DOQ4. The decision about the concrete algorithm (exact/approximate etc.) will be made internally.

- "**LIN2**", "LINEARIZER", "SEPARABLE-APPROXIMATE", "PERFORMANCE BOUNDS"

  These strings select LIN2.

- "**MARKOV**", "NUMERICAL", "MARKOVIAN", "MARK"

  These strings select the numerical solver (MARKOV).

In the declaration part of the EXPERIMENT block all kinds of declarations are permissible except:

- virtual declarations of components (ENCLOSE),
- declarations of streams (STREAM),
- declarations of services (TYPE *name* SERVICE), and
- declarations of process objects (PROCESS).

Especially the declaration of model types and component types is permissible, but should preferably be done before the experiment block in order to clearly separate the model description from the experiment description.

The body of the EXPERIMENT block consists of arbitrary statements, especially zero or more EVALUATE or AGGREGATE statements (*sequence_of_statements*) and as last statements zero or more GRAPH or HISTOGRAM statements (*plot_statement*). The EVALUATE and/or AGGREGATE statements may be enclosed by control constructs such as LOOP to perform evaluation series.

Example:

```
{ declaration part for model and component types }

EXPERIMENT analysis METHOD ANALYTICAL "Separable";

    { declaration part, mainly variables for evaluation control }

BEGIN

    { body, containing EVALUATE and/or AGGREGATE statements }

END EXPERIMENT analysis;
```

6. HI-SLANG Source Structure

A compilable HI-SLANG source (*hit_unit*) consists of an EXPERIMENT block (*experiment_block*) combined with an optional declaration part (*declaration*).

```
hit_unit ::=
    [declaration [ ...]] experiment_block


declaration ::=
    modelling_declaration  |  common_declaration


modelling_declaration ::=
    process_declaration
|   component_declaration
|   enclose_declaration
|   stream_declaration

common_declaration ::=
    record_declaration
|   variable_or_constant_declaration
|   procedure_declaration
|   type_declaration
```

Any type of (syntactically deductible) declaration, excepting streams, virtual components using ENCLOSE, services and process objects, is allowed in the declaration part preceding an EXPERIMENT block. Within the *common_declaration* part, procedures, variables, constants and any type of modelling objects can be declared.

A complete HI-SLANG source roughly falls into the parts of "model description" (Chapter 4) and "model analysis" (Chapter 5). The model description part contains the declaration of one or more model types and/or one or more component types. Model analysis is meant on the one hand for statistical evaluation of the model types via created models, on the other for pre-analysis of component types to generate aggregated submodels, or for both.

A more complex structuring is provided by definition of model type structures and component type structures as well as by the block structure of the programming language kernel of HI-SLANG.

Section 6.1. discusses the block concept of HI-SLANG. It treats the general programming language concepts of HI-SLANG as well as the structuring mechanisms of modelling and performance evaluation. The scopes of identifiers based on the block concept are explained in Section 6.2. following some simple rules.

6.1.The Block Concept

The essential structuring technique in HI-SLANG is the block concept. A block consists of a declaration part to define local objects and data types and of a statement part in which those objects and types are used. Blocks serve two different purposes:

- The EXPERIMENT block, the EVALUATE statement, the BLOCK statement and procedures are part of the structuring concept of HI-SLANG as a high level language.

- Model types, component types and services are the basic utilities for structured modelling. Their use and their arrangement depends on the system that is to be modelled.

One of the main tasks of blocks is to limit the scope of objects and types to provide means for efficient problem handling. The module-like blocks (component and model types) enhance this concept by yielding exactly defined interfaces.

The method of hierarchical ordering of blocks by nesting one block into another is useful in a structured problem handling following the idea of stepwise refinement. The embracing outer blocks contain and furnish general utilities as well as global objects and types.

Each complete HI-SLANG source consists of exactly one EXPERIMENT block that can be preceded by a declaration part. The EXPERIMENT block and the preceding declaration part are embedded in an outer block, the modelling environment, which contains all predefined objects and types. Part of this environment is represented by the HIT Standard Mobase, the rest is built into the HIT system itself such as standard procedures.

6.2.Scopes of Identifiers

All local identifiers declared within one block must be unambiguous. Multiple declarations of an identifier within one block leads to a HI-SLANG compile error message. The name of the EXPERIMENT block, the model name within an EVALUATE statement and the formal parameters of procedures, model types, component types and services are local in that block. These identifiers may not be redeclared within the declaration part of that block.

Regardless of the sequence of notation, every declared identifier is known in the whole declaration part and the statement part of the block. It is also known in every inner block unless the identifier has been newly declared there. If it has, only the latest declaration is valid for this block as well as for all following inner blocks (in terms of the block structure). Newly declaring an identifier is thus allowed and replaces (the validity of) all previous declarations in outer blocks.

Identifiers declared in the declaration part preceding the EXPERIMENT block are known in the EXPERIMENT block itself. Identifiers declared in the modelling environment (e.g., predefined procedures and constants) are global to the entire source.

Relevant in terms of the scope of identifiers in procedures, record types and any type of modelling object is only the declaration environment, not the block in which a procedure has been activated or an object has been generated.

Outside of a block its local declarations are not known. This is true for embracing blocks as well as parallel blocks. Some exceptions to this rule are discussed later on.

Services and procedures which are local within a component type may be made available to the next-higher component type by use of the PROVIDE declaration. The upper level component type can take access to one of these provided objects using a special mechanism, the REFER part. The procedures *popul*, *popul_announce*, *popul_entry*, *popul_service* and *popul_exit* are known in any component type without explicitely being provided. Services provided by a component type are also known and available within the statement part of the AGGREGATE statement.

The identifiers of utilized services or procedures being used within a service or procedure definition are specified in the USE declaration part, the interface of services and procedures. Just as the provided objects, they can only be used in the REFER part of the component type they belong to.

Access to block-local objects and types is limited within CONCURRENT statements in services. See Section 4.1.5.1.

The identifiers of components, services and used services may be accessed in the definition of evaluation objects and load filtering hierarchies. They must, however, be completely specified and qualified by navigating from the model down to the evaluation object.

References to files (by link names) are not influenced by the block concept. Therefore, all used link names must be unambiguous and globally declared in the control file. Regarding the control file, they may refer only to one file or mobase object.

7. Installation Dependent Properties

The HI-SLANG compiler is a pre-compiler generating SIMULA code (see Section 1.5.). A portation therefore depends upon a suitable SIMULA compiler on the target machine.

Presently, the following HIT installations exist, determined by hardware, operating system and SIMULA system. They are in use at different universities and companies.

1)	Siemens mainframe	, BS2000, (VM/CMS)	, IBM-SIMULA
2)	IBM mainframe	, MVS	, IBM-SIMULA
3)	SUN/3, SUN/4	, SunOS4.1	, Lund-SIMULA, CIM
4)	Apollo workstation	, Domain/OS SR 9.7 and 10.2	, Lund-SIMULA
5)	most PC'386	, interactiveUNIX, SCO/UNIX	, S-PORT-SIMULA, CIM
6)	DEC station	, Ultrix	, Lund-SIMULA, CIM

The following are or have been in the stage of planning or preparation:

8)	Targon 31	, UNIX System V	, TPH-SIMULA
9)	VAX	, UNIX	, Lund-SIMULA

For the mentioned Simula Systems see /IBMSim86/, /LundSim87/, /PCSim89/, /TPHSim87/ and /CIM91/.

Operating system command procedures, which facilitate the usage of the HIT system, exist for all installations. An explanation of these procedures is provided in the HIT User's Guide, e.g., /LeWe92/.

7.1.SIMULA System Dependencies

It is obvious that restrictions of the SIMULA system used also apply for HIT. Especially for installations based on the IBM SIMULA system there are certain limits for the number of identifiers declarable and the depth of nesting blocks and control statements. Since the HI-SLANG compiler inserts identifiers and blocks to the generated SIMULA code these restrictions cannot be exactly quantified for HI-SLANG. Moreover, a considerable amount of CPU time may be spend with garbage collection when compiling or analysing voluminous models.

If an optimizing SIMULA system is used the SIMULA compiler may find errors which normally are not detected before run time, as, e.g., "division by zero" within constant expressions. This kind of errors are the only ones allowed to occur during the SIMULA compilation phase of HIT.

Since HI-SLANG has no run time system of its own the SIMULA run time system is used for the HIT analyzers. Thus the format of basic run time errors depends on the SIMULA system used.

7.2.Operating System Dependencies

File names are specified only within the HIT control file (see Chapter 8.), not within the HI-SLANG source code. The file names have to correspond to respective operating system conventions. If the file names are too long, run time errors are possible, depending upon the installation. This applies especially for generated file names created from the name of the control file according to a fixed scheme (see Section 8.2.6.). For installations under UNIX, the distinction of upper- and lower-case characters and, in some versions, the limitation of the length of the name to 14 or even less characters has to be paid particular attention.

Within HI-SLANG sources files are only addressed via link names. The binding of link names to files or objects within a modelling base is only contained in the control file, so HI-SLANG models are completely portable. The link names are texts which can be chosen randomly.

Depending on the installation, HIT varies in the respect of

- the %CMD record within the HIT control file, since the argument is an operating system command,
- the STARTUP control file in total, since it defines file names and file name generation patterns.

For additional reference turn to Chapter 8.

7.3.Hardware Dependencies

In the following passage, only the installation-dependent properties of the standard types are listed. They depend upon the hardware employed and are adapted for HIT from the SIMULA system used.

type of data	**default value**	**range of values**
INTEGER	0	-maxint .. maxint
REAL	0.0	-maxlongreal .. maxlongreal
BOOLEAN	FALSE	TRUE, FALSE
CHARACTER	char(0)	all available characters
TEXT	""	"" or sequence of characters

The IBM- as well as the Siemens-installations use EBCDIC code, while all UNIX installations use ASCII code. The predefined procedures *rank* and *char* give access to the code (see Appendix D.).

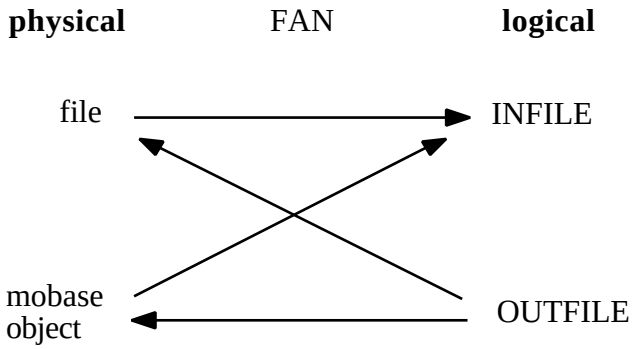
The typical value of *maxint* is 2**31 (approximately 2.14E9). On mainframes from IBM and Siemens, *maxlongreal* has the value of 16**63 (approximately 7.23E75), whereas on SUN workstations it has approximately the value 1.8E308. There, the output of the exponent always occurs in three digits. REAL values mostly have a precision of approx. 16 decimal digits. The exact values can be taken from the description of the hardware or the SIMULA system, or try the HI-SLANG constants MAXINT and MAXREAL.

8. Control of the HIT System

HIT is called by an installation dependend operating system procedure, sequentially activating the HI-SLANG compiler, the SIMULA compiler and linker and the generated, compiled and linked analyzer.

Control of the HIT system means defining files or objects as input or output for all these steps and setting of different parameters. Both is done in the so-called HIT control file, which is interpreted by the HIT-FAN system of the compiler or analyzer.

FAN means **F**ile **A**ccess **N**etwork. With the help of this network every logical file HIT deals with may be connected either to a physical file or to an object within a modelling base (shortly called mobase). For output files even both may apply at the same time.



The modelling base is designed to store all objects occuring in the modelling and evaluation process. It can be managed by the interactive program HIT-OMA (**O**bject **MA**nager) (see /Weis92b/), but the HIT system is able to read and write objects within such a mobase directly.
The FAN system thus combines the operating system interface and the modelling base interface to one standard interface for the HIT system, which may be "programmed" by the "user interface", being the control file in the simplest case. However, there are defaults for all control records (the contents of a control file), so one can use HIT even without a control file. In this case a HI-SLANG source file may be given as control file.

8.1.File Objects and Link Names

A file object is an operating system file or an object within a modelling base. For historical reasons file objects are often simply called files in HI-SLANG. As a rule file objects are denoted by their link names, being strings of arbitrary contents and length ≤ 132 characters. This is the most flexible way of accessing files.

Some link names are predefined by HIT, especially those having a fixed meaning as, e.g., "LISTING" or "TABLE". Special purpose link names may be invented by the user. Most link names are declared in the control file and occur in HI-SLANG source files at different places.

The following two tables give a complete overview about all standard link names and user-defined link names, their meaning and usage in HI-SLANG sources. The column titled # gives the maximal number of file objects of that kind (n : no restriction). An *L* in place of the link name denotes an arbitrary user-defined link name.

File Objects of the Compiler

	#	kind of file object	link name	occurance in the source
	1	control file	CONTROL	-
	1	HI-SLANG source	SOURCE	-
	1	standard input	SYSIN	-
I	1	initial input	SYSINIT	-
N	n	HI-SLANG module	L	%COPY "L"
	n	aggregated component type	L	%COPY "L"
	1	message library	MESSAGE	-
O	1	generated analyzer	CODE	-
U	1	compiler listing	LISTING	-
T	1	standard output	SYSOUT	-

File Objects of an Analyzer

	#	kind of file object	link name	occurance in the source
	1	control file	CONTROL	-
	1	standard input	SYSIN	-
	1	initial input	SYSINIT	-
I	n	aggregated component type	L, Typename	%COPY "L"
N	n	infile	L	OPEN f, "L" LENGTH n
	n	dump file	L, DUMP	GRAPH...INPUT "L" HISTOGRAM...INPUT "L"
	1	message library	MESSAGE	-
	n	tabular results	L, TABLE	...OUTPUT TABLE "L"
	n	dump file results	L, DUMP	...OUTPUT DUMPFILE "L"
	n	produced graph	L, GRAPH	GRAPH...OUTPUT "L"
	n	produced histogram	L,HISTOGRAM	HISTOGRAM...OUTPUT "L"
O	n	outfile	L	OPEN f, "L" LENGTH n
U	1	trace	TRACE	-
T	n	aggregated component type	L, PREANA	AGGREGATE ct OUTPUT "L"
	1	numeric matrix scheme	MATRIX	-
	1	numeric state table	STATES	-
	1	analyzer listing	LISTING	-
	1	standard output	SYSOUT	-
		(internal data structures)	PRINTDS	-

Notes:

L, link name means that *link name* is predefined and bound to a default file, but *link name* may be rebound and it can also be substituted by an arbitrary link name. The file objects with link names "TABLE", "DUMP" and "TRACE" are written in EXTEND mode to support evaluation series: The results of the next evaluation are appended to the current results.

Rebinding "PRINTDS" the corresponding file should be bound in EXTEND mode (to get the complete information).

For more information concerning these link names see Section 8.2.2. and Appendix G.

8.2.The Control File

If the name of the control file has not been passed to the operating system procedure calling *hit*, the HI-SLANG compiler or analyzer will ask for that name, e.g., by

```
%BIND "CONTROL" TO ?
```

Now you can enter a file name or specify an object within a modelling base (see syntax *file_object*, Section 8.2.2.). It is also possible to enter *SYSIN* here, whereafter the contents of the control file can be entered via keyboard.

The control file has record length ≤ 132 and the following structure:

%COMMON
control records valid for both compiler and analyzer

%COMPILER
control records for the compiler

%ANALYZER
control records for the analyzer

%END
HI-SLANG source may follow, e.g., %COPY directives

The FAN system of the compiler only interprets the %COMMON- and %COMPILER-part of the control file in the given order, while the analyzer's FAN scans the %COMMON- and %ANALYZER-part. Only the interpreted records occur in the listing. Every part may be missing (even the control file in total), and the sorting of these parts is arbitrary. Lower-case letters are converted to upper-case letters (with the exception of file names) and are listed in such style.

After the keyword %END the HI-SLANG source to be compiled may appear (record length ≤ 132). The control file part may also be totally missing, e.g., in the case that a source file is given as control file and thus all defaults are used and missing file bindings are queried during run time.

All control records start with a '%'-character, which is the first character different from blank or tab. The records may be completed in the following lines.

There are five different control records to be introduced in the next sections: %PARM and %BIND, as well as the less important records %MOBASE, %CMD and %DEFAULT.

8.2.1. Setting Parameters (%PARM)

For setting different parameters in the control file there exists the control record.

control_record ::= ...
| **%PARM =** parameter [, ...]

Different parameters may be separated by commas or distributed on different %PARM records. There are three classes of parameters: Compile options affect the operation of the compiler, while analyzer options take influence on analyzer outputs. These parameters do only make sense in the corresponding part of the control file, while printing options affect both the listing of the compiler and that of the analyzer.

In general, a parameter has one of the following forms:

parameter ::=
name = value
| [**NO**]name

Either a *value* is assigned to *name* or *name* is a switch which may be reset by *NOname*.

In the following three lists of parameters the defaults are given in brackets after the name of the parameter. A default YES means that no NO-prefix is given in the default.

8.2.1.1. Compiler Options

Compiler options modify the operation of the HI-SLANG compiler. There are the switches CHECK, COM as well as XREF, DEBUG and CIM, with the following meanings:

CHECK (NO)

The HI-SLANG source is only tested for syntactical and semantical correctness. No code is generated. Because CHECK is always the invers to COM it can also be set by NOCOM.

COM (YES)

COM is the normal compile mode. A complete compilation is performed and the SIMULA code is generated via link name "CODE". If no errors are detected, the SIMULA compiler and linker and the produced analyzer are started afterwards. Because COM is always the invers to CHECK, NOCOM is the same as CHECK.

XREF (NO)

If XREF is demanded the compiler will additionally generate a cross-reference-table of all the identifiers in the given source. The XREF-table is appended to the listing, see Appendix G.1.

DEBUG (NO)

If DEBUG is set, the generated SIMULA code contains the corresponding absolute HI-SLANG source line numbers (whenever possible) as prefixing comments in the style

!nnnnn; <code.line>

This might help when there are problems as described in Appendix. H. It is planned to construct a HI-SLANG debugger HIDe using this kind of information.

Example:

```
%PARM = CHECK, xref, debug
```

CIM

If the CIM switch is set, a slightly different code for the CIM SIMULA system is generated. The default is installation dependent.

Example:

```
%PARM = CIM
%PARM = NOCIM
```

8.2.1.2. Printing Options

The compiler as well as all the analyzers produce a listing. Both start with a protocol of all interpreted records of the control file, followed by a completion message of the FAN system. The compiler listing is continued by a numbered display of the assembled source. For further remarks see Appendix G.1.

For pretty printing there are the parameters MAXERROR, LINES and INDENT, which can be assigned a value, and the switches SOURCE, RESWD, WARN, WARNACCESS and RELATIVE.

They have the following meanings, with default values given in brackets:

SOURCE (YES)

Normally a listing of the control file and the source according to the following parameters is generated. If NOSOURCE is given there will be no listing at all, even if the source contains %SOURCE compiler directives. Especially for big models this can save a lot of time, because PASS 1 only needs half the time as usual.

RESWD (YES)

If RESWD is set all reserved words of HI-SLANG will be printed in capital letters, while all other identifiers are printed lower-case, independent of the style of writing in the source. NORESWD suppresses this preparation.

WARN (YES)

Not only error messages but also warnings may be given on the listing as well as on the standard output SYSOUT (e.g., terminal). NOWARN suppresses warnings.

WARNACCESS (YES)

Warnings about the usage of variables, constants, (self-defined) procedures, records and record types, which are not defined locally and are not accessed via a (parameter) interface, may be given on the listing as well as on the standard output SYSOUT (e.g., terminal). NOWARNACCESS suppresses these warnings.

RELATIVE (YES)

Normally every error message or warning of the compiler is followed by another message giving the relative line number (second row of numbers in the listing) and the file object which contains the error or caused the warning. This second message is only given if the relative line number is different from the absolute line number of the assembled source. With NORELATIVE it can be suppressed.

MAXERROR = number (200)

The *number* gives the number of error messages after which the HI-SLANG compiler or analyzer is stopped due to maxerror overflow.

LINES = number (65)

The *number* denotes the number of lines per page of the listing. The first two lines of the page, the title line and a blank line, are not included.
The value 0 means an unlimited number of lines per page.

INDENT = [character] number (blank 3)

The *number* gives the number of blanks per block for indenting the listing. Optionally, a *character* may appear immediately after the '='-character. This character is used to connect start and end of a block. Only digits are not allowed here, meaning to omit the *character.*

Example:

control file:

```
%PARM = INDENT = |  3
```

listing:

```
TYPE st SERVICE;
|       USE
|       |     SERVICE s;
|       END USE;
BEGIN
|  ...
|  ...
END TYPE st;
```

Examples:

```
%PARM = INDENT =: 5, noreswd, nowarn, maxerror = 10
%PARM = lines = 60, NORELATIVE
```

8.2.1.3. Analyzer Options

Currently there are these options to influence analyzer output: EXTERN, UPDATES, MINMAX, SOLVERINFO, FREQUENCYFORMAT, TRACEFORMAT and PRINTDS. They have the following meanings:

EXTERN (YES)

The matrix scheme and the states of the numerical solver are stored on external files, so the virtual storage of the user can completely be used for the solving algorithm. Normally this saves run time. If NOEXTERN is set the two data sets are managed internally.

UPDATES (NO)

In table outputs of simulations the number of updates recorded for all streams will be given after the mean value in the same field. An update is the execution of an UPDATE statement, or an automatic update of a standard stream (cf. Section 5.1.3).

This option can also be activated by %parm=events, but has been aliased now, since confusion with event streams as well as events occuring in STOP conditions (a process leaving a component) should be avoided.

MINMAX (NO)

In table outputs of simulations the minimum and maximum value recorded for the corresponding stream within the observation interval will be given below the mean value in the same field.

For COUNT / STATE / EVENT streams these values are the minimum and maximum interevent time / trajectory value / recorded value, respectively.

SOLVERINFO (YES)

The user will be informed why a special algorithm of the selected solver was applied. See Appendix G.1.3. for more information.

FREQUENCYFORMAT = number (1)

This option controls the table output of the estimator FREQUENCY. The default value of 1 corresponds to relative values for STATE streams and absolute values for EVENT streams. A value of 2 means relative values, whereas a value of 3 means absolute values of times for STATE streams and number of occurrences of EVENT streams.

TRACEFORMAT = number (1)

This option affects the format of the event trace of a simulation. The numbers 1, 2 and 3 are currently supported, other numbers cause an unpredictable behaviour. See Appendix G.5.1. for more information.

PRINTDS (NO)

Print out most data structures construced by the analytical solver applied. This option is mainly for debugging purpose, thus the output generated via the standard link name "PRINTDS" is not documented.

Example:

```
%PARM = noextern, updates, minmax
```

8.2.2. Binding Link Names (%BIND)

For the binding of a logical link name to a physical file or mobase object (or both) the HIT system provides the control record

```
control_record  ::= ...
|   %BIND "link_name" TO [io_mode] file_object
```

By using link names instead of file names or specifications of mobase objects within the HI-SLANG source the models become independent of the operating system and modelling base underlying HIT. As a rule, link names are always double-quoted, and using lower-case letters or upper-case letters is the same.

The keyword TO may be followed by a mode specification:

```
io_mode  ::=
    EXTEND
|   READONLY
```

Normally a file object may either be read or (over)written or sequentially both. Specifying READONLY protects the file object from being changed, while EXTEND means that anything written to that object will be appended to its end.

The name of a file or specification of a mobase object follows:

```
file_object ::=
    mobase_object
|   file_name
|   SYSIN
|   SYSOUT
|   "link_name"
|   DEFAULT
```

SYSIN and SYSOUT are special file names standing for standard input (e.g., keyboard) and standard output (e.g., terminal). Giving a link name as file object denotes that special file or object already bound to that link name. The alternative DEFAULT activates the HIT default file name generator to create a file name according to the pattern defined by the %DEFAULT record.

Every link name may be bound at most to one file and/or one mobase object, only the last such binding is valid. If a link name has a file binding and a mobase binding both file objects are written in parallel, while only the mobase object can be read in such a case. Different link names can be bound to the same file_object and vice versa, but this facility should be used carefully.

For some predefined link names there are some restrictions:

- If there is no binding to "SOURCE" within the compiler control part the compiler expects to find its HI-SLANG source following the %END of the control file.

- The predefined binding of "CODE" (generated SIMULA code) to a file (defined in the STARTUP file of HIT) may not be overridden for most installations of HIT. This is because the SIMULA compiler following HIT has to find its input.

- If "SYSINIT" is bound then "SYSIN" has initially the same binding. Not before the end of the "SYSINIT" file object is reached the default or explicit binding of "SYSIN" becomes active. For UNIX installations, "SYSINIT" is bound in the STARTUP file to deliver the name of the control file (normally read from standard input) to compiler and analyzer. Here it may not be rebound.

- The link name "MESSAGE" may only be bound to directfiles having the HIT message format. This is of no interest to the normal user.

The syntax of *file_name* and *mobase_name* is defined by the underlying operating system, but may not contain a '('-character or a blank (exception: in CMS it must contain one blank). Please also note that the check of *file_name* is realized in SIMULA, i.e., only the existence of the name can be checked (for example a directory (UNIX) is handled as a file but can lead to SIMULA Run Time Error at the first read attempt).

If the name is not followed by an opening bracket it denotes a file, otherwise a mobase, and the access attributes of the desired object are given in brackets. The following section defines these attributes for the standard modelling base of HIT.

If a file to be read cannot be found in the actual directory, an error message occurs. For UNIX installations the file is then also searched in the directory where the control file was found, if the file is not specified with absolute path.

Examples:

```
%bind "source"      to SYSIN
%bind "TRACE"       to SYSOUT
%bind "TABLE"       to extend SYSOUT
%bind "data"        to DEFAULT
%bind "input"       to readonly DEFAULT
```

8.2.3. Accessing Mobase Objects

Currently there is one standard modelling base for HIT simply called HIT-MOBASE. There are two interfaces for this mobase: One is used directly by the HI-SLANG compiler and all analyzers, the other is the object manager HIT-OMA described in /Weis92b/. Both use the same syntax to access objects. The objects are identified by a *mobase_name* followed by at most 4 attributes in the following order:

```
mobase_object ::=
   [mobase_name] ([module], [type], [object], [protection])
```

Mobase_name denotes the name of the mobase in which the object is (to be) stored. A user can work with an arbitrary number of mobases at the same time, one of which is the HIT standard mobase. If *mobase_name* is omitted the actual mobase is referred to. An actual mobase is defined by the %MOBASE record (see below). If the control file is itself stored in a mobase this mobase becomes the actual mobase if no %MOBASE record is used.

Module defines the representation of the object. If it is omitted the most efficient existing representation is used (e.g., PREANAlyzed before HI-SLANG).

Type denotes the contents of the object (e.g., COMPONENT or SERVICE). It may be omitted if the object is nevertheless identified uniquely (see below).

Object is the object's name composed of letters, digits, dots and underscore characters, the first character being a letter. Only the first 12 characters are significant. If the object name *object* is omitted the link name (see %BIND) is used as object name.

Protection. This attribute may have the values 'p' (protected) or 'u' (unprotected). Objects are created unprotected if this attribute is omitted or not set to 'p'. Protected objects may be read, but can only be overridden when protection is set to 'p'.

The attributes *module* and *type* may be abbreviated but must clearly identify the attribute value. Moreover *module* and *type* have to be consistent. The following combinations are allowed:

module	type
CONTROL	CONTROL, OTHERS
HISLANG	MODEL, COMPONENT, SERVICE, PROCEDURE, EXPERIMENT, OTHERS
PREANA	COMPONENT, OTHERS
SIMULA	ACCEPT, SCHEDULE, DISPATCH, OFFER, OTHERS
DATA	FILE, DUMPFILE, TABLE, GRAPH, HISTOGRAM, TRACE LISTING, MATRIX, STATES, OTHERS

To simplify the specification of a mobase object there is a four-level default mechanism:

1. If *object* has not been specified, its default value is defined as the *link name.*

2. When binding a standard link name, the values for *module* and *type* are already determined distinctively by the link name itself, following the table below. The only exception is marked by an asterisk in the table: Here the values following the asterisk are the default:

link name →	**module** &	**type**
CONTROL	CONTROL	CONTROL
SOURCE	HISLANG	*EXPERIMENT
PREANA	PREANA	COMPONENT
LISTING	DATA	LISTING
DUMP	DATA	DUMPFILE
TABLE	DATA	TABLE
GRAPH	DATA	GRAPH
HISTOGRAM	DATA	HISTOGRAM
TRACE	DATA	TRACE
MATRIX	DATA	MATRIX
STATES	DATA	STATES

3. If a user-defined link name was bound and the object is to be read, the most efficient object matching the attribute combination is searched for in the specified mobase. The *module* precedence rules are CONTROL < DATA < HISLANG < PREANA < SIMULA, the most efficient representation being SIMULA. If several objects of different *types* match the specification, the object with the type lowest in alphabetical order is addressed.

4. If no object can be found following the procedure described above, and if neither *type* nor *module* is given, an error message is produced. In all other cases the missing attribute is set according to the following table, an asterisk distinguishing the defaults, all other combinations being determined by the consistency table.

type →	**module**
CONTROL	CONTROL
COMPONENT	*PREANA
SERVICE	*HISLANG
PROCEDURE	*HISLANG
MODEL	*HISLANG
EXPERIMENT	*HISLANG
FILE	DATA
DUMPFILE	DATA
TRACE	DATA
TABLE	DATA
LISTING	DATA
MATRIX	DATA
STATES	DATA
ACCEPT	SIMULA
SCHEDULE	SIMULA
DISPATCH	SIMULA
OFFER	SIMULA
OTHERS	*SIMULA

modul →	**type**
CONTROL	CONTROL
DATA	*TABLE
HISLANG	*EXPERIMENT
PREANA	COMPONENT
SIMULA	*SCHEDULE

Examples (for %BIND):

```
%BIND "SOURCE"      TO  mylib (HISLANG,  , INSTALLATION)
%BIND "ST"          TO  mylib (, COMPONENT, CT)
%BIND "ct_agg"      TO  mylib (PREANA,,,P)
```

8.2.4. Declaring a Modelling Base (%MOBASE)

If several link names are to be bound to objects within a single mobase, the constant repetition of the mobase name can be annoying. If no name is specified, the mobase name declared by the last %MOBASE record is amended. If there was none and the control file itself is contained in a mobase, this mobase becomes the actual mobase automatically. The record has the following syntax:

control_record ::= ...
| **%MOBASE** [**READONLY**] mobase_name

The syntax of *mobase_name* depends on the underlying operating system. READ-ONLY specifies that objects may only be read from that mobase. In this case more than one user can read objects from that mobase at the same time. The HIT standard mobase is automatically declared in this way. If READONLY is omitted, only the user can read from and write into the mobase.

The READONLY modus can only be specified by using %MOBASE. Mobase names introduced within a *file_object* are therefore exclusive to the user.

Examples:

```
%MOBASE    READONLY    project.x.lib
%BIND      "input1"    TO   (DATA, FILE, in1)
%BIND      "input2"    TO   (DATA, FILE, in2)

%MOBASE    mylib
%BIND      "TABLE"     TO   (, , mytab, P)
%BIND      "DUMP"      TO   (, , mydump, P)
```

8.2.5. Specifying Operating System Commands (%CMD)

It may be useful to submit operating system commands before and/or after the run of the compiler and/or analyzer. This can be specified by the control file record

```
control_record  ::= ...
|    %CMD [AFTER] "operating_system_command"
```

where *operating_system_command* must comply with the syntax of commands of the underlying operating system. Specification of AFTER implies execution after the run of that part of HIT initially interpreting the record. The four possible alternatives are illustrated here:

```
                                %COMPILER
c1                              %CMD "c1"
run of HI-SLANG compiler        % compiler control file
c2                              %CMD AFTER "c2"
run of SIMULA compiler          ...
                                %ANALYZER
c3                              %CMD "c3"
run of analyzer                 % analyzer control file
c4                              %CMD AFTER "c4"
                                %END
```

This record is not necessarily supported by all HIT installations and may have no effect.

Examples: (BS2000)

```
%CMD    "TCHNG OFLOW=NO"
%CMD    "FSTAT"
%BIND   "link" TO ? "Please enter file name"
%CMD    AFTER "PRINT #HIT.LISTING, SPACE=E, LOCK=YES"
```

8.2.6. Defining File Name Defaults (%DEFAULT)

The standard link names of the HIT system (see table in Section 8.1.) can be left unbound. They either have a default binding to a file, which is true when the control file is a plain file, or they have a default binding to the mobase containing the control file. The default file name is generated by the HIT file name generator, which is "programmed" in the HIT STARTUP file by a %DEFAULT record and can be re-programmed by a %DEFAULT record in a control file created by the user.

control_record ::=
| **%DEFAULT** "pattern"

This *pattern* is not only applied to standard link names, but can also be employed for user-defined link names by writing

%BIND "link name" TO [EXTEND] DEFAULT

The file name generation *pattern* may be composed of the following elements:

<C> : For this element the name of the control file (which may also contain the source) is substituted.

<S> : Same as <C>, but the suffixes .CONTROL, .CTL or .HIT are stripped from the name of the control file, regardless whether they are written lower-case or upper-case.

<L> : The link name of the file which is to be named is substituted for this element. Special characters within the link name are replaced with 'Y'-characters. For UNIX systems, the link name is converted to lower-case.

<-z> : Any suffix starting with the character *z* is deleted from the file name generated so far. If no *z* is found the whole current construction is deleted.

<+> : Adds the suffix which has been deleted by the last <-z> element.

<1> : Deletes the whole current construction (sets pointer to 1).

</s/t/> : Adds either text *s* to the file name or, if the file contains print control characters (as e.g., tables, listings), adds text *t*.

<xd> : A digit *d* may be inserted preceding the closing bracket of any element *x*, expressing the maximum number of characters to be generated by the element. This element may not contain a blank.

others : Elements not included in such brackets are simply appended to the current construction.

Different patterns are reasonable for different operating systems:

Examples:

```
BS2000    :  %DEFAULT   "#<S>.<L>"
MVS       :  %DEFAULT   "<C><-.><L>.</DATA/TEXT/>"
VM/CMS    :  %DEFAULT   "HIT <L>"
UNIX      :  %DEFAULT   "<S><-/><1>t.<+>.<L3>"
```

The patterns drawn above generate the following file names for "LISTING" and "TRACE" in reaction to the respective control file names specified in the first line:

BS2000	:	MODEL.CONTROL #MODEL.LISTING #MODEL.TRACE
MVS	:	MODEL.DATA MODEL.LISTING.TEXT MODEL.TRACE.DATA
VM/CMS	:	MODEL CONTROL HIT LISTING HIT TRACE
UNIX	:	hit/model/vers1.ctl t.vers1.lis t.vers1.tra

The UNIX pattern contains nearly all existing elements. It operates in the following way (e.g., for hit/model/vers1.ctl) :

apply	result is	comment
<S>	hit/model/vers1	.ctl stripped off
<-/>	hit/model	vers1 stored in <+>
<1>		
t.	t.	
<+>	t.vers1	
.	t.vers1.	
<L3>	t.vers1.lis	first three letters of "LISTING" lower-case

The %OMA segment of the HIT STARTUP file contains further %DEFAULT records. These have slightly different meanings, explained in /Weis92b/.

8.2.7. Dialogue Support

To increase the portability of HIT, a simple dialogue component is included in the FAN system. HIT can therefore be comfortably used in any environment.

Dialogues may be specified within the control file by using

? ["query_text"]

as argument of a control file record.

When the FAN system interprets such an argument it prints the *query_text* on the terminal and waits for input. If no *query_text* is given the control record itself is used as query text. The input is registered on SYSOUT and substituted for the query text within the control file listing. Therefore the input may not be longer than the query text if more arguments follow in the same line.

Example:

```
%MOBASE     ?
%BIND       "SOURCE" TO ? "which SOURCE file ?"
%CMD        ? "You may enter an OS command now"
%PARM       = INDENT = ? "indent value ?", ? "please enter run option", XREF
```

There is an implicit activation of this dialogue component if I/O is to be made via an unbound user-defined link name. The FAN system automatically asks for the binding with

```
%BIND "link name" TO ?
```

The user's input must follow the *file_object* syntax (see Section 8.2.2.), optionally prefixed by EXTEND or READONLY. There is no query for standard link names (except "CONTROL") since they have a default binding to a file (see %DEFAULT and table of link names in Section 8.2.3.). There is also no query for the link names of all objects within the HIT standard mobase (e.g., "semaphor"), so they must not be bound. The same mechanism applies for objects found in the actual user mobase (see %MOBASE).

8.2.8. Comments

The control file may contain comments, starting with *%blank*. Moreover, empty lines can be used to structure control files. An arbitrary amount of comment may also follow directly upon the keywords %COMPILER, %ANALYZER, %COMMON and %END.

Examples:

```
% This is a comment
% CMD "Attention, this is a comment"

%COMPILER control
```

8.2.9. Defining Configurations

One of the purposes of the control file is to define all input files or mobase objects for the HI-SLANG compiler. Thus the control file serves to define configurations of models without modifying the source text of the model.

Depending on the version (e.g., via number suffix of object names) and representation (e.g., HISLANG, PREANA) for instance of a component type which is to be added to the model by %COPY "link name", i.e., depending on the "environment" defined in the control file, a different efficiency and/or performance will be reached when the analyzer is run.

It is advisable to use the control file as documentation for every experiment performed. Apart from providing configuration information, such documents informally describe the changes with respect to the previous experiment and the observations and conclusions from performing the experiment.

Example:

```
%COMMON  ----------------------------------For both COMPILER and ANALYZER  ----------------------
%MOBASE  office.lib
%BIND    "evaluation"    TO  (PREANA,     COMPONENT,     evaluation2)

%COMPILER  ---------------------------------- Configuration Segment  ------------------------------------------
%BIND    "office"        TO  (HISLANG,    MODEL,         office1)
%BIND    "handling"      TO  (HISLANG,    COMPONENT,     handling4)
%BIND    "experiment"    TO  (HISLANG,    EXPERIMENT,    expsep2)

%ANALYZER  ---------------------------------- Result Directing  ----------------------------------------------
% Usage of DEFAULT-bindings to the mobase containing this control file.

%END       ---------------------------------- Model Structure  ----------------------------------------------
%TITLE   Office Model
%COPY    "office"
%COPY    "handling"
%COPY    "evaluation"
%COPY    "experiment"
% The source consists of 4 modules with versions and representations as described above

%EOF       ---------------------------------- Experiment Description  ---------------------------------------

EXPERIMENT    :  exp5                AUTHOR: N. Weissenberg                Date: 25.2.88

BASED ON      :  exp3, exp4

CHANGES       :  The DISPATCH strategy of server link was changed to equal (100) (exp3) and
                 evaluation was aggregated (exp4).
-------------------------------------------------------------------------------------------------------------------------
OBSERVATION   :  The results are the same, except for server link.

CONCLUSION    :  Both steps could be performed.
```

8.3. Compiler Directives

The compiler directives described here may appear anywhere within a HI-SLANG source, whereas the control records defined above may only be contained in the control file. All keywords for compiler directives start with a percent symbol which has to be the first character different from a blank or a tab of the line.

The following compiler directives can be utilized: %[NO]SOURCE, %PAGE, %TITLE, %COPY, %SPEEDS, %EOF, and additionally, for conditional compilation: %IF, %ELSE, %FI, %[RE]SET. They may all be written either upper-case or lower-case.

Only %COPY and the directives for conditional compilation appear in the listing. The directives have the following meaning:

%NOSOURCE

The succeeding lines are not listed until a %SOURCE is encountered.

%SOURCE

The succeeding lines will be listed until a %NOSOURCE is encountered. %SOURCE has no effect if %PARM=NOSOURCE was specified in the control file.

%PAGE

The listing is continued on a new page with the title defined by the last %TITLE command. When the amount of lines defined by %PARM=LINES=n is reached, this command is performed implicitely.

%TITLE title_text

The listing is continued on a new page titled by *title_text* (or its leading 43 characters, see Appendix G.1.). The title appears on every succeeding page until it is overwritten by a new %TITLE command. The default *title_text* is the name of the control file used.

Example:

```
%TITLE  This is a title, but only the first 43 characters will appear.
```

%COPY "link_name"

By %COPY a further module is included in the source. The module, identified by its *link_name*, may have one of two representations, which the compiler can distinguish by the first character of their contents:

- HI-SLANG source codes are completely copied and listed.
- In the case of an aggregated component type only the HI-SLANG interface is copied and listed. The table of speeds following the %SPEEDS command is discarded by the HI-SLANG compiler, but will be interpreted by the analyzer.

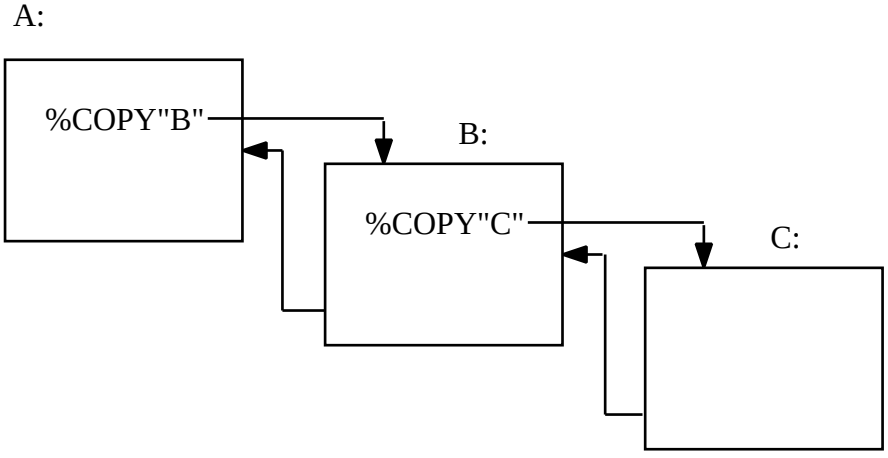
The *link_name* should be bound in the control file or else the dialogue component of FAN will ask for the binding. The link names of all objects of the HIT standard mobase (see Appendix F.) are bound automatically. Lower-case letters within link names are converted to upper-case.

Example:

```
%COMPILER
%MOBASE   mylib
%BIND    "COPY"   TO copyfile
%BIND    "cpu1"    TO ( )
%BIND    "cpu2"    TO ( )
%END

%COPY   "COPY"
...
%COPY   "Cpu1"
%COPY   "CPU2"
...
%COPY   "semaphor"
```

Nesting %COPY is permitted:

Example:



A short-cut avoiding link names and their bindings can be taken as follows:

%COPY FILE file_object
%INCLUDE file_object

Both have the same meaning. Here a file name or a mobase object can be specified directly, with the disadvantage of the source now being dependent on naming conventions of the operating system or modelling base. This dependency is usually restricted to the control file.

Examples:

```
%COPY FILE system.installation
%INCLUDE   (HISLANG,, cpu)
%INCLUDE   sysin
```

The second example relates to the last %MOBASE record in the control file.

%SPEEDS

This compiler directive is only used internally for aggregated component types to separate the HI-SLANG interface from the speeds table. For the compiler it is identical to %EOF.

%EOF

This compiler directive terminates the actual source. It is useful when entering a HI-SLANG source line by line while the compiler is running (e.g., by entering SYSIN as name of the control file).

Another application is to append arbitrary comments following %EOF at the end of a source (see Section 8.2.8.).

%IF [**NOT**] switch **THEN**
source1
[**%ELSE**
source2]
%FI

This combination of directives is used to specify conditionally compiled segments of the source. Either *source1* or *source2* is compiled, depending upon the setting (via %SET) of the *switch*. Both sources may consist of an arbitrary number of lines. %ELSE may be omitted if *source2* is empty. These commands cannot be nested (within one file).

%SET switch
%RESET switch

These directives serve to set or reset the *switch*, which can be used within an %IF directive. The *switch* may have an arbitrary name composed of letters, digits and underscores, the first character being a letter. The default value of a switch is RESET.

Example:

```
%RESET quick_variant
…
%IF quick_variant THEN
     spend (negexp (1/10));
%COPY "sequence"
%ELSE
     spend (negexp (1/100));
%FI
```

% comment

All lines starting with percent-blank are treated as comments.

Examples:

```
% This is a comment
%no comment; result in the warning: %no unknown.
```

8.4.Control File Example

This example demonstrates most control records and the way file objects are referred in HI-SLANG sources:

```
%COMMON ------------------------------------------------------------------------------
%MOBASE      mylib
%BIND        "CT2_AGG"      TO    (PREANA, COMPONENT, CT2, P)

%COMPILER ----------------------------------------------------------------------------
%PARM = INDENT=5, MAXERROR=10, XREF, COM
%BIND        "LISTING"      TO    ( ,  , example1_lst)
%BIND        "LISTING"      TO    EXTEND #HIT.LISTING
% listing doubled!
%BIND        "COPYEXP"      TO    READONLY example1.experiment.1
%BIND        "my_sched"     TO    (SIMULA, SCHEDULE, my_sched)
%CMD         "INF"
%CMD         AFTER "PRINT #HIT.LISTING, form=std3, space=e"

%ANALYZER ----------------------------------------------------------------------------
%BIND        "LISTING"      TO    EXTEND  #HIT.LISTING
%CMD         "FSTAT   *.INFILE"
%BIND        "IN1"          TO    ?  "Please enter name of data file"
%BIND        "IN2"          TO    input.lib (FILE,  , example1_dat, P)
%BIND        "OUT"          TO    example1.out
%BIND        "TABLE"        TO    ( ,  , example1_tab)
%BIND        "DUMP"         TO    example1.dump
%BIND        "TRACE"        TO    ( ,  , example1_trace)

%END ---------------------------------------------------------------------------------
% The source text follows, since no %BIND "SOURCE" is specified.

TYPE mt MODEL;

%COPY    "CT2_AGG"
%COPY    "nowaitsend"

    COMPONENT      c1 :  ct1   (LET dispatch     :=  equal (1);
                                LET schedule     :=  my_sched);
                   c2 :  ct2;
                   c3 :  nowaitsend;

    REFER st TO c1 EQUATING ...
    END REFER;
END TYPE mt;

%COPY "COPYEXP"
{contents:}
…
VARIABLE     fa :   ARRAY [1...2] OF INFILE;
                g :   OUTFILE;
…
OPEN    fa  [1],   "IN1" LENGTH 80;
OPEN    fa  [2],   "IN2" LENGTH 80;
OPEN    g,         "OUT" LENGTH 132;
…
```

9. Literature

/Beil85/ Beilner, H.:
Workload Characterization and Performance Modelling Tools,
in: Proc. of the International Workshop on Workload Characterization of Computer Systems, Pavia, Italy, 1985, North-Holland

/Beil88/ Beilner, H.:
Zur Methodik der modellgestützten Bewertung von Produktionssystemen - Grundlagen der Simulation mit HIT,
in: Simulation in der Fertigungstechnik, B. Schmidt (ed.), Springer Verlag, 1988

/Beil89/ Beilner, H.:
Structured Modelling - Heterogenous Modelling,
in: Proc. of the European Simulation Multiconference ESM '89, Rome, Italy, 1989

/BeMW88/ Beilner, H. / Mäter, J. / Weißenberg, N.:
Towards a Performance Evaluation Environment: News on HIT,
in: Proc. "Modelling Techniques and Tools for Computer Performance Evaluation", Palma de Mallorca, Spain, 1988

/BeSt87/ Beilner, H. / Stewing, F.-J.:
Concepts and Techniques of the Performance Modelling Tool, HIT,
in: Proc. of the European Simulation Multiconference ESM '87, Wien, Austria, 1987

/BuPS88/ Büser, M. / Pape, D. / Stewing, F.-J.:
Simulation of Integrated Information and Material Flow in Logistics Systems: An Application of the Modelling Tool, HIT,
in: Proc. of the 2nd European Simulation Multiconference, Nizza, France, 1988

/BuSt90/ Büttner, M./Strell, V.:
Implementationsbeschreibung des HIT-Simulators (SIMUL),
Universität Dortmund, Informatik IV, 1990

/CIM91/ Johansen, S.; Krogdahl, S.; Mjøs, T.:
User´s and Installation Guide - Portable Simula System based on C,
University of Oslo, Department of Informatics, 1991

/Deik89/ Deike-Glindemann, H. (ed.):
SIQUEUE-PET Benutzerhandbuch,
Universität Dortmund, Informatik IV, 1989

/Hoof92/ Hoof, A.:
HIT Installation Guide for VM/CMS,
Universität Dortmund, Informatik IV, 1992

/IBMSim86/ SIMULA Programmer´s Guide for Siemens System BS2000
Informatikrechner-Betriebsgruppe, Universität Dortmund, 1986
(revision of previous edition: "SIMULA Programmer´s Guide for IBM System 360/370")

/KLMN88/ Knaup, W. / Litzba, D. / Müller-Clostermann, B. / Noack, F. / Sczittnick, M. / Stahl,H.:
Approximative Lösungsverfahren für nicht-separable Warteschlangennetze, Universität Dortmund, Informatik IV, 1988

/Knau88/ Knaup, W.:
Approximative Verfahren zur Analyse von Warteschlangennetzen mit Prioritätsstationen,
Universität Dortmund, Informatik IV, 1988

/KnND89/ Knaup, W. / Noack, F. / Deike-Glindemann, H.:
Performance-Bounds-Verfahren für separable Netze,
Universität Dortmund, Informatik IV, 1989

/LeWe92/ Lengewitz, P. / Weißenberg, N.:
HIT User's Guide for UNIX Systems,
Universität Dortmund, Informatik IV, 1992

/LiSS89/ Litzba, D. / Sczittnick, M. / Stewing, F.-J.:
Eine Online-Update-Auswertungskomponente auf Basis autoregressiver Modelle,
Universität Dortmund, Informatik IV, 1989

/LiSS89b/ Litzba, D. / Sczittnick, 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, Sept. 5-8, 1989

/Litz85/ Litzba, D.:
Auswertung von Simulationsdaten mittels autoregressiver Modelle,
Grüne Reihe, Bericht Nr. 203, Universität Dortmund, Informatik

/LundSim87/ Holm, P. / Taube, M.:
SIMULA User´s Guide for UNIX,
Lund Software House AB, Lund, 1987

/MiKe86/ Mitra, D. / McKenna, J :
Asymptotic Expansions for Closed Markovian Networks with State-Dependent Service Rates,
Journal of the ACM, Vol.33, No.3, 1986, pp.568-592

/MuNS85/ Müller-Clostermann, B. / Noack, F. / Sczittnick, M.:
Analytische Lösungsverfahren für Rechensystemmodelle,
Universität Dortmund, Informatik IV, 1985

/MuRo87/ Müller-Clostermann, B. / Rosentreter, G.:
Synchronized Queueing Networks: Concepts, Examples and Evaluation Techniques,
in: Proc. 4. GI/NTG-Fachtagung "Messung, Modellierung und Bewertung von Rechensystem", Erlangen, 1987, Springer Verlag

/MuWe87/ Müller-Clostermann, B. / Weißenberg, N.:
Using SIMULA for the Implementation of the Hierarchical Modelling and Performance Evaluation Tool, HIT,
in: Proc. 14th SIMULA User's Conference, Stockholm, Sweden, 1987

/PaSt89/ Pape, D.F. / Stewing, F.-J.:
Strategic Planning of Logistic Systems with Simulation,
in: Proc. of the European Simulation Multiconference ESM '89, Rome, Italy, 1989

/PCSim89/ SIMULA Programmers Reference Manual under the operating systems MS-DOS, OS-2, XENIX386, UNIX386,
Simula a.s., Oslo, 1989

/Pool87/ Pooley, R.:
An Introduction to Programming in SIMULA,
Blackwell Scientific Publications, 1987

/SaMN84/ Sauer, C.H. / McNair, E.:
The Evolution of the Research Queueing Package RESQ,
in: Proc. "Modelling Techniques and Tools for Performance Analysis", Paris, 1984, D.Potier (ed.), North Holland

/Sczi93/ Sczittnick, M. (ed.):
HITGRAPHIC User's Guide,
Universität Dortmund, Informatik IV, 1993

/SIMULA87/ Databehandling - Programspråk - SIMULA
(Dataprocessing - Programming language - SIMULA)
Swedish Standard SS 63 61 14
SIS Standardiseringskommissionen; Sverige, 1987
(ISBN 91-7162-234-9)

/Stew89/ Stewing, F.-J. (ed):
HI-SLANG Reference Manual (in German),
Universität Dortmund, Informatik IV, 1989
(predecessor of this document, no more available)

/TPHSim87/ Philippot, G.P.:
TPH Simula Reference Manual, Unix version,
TPH Data A.S., Oslo, 1987

/WaHo92/ Wadulla, G. / Hoof, A.:
HIT Installation Guide for BS2000,
Universität Dortmund, Informatik IV, 1992

/Weis92a/ Weißenberg, N. (ed):
HIT and HI-SLANG: An Introduction,
Universität Dortmund, Informatik IV, 1992

/Weis92b/ Weißenberg, N.:
HIT-OMA User's Guide,
Universität Dortmund, Informatik IV, 1992

/Weis92c/ Weißenberg, N.:
HIT Installation Guide for UNIX Systems,
Universität Dortmund, Informatik IV, 1992

/Weis92d/ Weißenberg, N. (ed):
Implementationsbeschreibungen des HIT Systems,
Internal Reports, Universität Dortmund, Informatik IV, 1992

/Wolf86/ Wolf, H.:
Outil de Modelisation et d'Evaluation HIT
in: Proc. of the Workshop on Computer Performance Evaluation, Sophia Antipolis, France, 1986

A. HIT Syntax Rules

Here the syntax of HI-SLANG is given in a BNF-like notation. The syntax rules are ordered logically, while the syntax diagrams in Appendix B. are in alphabetic order. The following conventions apply:

sem_comment-nonterminal	Nonterminal *nonterminal,* prefixed by a semantical comment *sem_comment*
TERMINAL	HI-SLANG terminal (may be written lowcase)
[unit]	Optional *unit*
unit [separator ...]	Repetition of *unit*s, separated by *separator*
{ a b c } [separator ...]	The meta-parenthesed group *a b c* may be repeated
⟦ ⟧ and { }	denote the corresponding HI-SLANG terminals, not the meta characters as defined above.

Example:

The rule: identifier ::=
{name [⟦ simple_real_expression [, …] ⟧] [actual_parameters] } [. ...]

can, e.g., be derived to: name1 [1, 2] (3) . name2 (4) . name3 . name4 [5]

A.1. HI-SLANG Syntax

HI-SLANG Programs

```
hit_unit ::=
    [declaration [ ...]] experiment_block


experiment_block ::=
    EXPERIMENT experiment-name METHOD method ;
        [declaration [ ...]]
    BEGIN
        [sequence_of_statements]
        [plot_statement [ ...]]
    END EXPERIMENT [experiment-name] ;


method ::=
    ANALYTICAL simple_text_expression
|   SIMULATIVE


declaration ::=
    common_declaration
|   modelling_declaration


sequence_of_statements ::=
    statement [ ...]
```

Declarations

modelling_declaration ::=
    process_declaration
| component_declaration
| enclose_declaration
| stream_declaration

process_declaration ::=
    PROCESS
    { process-name [**,** ...] : [**ARRAY** ⟦ array_bounds ⟧ **OF**]
        process_name_or_object_declaration **;** } [...]

process_name_or_object_declaration ::=
    service-name [actual_parameters]
| **NAME FOR** service-name

component_declaration ::=
    COMPONENT
    { component-name [**,** ...] **:** [**ARRAY** ⟦ array_bounds ⟧ **OF**]
        componenttype-name [actual_parameters] **;** } [...]
| **COMPONENT**
        component-name [**,** ...]
        [actual_parameters] **;**
        [provide_declaration_part]
        [collect_block]
        [control_declaration_part]
        [declaration [...]]
        [refer_part]
        [**BEGIN** sequence_of_statements]
        END COMPONENT [component-name] **;**

enclose_declaration ::=
    ENCLOSE
    {component-name [**,** ...] [**:** [**ARRAY OF**] componenttype-name] **;** } [...]

stream_declaration ::=
    STREAM
    {stream-name [**,** ...] **:** stream_type **;** } [...]

stream_type ::=
    COUNT
| **EVENT**
| **STATE**

common_declaration ::=
    record_declaration
| variable_or_constant_declaration
| procedure_declaration
| type_declaration

```
record_declaration ::=
    RECORD
    {record-name [, ...] :
    [ARRAY [ array_bounds [, ...] ] OF]
    recordtype-name [actual_parameters] ; } [ ...]


variable_or_constant_declaration ::=
    variable_or_constant   simple_object_declaration [ ...]
|   variable_or_constant   array_object_declaration   [ ...]


simple_object_declaration ::=
    object-name [, ...] : simple_type [DEFAULT expression] ;


array_object_declaration ::=
    object-name [, ...] :
    ARRAY [ array_bounds [, ...] ] OF simple_type  [DEFAULT expression_or_aggregate] ;


variable_or_constant ::=
    VARIABLE
|   CONSTANT


array_bounds ::=
    simple_real_expression ..  simple_real_expression


procedure_declaration ::=
    PROCEDURE procedure-name [formal_parameters]  [RESULT simple_type [, ...]] ;
        [use_declaration_part]
        [common_declaration [ ...]]
    [BEGIN sequence_of_statements]
    END PROCEDURE [procedure-name] ;


type_declaration ::=
    service_declaration
|   componenttype_declaration
|   modeltype_declaration
|   recordtype_declaration


service_declaration ::=
        TYPE service-name SERVICE [formal_parameters]  [RESULT simple_type [, ...]] ;
            [use_declaration_part]
            [declaration [ ...]]
        [BEGIN sequence_of_statements]
        END TYPE [service-name] ;

    |   SERVICE service-name [formal_parameters]  [RESULT simple_type [, ...]] ;
            [use_declaration_part]
            [declaration [ ...]]
        [BEGIN sequence_of_statements]
        END SERVICE [service-name] ;
```

componenttype_declaration ::=
    TYPE componenttype-name **COMPONENT** [formal_parameters] **;**
        [provide_declaration_part]
        [collect_block]
        [control_declaration_part]
        [declaration [...]]
        [refer_part]
    [**BEGIN** sequence_of_statements]
    END TYPE [componenttype-name] **;**

modeltype_declaration ::=
    TYPE modeltype-name **MODEL** [formal_parameters] **;**
        [collect_block]
        [declaration [...]]
        [refer_part]
    [**BEGIN** sequence_of_statements]
    END TYPE [modeltype-name] **;**

recordtype_declaration ::=
    TYPE recordtype-name **RECORD** [formal_parameters] **;**
        [common_declaration [...]]
    [**BEGIN** sequence_of_statements]
    END TYPE [recordtype-name] **;**

use_declaration_part ::=
    USE
        use_declaration [...]
    END USE ;

use_declaration ::=
    procedure_or_service [**ARRAY**]
      {procedure_or_service-name [formal_parameters] [**RESULT** simple_type [**,** ...]] **;** } [...]

procedure_or_service ::=
    PROCEDURE
| **SERVICE**

provide_declaration_part ::=
    PROVIDE
        provide_declaration [...]
    END PROVIDE ;

provide_declaration ::=
    procedure_or_service
      {procedure_or_service-name [formal_parameters] [**RESULT** simple_type [**,** ...]] **;** } [...]

refer_part ::=
    REFER procedure_or_service-name [**,** ...] **TO** component-name [**,** ...] **EQUATING**
        {use-identifier **WITH** provide-identifier [**OF** service-name] **;** } [...]
    END REFER ;

control_declaration_part ::=
CONTROL
control_procedure_declaration [...]
END CONTROL ;

control_procedure_declaration ::=
PROCEDURE control_procedure-name **;**
[common_declaration [...]]
[**BEGIN** sequence_of_statements]
END PROCEDURE [control_procedure-name] **;**

collect_block ::=
COLLECT
{ [service-name .] stream-name [**AS** external_stream-name] **;** } [...]
END COLLECT ;

Expressions

expression_or_aggregate ::=
    expression
| aggregate

aggregate ::=
    ⟦ expression_or_aggregate [**,** …] ⟧

boolean_expression ::=
    TRUE
| **FALSE**
| expression

expression ::=
    disjunction [**EQV** disjunction]

disjunction ::=
    conjunction [or_else ...]

or_else ::=
    OR [**ELSE**]

conjunction ::=
    { [**NOT**] relation} [and_then ...]

and_then ::=
    AND [**THEN**]

relation ::=
    simple_expression [relational_operator simple_expression]

relational_operator ::=
    = | < | > | <= | >= | <> | #

simple_expression ::=
    NONE
| character
| simple_text_expression
| simple_real_expression
| identifier
| **(** expression **)**

simple_text_expression ::=
    simple_text [& ...]

simple_text ::=
    string
| identifier
| **(** simple_text_expression **)**

simple_real_expression ::=
    [unary_operator] term [adding_operator ...]

adding_operator ::=
    + | **-**

term ::=
    factor [multiplying_operator ...]

multiplying_operator ::=
    ***** | **/** | **//** | **MOD**

factor ::=
    primary [****** ...]

primary ::=
    identifier [**OF** identifier]
| number
| **(** simple_real_expression **)**

identifier ::=
    {name [⟦ simple_real_expression [**,** ...] ⟧] [actual_parameters] } [**.** ...]

simple_type ::=
    INTEGER
| **REAL**
| **BOOLEAN**
| **CHARACTER**
| **TEXT**
| **INFILE**
| **OUTFILE**
| **POINTER FOR** recordtype-name

Statements

statement ::=
    simple_statement
| compound_statement
| aggregate_statement
| evaluate_statement

simple_statement ::=
    empty_statement
| assignment_statement
| new_statement
| result_statement
| io_statement
| update_statement
| procedure_or_service_call
| create_or_submit_statement
| control_procedure_statement

compound_statement ::=
    block_statement
| with_statement
| concurrent_statement
| conditional_statement
| chain_statement
| loop_statement
| inspect_statement

empty_statement ::= **;**

assignment_statement ::=
    common_assignment
| result_assignment

common_assignment ::=
    identifier [**,** ...] := expression_or_aggregate **;**

result_assignment ::=
        identifier **:=** procedure_or_service_call
| **(** identifier [**,** …] **)** **:=** procedure_or_service_call

result_statement ::=
    RESULT expression [**,** ...] **;**

update_statement ::=
    UPDATE stream-name **BY** simple_real_expression **;**

procedure_or_service_call ::=
    procedure_or_service-identifier **;**

block_statement ::=
BLOCK
common_declaration [...]
BEGIN
sequence_of_statements
END BLOCK ;

with_statement ::=
WITH record_or_pointer-identifier
DO
sequence_of_statements
END WITH ;

concurrent_statement ::=
CONCURRENT
sequence_of_statements
{**TO**
sequence_of_statements } [...]
END CONCURRENT ;

aggregate_statement ::=
AGGREGATE componenttype-name [**OUTPUT** simple_text_expression] ;
create_or_submit_statement [...]
END AGGREGATE ;

evaluate_statement ::=
EVALUATE MODEL model-name **:** modeltype-name [actual_parameters] **;**
evaluate_declaration [...]
BEGIN
measure_statement [...]
[control_statement]
END EVALUATE ;

new_statement ::=
NEW recordtype-name [actual_parameters] **POINTER** pointer-identifier [**,** ...] **;**

io_statement ::=
open_or_close_statement
| read_statement
| write_statement

open_or_close_statement ::=
OPEN file-identifier **,** simple_text_expression **LENGTH** simple_real_expression **;**
| **CLOSE** file-identifier **;**

read_statement ::=
READ [**TEXT** file-identifier**,**] input_list **;**
| **READ** [**FILE** text-identifier**,**] input_list **;**
| **READLN** [input_list]**;**
| **READLN FILE** file-identifier [**,** input_list] **;**

input_list ::=
{identifier [**::** simple_real_expression] } [**,** ...]

```
write_statement ::=
     WRITE      [TEXT file-identifier, ] output_list ;
|    WRITE      [FILE text-identifier, ] output_list ;
|    WRITELN    [output_list] ;
|    WRITELN    FILE file-identifier [, output_list] ;


output_list ::=
    {expression [:: simple_real_expression [:: simple_real_expression] ] } [, ...]


create_or_submit_statement ::=
    CREATE      simple_real_expression  PROCESS service-name [actual_parameters]
        [ [LIMIT  simple_real_expression] timing_condition] ;
|   SUBMIT      service-name [actual_parameters]
        NAME    process-identifier [timing_condition] ;


timing_condition ::=
    time_specification  simple_real_expression


time_specification ::=
    AT
|   AFTER
|   EVERY


conditional_statement ::=
    if_statement
|   case_statement
|   branch_statement


if_statement ::=
    IF boolean_expression
    THEN sequence_of_statements
    [ ELSE sequence_of_statements ]
    END IF ;


case_statement ::=
     CASE    simple_expression
    { WHEN simple_expression [, ...]    : sequence_of_statements } [ ...]
    [ ELSE                              : sequence_of_statements ]
    END CASE ;


branch_statement ::=
    BRANCH
    { PROB  simple_real_expression : sequence_of_statements } [ ...]
    [ ELSE                         : sequence_of_statements ]
    END BRANCH ;


prob_part ::=
    { PROB simple_real_expression  : node-identifier ; } [ ...]
    [ ELSE                         : node-identifier ; ]
```

```
chain_statement ::= ...
    open_chain_statement
|   closed_chain_statement


open_chain_statement ::=
    OPEN_CHAIN [ arrival-prob_part ]
        qnode [ ...]
    END OPEN_CHAIN ;


closed_chain_statement ::=
    CLOSED_CHAIN
        qnode [ ...]
    END CLOSED_CHAIN ;


qnode ::=
    QNODE qnode-identifier [prob_part]


loop_statement ::=
    infinite_loop
|   while_loop
|   until_loop
|   for_loop
|   times_loop


infinite_loop ::=
    basic_loop ;


basic_loop ::=
    LOOP
        sequence_of_statements
    END LOOP


while_loop ::=
    WHILE boolean_expression basic_loop ;


until_loop ::=
    basic_loop UNTIL boolean_expression ;


for_loop ::=
    FOR variable-identifier := loop_value_list basic_loop ;


loop_value_list ::=
    simple_real_expression STEP simple_real_expression UNTIL simple_real_expression
|   expression [, ...]


times_loop ::=
    AVERAGE simple_real_expression TIMES basic_loop ;
```

```
inspect_statement ::=
    INSPECT area [WHILE boolean_expression]
    LOOP [REVERSE]
        when_or_sequence
    END LOOP ;


control_procedure_statement ::=
    SELECT ;
|   SETSPEED simple_real_expression ;
|   TIMESLICE simple_real_expression ;


when_or_sequence ::=
    sequence_of_statements
|   {WHEN service-identifier      : sequence_of_statements } [ ...]
    [ELSE                         : sequence_of_statements]


area ::=
    ANNOUNCE_QUEUE
|   ENTRY_AREA
|   SERVICE_AREA
|   EXIT_AREA
```

Evaluation Declarations

```
evaluate_declaration ::=
    evaluationobject_declaration
|   hierarchy_declaration


evaluationobject_declaration ::=
    EVALUATIONOBJECT
    { {evaluationobject-name VIA [area OF] component-identifier } [, ...]
        [DEFAULT estimator_part] ; } [ ...]


hierarchy_declaration ::=
    HIERARCHY
    {hierarchy-name [, ...] default_or_merge ; } [ ...]


default_or_merge ::=
    DEFAULT hierarchy_part [, ...]
|   MERGE   hierarchy-name [, ...]


hierarchy_part ::=
    hierarchy-name
|   ( component-identifier [, service-name [, use-name] ] )
```

Evaluation Statements

measure_statement ::=
    MEASURE {stream [**DEGREE** simple_real_expression] } [**,** ...]
    AT evaluationobject-name
    [**DUE TO** hierarchy-name [**,** ...]]
    [**ABSCISSA** simple_real_expression]
    [estimator_part]**;**

stream ::=
    THROUGHPUT
| **TURNAROUNDTIME**
| **POPULATION**
| **OCCUPATION**
| **UTILIZATION**
| **SCHEDULE_RATE**
| **PREEMPT_RATE**
| stream-identifier

estimator_part ::=
    [**ESTIMATOR** estimator[**,** ...]]
    [**OUTPUT** output_link [**,** ...]]
    [**START** start_or_stop_condition]
    [**STOP** start_or_stop_condition
      | **GLOBALSTOP** stop_expression]

estimator ::=
    MEAN
| **BOUNDS**
| **STANDARDDEVIATION**
| **CONFIDENCE LEVEL** simple_real_expression
| **FREQUENCY INTERVAL** ⟦ array_bounds [**,** ...] ⟧

output_link ::=
    TABLE simple_text_expression
| **DUMPFILE** simple_text_expression

control_statement ::=
    CONTROL [**TRACEALL**]
    { [**AT** evaluationobject-name]
      [**STOP** start_or_stop_condition]
      [**TRACE**] } [...] **;**

start_or_stop_condition ::=
    basic_condition [and_or ...]

```
basic_condition ::=
    CPUTIME          simple_real_expression
|   MODELTIME        simple_real_expression
|   ACCURACY         simple_real_expression
|   EVENTS           simple_real_expression
        [DUE TO      hierarchy-name]
|   CONFIDENCE LEVEL simple_real_expression
        WIDTH        simple_real_expression
        MEASURE      stream
        [DEGREE      simple_real_expression]
        [DUE TO      hierarchy-name]

stop_expression ::=
    WIDTH            simple_real_expression
|   UPDATES          simple_real_expression
|   WIDTH            simple_real_expression  and_or
    UPDATES          simple_real_expression
|   UPDATES          simple_real_expression  and_or
    WIDTH            simple_real_expression


and_or ::=
    AND
|   OR
```

Parameters

```
formal_parameters ::=
    ( {[mode] parameter_declaration } [; ...] )


parameter_declaration ::=
    [VARIABLE]  parameter-name [, ...]  :  [ARRAY OF]   simple_type
        [DEFAULT  expression_or_aggregate]
|   RECORD      parameter-name [, ...]  :  [ARRAY OF]   recordtype-name


mode ::=
    VALUE
|   NAME
|   REFERENCE


actual_parameters ::=
    ( { [ [LET parameter-name := ] expression_or_aggregate] } [, ...] )
```

Representation of Results

plot_statement ::=
    graph_statement
| histogram_statement

graph_statement ::=
    GRAPH [inscription]
    {**PLOT** simple_text_expression
        plot_specification_graph
        INPUT simple_text_expression [**,** ...] } [...] **;**

histogram_statement ::=
    HISTOGRAM [inscription]
    PLOT plot_specification_histo
    INPUT simple_text_expression **;**

plot_specification_graph ::=
    MEASURE simple_text_expression
    ESTIMATOR simple_text_expression
    EVALUATIONOBJECT simple_text_expression
    HIERARCHY simple_text_expression

plot_specification_histo ::=
    MEASURE simple_text_expression
    EVALUATIONOBJECT simple_text_expression
    HIERARCHY simple_text_expression

inscription ::=
    [simple_text_expression]
    ABSCISSA simple_text_expression
    ORDINATE simple_text_expression
    [**OUTPUT** simple_text_expression]

A.2. Token Syntax

name ::=
    letter [letter_or_digit_or_underscore [...]]

letter_or_digit_or_underscore ::=
    letter
| digit
| _

number ::=
    digit [...] [**.** digit [...]] [**E** [unary_operator] digit [...]]

unary_operator ::=
    + | **-**

character ::=
    'ascii_or_ebcdic_character'

ascii_or_ebcdic_character ::=
    <one of the ASCII- or EBCDIC-Characters>

string ::=
    "[ascii_or_ebcdic_character [...]]"

comment ::=
    { [ascii_or_ebcdic_character [...]] }
| **%** blank [ascii_or_ebcdic_character [...]]

letter ::=
    A | **B** | ... | **Z** | **a** | **b** | ... | **z**

digit ::=
    0 | **1** |**2** | ... | **9** |

special_character ::=
    " | **#** | ... | **{** | **}** | **_** | blank

blank ::=

A.3. Compiler Directives

The following directives may be used anywhere within HI-SLANG sources (but not in control file). The %-character must appear in the first column, immediately followed by the keyword.

```
compiler_directive ::=
    %COPY link_name
|   %COPY FILE file_object
|   %INCLUDE   file_object
|   %[NO]SOURCE
|   %PAGE
|   %TITLE any_comment
|   %IF [NOT] condition-name THEN
        <parts of HI-SLANG source>
    [%ELSE
        <parts of HI-SLANG source>]
    %FI
|   %[RE]SET condition-name
|   %EOF
|   %SPEEDS
|   % blank any_comment
```

A.4. Control File Syntax

This syntax is interpreted by the HIT-FAN system (File Access Network). All the terminals starting with % must appear on a new line.

```
hi_slang_source ::=
    [control_file] hit_unit

control_file ::=
    { control_section
      control_record
        [ ...] } [ ...]
    %END

control_section ::=
    %COMMON
|   %COMPILER
|   %ANALYZER

control_record ::=
    %PARM = parameter [, ...]
|   %BIND link_name TO [io_mode] file_object
|   %MOBASE mobase_name
|   %CMD [AFTER] operating_system_command
|   %DEFAULT string
|   % blank any_comment
```

parameter ::=
    INDENT = ascii_or_ebcdic_character number
| **MAXERROR** = number
| **LINES** = number
| **FREQUENCYFORMAT** = number
| **TRACEFORMAT** = number
| **CHECK** | **COM** | **PRECOM**
| [**NO**]**SOURCE**
| [**NO**]**RESWD**
| [**NO**]**XREF**
| [**NO**]**DEBUG**
| [**NO**]**WARN**
| [**NO**]**WARNACCESS**
| [**NO**]**RELATIVE**
| [**NO**]**EXTERN**
| [**NO**]**UPDATES**
| [**NO**]**MINMAX**
| [**NO**]**PRINTDS**
| [**NO**]**SOLVERINFO**
| [**NO**]**CIM**

io_mode ::=
    EXTEND | **READONLY**

link_name ::=
    string

file_name ::=
    <operating system dependent syntax>

mobase_name ::=
    file_name

operating_system_command ::=
    <operating system dependent syntax>

any_comment ::=
    ascii_or_ebcdic_character [...]

file_object ::=
    file_name
| [mobase_name] **(** [module] [**,** [type] [**,** [member-name] [**,** [protection]]]] **)**
| **SYSIN**
| **SYSOUT**
| **DEFAULT**
| link_name

module ::=
    CONTROL | **DATA** | **HISLANG** | **PRECOM** | **PREANA** | **SIMULA**

type ::=
    ACCEPT | **CONTROL** | **COMPONENT** | **DISPATCH** | **DUMPFILE**
| **EXPERIMENT** | **FILE** | **LISTING** | **MATRIX** | **MODEL** | **OFFER**
| **OTHERS** | **PROCEDURE** | **SCHEDULE** | **SERVICE** | **STATES**
| **TABLE** | **TRACE**

protection ::=
    P | **U**

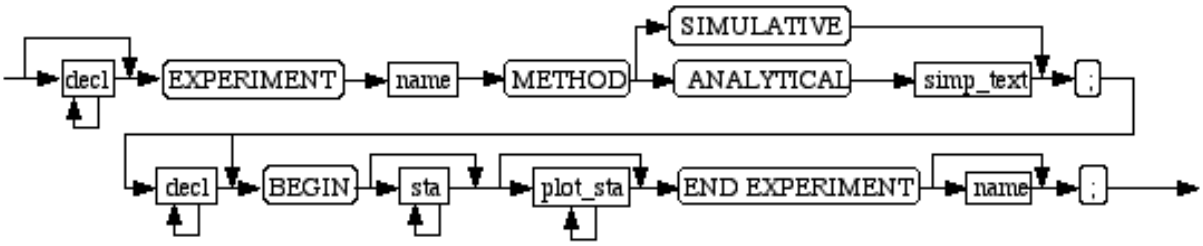
A.5. Nonterminal-Index

B. HIT Syntax Diagrams

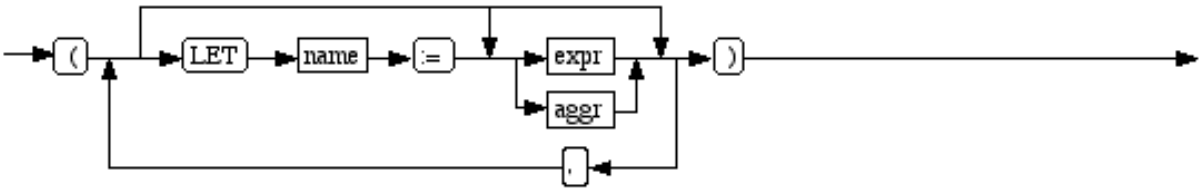
Due to lack of space most non-terminals are abbreviated compared to the rest of the manual. They are ordered alphabetically, with the exception of *hit_unit*, the start symbol.

B.1. HI-SLANG Syntax

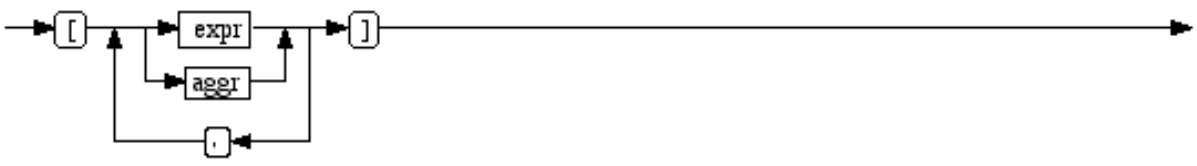
hit_unit



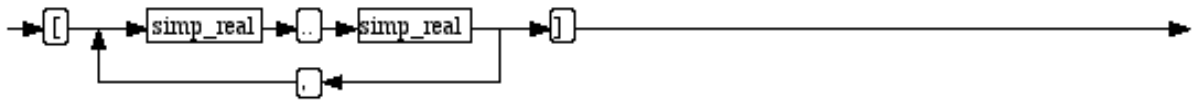
actual_par



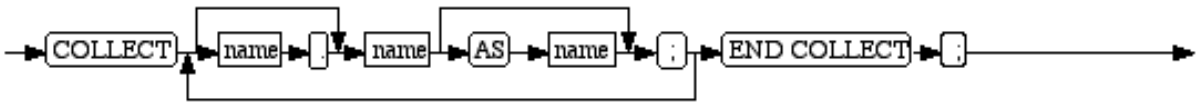
aggr



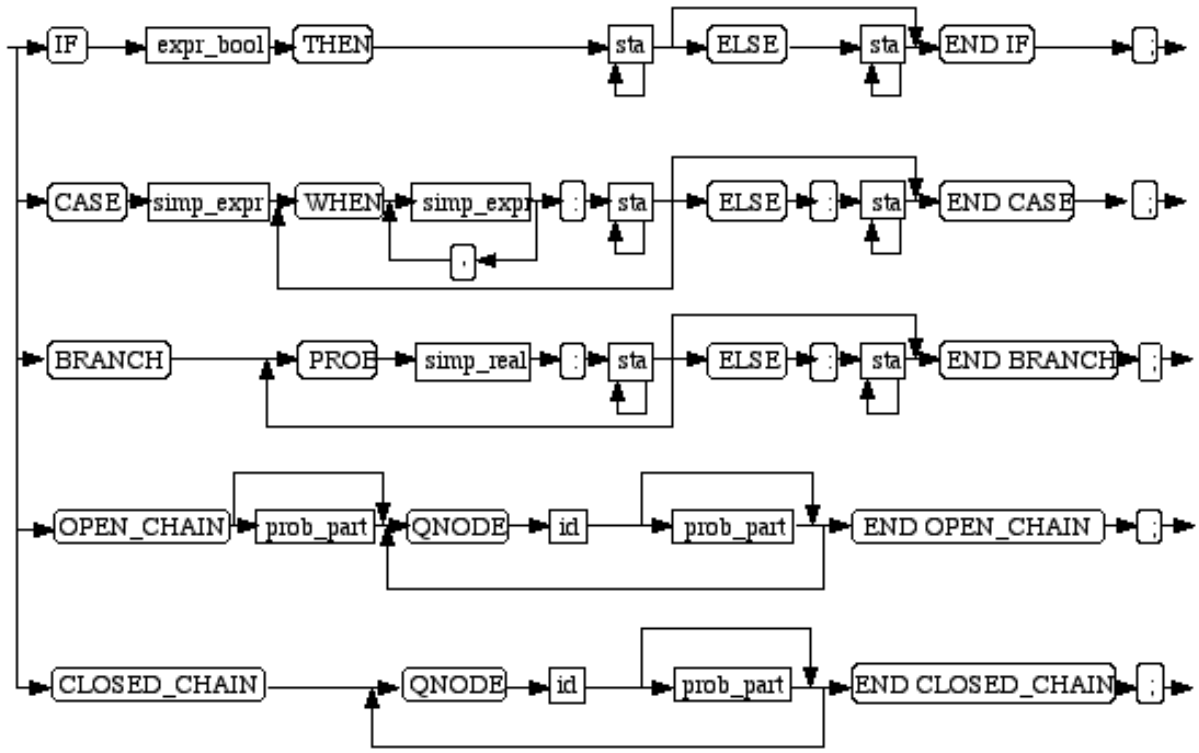
bounds



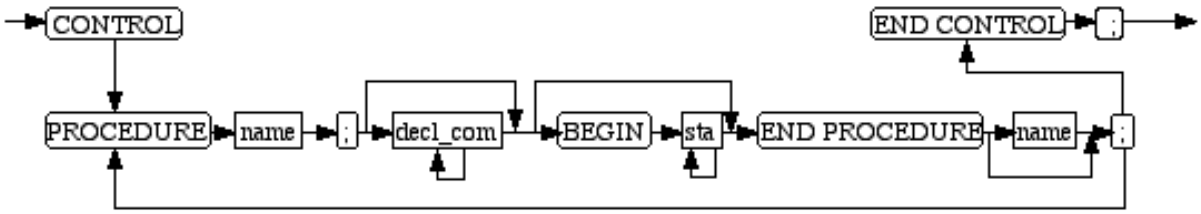
collect



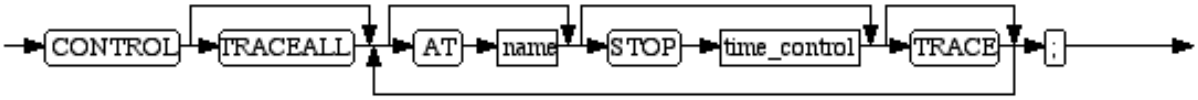
cond_sta



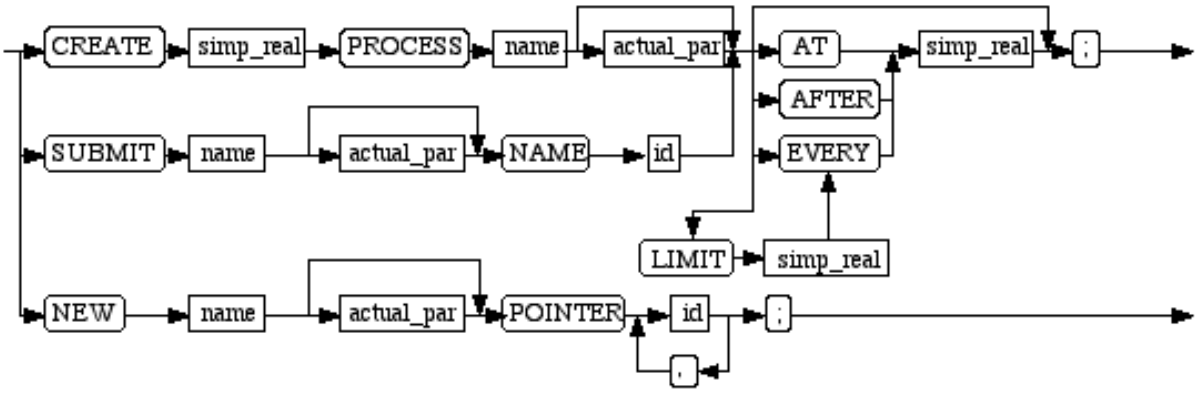
control_procs



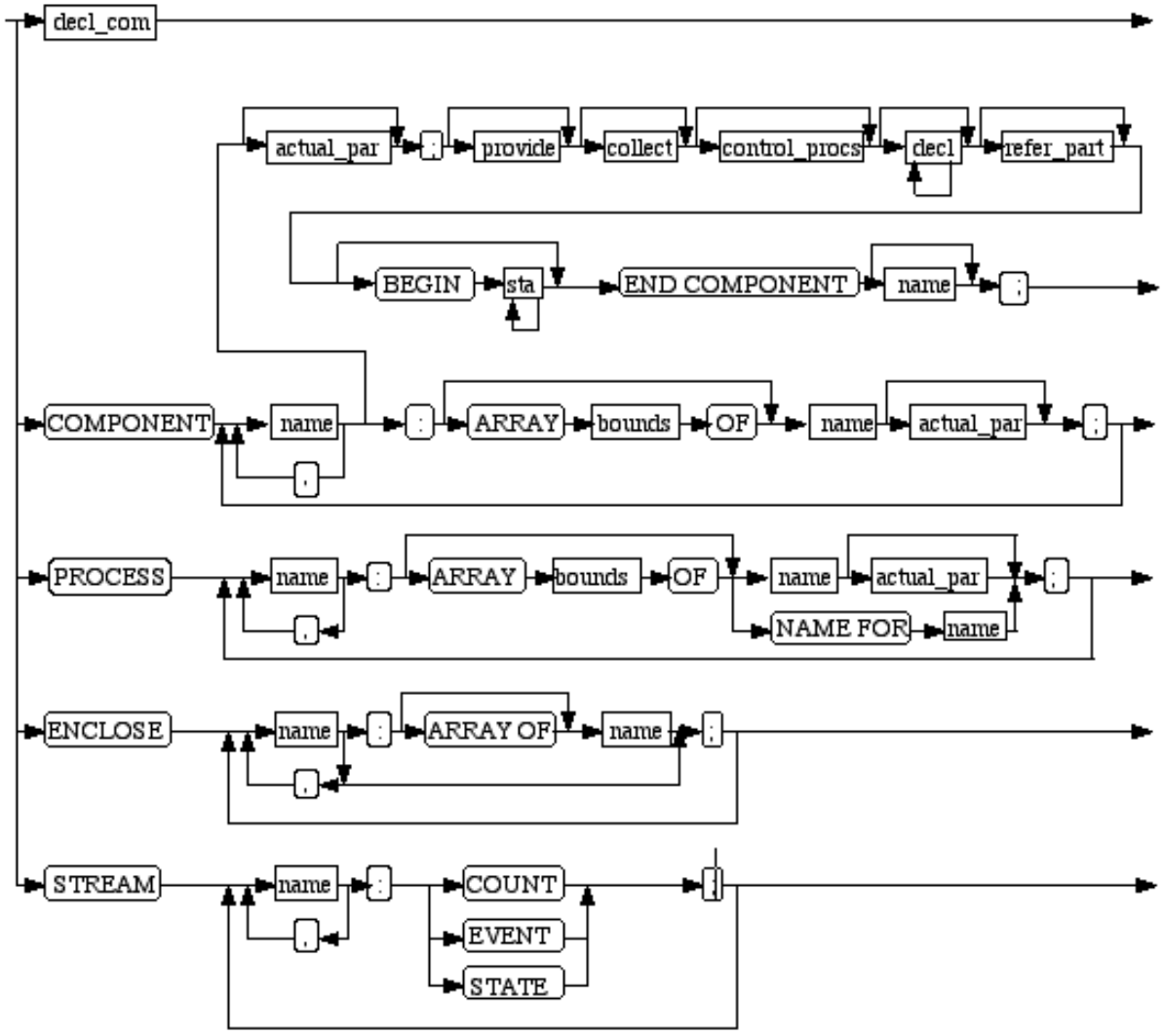
control_sta



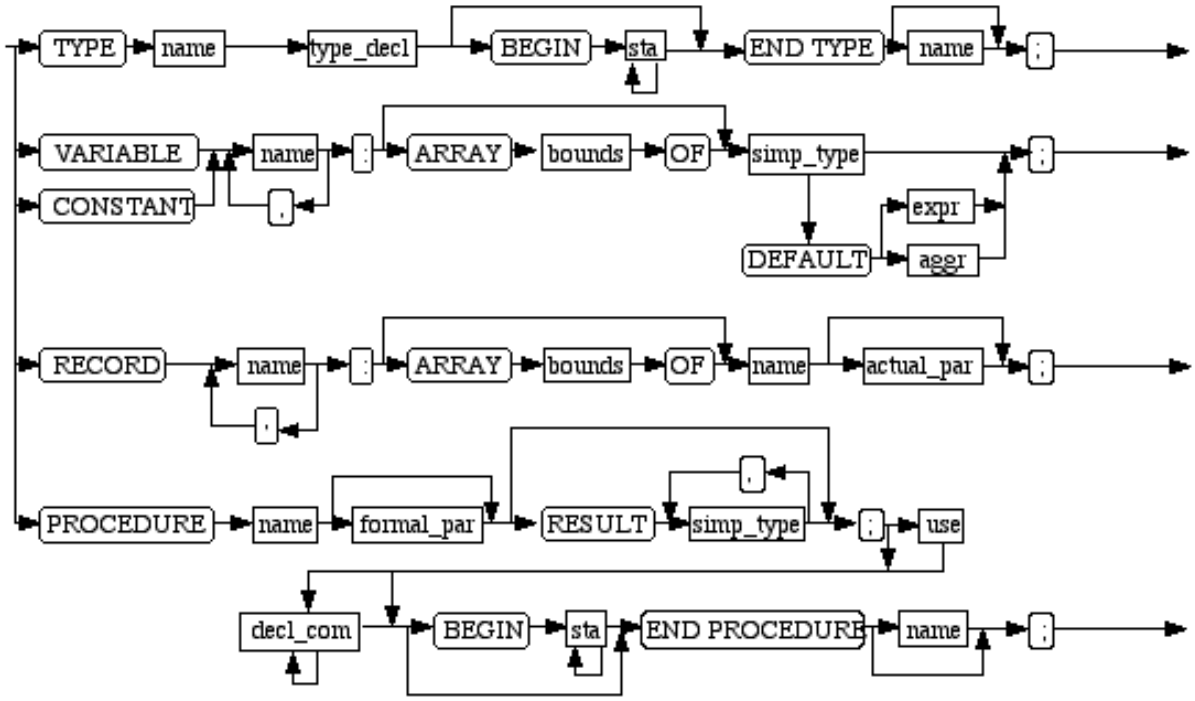
create_sta



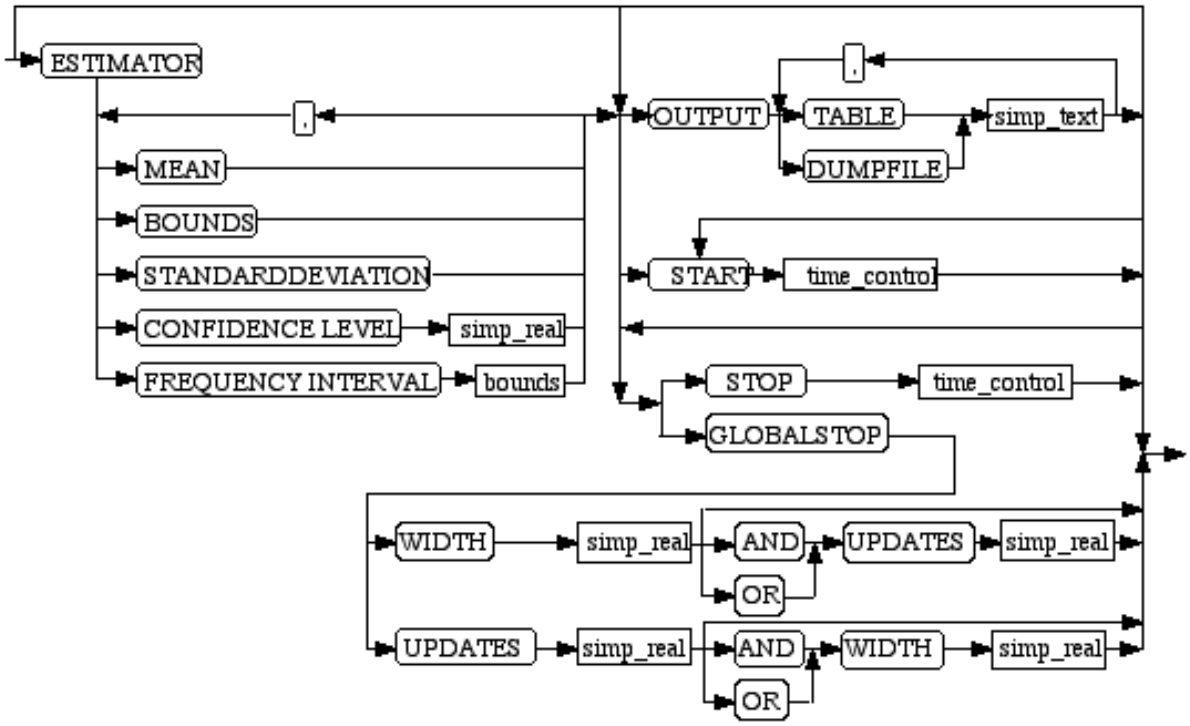
decl


decl_com
actual_par
;
provide
collect
control_procs
decl
refer_part
BEGIN
sta
END COMPONENT
name
;
COMPONENT
name
,
:
ARRAY
bounds
OF
name
actual_par
;
PROCESS
name
,
:
ARRAY
bounds
OF
name
actual_par
NAME FOR
name
;
ENCLOSE
name
,
:
ARRAY OF
name
;
STREAM
name
,
:
COUNT
EVENT
STATE
;

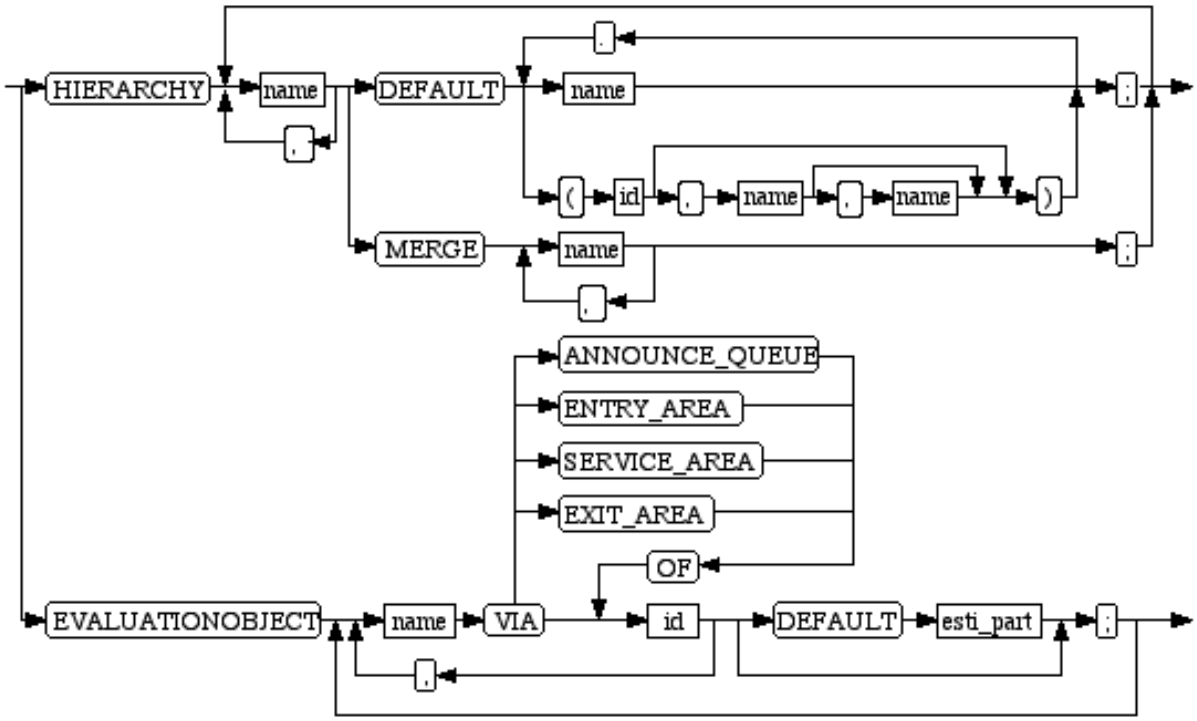
decl_com



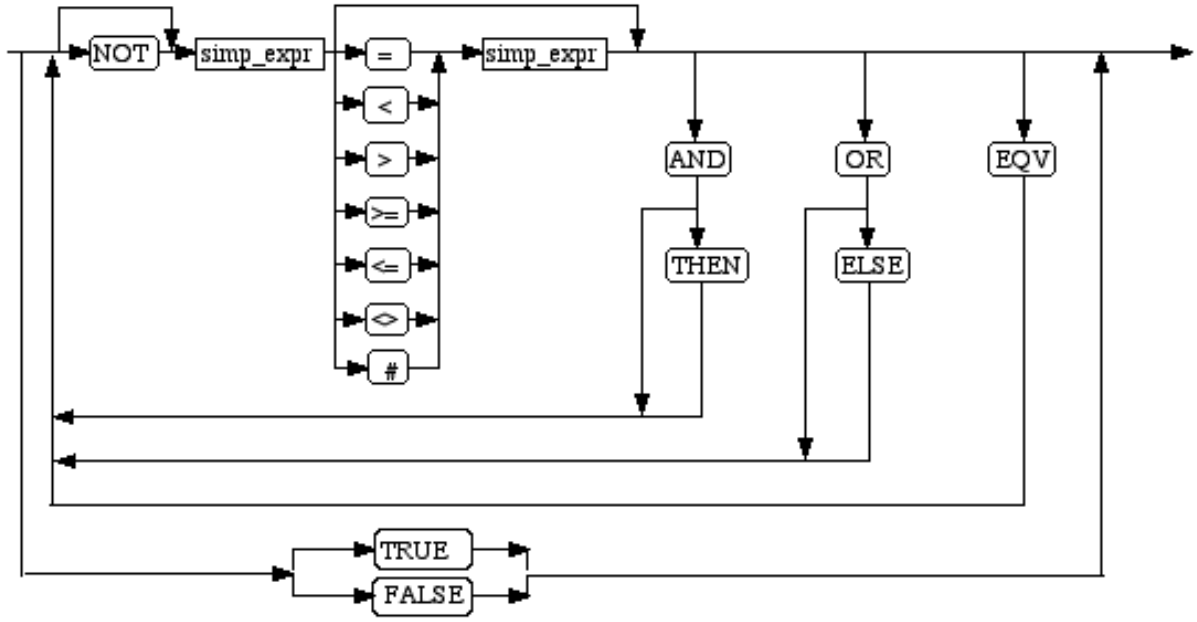
esti_part



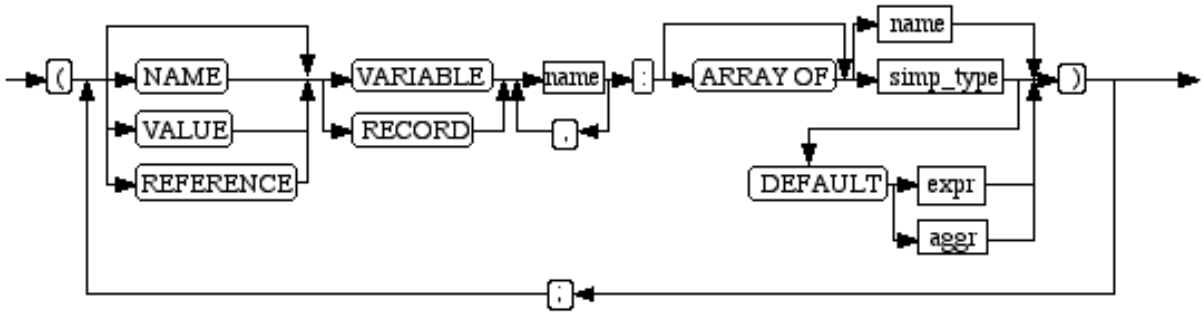
evaluate_decl



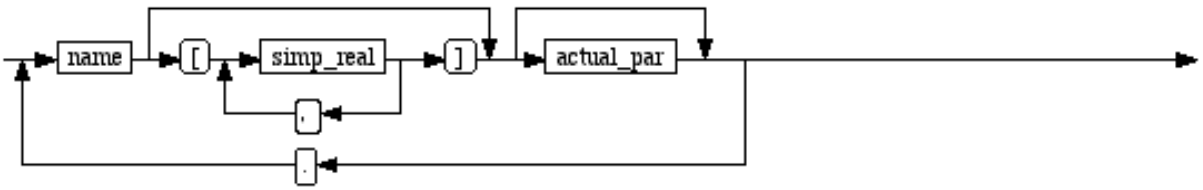
expr, expr_bool



formal_par

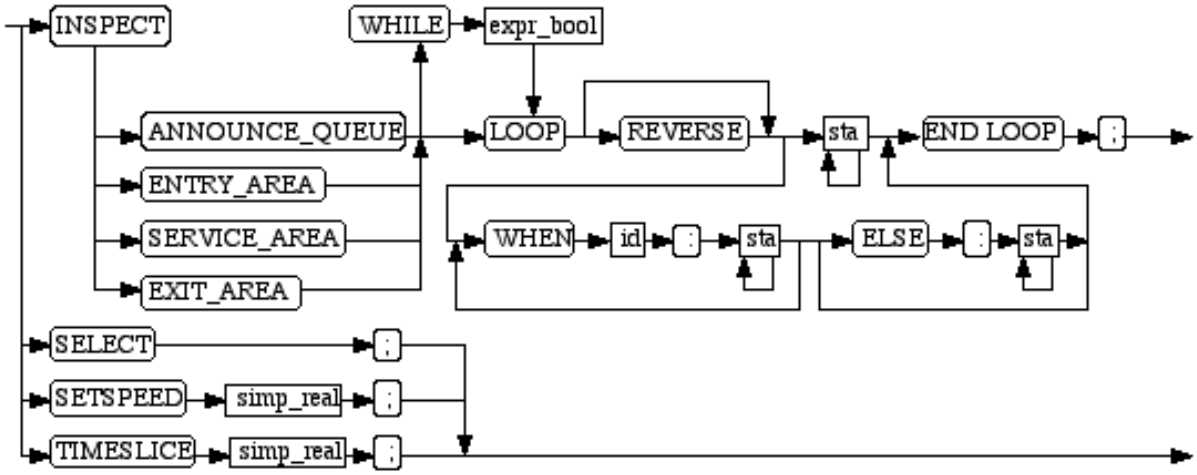


id

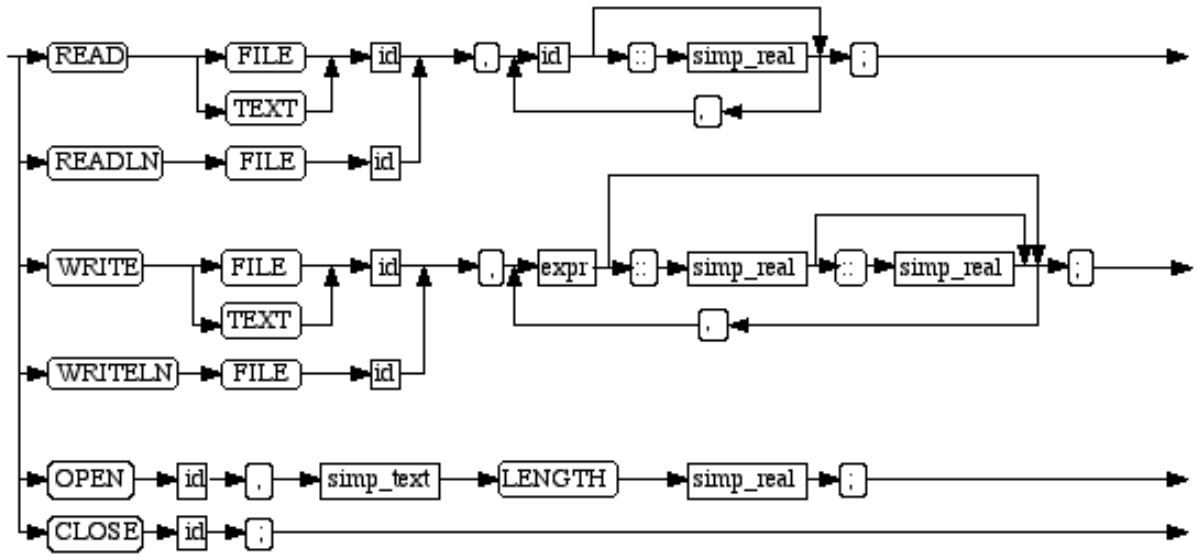


inscription

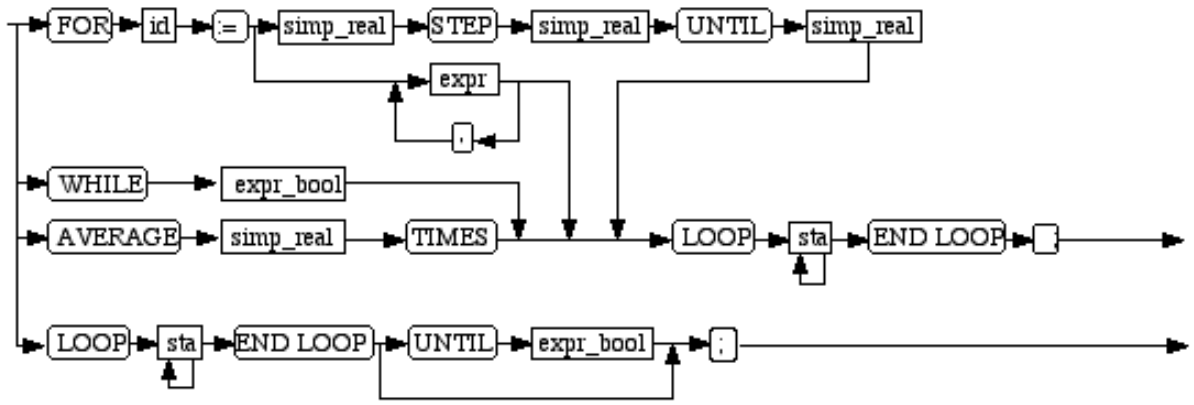
inspect_sta



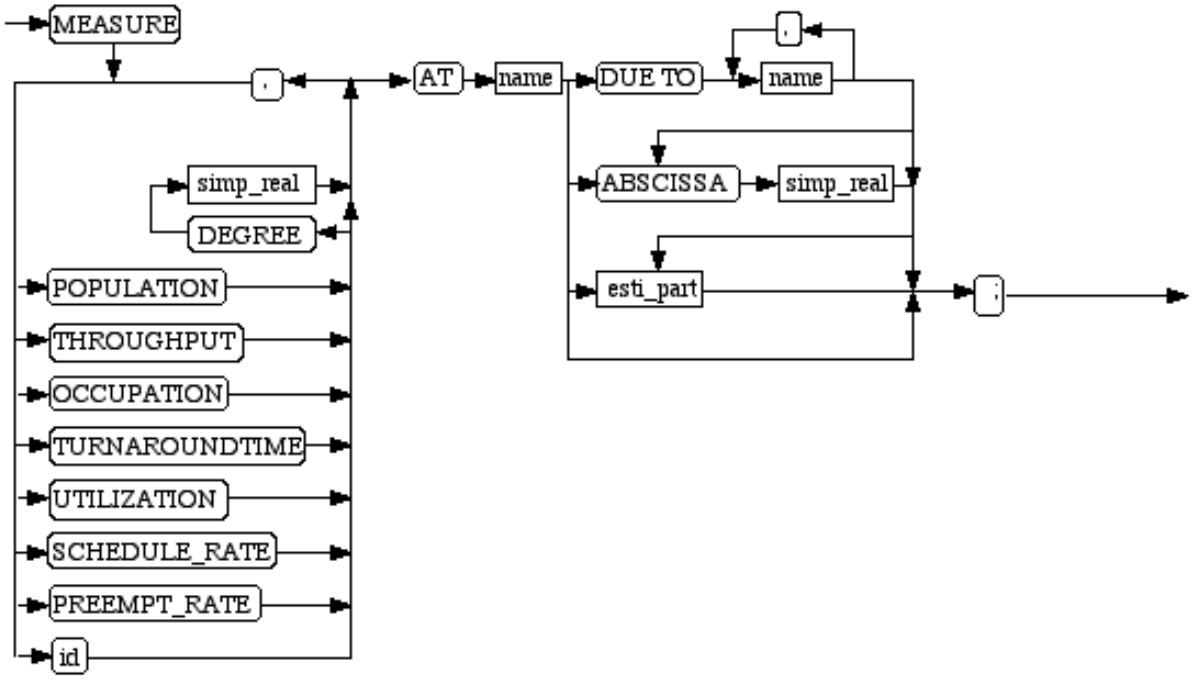
io_sta



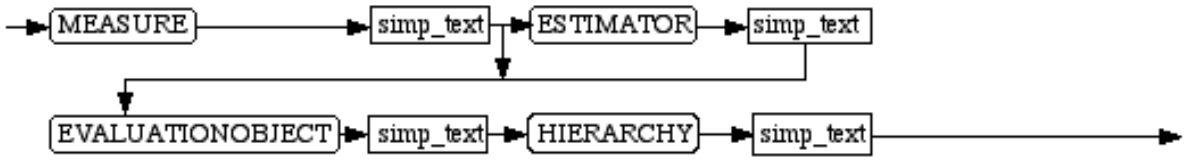
loop_sta



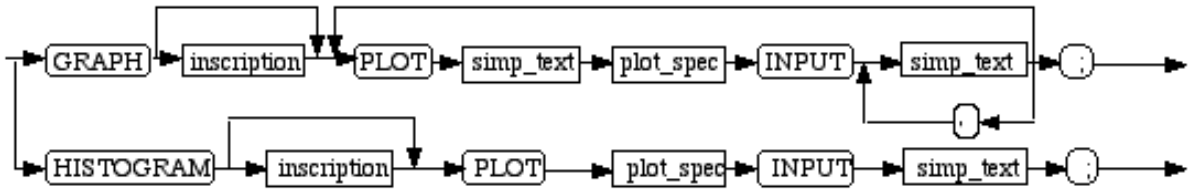
measure_sta



plot_spec

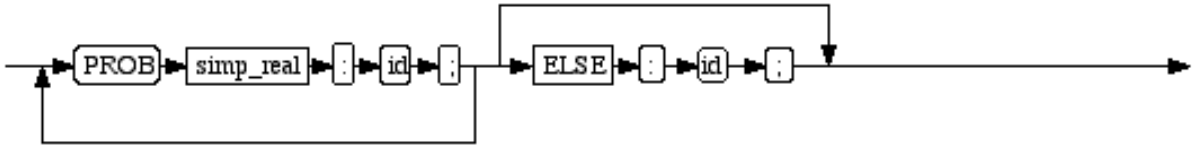


plot_sta

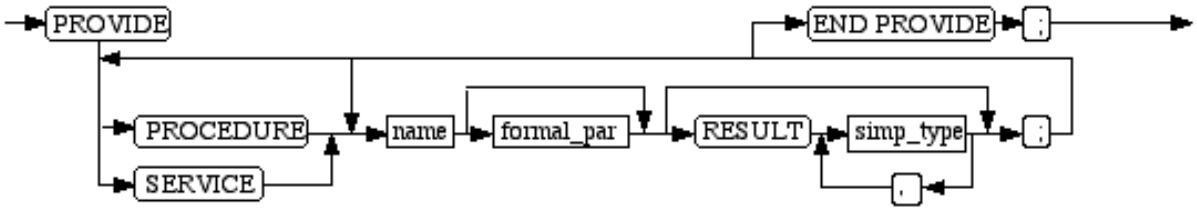


primary

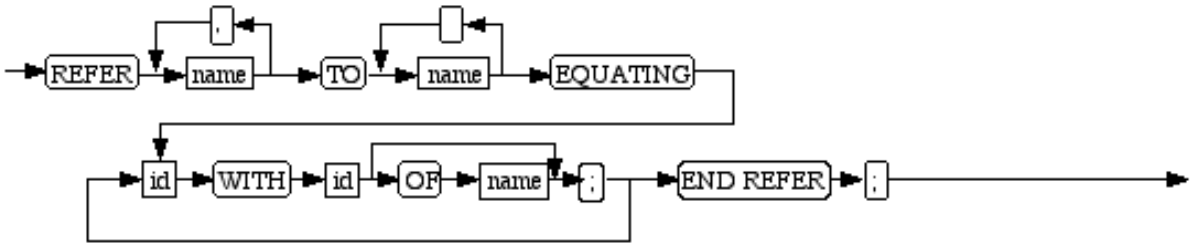
prob_part



provide

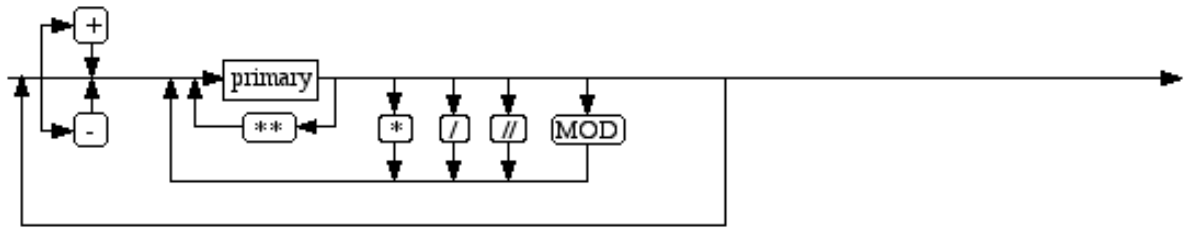


refer_part



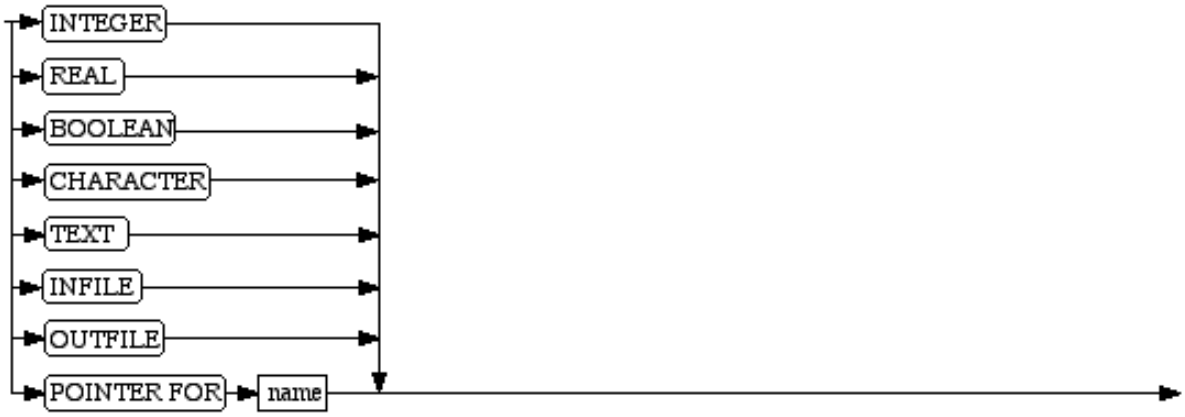
simp_expr

simp_real

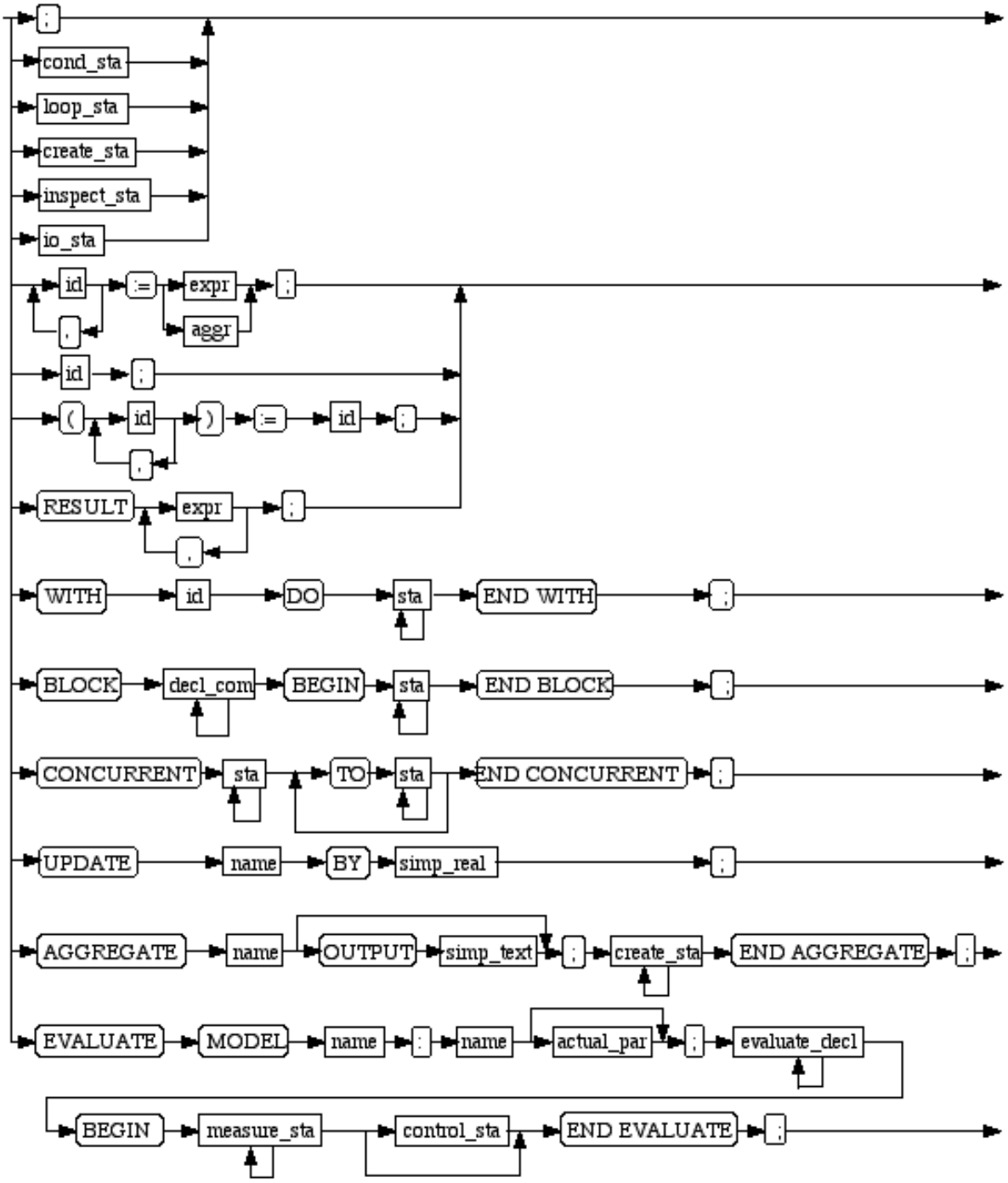


simp_text

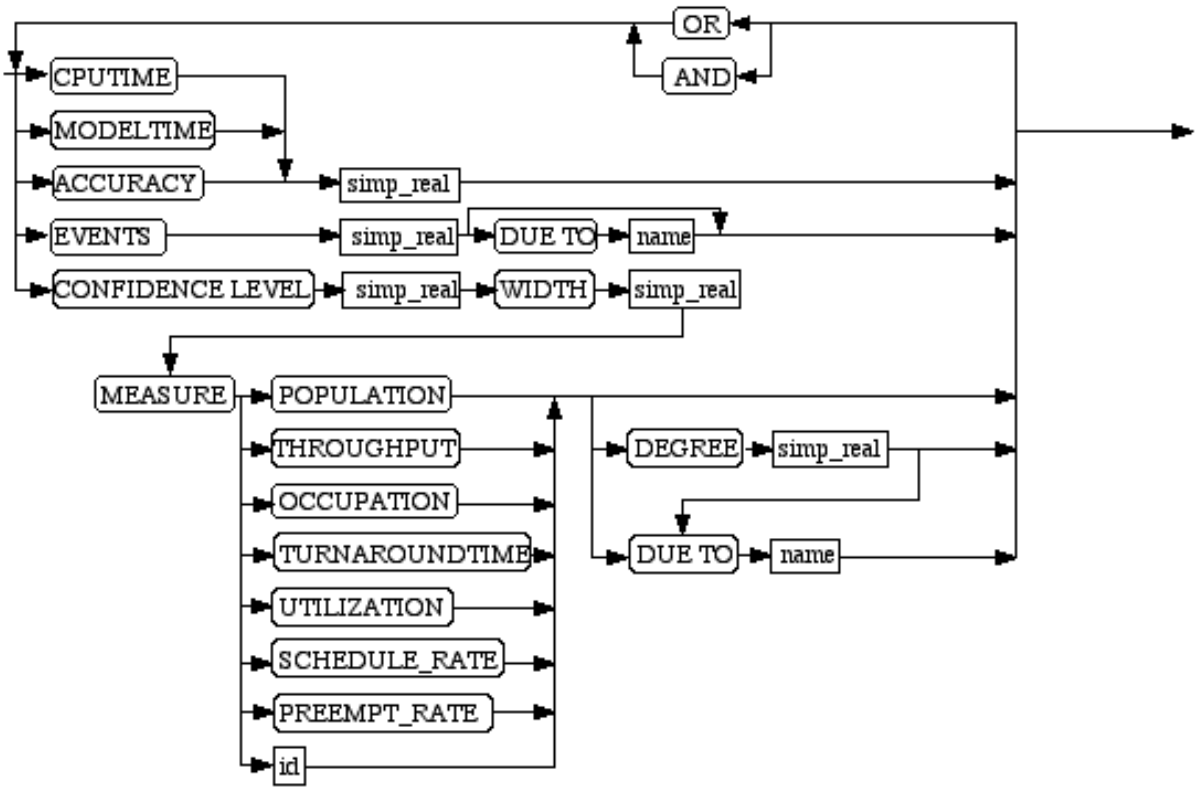
simp_type



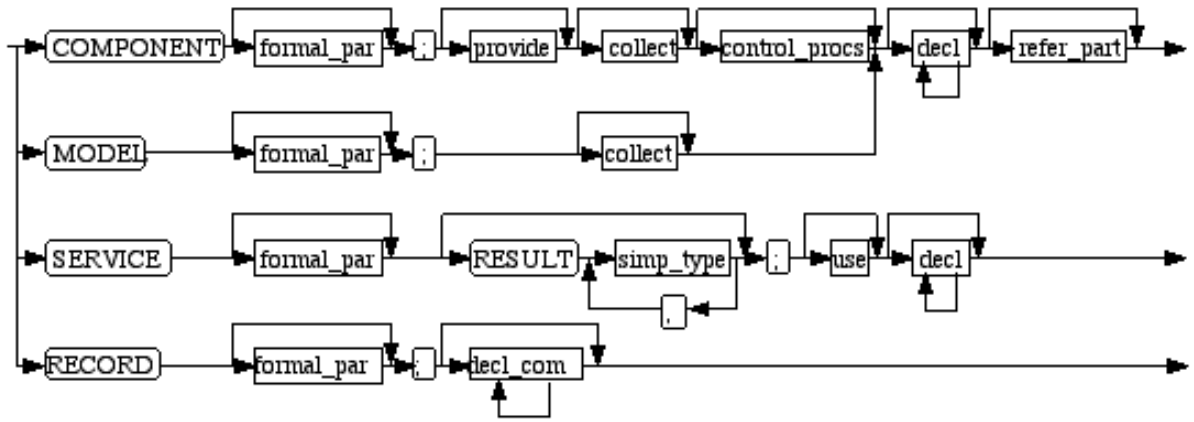
sta


;
cond_sta
loop_sta
create_sta
inspect_sta
io_sta
id
:=
expr
;
,
aggr
id
;
(
id
)
:=
id
;
,
RESULT
expr
;
,
WITH
id
DO
sta
END WITH
;
BLOCK
decl_com
BEGIN
sta
END BLOCK
;
CONCURRENT
sta
TO
sta
END CONCURRENT
;
UPDATE
name
BY
simp_real
;
AGGREGATE
name
OUTPUT
simp_text
;
create_sta
END AGGREGATE
;
EVALUATE
MODEL
name
:
name
actual_par
;
evaluate_decl
BEGIN
measure_sta
control_sta
END EVALUATE
;

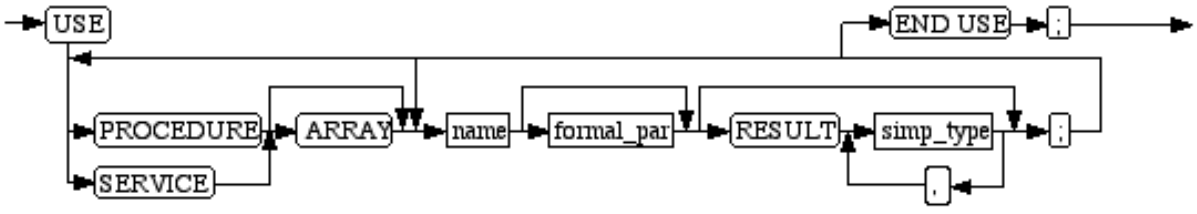
time_control



type_decl

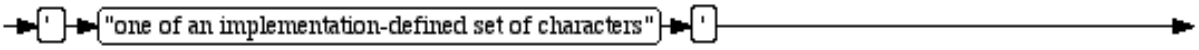


use

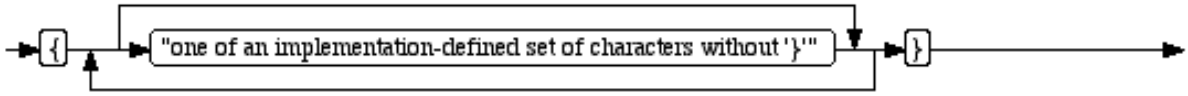


B.2. Token Syntax

character



comment

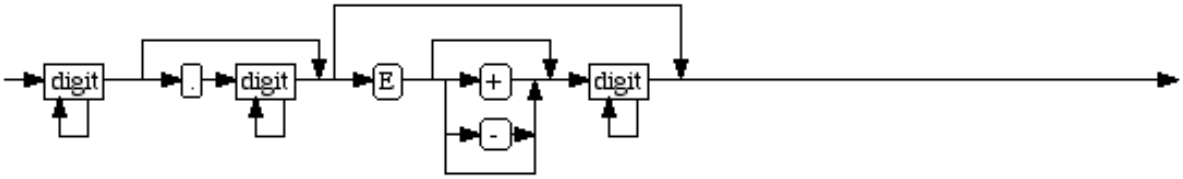


digit

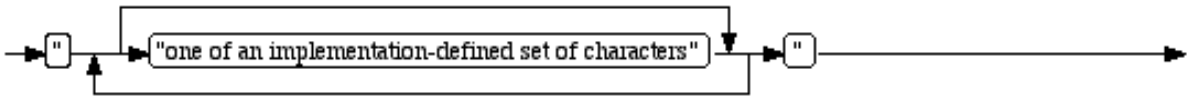
letter

name

number



string



1C. Lexical Units

This appendix list all HI-SLANG keywords and symbols and all available characters.

C.1. Reserved HI-SLANG Keywords

The following 144 words are reserved for use as keywords in HI-SLANG and must not be used otherwise:

ABSCISSA
ACCURACY
AFTER
AGGREGATE
ANALYTICAL
AND
ANNOUNCE_QUEUE
ARRAY
AS
AT
AVERAGE
BEGIN
BLOCK
BOOLEAN
BOUNDS
BRANCH
BY
CASE
CHARACTER
CLOSE
CLOSED_CHAIN
COLLECT
COMPONENT
CONCURRENT
CONFIDENCE
CONSTANT
CONTROL
COUNT
CPUTIME
CREATE
DEFAULT
DEGREE
DO
DUE
DUMPFILE
ELSE
ENCLOSE
END
ENRTY_AREA
EQUATING
EQV
ESTIMATOR
EVALUATE
EVALUATIONOBJECT
EVENT
EVENTS
EVERY
EXIT_AREA
EXPERIMENT
FALSE
FILE
FOR
FREQUENCY
GLOBALSTOP
GRAPH
HEADER
HIERARCHY
HISTOGRAM
IF
INFILE
INPUT
INSPECT
INTEGER
INTERVAL
LENGTH
LET
LEVEL
LIMIT
LOOP
MEAN
MEASURE
MERGE
METHOD
MOD
MODEL
MODELTIME
NAME
NEW
NONE
NOT
OCCUPATION
OF
OPEN
OPEN_CHAIN
OR
ORDINATE
OUTFILE
OUTPUT
PLOT
POINTER
POPULATION
PREEMPT_RATE
PROB
PROCEDURE
PROCESS
PROVIDE
QNODE
READ
READLN
REAL
RECORD
REFER
REFERENCE
RESULT
REVERSE
SCHEDULE_RATE
SELECT
SERVICE
SERVICE_AREA
SETSPEED
SIMULATIVE
STANDARDDEVIATION
START
STATE
STEP
STOP
STREAM
SUBMIT
TABLE
TEXT
THEN
THROUGHPUT
TIMES
TO
TIMESLICE
TRACE
TRACEALL
TRUE
TURNAROUNDTIME
TYPE
UNTIL
UPDATE
UPDATES
USE
UTILIZATION
VALUE
VARIABLE
VIA
WHEN
WHILE
WIDTH
WITH
WRITE
WRITELN

C.2. Reserved HI-SLANG Symbols

The following list contains all the 36 symbols of HI-SLANG. Symbols are language terminals consisting of one or two unseparated special characters. For some symbols there exist substitutive symbols.

Symbol	Meaning
.	Access to local objects via dot notation, e.g., for RECORDS and OUTPUT or Separation of parenthesized HIERARCHY tripels or Decimal point
..	Separation of lower bounds and upper bounds of ARRAYs or Definition of FREQUENCY intervals
;	Separator
,	Separator
:	Separator
Blank	Separator
::	Separation of format values in READ and WRITE
:=	Assignment
'	CHARACTER delimiter
"	TEXT delimiter
(	Opening bracket
)	Closing bracket
[, (.	Start of indexing
], .)	End of indexing
{, (*	Start of comment
}, *)	End of comment
=	Comparator "equal"
#, <>	Comparator "not equal"
<	Comparator "less than"
<=	Comparator "less than or equal"
>	Comparator "greater than"
>=	Comparator "greater than or equal"
+	Operator "addition"
-	Operator "subtraction"
*	Operator "multiplication"
/	Operator "division"
//	Operator "integer division"
**	Operator "exponentiation"
&	Operator "text concatenation"
_	Underscore within identifiers (not a symbol due to the definition above).
%	Used as first character different from blank or tab of a source line the percent character indicates the beginning of a compiler control statement interpreted by the HIT-FAN system (not a symbol due to the definition above)

Table 0: HI-SLANG Symbols

C.3. ASCII and EBCDIC Tables

The first table shows the ASCII code, the second table shows the EBCDIC code. Every character is defined by a pair of hexadecimal digits. The left hex-digit corresponds to the upper half byte of the character, and it is declared in the head of the tables. The right hex-digit corresponds to the lower half byte of the character, and it is specified on the left hand side of the tables. The third table shows the corresponding decimal value.

Examples:

Characters	ASCII code		EBCDIC code	
	hex.	dec.	hex.	dec.
'5'	35	53	F5	245
'S'	53	83	E2	226
'm'	6D	109	94	148
'?'	3F	63	6F	111

You can take access to the codes by using *rank* and *char*, explained in Appendix D.

The upper bit of the ASCII coding is used as the parity bit. Both even and odd parity are common. Characters within the range '00' to '1F', and the character '7F' are control characters in the ASCII code. EBCDIC control bytes are within the range '00' to '3F'. Unfortunately, the coding of control information is not standardized.

The characters '[', ']', '{' and '}' do not exist in standard EBCDIC. Yet, since they are symbols in HI-SLANG (see Appendix B.), they are elements of the EBCDIC tables. They are coded as installed on most Siemens and IBM computers. This is important to know, for any HI-SLANG source contains characters that are coded in ASCII or EBCDIC. Since the characters mentioned above are **not** available on IBM keyboards, it is necessary to substitute them using special editor options. It is also possible to substitute those characters by '(.', '.)', '(*', '*)'.

Vacant spaces within the EBCDIC table indicate that their codes are not used. Correlating to control bytes, these vacancies are not standardized, either.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	X-ON	!	1	A	Q	a	q
2	STX	TP-ON	"	2	B	R	b	r
3	ETX	X-OFF	#	3	C	S	c	s
4	EOT	TP-OFF	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	TAB	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table 1: ASCII Code

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL				SP	&	-									0
1							/		a	j	~		A	J		1
2									b	k	s		B	K	S	2
3									c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	EOB	UC					f	o	w		F	O	W	6
7	DEL	IL	PR	EOT					g	p	x		G	P	X	7
8									h	q	y		H	Q	Y	8
9									i	r	z		I	R	Z	9
A			SM		`	!	^	:								
B					.	$	,	#				[				{
C					<	*	%	@				\				
D					(	)	_	'				]				}
E					+	;	>	=								
F					|	¬	?	"								

Table 2: EBCDIC Code

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	000	016	032	048	064	080	096	112	128	144	160	176	192	208	224	240
1	001	017	033	049	065	081	097	113	129	145	161	177	193	209	225	241
2	002	018	034	050	066	082	098	114	130	146	162	178	194	210	226	242
3	003	019	035	051	067	083	099	115	131	147	163	179	195	211	227	243
4	004	020	036	052	068	084	100	116	132	148	164	180	196	212	228	244
5	005	021	037	053	069	085	101	117	133	149	165	181	197	213	229	245
6	006	022	038	054	070	086	102	118	134	150	166	182	198	214	230	246
7	007	023	039	055	071	087	103	119	135	151	167	183	199	215	231	247
8	008	024	040	056	072	088	104	120	136	152	168	184	200	216	232	248
9	009	025	041	057	073	089	105	121	137	153	169	185	201	217	233	249
A	010	026	042	058	074	090	106	122	138	154	170	186	202	218	234	250
B	011	027	043	059	075	091	107	123	139	155	171	187	203	219	235	251
C	012	028	044	060	076	092	108	124	140	156	172	188	204	220	236	252
D	013	029	045	061	077	093	109	125	141	157	173	189	205	221	237	253
E	014	030	046	062	078	094	110	126	142	158	174	190	206	222	238	254
F	015	031	047	063	079	095	111	127	143	159	175	191	207	223	239	255

Table 3: Conversion from Hexa-Decimal to Decimal

D. Modelling Environment

All HI-SLANG models can make use of the standard modelling environment of HIT. It consists of all objects stored in the HIT Standard Mobase (see Appendix F) as well as the built-in procedures listed in this appendix.

All the predefined procedures and operators (i.e., built-in procedures using infix notation) listed in Appendix D.1 and D.2 are "globally declared" for every HI-SLANG source, while the predefinitions listed in Appendix D.3 are only available in a special context, e.g., only within services or component types.

In contrast to reserved words it is possible to declare other objects using the same name (e.g., *negexp*), but then the corresponding predefined procedure is no longer accessible in that block. This is not valid for the built-in infix operators, since their names are either keywords or symbols.

D.1. Operators and Precedence Rules

The following tables list the type of result (or signatures) of all HI-SLANG operators, depending on the type of their arguments.

unary operators:

operand	operator +	-
INT	INT	INT
REAL	REAL	REAL

binary operators:

left operand	right operand	operator +	-	*	/	//	MOD	**
INT	INT	INT	INT	INT	REAL	INT	INT	REAL
INT	REAL	REAL	REAL	REAL	REAL	INT	INT	REAL
REAL	INT	REAL	REAL	REAL	REAL	INT	INT	REAL
REAL	REAL	REAL	REAL	REAL	REAL	INT	INT	REAL

left operand	right operand	operator EQV	AND	AND THEN	OR	OR ELSE
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE

left operand	right operand	operator &
TEXT	TEXT	TEXT

The result of a comparison is of type BOOLEAN. Both arguments to be compared must have the same type. There is one exception: Comparing operands of type REAL and INTEGER is possible, as the INTEGER operand is converted to REAL.

Precedence rules for operators:

(1) = lowest precedence	EQV
(2)	OR, OR ELSE
(3)	AND, AND THEN
(4)	NOT
(5)	=, <, >, <=, >=, <>, #
(6)	+, -, &
(7)	*, /, //, MOD
(8)	**
(9) = highest precedence	(expression), OF

With n<m operators of precedence operators of class n are evaluated after those of class m. Operators of the same class are evaluated from the left to the right (left associative).

D.2. Predefined Procedures

name	type of result	name of 1. par	type of 1. par	name of 2. par	type of 2. par	usable[1] for array bounds	comments
abs	REAL	X	REAL	-	-	y	
arccos	REAL	X	REAL	-	-	y	
arcsin	REAL	X	REAL	-	-	y	
arctan	REAL	X	REAL	-	-	y	
cos	REAL	X	REAL	-	-	y	
cosh	REAL	X	REAL	-	-	y	
entier	INT	X	REAL	-	-	y	
exp	REAL	X	REAL	-	-	y	arithmetic
ln	REAL	X	REAL	-	-	y	functions
log	REAL	X	REAL	-	-	n	
maxint	INT	-	-	-	-	y	
maxreal	REAL	-	-	-	-	y	
sign	INT	X	REAL	-	-	y	
sin	REAL	X	REAL	-	-	y	
sinh	REAL	X	REAL	-	-	y	
sqrt	REAL	X	REAL	-	-	y	
tan	REAL	X	REAL	-	-	y	
tanh	REAL	X	REAL	-	-	y	
undefined	REAL	-	-	-	-	y	
char	CHAR	I	INT	-	-	-	
digit	BOOL	C	CHAR	-	-	-	character
letter	BOOL	C	CHAR	-	-	-	functions
rank	INT	C	CHAR	-	-	y	
cox	REAL	A	REAL	B	REAL	n	
coxg	REAL	A	ARRAY	-	-	n	
discrete	INT	A	ARRAY	-	-	n	
draw	BOOL	A	REAL	-	-	n	
erlang	REAL	A	REAL	B	REAL	n	
histd	INT	A	ARRAY	-	-	n	random
linear	REAL	A	ARRAY	B	ARRAY	n	number
negexp	REAL	A	REAL	-	-	n	generators
normal	REAL	A	REAL	B	REAL	n	
poisson	INT	A	REAL	-	-	n	
randint	INT	A	INT	B	INT	n	
uniform	REAL	A	REAL	B	REAL	n	
cpu_time	REAL	-	-	-	-	n	
get_result	REAL	*	*	*	*	n	
last_seed	INT	-	-	-	-	n	
put_header	-	link	TEXT	info	TEXT	-	
put_footer	-	link	TEXT	info	TEXT	-	
put_result	-	*	*	*	*	-	modelling
set_seed	-	new_seed	INT	-	-	-	support
stop_evaluation	-	message	TEXT	-	-	-	procedures
time	REAL	-	-	-	-	n	
trace_off	-	-	-	-	-	-	
trace_on	-	-	-	-	-	-	
trace_state	-	-	-	-	-	-	
transfer_results	-	-	-	-	-	-	
was_message	BOOL	kind	CHAR	no	INT	-	
eof	BOOL	F	INFILE	-	-	-	
eoln	BOOL	F	INFILE	-	-	-	
lastitem	BOOL	F	INFILE	-	-	-	
lowten	-	C	CHAR	-	-	-	I/O support
sysin	INFILE	-	-	-	-	-	procedures
sysout	OUTFILE	-	-	-	-	-	
tracefile	OUTFILE	-	-	-	-	-	

Legend: ARRAY ⇔ ARRAY [...] OF REAL - ⇔ non existent
* ⇔ more than two parameters, see below

[1]Usage possible for array bounds definition within hit unit and experiment block

D.2.1.Arithmetic Functions

The exact definitions (concerning precision, allowed parameter values, etc.) for most arithmetic functions are implementation defined. All functions return best possible approximations to the exact mathematical results. For argument values where the mathematical functions are not defined a run time error occurs.

D.2.1.1. Trigonometric Functions

All trigonometric functions deal with angles expressed in radians. The following functions are available:

sin (X: REAL) RESULT REAL

cos (X: REAL) RESULT REAL

tan (X: REAL) RESULT REAL

arcsin (X: REAL) RESULT REAL

arccos (X: REAL) RESULT REAL

arctan (X: REAL) RESULT REAL

sinh (X: REAL) RESULT REAL

cosh (X: REAL) RESULT REAL

tanh (X: REAL) RESULT REAL

D.2.1.2. Other Arithmetic Functions

The following arithmetic functions are built into the HIT system:

abs (X: REAL) RESULT REAL

The result is the magnitude of X.

entier (X: REAL) RESULT INTEGER

The result is the integer "floor" of the real X, the value always being less than or equal to X. Thus *entier* (1.8) returns 1, while *entier* (-1.8) returns -2.

exp (X: REAL) RESULT REAL

The result is e^X.

ln (X: REAL) RESULT REAL

The result is ln X. X must be positive.

log (X: REAL) RESULT REAL

The result is $\log_{10}$ X. X must be positive.

sign (X: REAL) RESULT INTEGER

The result is zero if X is zero, +1 if X is positive and -1 if X is negative.

sqrt (X: REAL) RESULT REAL

The result is $\sqrt{X}$. X must not be negative.

D.2.1.3 Installation-dependent Functions

These functions result values defined by the underlying hardware or Simula system.

maxreal RESULT REAL

The *maxreal* value is the maximal real representable by your hardware. If this value is exceeded during REAL calculation an overflow results from the SIMULA Run Time System.

The smallest REAL is - MAXREAL.

undefined RESULT REAL

The *undefined* value is defined as *maxreal* / 4 (to avoid calculation overflows). It is especially used in dump files for performance measures which could not be determined (in table files the word "Undefined" is displayed instead).

If you read values from a dump file by *get_result* you should take into account that they may be *undefined.*

maxint RESULT REAL

The *maxint* value is the maximal integer representable by your hardware. If this value is exceeded during INTEGER calculation an integer overflow results from the SIMULA Run Time System.

D.2.2.Character Functions

All character functions deal with characters. In HI-SLANG the set of characters is either coded in ASCII or EBCDIC (see Appendix C.3.).

char (I: INTEGER) RESULT CHARACTER

The result is the character obtained by converting I according to the implementation defined coding of characters. I must be in the range 0..255, otherwise a run time error occurs. The inversion of *char* is *rank*. *char (rank (c)) = c* always holds.

digit (C: CHARACTER) RESULT BOOLEAN

The result is TRUE if the character C represents a decimal digit.

letter (C: CHARACTER) RESULT BOOLEAN

The result is TRUE if the character C is a letter of the alphabet ('a'..'z', 'A'..'Z').

rank (C: CHARACTER) RESULT INTEGER

The result is an integer in the range 0..255, obtained by converting C according to the implementation defined character code. The inversion of *rank* is *char*. For i in 0..255 *rank (char (i)) = i* always holds.

D.2.3.Random Number Generators

All (pseudo-)random drawing procedures of HI-SLANG are based on the technique of obtaining "basic drawings" from the uniform distribution in the interval]0,1[. A basic drawing (call of a random number generator) replaces the value of the predefined integer variable *seed* by a new value according to an implementation defined algorithm, e.g.,

seed (i+1) = mod (seed (i) ∗ 5∗∗ (2∗p+1), 2∗∗n)

and generates a stream of pseudo-random numbers in the interval [1,(2**n) -1]. The result of the i+1th basic drawing called u(i+1) is calculated by

u(i+1) = seed (i+1) / 2∗∗n

The *n* is an integer related to the size of a computer word (e.g., 35) and *p* is a positive integer (e.g., 6). The initial value of *seed* is 13 within a globally declared procedure or within the experiment block. Within a model, component or service the predefined parameter *seed* (with default 13) of the model type denoting the root of the hierarchy is used (see chapter 4.).

It can be proved that if *seed* (= seed (0)) is a positive odd integer, the same holds for seed (i), and the sequence seed (0), seed (1), ... is cyclic with period 2∗∗n-2.

Of course the user may write his own random drawing procedures based on those listed below, e.g.,

```
PROCEDURE geometric_distribution (m : REAL) RESULT REAL;
    VARIABLE res : INTEGER;
BEGIN
    WHILE draw (m) LOOP res := res + 1; END LOOP;
    RESULT res;
END PROCEDURE;
```

The following random drawing procedures are available in HI-SLANG:

cox (A, B: REAL) RESULT REAL

The result is a COX-distributed pseudo-random number with rate A (reciprocal mean value) and variation coefficient B. If A ≤ 0 or B < 0.1, a run time error occurs.

coxg (A: ARRAY OF REAL) RESULT REAL

The result is a COX-distributed pseudo-random number. A must be a two dimensional ARRAY OF REAL or ARRAY aggregate denoting step-progress-probabilities for all service rates given in the first "line" of the array.

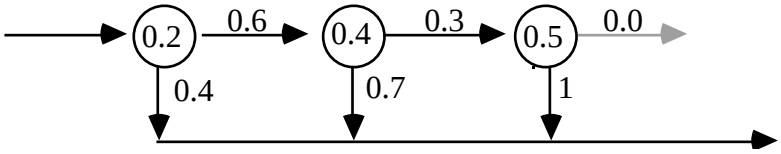
Example:

The array

A	1	2	3
1	0.2	0.4	0.5
2	0.6	0.3	0.0

or aggregate [[0.2, 0.4, 0.5], [0.6, 0.3, 0.0]]

describes the following COX-distribution:



The values on the vertical arrows are not found in the array; they are the complement of the values in the second sub array.

Please note that the last value (in the second sub array) must be zero, while all the other array elements must be positive, otherwise a run time error occurs.

discrete (A: ARRAY OF REAL) RESULT INTEGER

The one-dimensional ARRAY OF REAL A, augmented by the element 1 to the right (for safety reasons), is interpreted as a step function of the subscript, defining a discrete (cumulative) distribution function. All array elements must be in the interval [0, 1] and in ascending order. The result of the function is the smallest index i such that A[i] > u, where u is the value of the basic drawing. The result thus is an integer in the range A.lower_bounds[1] .. A.upper_bounds[1]+1.

draw (A: REAL) RESULT BOOLEAN

The result is TRUE with the probability A, FALSE with the probability 1-A. For A ≤ 0 the result is always FALSE, for A ≥ 1 the result is always TRUE.

erlang (A, B: REAL) RESULT REAL

The result is a drawing from the Erlang distribution with mean 1/A and standard deviation 1/(A∗sqrt(B)). Both the rate A and the number of phases B must be greater than zero.

histd (A: ARRAY OF REAL) RESULT INTEGER

The one-dimensional ARRAY OF REAL A is interpreted as a histogram defining the relative frequencies of the array elements. The relative frequencies must be less or equal 1, the sum over all elements must be one. The result of the function is an index, that is an integer in the range A.lower_bounds[1] ..A.upper_bounds[1].

linear (A, B: ARRAY OF REAL) RESULT REAL

A and B must be one-dimensional ARRAYs OF REAL with equal lower and upper bounds lb and ub. They define a cumulative distribution f. The result of type REAL is determined by linear interpolation in the non-equidistant distribution table, defined by A and B. The function f is defined by

A[i] = f(B[i]) for lb ≤ i ≤ ub

To avoid run time errors or installation dependent results, the following conditions must hold:

A[lb] = 0 and A[ub] = 1 and both arrays must be monotonous, i.e., for lb ≤ i ≤ ub, both, A[i] ≤ A[i+1] and B[i] ≤ B[i+1] hold.

negexp (A: REAL) RESULT REAL

The result is a drawing from the negative exponential distribution with mean 1/A (rate A), defined by - ln(u)/A, where u is the basic drawing. This is the same as a random "waiting time" in a Poisson distributed arrival pattern with expected number of arrivals per time unit equal to A. If A is non-positive, a run time error occurs.

normal (A, B: REAL) RESULT REAL

The result is normally distributed with mean A and standard deviation B. B must be non-negative.

poisson (A: REAL) RESULT INTEGER

The result is a drawing from the Poisson distribution with parameter A. The result *n* is defined by n+1 basic drawings u(i), such that n is the smallest non-negative integer with u(0)∗u(1)∗…∗u(n) < exp(-A). For negative A, the result is defined to be zero. When A is greater than some implementation defined value, e.g., 20, the result may be approximated by the maximum of *entier (normal (A, sqrt(A) + 0.5))* and zero.

randint (A, B: INTEGER) RESULT INTEGER

The result is one of the integers A, A+1,..., B-1, B with equal probability. If B < A, an error occurs.

uniform (A, B: REAL) RESULT REAL

The result is a uniformly distributed pseudo-random REAL number in the interval [A..B]. If B < A, an error occurs.

D.2.4.Modelling Support Procedures

HI-SLANG offers some procedures which make modelling more easy. They allow to query the current time (*cpu_time, time*) or control evaluations (*transfer_result, stop_evaluation*). Most of these procedures can only be used in simulations (implied by **(S)**; if also allowed for analytical solvers **(A,S)** is used).

(S) cpu_time RESULT REAL

The result is the amount of cpu time in seconds spent since program start, i.e., since the start of the first evaluation if there are more than one.

(A,S) get_result (esti : TEXT DEFAULT "MEAN";
meas : TEXT;
evaobj : TEXT;
hier : TEXT DEFAULT "ALL";
link : TEXT DEFAULT "DUMP";
absc : REAL DEFAULT 0.0;
NAME err : REAL DEFAULT -1.0;) RESULT REAL

The result of this procedure is one performance value read from a dump file. The text parameters *esti*mator, *meas*ure, *eva*luation*obj*ect, *hier*archy and the real parameter *absc*issa together specify the value to search for. The dump file to search in is specified by its *link* name.

The use of upper-case or lower-case characters within the text arguments is tantamount. The interpretation of the result value depends on the first parameter *esti*mator:

"MEAN"	: mean value
"STANDARDDEVIATION"	: standard deviation
"CONFIDENCE"	: half the size of the confidence interval (%)
"FREQUENCY n"	: number of events in the *n*'th interval
"LOWERBOUND"	: lower bound
"UPPERBOUND"	: upper bound

If there is no such performance value in the dump file as specified by the first six parameters, the result will be the value of the name parameter *err* (error_value). This value can be specified by another call of *get_result*, which will only be executed in case of an error, since it is a name parameter, which is not modified.

If there are more than one of such values in the dump file (caused by an evaluation series), the result will be the performance value of the last completed evaluation written to that dump file. By writing different abcissa values (see MEASURE statement) for different evaluations of a series, the values of a previous evaluation can be accessed.

The main application for this procedure is to control evaluation series, but *get_result* may appear anywhere in HI-SLANG sources.

Note that a warning occurs if the specified name of evaluation object, measure or hierarchy is cut. Please take care that your search criterias could be fulfilled after this truncation.

Example:

```
TYPE mt MODEL (...);
   …
   CREATE get_result ("mean","POPULATION","cpu", "all","dump", 0.0, LET err := 0) + 1
         PROCESS st;
END TYPE mt;

EXPERIMENT e METHOD SIMULATIVE;
   VARIABLE pop : INTEGER;
BEGIN
   WHILE get_result (, "TURNAROUNDTIME", "cpu", "cpu_hier",
                     "DUMP", 2, get_result (, "TURNAROUNDTIME", "cpu2")) < 10;
   LOOP
      EVALUATE MODEL m : mt (pop);
      …
      BEGIN
         MEASURE POPULATION AT cpu OUTPUT DUMPFILE "DUMP";
         CONTROL AT cpu STOP MODELTIME
            may_be_ok_at (get_result ("CONFIDENCE", "POPULATION",
                                      "cpu", "hi", "DUMP");
      END EVALUATE;
         pop := pop + 1;
   END LOOP;
END EXPERIMENT;
```

Appropriate declarations are assumed above.

(A,S) last_seed RESULT INTEGER

This procedure results the current *seed* value, determined by the previous action of that list.

There are three ways to modify the *seed* value:

- calling a random drawing procedure (for simulative analysis in model description part and experiment part; for analytical methods only in experiment part),
- calling *set_seed*,
- setting *seed* via the predefined model parameter *seed* (within an EVALUATE statement).

Note:

Normally every simulation in an evaluation series starts with *seed* value 13, if no other value is set in the EVALUATE statement. To have a continuous sequence of *seed* values please use: EVALUATE MODEL m: mt (LET seed := last_seed);

```
(A,S) put_result ( esti     :TEXT    DEFAULT "MEAN";
                   meas     :TEXT;
                   evaobj   :TEXT;
                   hier     :TEXT    DEFAULT "ALL";
                   link     :TEXT    DEFAULT "DUMP";
                   absc     :TEXT    DEFAULT 0.0;
                   res      :REAL    DEFAULT undefined;
                   lev      :INTEGER DEFAULT 95;
                   wid      :REAL    DEFAULT 0.0;
                   int      :TEXT    DEFAULT "")
```

This procedure appends one performance entry to a dump file. The first four text parameters *esti*mator, *meas*ure, *eva*luation*obj*ect and *hier*archy together specify the meaning of the values to be added, the parameter *link* denotes the link name of the dump file concerned. The use of upper- or lower-case letters within these text arguments is tantamount.

For producing graphs from dump files an abscissa value *absc* (x-value) may be assigned to each performance value (y-value).

The last four parameters *res*ult, *lev*el, *wid*th and *int*ervals define the (possibly structured) performance value itself. Depending on the value of the first parameter *esti* it consists of:

"MEAN"	:	The mean value *res.*
"STANDARDDEVIATION"	:	The standard deviation *res.*
"CONFIDENCE"	:	The mean value *res* of the confidence interval, the half size *wid* of the interval in percent and the confidence level *lev*.
"FREQUENCY"	:	The number of intervals *lev* and a text *int*, which contains *lev* times three blank-separated REAL values, the first and second denoting lower and upper bound of that interval respectively, while the third gives the number of events with values lying in that interval.
"LOWERBOUND"	:	The lower performance bound *res.*
"UPPERBOUND"	:	The upper performance bound *res.*

For the format of a dump file compare Appendix G.3.2. All parameters but *meas* and *evaobj* have default values (see above), so they may be omitted when calling *put_result*. Before the first call of *put_result* the procedure *put_header* has to be called, and after the last call *put_footer* must be activated.

Note that a warning occurs if the specified name of evaluation object, measure, estimator or hierarchy is cut. Please take care that your search criterias could be fulfilled after this truncation.

Examples:

```
put_result (  , "POPULATION", "eva", LET res := 3); {mean population at eva set to 3}

put_result ("STANDARDDEVIATION", "TURNAROUNDTIME", "e", "h1", "dmp", 1.5, 4.7);

put_result ("CONFIDENCE", "Occupation", "e", LET res := 12.0,
            LET wid := 10.0, LET lev := 99);

put_result ("FREQUENCY", "POPULATION", "e", LET lev := 3,
            LET int := "1 2 7  3 5 2  5 10 1");

put_result ("LOWERBOUND", "THROUGHPUT", "e", LET res := 1.72E-3);
```

(A,S) put_header (link : TEXT DEFAULT "DUMP";
info : TEXT DEFAULT "")

This procedure has to be called before the first call of *put_result*. It opens the dump file denoted by its link name *link* (which should have an EXTEND binding, see 8.2.2.) and appends a header. This header is only a short form of the standard header described in Appendix G.3.2., but contains the text passed by parameter *info*.

(A,S) put_footer (link : TEXT DEFAULT "DUMP";
info : TEXT DEFAULT "")

This procedure has to be called after the last call of *put_result*. It appends a footer to the dump file denoted by its link name *link* and closes it. This footer is only a short form of the standard footer described in Appendix G.3.2., but contains the text passed by parameter *info*.

Example:

```
put_header   (, "special_results");
put_result   (...);
...
put_footer;
```

(A,S) set_seed (new_seed: INTEGER DEFAULT 13)

This procedure sets the *seed* value for the next call of a random number generator to the given value. It enables to use a defined seed start value for the next sequence of random number generators, which may be useful, e.g., in experiment descriptions to determine some model data stochastically and with a seed value independent of the previous evaluation.

(S) stop_evaluation (message: TEXT)

Within a model, component, or service a call of *stop_evaluation* prints *message* to the standard output *sysout* and stops the current evaluation. The performance values determined so far are given as specified in MEASURE statements. Then the next evaluation is started, if it exists.

Within a global procedure (i.e., outside the model description) or the statement part of the experiment, a call of *stop_evaluation* has no effect but generates a warning.

(S) time RESULT REAL

Within a model, component, or service the result is the current model time, i.e., the simulation time. Within a global procedure or the statement part of an experiment, *time* gives the model time reached by the last simulative evaluation.

(S) trace_state

Within a model, component or service, a call of *trace_state* adds a state trace to the trace file, which documents the current state of the model. The format of the state trace is described in Appendix G.5.

Outside a model or component type a call of *trace_state* produces a warning. The procedure can only be used for simulative evaluations, otherwise a warning occurs.

If the model becomes empty and no new processes are to be generated, or all processes in the model can not perform any further activities, it is possible that a deadlock has occurred and an appropriate warning is written. In this case the procedure trace_state is called automatically. If the stop condition in the CONTROL statement does not contain MODELTIME as a basic condition, the value 'Reached Model Time' is not the actual model time but the time the last event in the model has happened. Be aware that the results are most likely useless.

(S) trace_off

Within a model, component or service, a call of *trace_off* stops the output of simulative event (not state) trace information, until *trace_on* is called. The comment

```
%STOP OF EVENT TRACE.  MODELTIME  n.nnnnnnEznn
```

is written to the trace file. It contains a real value which denotes the actual model time when the call of *trace_off* has taken place.

If there are several consecutive calls of *trace_off*, only the first call has the effect described above, all following calls are ignored. If you want to use the procedure *trace_off*, you must give a TRACE or TRACEALL command in the CONTROL statement, otherwise all calls of *trace_off* are ignored.

Outside a model or component type a call of *trace_off* has no effect but generates a warning.

(S) trace_on

Within a model, component or service, a call of *trace_on* restarts the output of the simulative event (not state) trace information, if it has been switched off by *trace_off* before. The comment

```
%START OF EVENT TRACE.  MODELTIME  n.nnnnnnEznn
```

is written to the trace file. It contains a real value which denotes the actual model time when the call of *trace_on* has taken place.

If there are several consecutive calls of *trace_on*, only the first call has the effect described above, all following calls are ignored. If you want to use the procedure *trace_on*, you must give a TRACE or TRACEALL command in the CONTROL statement (which also starts the trace at the beginning of the experiment), otherwise all calls of *trace_on* are ignored.

Outside a model or component type a call of *trace_on* has no effect but generates a warning.

(S) transfer_results (solver_info : BOOLEAN DEFAULT FALSE)

Within a model, component or service a call of *transfer_results* immediately starts calculation and output of all the performance values as specified in MEASURE statements. The evaluation continues. Within a global procedure or the statement part of the experiment, a call has no effect but generates a warning.

The output format used for these intermediate results is the same as the format of the final output into tables or dump files performed automatically (see Appendix G.3.). Calculation of intermediate results is very time consuming and should be used economically.

If the value of parameter *solver_info* is TRUE additional solver information will be produced (see Section G.1.3.4.).

See also the standard component type *observer*, which contains a call of *transfer_results*.

Example:

```
TYPE results SERVICE;
    VARIABLE t : REAL;
BEGIN
    LOOP
        WRITELN "Please enter model time for next intermediate results";
        READLN t;
        hold (t-time);
        transfer_results;
    END LOOP;
END TYPE results;
...
PROCESS output_results : results;
```

(A,S) was_message (kind : CHARACTER DEFAULT 'E';
number : INTEGER DEFAULT 0)
RESULT BOOLEAN

This procedure results TRUE if a HI-SLANG message of the given *kind* and the given *number* has previously occured. Previously here means: messages which have occured after the generation of the last completion message (see G.1.1.3), or just before the last completion message if there have not yet been messages hereafter.

For *kind* the values 'W' (for warnings), 'E' (errors) and 'A' (abort errors) are admissable, even in lower-case. If parameter *number* is set to zero (the default), this is interpreted as "any" message of the given *kind*. Thus *was_message* without setting any parameter returns TRUE if some errors have previously occured.

Note:

Be careful that the given number has the given kind, otherwise it will never be found.

Example:

```
num := 1;

WHILE NOT was_message AND num<100
LOOP
    EVALUATE MODEL m: mt (num);
    …
    END EVALUATE;
    num:=num+1;
END LOOP;
```

Normally the next evaluation of a series is started if there have been errors or warnings for the previous one. Using *was_message* an evaluation series can be better controlled. This enables saving much cpu time in cases where it is obvious that all further evaluations are senceless if any or a special kind of error was detected for a previous one.

Calling *was_message* is especially useful for the following message numbers:

'W' 166 : Possible numerical instabilities
'E' 165 : No computation possible

The procedure can also be applied to modify the model parameters for the next evaluation if such a message has previously occured.

D.2.5.I/O Support Procedures

Working with files or mobase objects is supported by procedures to determine the end of a line (*eoln*) or a file/object (*eof*, *lastitem*) or to determine the exponent sign of REAL values (*lowten*). Moreover, via procedures *sysin*, *sysout* and *tracefile* references to special file_objects can be obtained.

eof (F: INFILE DEFAULT sysin) RESULT BOOLEAN

When the position pointer of F has reached the end of the INFILE (or mobase object) *eof* results TRUE, otherwise FALSE. The result is also TRUE if F has not just been opened or has already been closed (see Section 3.6.). When calling *eof* with no actual parameter, the default *sysin* (see below) is used. A similar procedure is *lastitem* (see below).

Example: Reading all the records of INFILE f

```
VARIABLE f : INFILE;
...
OPEN f, "link name" LENGTH 80;
    {"link name" has to be bound in the control file}

WHILE NOT eof(f) LOOP
    {process current record, e.g., READ it}
    READLN FILE f;
END LOOP;
...
CLOSE f;
```

eoln (F: INFILE DEFAULT sysin) RESULT BOOLEAN

The result is TRUE, if no more characters can be read from the current record of the file (or mobase object) F, i.e., the position pointer has reached the end of the record. The result is also TRUE if F has not just been opened or has already been closed (see Section 3.6.). When calling *eoln* with no actual parameter the default *sysin* is used.

Example: Reading all the characters of the actual *sysin* record

```
VARIABLE c : CHARACTER;
...
WHILE NOT eoln LOOP
    READ c;   {process character c};
END LOOP;
```

lastitem (F: INFILE DEFAULT sysin) RESULT BOOLEAN

The procedure *lastitem* first skips past all blanks, tab characters, and line ends and thus may even skip some INFILE records. Beside this side-effect the result is the same as for *eof*.

lowten (C: CHARACTER)

In the textual representation of REAL values the default separator of mantissa and exponent is the character 'E'. *lowten* replaces this character by the value C. C should not be '.', ',', '+', '-', or a control character. Illegal values may cause run time errors.

Example:

```
WRITE 1234.5          normally  outputs  1.234500E+03
Calling lowten ('&')  before    outputs  1.234500&+03
```

After a *lowten* ('&') call a value "1.234500E+03" is interpreted as 1.2345 by a subsequent READ; the position pointer pointing at 'E' which is not part of the REAL.

sysin RESULT INFILE

sysin is the standard input of HI-SLANG, normally bound to the keybord, but it may be bound to any other file or mobase object (see Chapter 8.). It is automatically opened at the beginning of an analyzer run and closed at the end of a run. The record length is 80 characters.

All READ and READLN statements not followed by the keywords FILE or TEXT concern *sysin*.

sysout RESULT OUTFILE

sysout is the standard output of HI-SLANG, normally bound to the terminal, but it may be bound to any other file or mobase object (see Chapter 8.). It is automatically opened at the beginning of an analyzer run and closed at the end of a run. The record length is 132 characters.

All WRITE and WRITELN statements not followed by the keywords FILE or TEXT concern *sysout*.

tracefile RESULT OUTFILE

The procedure *tracefile* results a reference to the trace file of the simulator, which is automatically opened before each evaluation and closed afterwards. It has the link name "TRACE", which has a standard EXTEND binding to a file, but can be re-bound (see Chapter 8.). The file record length is 115 characters, the exact format of the trace is described in Appendix G.5.

The *tracefile* can be used within HI-SLANG models to write additional information to this file, which will be sorted chronologically within the regular trace information. Accessing *tracefile* within the statement part of the experiment makes no sence because the trace file is closed then.

Example:

```
WRITELN FILE tracefile, "So late: ", time;
```

D.3. Context-Dependent Predefinitions

Besides the entities listed in the next sections the following predefinitions exist:

- For each ARRAY declared in a HI-SLANG source the array attributes *dimension, lower_bounds* and *upper_bounds* are predefined and accessible via dot notation.
- For every service all the procedures to access the implicit state of the executing process and the services *spend* and *hold* are predefined.
- For every process all the procedures to access its implicit state are available.
- For every component type all *component control procedures* and *popul* procedures are predefined.
- For every evaluation object there exists a HIERARCHY all MERGE ... The dots are standing for a MERGE of all load filtering hierarchies ending at that evaluation object. Omitting the DUE TO-part in MEASURE statements means DUE TO all.

The result types and parameters (if any) of these predefinitions are listed in the following table:

name	type of result	name of parameter	type of parameter	comments
A.dimension	INT	-	-	
A.lower_bounds[]	INT	n	INT	array
A.upper_bounds[]	INT	n	INT	attributes
arrival_announce	REAL	-	-	
arrival_entry	REAL	-	-	
arrival_service	REAL	-	-	predefinitions
arrival_exit	REAL	-	-	for
arrival	REAL	-	-	services /
preempted	BOOL	-	-	processes
speed	REAL	-	-	
hold	-	t	REAL	… for
spend	-	t	REAL	services
popul_announce	INT	-	-	
popul_entry	INT	-	-	
popul_service	INT	-	-	
popul_exit	INT	-	-	predefinitions
popul	INT	-	-	for
accept	-	-	-	component types
schedule	-	-	-	
dispatch	-	-	-	
offer	-	-	-	

D.3.1.Predefinitions for Arrays

Each array of simple data type or record type has the attribute *dimension, lower_bounds* and *upper_bounds*, all of which can only be read. These attributes support the access to dynamic arrays and to arrays being surrendered as parameters. For instance, if arrays are defined as formal parameters, their dimension is not specified. Specification occurs when the current array object is handed over. Array attributes are accessed by dot notation, the attribute following the array name.

A.dimension RESULT INTEGER

Results the number of indices of array *A* which have to be supplied when accessing array elements.

A.lower_bounds [n: INTEGER] RESULT INTEGER

Results the lower bound of the *n´*th index of array *A*.

A.upper_bounds [n: INTEGER] RESULT INTEGER

Results the upper bound of the *n´*th index of array *A*.

D.3.2.Predefinitions in Services

For every service all implicit state procedures and the services *spend* and *hold* are predefined.

arrival_announce RESULT REAL

Returns the arrival time of this process at the *announce queue.*

arrival_entry RESULT REAL

Returns the arrival time of this process at the *entry area* (this value is also updated when a process is preempted, i.e., transferred from *service area* to *entry area*), or -1, if it has not yet arrived.

arrival_service RESULT REAL

Returns the arrival time of this process at the *service area* (this value is also updated when a process resumes its activities after a preemption), or -1, if it has not yet arrived.

arrival_exit RESULT REAL

Returns the arrival time of this process at the *exit area*, or -1, if it has not yet arrived.

arrival RESULT REAL

Returns the arrival time of this process at the component, i.e., the time of the first arrival at the entry area, or -1, if it has not yet arrived.

preempted RESULT BOOLEAN

Returns TRUE if the process has been preempted (transferred from *service area* to *entry area* by a rescheduling) at least once.

speed RESULT REAL

Returns the service speed most recently assigned to the process by a SETSPEED statement in a *dispatch* procedure, or -1, if this has not yet happened.

hold (T: REAL)

This service can only be called within (the statement part of) services. Here a call of *hold* means that a process following the service pattern is delayed for T time units (model time), independent of the *dispatch* procedure of the component. Service *hold* can be used to model, e.g., terminals, time outs, or timer interrupts.

hold may also be used if the model is to be solved analytically with restrictions listed in Appendix E. If T is negative within a simulation run, this is interpreted as *hold*(0) and a warning is generated.

spend (T: REAL)

This service can only be called within (the statement part of) services. Here, a call of *spend* requests the hosting component to give service for T units. This is not model time, since this request is dispatched. The *dispatch* procedure of this component can affect the "speed" of this service (see Section 4.2.2.).

Time consuming components comparable to ones of type *server* or *prioserver* can be defined by the user by using a call of *spend* in the provided service. Using *spend* within a service local to a model is equal to using *hold*, because a model has no *dispatch* procedure, i.e., uses *dispatch := equal (1.0)*.

The service *spend* may also be used if the model is to be solved analytically, with restrictions listed in Appendix E. If T is negative within a simulation run, this is interpreted as *spend*(0) and a warning is generated.

D.3.3.Predefinitions in Component Types

For every component type all *component control procedures* (*accept, schedule, dispatch, offer*) are predefined, but can be overridden in the type body or via corresponding actual component type parameters.

accept

Controls the transfer of processes from the *announce queue* to the *entry area* of the component.

schedule

Controls the transfer of processes between *entry* and *service area.* Processes in the *entry area* can be transferred to the *service area* (progress is granted) and vice versa (further progress is denied).

dispatch

Processes in the *service area* are further controlled by a *dispatch* procedure. The procedure assigns service speeds to processes.

offer

Processes in the *exit area* waiting to be accepted by the next component (providing the next requested service), but only those processes selected by an *offer* procedure are actually allowed to leave the component.

Moreover the *popul* procedures are predefined and automatically provided. They enable to determine the current number of all processes in a special component area. Additionally the OF operator can be used to retrieve population numbers of a specific service only. It takes a *popul* procedure and a *service-name* as arguments and yields a new restricted *popul* procedure.

popul_announce RESULT INTEGER

Returns the number of process references in the *announce queue*.

popul_entry RESULT INTEGER

Returns the number of processes in the *entry area.*

popul_service RESULT INTEGER

Returns the number of processes in the *service area.*

popul_exit RESULT INTEGER

Returns the number of processes in the *exit area.*

popul RESULT INTEGER

Returns the total number of processes in the component (=*popul_entry* + *popul_service* + *popul_exit*).

E. Restrictions for the HIT Solvers

This appendix lists all restrictions which must be fulfilled for the application of a specific HIT solver. It starts with a section for all analytical solvers including solver-specific subsections. The restrictions for the simulative solver conclude this appendix. Please see also Appendix G.1.3., describing solver information.

E.1. Restrictions for the Analytical Solvers

HIT offers the analytical solvers DOQ4, LIN2 and MARKOV. Most restrictions apply to all three solvers in common. Generally these restrictions are tested by the analytical code generator ACG, but partly the restrictions are not verified until the execution of the experiment.

E.1.1. Restrictions Common to all Analytical Solvers

These restrictions concern the allowed station types, the modelling of chains, allowed distributions, the use of control statements and other programming features and the allowed evaluations.

E.1.1.1. Stations

Service stations in the sense of this model class are described by definition of component objects of a predefined type. *Servers* can be used by all solvers, DOQ4 additionally allows *prioservers*, while MARKOV can additionally deal with *counters*, *ftservers* and *prioservers*. The use of further standard component types is not permitted.

Standard Type	DOQ4	LIN2	MARKOV	SIMUL
Server	x	x	x	x
Prioserver	x	x	x	x
Ftserver	-	-	x	-
Counter	-	-	x	x
others	-	-	-	x

The service strategies permitted have to be described with support of the component control procedures (see also Appendix F.3.). In general the procedure *dispatch* determines the service speed. State-independent service speeds are described by EQUAL (*speed*) or SHARED (*speed*), state-dependent service speeds are described by the procedures SDEQUAL (...) or SDSHARED (...). The allowed combinations of *dispatch* and *schedule* depend on the solver used and are therefore listed in the solver-specific parts below. However, only ALWAYS and ALL may be used as *accept*- or *offer*-procedure, respectively.

User-defined component types may contain or enclose stations and objects of other user-defined component types. They have to fulfil all restrictions listed in the solver-specific sections. For objects of user-defined component types only the default values for *accept, schedule* and *offer* are allowed. For *dispatch* all standard procedures can be used. However, for objects of aggregated (user-defined) component types, no specifications concerning any control procedure are allowed.

E.1.1.2. Chains

Chains in the sense of this model class are defined by services together with the creation of process objects through the CREATE statement.

Closed chains have to be described by

```
CREATE n PROCESS service-name;
```

The outermost control statement of the service has to be a loop without the possiblility of termination (the infinite LOOP...END LOOP), in order to keep the customers permanently in the system, but none of the used services may contain an infinite loop.

Open chains have to be described by

```
CREATE 1 PROCESS service-name EVERY negexp (arrival_rate);
```

The corresponding service must not contain an infinite loop, and none of its used services (and the used services of the latter etc) may contain an infinite loop.

The arrival rate has to be either larger than, or equal to zero. Declarations as for instance SUBMIT statements and declarations of process names are not possible. Restrictions by LIMIT may only be used for MARKOV.

E.1.1.3. Distributions

Allowed distributions for describing the required amount of service time are

- *cox* (rate, coefficient of variation)
- *coxg* (description of phases)
- *negexp* (rate)

When requesting services from components which have a non-predefined type the functions *cox*, *coxg* and *negexp* are not allowed within actual parameter expressions. In this case, only constants, parameters and terms over these are permitted.

E.1.1.4. Control Statements

Within the bodies of services, only the following control statements are allowed:

- LOOP...END LOOP (infinite loop)
- LOOP...END LOOP UNTIL draw(prob) (UNTIL loop)
- WHILE draw(prob) LOOP...END LOOP (WHILE loop)
- AVERAGE n TIMES LOOP...END-LOOP (TIMES loop)
- IF draw(prob) THEN...ELSE...END IF (IF statement)
- BRANCH-END BRANCH (BRANCH statement)
- OPEN_CHAIN-END OPEN_CHAIN (OPEN_CHAIN statement)
- CLOSED_CHAIN-END CLOSED_CHAIN (CLOSED_CHAIN statement)

An infinite loop (LOOP...END LOOP) is only allowed in the case of defining a closed chain, and may at most occur once in a service.

The probabilities *prob* within the control statements above must be greater than 0.0 and lower than 1.0. In some cases the values 0.0 and 1.0 are allowed as well, depending on the structure of the chain and the requested analytical method. Moreover, service parameters are not allowed within the expression *prob*, but parameters of the enclosing component type are allowed. Thus service parameters are only useful to define service requests, not to define routing. All these restritions are made to achieve that the "structure" of a modelled chain cannot be modified stochastically or by service parameters.

E.1.1.5. Programming Features

In all model elements (model types, component types, services), only the declaration of constants is allowed. Variables, procedures and value assignments are not permitted. Consequently, services may not possess RESULT parameters. All statements for file and text handling (READ, READLN, WRITE, WRITELN, OPEN, CLOSE) are forbidden. Note that these restrictions do not apply for the experiment specification.

E.1.1.6. Evaluations

No user-defined streams are permitted, and consequently, neither are UPDATE statements and COLLECT blocks. Only the standard streams POPULATION, THROUGHPUT, TURNAROUNDTIME and UTILIZATION are allowed. The evaluation of component areas is not permitted.

Within a CONTROL statement, no evaluation object may be referenced. Only CPUTIME and ACCURACY are allowed (singular or connected by OR) as termination criteria. CPUTIME is ignored by fast solvers (DOQ4, LIN2), while ACCURACY is ignored by exact solvers (DOQ4).

As an estimator for performance measurements, only the declaration of MEAN is permitted. FREQUENCY is ignored, while STANDARDDEVIATION, CONFIDENCE LEVEL and BOUNDS are transformed to MEAN. In this cases, a warning is given.

Estimator	DOQ4	LIN2	MARKOV	SIMUL
MEAN	x	x	x	x
BOUNDS	mean	x	mean	mean
FREQUENCY	ignored	ignored	ignored	x
STANDARDDEVIATION	mean	mean	mean	x
CONFIDENCE LEVEL	mean	mean	mean	x

For the declaration of experiments and for the control of series of evaluations (in contrast to the declaration of model elements), the complete extent of the language HI-SLANG is permitted.

E.1.2.Further Restrictions for DOQ4

The analytic-algebraical solver DOQ4 allows the exact analysis or aggregation of separable networks as well as the approximative analysis or aggregation of non-separable networks (see /MuNS85/, /KLMN88/). Verification of part of the restrictions does not occur until the experiment is executed. If possible, the exact solver is applied. Otherwise the approximate solver is automatically selected at run time.

In the following, the restrictions relating to the solvable class of models of the DOQ4 solver are listed. Moreover, the restrictions listed in E.1.1 apply.

E.1.2.1. Stations

Service stations in the sense of this model class are described by definition of component objects of the type *server* or *prioserver*. The service strategies permitted have to be described with support of the component control procedures (see also Appendix F.3.). The following combinations are allowed:

server strategy:	*schedule*	*dispatch*
FCFS:	FCFS (1)	(SD)EQUAL or (SD)SHARED
RANDOM:	RANDOM(1)	(SD)EQUAL or (SD)SHARED
LCFSPR:	LCFSPR	(SD)EQUAL or (SD)SHARED
PS:	IMMEDIATE	(SD)SHARED
IS / NODELAY:	IMMEDIATE	(SD)EQUAL

prioserver strategy:	*schedule*	*dispatch*
PREEMPTIVE REPEAT:	PRIOPREP	(SD)EQUAL
NONPREEMPTIVE:	PRIONP	(SD)EQUAL
FCFS:	FCFS (1)	(SD)EQUAL or (SD)SHARED
RANDOM:	RANDOM(1)	(SD)EQUAL or (SD)SHARED
LCFSPR:	LCFSPR	(SD)EQUAL or (SD)SHARED
PS:	IMMEDIATE	(SD)SHARED
IS / NODELAY:	IMMEDIATE	(SD)EQUAL

E.1.2.2. Chains

Open chains in models with state-dependent stations are allowed, unless aggregated stations with more than one provided service are visited by open chains.

E.1.2.3. Distributions

When the exact analytic algebraic solution methods are applicable and thus automatically selected, *cox* is transformed to *coxg* by selecting a set of rates and phase probabilities consistent with the parameters of *cox*. The coefficient of variation may not be negative. All (phase description-)rates have to be larger than zero.

E.1.2.4. Evaluations

There are no further restrictions concerning evaluations.

E.1.2.5. Aggregation and Use of Aggregates

The component type to be aggregated and its provided services have to be without parameters. The aggregate produced can be embedded into models which can then be evaluated by an arbitrary solution method if certain restrictions are obeyed. These restrictions are listed in the corresponding sections of this appendix.

By applying DOQ4, aggregated component types can only be used

- within component types, which themselves can be aggregated,
- within models containing only closed chains,
- and within models with open and closed chains, if the services of the aggregated component type are only used by closed chains.

At this point, it has to be said that aggregated component types may have been originated from approximate aggregation. The performance measurements determined during the use of the aggregate are generally also approximate. At present, no corresponding warning is given if such an aggregate is used.

E.1.2.6. Algorithm Selection

E.1.2.6.1. Exact Analysis

The exact algorithm is only applicable and used, if

- neither schedule procedure PRIOPREP nor PRIOPNP are used,
- the distribution of the needed amount of service in the strategy FCFS or RANDOM is *negexp* with equal service rates for all service requests.

Otherwise the approximative algorithm is used.

E.1.2.6.2. Approximative Analysis

When utilizing the approximative algorithm, the service requests have to satisfy the following conditions:

- At stations with strategy RANDOM, the distribution of the required amount of service has to be *negexp* with equal service rates for all service requests. This also applies for state-dependent stations with strategy FCFS.
- At state-independent stations with strategy FCFS, the service rates and the coefficients of variation of the required amount of service have to be equal for all service requests of one chain. Service requests of different chains yet may possess different rates and coefficients of variation, in particular, they need not be distributed *negexp*.
- At strategy PRIOPREP and PRIONP, the distribution of the required amount of service has to be *negexp* with equal service rates and priorities for all service requests of one chain. The priorities of service requests of different chains may not coincide.

E.1.2.6.3. Remarks on Scaling

In the course of evaluating the performance measurements desired by the user, the solver DOQ4 determines the relative visit of the service stations for the normalizing constants as intermediate results.

Here, due to the arrival and service rates determined by the user within the model, scaling problems could occur. These could cause an under- or overflow in the evaluation of the normalizing constants.

Principally, scaling methods determining a scaling factor for every normalizing constant to be calculated could be used here, but this would cause the solver to occupy twice the amount of storage and time. DOQ4 does not contain such a scaling method. If under- or overflow problems occur, changes in the description of services (e.g., reordering sequences of statements in services) may help.

E.1.3.Further Restrictions for LIN2

With support of the approximate analytic-algebraical solver LIN2, models of the class of "large" separable queuing systems can be solved (/Knau88/). Furthermore, the solver LIN2 permits the calculation of exact lower and upper bounds (performance bounds) for the mean values of the desired performance measurements (see also /KnND89/).

In the following, the restrictions relating to the solvable class of models of the LIN2 solver are listed. Moreover the restrictions listed in E.1.1 apply.

E.1.3.1. Stations

Service stations in the sense of this model class are described by the definition of component objects of the type *server* or *prioserver*. The service strategies permitted have to be described with support of the component control procedures (see also Appendix F.3.). The following combinations are allowed:

server strategy:	*schedule*	*dispatch*
FCFS:	FCFS (1)	(SD)EQUAL or (SD)SHARED
RANDOM:	RANDOM	(SD)EQUAL or (SD)SHARED
LCFSPR:	LCFSPR	(SD)EQUAL or (SD)SHARED
PS:	IMMEDIATE	(SD)SHARED
IS / NODELAY:	IMMEDIATE	(SD)EQUAL

prioserver strategy:	*schedule*	*dispatch*
RANDOM:	RANDOM(1)	(SD)EQUAL or (SD)SHARED
PS:	IMMEDIATE	(SD)SHARED
IS / NODELAY:	IMMEDIATE	(SD)EQUAL

E.1.3.2. Chains

Open chains may not use any service provided by state dependent stations.

E.1.3.3. Distributions

When the approximative analytic-algebraical solver is applied, *cox* is transformed to *coxg* by selecting a set of rates and phase probabilities consistent with the parameters of *cox*. The coefficient of variation may not be negative. All (phase description-)rates have to be larger than zero.

When using the strategy FCFS or RANDOM, the distribution has to be *negexp* with equal service rates for all service requests.

E.1.3.4. Evaluations

As an estimator for performance measurements, only the declaration of MEAN or BOUNDS (see below) is permitted. For a CONTROL statement in connection with estimator BOUNDS see below.

E.1.3.5. Aggregation and Use of Aggregates

The aggregation of component types by LIN2 is not possible, meaning the AGGREGATE statement is not permitted.

Neither is the use of aggregated component types with more than one provided service permitted. Because of the speed of the solver LIN2 the original component type can be used instead of the aggregated component type, if possible.

E.1.3.6. Algorithm Selection

When using the solver LIN2, the calculation of exact upper and lower bounds for the mean values of the desired performance measurements (performance bounds) is only possible if ESTIMATOR BOUNDS is demanded within the corresponding MEASURE statements. In addition to the restrictions specified above, calculation of bounds is only possible if

- all state-dependent stations have monotonously increasing speed functions

  or if

- every closed chain has at least one state-independent station with service strategy IS with "sufficiently" high relative occupation.

The mean values furthermore are determined by the "normal" LIN2 approximation.

The ACCURACY statement has the following effects:

- A performance bounds algorithm is selected, if ACCURACY is set > 0. Even if no estimator BOUNDS occurs in the source, but ACCURACY is > 0, the approximate evaluation of mean values may be influenced.

- With the stop condition ACCURACY within the CONTROL statement, the user can take influence on the quality of the bounds which are to be calculated. Only values less than or equal 4 are valid concerning accuracy. Although the definition of larger values implies the calculation of closer bounds, it occurs at a higher effort than with lower values. Depending upon the size of the investigated model, (number of closed chains), LIN2 reduces the accuracy to hold the effort of computation within limits. In these cases, a warning is given.

  Real values used as accuracy are rounded to INTEGER values. If the value is lower than 0.5 or if the ACCURACY stop condition does not exist no bounds are calculated.

  The stop conditions CPUTIME is ignored by LIN2 in calculation of "performance bounds".

E.1.4.Further Restrictions for MARKOV

The analytic-numerical solver (MARKOV) allows the analysis of general, not separable queueing networks (/MuRo87/). In the following, the restrictions relating to the solvable class of models of the MARKOV solver are listed. Moreover the restrictions listed in E.1.1 apply.

E.1.4.1. Stations

Service stations in the context of this model class are described by definition of component objects of the type *server*, *prioserver*, *ftserver* and *counter*. The allowed service strategies have to be described using the component control procedures (see also Appendix F.3.). The following combinations are allowed:

server strategy:	*schedule*	*dispatch*
RANDOM:	RANDOM (1)	(SD)EQUAL or (SD)SHARED
PS:	IMMEDIATE	(SD)SHARED
IS / NODELAY:	IMMEDIATE	(SD)EQUAL

prioserver strategy:	*schedule*	*dispatch*
RANDOM:	RANDOM (1)	(SD)EQUAL or (SD)SHARED
PS:	IMMEDIATE	(SD)SHARED
IS / NODELAY:	IMMEDIATE	(SD)EQUAL
PREEMPTIVE REPEAT:	PRIOPREP	(SD)EQUAL or (SD)SHARED
NONPREEMPTIVE:	PRIONP	(SD)EQUAL or (SD)SHARED

ftserver strategy:	*schedule*	*dispatch*
RANDOM:	RANDOM (1)	(SD)EQUAL or (SD)SHARED
PREEMPTIVE REPEAT:	PRIOPREP	(SD)EQUAL or (SD)SHARED
NONPREEMPTIVE:	PRIONP	(SD)EQUAL or (SD)SHARED

Although the RANDOM parameter has to have value 1 for *ftserver*, it will actually be redefined by the value of the parameter *processors* of the component type *ftserver.*

counter strategy:	*schedule*	*dispatch*
RANDOM:	CRANDOM	ignored
PRIORITY:	CPRIO	ignored

For objects of type *counter*, only CRANDOM (random choice of one waiting service call) and CPRIO (choice of the service call with the highest priority) are allowed as *schedule* procedure. The *dispatch* procedure is of no significance in this case.

The following restrictions have to be taken into account when employing the component type *counter*:

- If the *counter* object uses the *schedule* procedure CRANDOM as a parameter, the state vector may only have one dimension and all *change* requests (= parameters of the corresponding service calls) have to be identical in regard of their absolut values.

- If the *counter* object has the *schedule* procedure CPRIO as a parameter, all priorities should be different. If two or more service calls have the same priority, the solver internally assigns distinctive priorities.

- Service calls may not be blocked at a *counter* object due to capacity restrictions of one of the following components. Consequently, if the following component uses RESTRICT (...), the capacity of a succeeding component may not be exhausted when the preceding *counter* service call terminates.

- Only ALWAYS and ALL may be used as *accept* or *offer* procedures, respectively.

Component objects of limited capacity are defined by the *accept* procedure RESTRICT (...). Apart from RESTRICT (...), only ALWAYS and ALL may be used as *accept* or *offer* procedures, respectively. RESTRICT (…), however, has a different meaning from the *accept* procedure LIMITED (...) in the case of a simulative solver: If the capacity of a component is exhausted, the admission of further services is not possible. Services calling for service are rejected and repeat the service (or the last phase of service in the case of *cox* service time distribution with several phases) at the last visited component.

E.1.4.2. Chains

The population of open chains always has to be limited. Thus open chains have to be described by

```
CREATE 1 PROCESS service-name LIMIT n EVERY negexp (arrival_rate);
```

E.1.4.3. Distributions

When using *cox*, the coefficient of variation should not lie under the value of 0.32, since such service requests would be transformed to a *cox* distribution with more than 10 phases. Here, however, only 10 phases are assigned for this transformation, and therefore the distribution of phases should explicitly be defined with support of *coxg*.

E.1.4.4. Evaluations

Within a CONTROL statement no evaluation object may be defined, and only CPUTIME and ACCURACY are eligible as stop conditions (separately or connected by OR). These, however, refer only to the CPU time necessary for solving the system of equations and to the accuracy of the iterative solution, respectively.

Reasonable values for ACCURACY should lie between zero and one. They are interpreted as accuracy of the numerical solution in percent, whereas the accuracy refers to the relative error of the calculated state probabilities of the underlying Markov chain. The attained accuracy is verified by estimation.

E.1.4.5. Aggregation and Use of Aggregates

The aggregation of component types by MARKOV is currently not possible, and the AGGREGATE statement, consequently, is not permitted. However, aggregates can be used.

E.1.4.6. Algorithm Selection

The solver MARKOV determines performance measures in five steps.

1. The model data is transformed and a model structure is determined heuristically to partition the state space in the following steps.

2. The state space is generated.

3. The generator matrix is build up.

4. The system of linear equations is solved.

5. Basic performance measures are computed using the stationary probability distribution of the states.

The algorithm selection is related to step 4, the solution of the systems of linear equations. One of the following three algorithms can be selected: a direct algorithm (Grassmann algorithm), an iterative algorithm (Gauss-Seidel) combined with aggregation/disaggregation (a/d) steps and a point iteration algorithm (SOR).

The direct algorithm solves the equations with a fixed number of operations, but is not applicable to large systems because of its computation and storage requirements. A measure of the size of a system is the product of order and bandwidth of the matrix.

The Gauss-Seidel algorithm with a/d steps is chosen if the number of macro states is in an acceptable domain, otherwise the SOR algorithm with an heuristical determination of a relaxation parameter is selected.

Note that the algorithm selection uses results of the previous steps 1-3.

E.2. Restrictions for the Simulative Solver

The simulative solver SIMUL allows for the analysis of nearly all models specifyable via HI-SLANG, but compared to analytical solvers simulation is very time-consuming and the results are only approximate. Thus analytical solvers should be applied whenever possible.

If an analytic-algebraical HI-SLANG source code is (e.g., due to extensions made in later analysis steps) to be analyzed with the simulative solver, the following modifications are necessary:

- METHOD ANALYTICAL "..." has to be replaced by METHOD SIMULATIVE.
- A CONTROL statement has to be added or the given CONTROL statement has to be modified in order to fulfill the requirements of simulation (i.e., evaluation objects should be supplied for control). An ACCURACY declaration within the CONTROL statement is ignored.

Generally, when changing from an analytical to a simulative solver, not only mean values (estimator MEAN ...) but also confidence intervals (estimator CONFIDENCE ...) should be demanded for reasons of statistical certainty. Mean value and standard deviation are then automatically computed and given out. Fading out the transient phase is also reasonable. The declaration of estimator BOUNDS is ignored by the simulative solver.

E.2.1. Restrictions on Modelling

Almost all analytically solvable models can be solved simulatively, with the following exceptions:

- The *accept* procedure RESTRICT can be used only analytic-numerically and not for the simulative solver, since no implementation is possible. The component control procedure, registered in the HIT standard mobase, only supplies a run time error message and stops the evaluation run.
- The component type *ftserver* can only be used analytic-numerically since the results would be questionable at least. The HI-SLANG source code, filed in the HIT standard mobase, only contains the interface of the component type *ftserver* and of its provided service *request*. Using it in the simulative case would entail all service execution of this *ftserver-request* without time; a warning will be additionally supplied.
- When using the component type *prioserver* with the *schedule* procedure PRIOPREP for a simulative solver, attention has to be paid to the circumstance that the next service in execution, if he has been preempted before, is carried out with exactly the same service demand (identical), while using PRIOPREP in the case of the numerical solver, the service demand is drawn anew (resampling). Although both strategies are close together this can lead to different results in extreme cases. Another, different implementation is neither possible for the simulative nor for the numerical solver.

E.2.2.Restrictions on Evaluation

When actualizing a user-defined stream with the UPDATE statement, the following points are of importance:

- In the case of the stream type STATE, the given value is interpreted as the difference to the last observed value. Thus the change of the state is specified.
- In the case of the stream type EVENT, the given value is interpreted as the value of the observed value itself.
- In the case of the stream type COUNT, the given value is 1, therefore higher numbers are attained only with several UPDATE statements.

The estimator MEAN for a stream of the type STATE results from the average of the observed values, weighted with the respective amount of time being observed. For a stream of the type EVENT this equals the average of the observed values. Regarding a stream of the type COUNT, the estimator results from the quotient of the number of counted events and the full length of the time interval of the observation.

The estimator CONFIDENCE INTERVAL refers to the mean value estimator for all three types of streams.

The estimator STANDARD DEVIATION refers to the mean value for STATE and EVENT streams and to the time interval between the events for COUNT streams.

E.2.3.Aggregation and Use of Aggregates

A simulative aggregation (AGGREGATE … END AGGREGATE) is not possible, whereas the use of an aggregate is legitimate. Only ACCEPT and OFFER control procedures can be specified for aggregates.

F. The HIT Standard Modelling Base

This appendix describes the objects (members) of the HIT standard mobase. Working on mobases by means of HIT-OMA is explained in /Weis92b/. The possibilities of access to mobases by HI-SLANG are treated in Chapter 8.

The HIT standard mobase contains on the one hand HI-SLANG sources and on the other hand predefined component control procedures, written in Standard SIMULA. Correspondingly, this appendix is divided into three sections: The first two concentrate on the explanation of the HI-SLANG sources (predefined component types and services), the third treats the predefined SIMULA component control procedures.

There are certain restrictions for the application of the mobase objects. They are indicated in the following way throughout this appendix:

(∗) implies: application possible for all HIT solvers

(S) implies: only allowed for METHOD SIMULATIVE

(M) implies: only allowed for METHOD ANALYTICAL "MARKOV"

(D) implies: only allowed for METHOD ANALYTICAL "DOQ4"

(L) implies: only allowed for METHOD ANALYTICAL "LIN2"

All but the first mark may occur in combinations, e.g.,

(S,M) means: applicable for the simulative and the Markov solver.

F.1. Standard Component Types

The following table gives an overview of all standard component types of HI-SLANG and their provided services. Moreover the default for the component procedure *schedule* is given. Here *hi-slang* means that the default procedure is contained in the component type, formulated in HI-SLANG. The defaults for *accept*, *dispatch* and *offer* are ALWAYS, EQUAL(1.0) and ALL, respectively. As before, the method range of application is denoted by (S), (M), (D), (L) and (*) marks.

Component Type	Allowed Methods	Provided Services				Schedule Default
Server	*	request				IMMEDIATE
Counter	S,M	change				CRANDOM
Semaphor	S	p	v			SEMSCHED
Tokenpool	S	allocate	release	destroy	produce	TOKSCHED
Synchsend	S	send	receive			hi-slang
Nowaitsend	S	send	receive			hi-slang
Ftserver	M	request				PRIONP
Prioserver	S,M,D	request				PRIONP
Observer	S	-				IMMEDIATE

These types are described in detail on the next pages. Before that the following table lists all component control procedures that are admissable for those types:

Control Procedure	Component Type										
	Server	Counter	Sema-phor	Token-pool	Synch-send	Nowait-send	Ft-server	Prio-server	Oserver	Aggre-gate-type	others
ALWAYS	**D**	**F**	**F**	**F**	**F**	**F**	**D**	**D**	**F**	**D**	**D1**
LIMITED	O	-	-	-	-	-	-	O	-	O	O
RESTRICT	O	-	-	-	-	-	O	O	-	-	-
hi-slang	-	-	-	-	-	-	-	-	-	-	**D2**
CPRIO	-	O	-	-	-	-	-	-	-	-	-
CRANDOM	-	**D**	-	-	-	-	-	-	-	-	-
FCFS	O	-	-	-	-	-	-	-	-	-	O
IMMEDIATE	**D**	-	-	-	-	-	-	**O**	**F**	**F**	**D1**
LCFS	O	-	-	-	-	-	-	-	-	-	O
LCFSPR	O	-	-	-	-	-	-	-	-	-	O
PRIONP	-	-	-	-	-	-	**D**	**D**	-	-	-
PRIOPREP	-	-	-	-	-	-	O	O	-	-	-
PRIOPRES	-	-	-	-	-	-	-	O	-	-	-
RANDOM	O	-	-	-	-	-	O	O	-	-	O
SEMSCHED	-	-	**F**	-	-	-	-	-	-	-	-
TOKSCHED	-	-	-	**F**	-	-	-	-	-	-	-
hi-slang	-	-	-	-	**F**	**F**	-	-	-	-	**D2**
AGGRDISP	-	-	-	-	-	-	-	-	-	**F**	-
EQUAL	**D**	**F**	**F**	**F**	**F**	**F**	**D**	**D**	**F**	**-**	**D1**
SDEQUAL	O	-	-	-	-	-	O	O	-	-	O
SDSHARED	O	-	-	-	-	-	-	O	-	-	O
SHARED	O	-	-	-	-	-	-	O	-	-	O
hi-slang	-	-	-	-	-	-	-	-	-	-	**D2**
ALL	**D**	**F**	**F**	**F**	**-**	**F**	**D**	**D**	**F**	**D**	**D1**
hi-slang	-	-	-	-	**F**	**-**	-	-	-	-	**D2**

Legend: O: optional, - : illegal,
F: fixed, setting this control procedure makes no sense,
D: default,
D1: default if no HI-SLANG is given in the type,
D2: default if HI-SLANG is given in the type

F.1.1. Server

The component type *server* (which can be used with all solvers) provides a base service *request* with the parameter *amount* of the type REAL. The parameter *amount* specifies the amount of work to be done by the *server*, creating the possibility of modelling time consumptions with aid of a *server* and the service *request* it provides. In HI-SLANG notation the component type *server* has the following structure:

```
TYPE server COMPONENT;
    PROVIDE
        SERVICE request     (amount : REAL);
    END PROVIDE;

    TYPE request SERVICE    (amount : REAL);
    BEGIN
        spend (amount);
    END TYPE request;
END TYPE server;
```

In accordance to the other component types, the type *server* has the predefined parameters *accept, schedule, dispatch* and *offer*, and provides as a standard the functional procedures *popul, popul_announce, popul_entry, popul_service* and *popul_exit.*

Please note that the component type *server* can´t be redefined in HI-SLANG since the compiler recognizes the *server* as a standard.

At the point of declaration, objects of the type *server* can be provided with the standard procedures for component control as parameters, initiating a possibility to imitate strategies such as "processor sharing" or "infinite server", which are important for modelling:

Example:

```
COMPONENT       bus :    server (LET dispatch := shared);
              terminal :  server (LET dispatch := equal);
```

F.1.2. Counter

The component type *counter* (Simulative and Markov only), which is used for modelling space consumption, serves as a counter part to the component type *server*, allowing the modelling of time consumption. Objects of this type can be interpreted as passive servers. Important possibilities of application are for example the realization of a semaphore, of a tokenpool or of mechanisms for synchronization. In the case of simulation, though, components more efficient under the aspect of run time can be employed. These will be explained later.

A component of the type *counter* provides the change of a state vector as a service:

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

  VARIABLE
    state_vector : ARRAY [init.lower_bounds[1] ..
                          init.upper_bounds[1]] OF INTEGER;


  TYPE change SERVICE (  amount   : ARRAY OF INTEGER;
                         prio     : INTEGER DEFAULT 32767);


  PROCEDURE change_state (amount : ARRAY OF INTEGER);

  VARIABLE run : INTEGER;

  BEGIN
    FOR run :=  state_vector.lower_bounds[1] STEP 1 UNTIL
                state_vector.upper_bounds[1]
    LOOP
      state_vector[run] := state_vector[run] + amount[run];
    END LOOP;
  END PROCEDURE CHANGE_STATE;

  BEGIN
    change_state (amount);
  END TYPE change;

  PROCEDURE init_state_vector (init : ARRAY OF INTEGER);
  VARIABLE run : INTEGER;
  BEGIN
    FOR run :=  state_vector.lower_bounds[1] STEP 1 UNTIL
                state_vector.upper_bounds[1]
    LOOP
        state_vector[run] := init[run];
    END LOOP;
  END PROCEDURE init_state_vector ;

BEGIN
  IF init.dimension<> 1 THEN
    stop_evaluation ("Illegal actual parameter for INIT at COUNTER.");
  ELSE
    init_state_vector (init);
  END IF;
END TYPE counter;
```

The lower and upper limits of the state vector and its initial value are given by the parameters *min*, *max*, and *init*, respectively. Changes of the state vector achieved by *change* occur without consuming time. Waiting situations may arise if *change* calls are temporarily not allowed. CRANDOM (default) and CPRIO are allowed as values for the component control procedure *schedule* in order to solve a waiting situation.

To illustrate the application, the realization of a binary semaphore is presented here, the component type *counter* being introduced by a %COPY statement to the HI-SLANG program. Calls of *change ([+1])* and *change ([-1])* correspond to the known *v* and *p* semaphore operations:

Example:

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

Note that this implementation of the semaphore has the property that *v* is blocking in the case that more *v*- than *p*-operations have been executed: *p* and *v* are symmetric.

Services intending to change the state vector cannot do this before a situation occurs in which the permitted limits (between *min* and *max*) are not exceeded by the change. If the change is allowed, it happens without time consumption. Otherwise the service has to wait until the change is allowed. Waiting services with legitimate changes are either selected according to their priority by *LET schedule := cprio*, or randomly by *LET schedule := crandom*. Corresponding to the *prioserver* (see below), the priorities here are also assigned at the point of the service call (*prio* is a parameter of the provided service *change*).

The following restrictions are of importance in the case of applying the numerical solver:

When using the *schedule* procedure CRANDOM, which is the default, only a state vector with one dimension, in other words a state variable, is allowed. Furthermore for the Markov solver all changes of this state variable should be identical regarding their absolute value. If, applying the procedure CPRIO, two or more service calls have the same priority, the solver internally gives out distinctive priorities.

Only ALWAYS and ALL may be used as *accept* or *offer* procedures, respectively. *Change* requests to a counter object may not be blocked because of capacity restrictions of a following component. Consequently, when using RESTRICT (...) on the component following a *counter* object, the capacity of this component may not be exhausted at the moment a service call of the preceding *counter* object terminates.

F.1.3. Semaphor

In the case of simulation only, a more efficient component type, regarding time consumption, can be employed as semaphore. It is named *semaphor* (note the missing 'e' at the end), provides the (parameterless) services *p* and *v* and has a formal INTEGER parameter *sem_init*, to initialize a *semaphor*. The default value of *sem_init* is "1", implying that the default of this component type realizes a binary semaphore. This is not completely right, since 1 is only the default value and can arbitrarily be incremented by successive *v*-calls.

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

    VARIABLE sem : INTEGER DEFAULT sem_init;

    TYPE p SERVICE;
    BEGIN
        sem := sem - 1;
    END TYPE p;

    TYPE v SERVICE;
    BEGIN
        sem := sem + 1;
    END TYPE v;
END TYPE semaphor;
```

When declaring the component object, no assignments should be done concerning the four component control procedures, since only the internally defined default values are meaningful.

The component type *semaphor* has to be notified by %COPY to a HI-SLANG program.

Example:

```
%COPY "SEMAPHOR"
...
COMPONENT           bin_sem : semaphor;
                 common_sem : semaphor (LET sem_init := 3);
...
TYPE serv SERVICE;
    USE
        SERVICE passeer; verlaat;
        ...
    END USE;

BEGIN
    passeer;
    ...          { critical section }
    verlaat;
END TYPE serv;

REFER serv, ... TO  bin_sem, ... EQUATING
    serv.passeer    WITH bin_sem.p;
    serv.verlaat    WITH bin_sem.v;
END REFER;
```

F.1.4. Tokenpool

As a further variant of modelling space-consumption in the simulative case, HI-SLANG provides the component type *tokenpool* (Simulative only). Tokenpool is a modelling element, known for example by RESQ (see also /SaMN84/). A tokenpool contains a number of objects (tokens). Processes can take access to these objects by certain functions (services). Tokens may be reserved exclusively (*allocate*) and can be set free again in the following (*release*). Newly created objects can be added to the pool (*produce*) and finally, objects may be destroyed (*destroy*). The tokens within the pool cannot be identified individually. The services have an INTEGER parameter named *number* which declares how many tokens should be reserved, released, created or destroyed:

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

  VARIABLE tokens : INTEGER DEFAULT no_of_tokens;

  TYPE allocate SERVICE (number : INTEGER);
  BEGIN
    tokens := tokens - number;
  END TYPE allocate;

  TYPE release SERVICE (number : INTEGER);
  BEGIN
    tokens := tokens + number;
  END TYPE release;

  TYPE destroy SERVICE (number : INTEGER);
  BEGIN
    tokens := tokens - number;
    max_avail_tokens := max_avail_tokens - number;  {used in TOKSCHED}
  END TYPE destroy;

  TYPE produce SERVICE (number : INTEGER);
  BEGIN
    tokens := tokens + number;
    max_avail_tokens := max_avail_tokens + number;  {used in TOKSCHED}
  END TYPE produce;
END TYPE tokenpool;
```

The component type *tokenpool* has an INTEGER parameter *no_of_tokens* which defines the initial number of available tokens. No assignments should be done concerning the four component control procedures, since only their default values are meaningful. The special built-in schedule procedure TOKSCHED cannot be displayed here. It causes calls of *allocate* and *destroy* to wait, if there are not enough tokens available.

The *tokenpool* is very convenient for instance for the plain modelling of main storage: A token here represents a page of the main storage. The component type *tokenpool* has to be introduced to a HI-SLANG program by %COPY:

Example:

```
%COPY "TOKENPOOL"
...
 COMPONENT memory : tokenpool (LET no_of_tokens := 1024);
```

The following service calls can then be made:

allocate (5)	:	Five tokens are reserved. If they are not available at the moment, they are reserved as soon as an appropriate number of tokens has been set free by *release* or has been created by *produce.*
release (5)	:	Five tokens are released.
produce (1)	:	One new token is created.
destroy (7)	:	Seven tokens are destroyed. If they are not available at the moment, they are destroyed as soon as an appropriate number of tokens has been set free by *release* or has been created by *produce.*

You will get a warning if more tokens were released than allocated.

F.1.5. Synchsend

The parameterless component type *synchsend* (usable only in simulation) enables processes to communicate with each other, that means, to exchange messages. A component object of the type *synchsend* allows communication between two processes in one direction, one of the processes acting as the sender, the other as the receiver. If the communication is to occur in both directions or if more than two processes are to exchange messages, several component objects of the type *synchsend* or even an array of components should be declared.

In process communication by *synchsend*, sender and receiver synchronize each other by exchanging messages:

In the first case (sender "arrives" prior to receiver) the sender waits for the receiver to make sure that the receiver actually gets the message, in the other case (receiver "arrives" prior to sender) the receiver naturally waits until the sender produces the message and makes it accessible to him. Due to this kind of synchronization between sender and receiver, no additional buffer is necessary for message exchange, because the sender does not produce and send messages "on store". A text variable is used to give the receiver a copy of the message.

Synchsend provides the services *send* and *receive*: the sender calls *send* if he wants to transmit a message. The actual parameter contains the text of the message (name of the formal parameter : *what*). On the other side the receiver calls *receive* and gets the text of the message as RESULT. The text handling in HI-SLANG (treated in Section 3.6.) allows the construction and processing of message of a complex structure.

According to this pattern the user may define a *synchsend* component type operating on an arbitrary record type instead of text.

```
TYPE synchsend COMPONENT;
  PROVIDE
  SERVICE   send   (what : TEXT);
            receive RESULT TEXT;
  END PROVIDE;

  CONTROL

    PROCEDURE schedule;
    BEGIN
      IF POPUL_SERVICE = 0 THEN
        IF last_service = 'R' THEN
        IF POPUL_ENTRY OF send > 0 AND POPUL_ENTRY OF receive > 0 THEN
          INSPECT ENTRY_AREA WHILE last_service = 'R' LOOP
          WHEN SEND :  SELECT;
                       last_service := 'S';
          END LOOP;
          END IF;
        ELSE
          INSPECT ENTRY_AREA WHILE last_service = 'S' LOOP
          WHEN RECEIVE : SELECT;
                         last_service := 'R';
          END LOOP;
        END IF;
      END IF;
    END PROCEDURE SCHEDULE;

  END CONTROL;
```

```
    VARIABLE   buffer        : TEXT;
               last_service  : CHARACTER DEFAULT 'R';

    TYPE send SERVICE (what : TEXT);
    BEGIN
        buffer := what;
    END TYPE send;

    TYPE receive SERVICE RESULT TEXT;
    BEGIN
        RESULT buffer;
        buffer := "";
    END TYPE receive;

END TYPE synchsend;
```

Synchsend has to be copied (by a %COPY statement) when being used. No setting of control procedures make sense in the declaration of objects of the type *synchsend*:

Example:

```
%COPY "SYNCHSEND"
...
COMPONENT commun : synchsend;
```

sender:

```
. { produce message }
.
send (message);
.
.
```

receiver:

```
.
.
message := receive;
.
. { process message }
```

F.1.6. Nowaitsend

The communication between processes allowed by the component type *nowaitsend* (usable only in simulation) is widely comparable to that of *synchsend*. Therefore only the differences are explained here.

In contrast to *synchsend*, the sender does not wait for the receiver to collect the message but may go on ("no-wait-send") and produce and send further messages as long as the space of the buffer is not exhausted. If it is, the sender is forced to pause until the collection of a message by the receiver provides a new vacancy in the buffer.

The buffer is written and read as a ring buffer, i.e., the receiver always processes the oldest message first. Naturally the receiver has to wait for the sender if no message is available to be processed, in other words if the buffer is empty.

The buffer is represented by a TEXT array. The size of the array (number of buffer cells) is defined by the INTEGER parameter *no_of_buffers* with default 1.

```
TYPE nowaitsend COMPONENT (no_of_buffers : INTEGER DEFAULT 1);
  PROVIDE
    SERVICE  send (what : TEXT);
             receive RESULT TEXT;
  END PROVIDE;

  CONTROL

    PROCEDURE schedule;
    BEGIN
      INSPECT ENTRY_AREA LOOP
      WHEN send    : IF no_of_filled_pos  < no_of_buffers AND POPUL_SERVICE < 1
                     THEN   SELECT;
                            no_of_filled_pos := no_of_filled_pos + 1;
                     END IF;
      WHEN receive : IF no_of_filled_pos  > 0 AND POPUL_SERVICE < 1
                     THEN   SELECT;
                            no_of_filled_pos := no_of_filled_pos - 1;
                     END IF;
      END LOOP;
    END PROCEDURE schedule;

  END CONTROL;


  VARIABLE  buffer          : ARRAY [1..no_of_buffers] OF TEXT;
            no_of_filled_pos: INTEGER DEFAULT 0;
            pos_send,
            pos_receive     : INTEGER DEFAULT 1;


  TYPE send SERVICE (what : TEXT);
  BEGIN
    buffer[pos_send] := what;

    IF pos_send = no_of_buffers
    THEN pos_send := 1;
    ELSE pos_send := pos_send + 1;
    END IF;
  END TYPE send;
```

```
    TYPE receive SERVICE RESULT TEXT;
    BEGIN
        RESULT buffer[pos_receive];
        buffer[pos_receive] := "";

        IF pos_receive = no_of_buffers
        THEN pos_receive := 1;
        ELSE pos_receive := pos_receive + 1;
        END IF;
    END TYPE receive;

END TYPE nowaitsend;
```

The component type *nowaitsend* has to be introduced to a HI-SLANG program by a %COPY statement. No setting of component control procedures is meaningful for *nowaitsend* and is thus not allowed.

Example:

```
%COPY "NOWAITSEND"
...
COMPONENT proc_comm : nowaitsend (100);
```

sender:

```
. { produce messages }
.
send (message);
.
.
```

receiver:

```
.
.
message := receive;
.
. { process message }
```

F.1.7. Ftserver

The component type *ftserver* (usable with solver Markov only) (fault tolerant server) realizes a fault tolerant standard element, with access to one or more processors, the number being defined by *processors* ≥1. Each of these processors is subject to a Poisson failure process with rate *failure_rate* or rate *dormancy* ∗ *failure_rate*, respectively, where *failure_rate* denotes the failure rate of an active processor and *dormancy*, $0 \leq dormancy \leq 1$, denotes the "dormancy" factor.

Every defective processor is repaired with rate *repair_rate*. Depending on the parameter *repair_units*, a processor not operating might have to wait for the beginning of its repair.

RANDOM (random choice with identical probabilities) is the standard strategy for the choice of services, which have to be deactivated or reactivated. Furthermore (essentially for reasons of the techniques used for the solvers) the maximum degradation *degmax*, $0 \leq degmax \leq processors$, can be specified.

The parameters relevant for *ftserver* are:

processors	number of processors, $processors \geq 1$
degmax	maximal degradation, $0 \leq degmax \leq processors$
repair_units	number of repair units, $1 \leq repair\_units \leq processors$
failure_rate	failure rate of an (active) processor, $failure\_rate > 0.0$
repair_rate	repair rate (per repair unit), $repair\_rate > 0.0$
dormancy	"dormancy" factor, $0 \leq dormancy \leq 1$ (The failure rate of a passive processor is equal to *dormancy* ∗ *failure_rate*)

The component type *ftserver* provides the service *request*. Apart from the demanded amount of service, *request* disposes of a parameter for the specification of a priority *prio*, 0 resembling the highest priority. Although this parameter has a default it must be given in a corresponding USE 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;

  TYPE request SERVICE (  amount    : REAL;
                          prio      : INTEGER DEFAULT 32767);
    {body not implemented in HI-SLANG}
  END TYPE REQUEST;

  BEGIN
      … { writing the warning (for simulative solution)}
  END TYPE ftserver;
```

In a component of type *ftserver*, only RANDOM, PRIOPREP and PRIONP are allowed for the component control procedure *schedule* (as in *prioserver*, see below). PRIONP is the default, but if parameter *prio* of *request* is not used or all priorities are set equal, this equals RANDOM. Only EQUAL and SDEQUAL are allowed for *dispatch*.

Note that there may occur conflicts between parameter *processors* and the component control procedures: e.g., *processors* := 2 and *schedule* := RANDOM (3) actually means RANDOM (2). For numerical solution only RANDOM(1) is allowed.

The component type *ftserver* has to be introduced to a HI-SLANG program by a %COPY statement.

Example:

```
%COPY "FTSERVER"
...
COMPONENT triple_computer : ftserver (3, 1, 1, 1E-5, 0.005);
```

The use of the service *request* of *triple_computer* is just as in the case of a normal *server* or *prioserver*. Employment of *ftserver* in the case of simulation is not yet possible (see also Appendix E.2.).

F.1.8. Prioserver

The component type *prioserver* (for Simulative, Markov and DOQ4) allows modelling of priority-controlled service strategies (zero being the highest priority):

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

    TYPE request SERVICE (amount :   REAL;
                          prio   :   INTEGER DEFAULT 32767);
    BEGIN
        spend(amount);
    END TYPE request;
END TYPE prioserver;
```

In the simulative case the body equals that of the normal *server.* The additional parameter *prio* is only treated in the component control procedures available: PRIONP, PRIOPREP, PRIOPRES (only for simulation) and RANDOM are eligible for the component control procedure *schedule* of a *prioserver,* see below. As for the *ftserver,* PRIONP is the default. All strategies for *accept, dispatch* and *offer* legitimate for the normal *server* in the case of the analytic-algebraic solver are allowed for the *prioserver* in the numerical case; in the case of simulation, any predefined component control procedure (legitimate for the normal *server*) may also be specified.

Just as all the other standard component types (with the exception of *server*) the type *prioserver* has to be introduced to a HI-SLANG program by a %COPY statement.

Example:

```
%COPY "PRIOSERVER"
…
TYPE diaproc SERVICE (my_prio : INTEGER);
    USE
        SERVICE compute (amount : REAL; prio : INTEGER);
    END USE;
BEGIN
    …
    compute (negexp(3/7), my_prio);
    …
END TYPE diaproc;

COMPONENT cpu : prioserver (LET schedule := prionp);

REFER diaproc TO cpu EQUATING
    diaproc.compute WITH cpu.request;
    …
END REFER;
```

Five strategies for the control procedure *schedule* are eligible for components of type *prioserver*, the first four of which allow only one process at a time to be in the *service area* of the component.

- **PRIOPREP:** In the preemptive priority strategy (*schedule* procedure) PRIOPREP ("**PRIO**rity **P**reemptive **REP**eat"), processes with high priority are served foremost. If several processes have the same priority, one of them is chosen randomly.

  A process of lower priority will be interrupted if one of higher priority arrives. When the process is restarted, the time consumed before the interruption is neglected; the process starts from the beginning ("repeat" strategy). While the new process is done with exactly the same demand of time (identical) in the simulative method, the demand of time is "drawn" again (resampling) in the numerical method. In extreme cases, this can lead to different results. Note that in the numerical case PRIOPREP is interpretable as PRIOPRES with resampling, since the distribution must be *negexp*.

- **PRIOPRES:** The preemptive priority strategy (*schedule* procedure) PRIOPRES ("**PRIO**rity **P**reemptive **RES**ume") proceeds in a way simular to PRIORPREP. At resumption, however, the process continues from the position reached prior to being interrupted. The consumed time, therefore, is not lost ("resume" strategy). PRIOPRES can only be used for the simulative solver.

- **PRIONP:** When using the non-preemptive priority strategy (*schedule* procedure) PRIONP ("**PRIO**rity **N**on **P**reemptive") no preemption results from the arrival of a process. Apart from this distinciton, PRIONP proceeds just as the preemptive priority strategy PRIOPREP.

- **RANDOM:** When the *schedule* mechanism RANDOM is selected, one of all waiting processes is randomly chosen. This strategy can also be used for components of the ("normal") type *server.*

- **IMMEDIATE**: When using the strategy IMMEDIATE, all processes in the *entry area* are immediately moved to the *service area.* This strategy can also be used for components of the ("normal") type *server.*

In contrast to service calls of *request* from a "normal" *server,* the assigning of priorities to service calls of *request* from a *prioserver* requires an additional parameter.

F.1.9. Observer

The standard component type *observer* (Simulative) should be used instead of the former standard service *watcher* to produce intermediate result outputs. Opposite to the *watcher* it does not influence the hosting model and it can be used in a more easy and flexible way.

The *observer* has no provided services, but internally creates one *watcher*-like 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 is implemented as follows:

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

   TYPE watcher SERVICE (watch_interval : REAL);
      VARIABLE
         answer     : TEXT;
         first      : CHARACTER;
         ok         : BOOLEAN;

   BEGIN
      LOOP

         hold (watch_interval);
         transfer_results;

         WRITELN "Current model time   :", time;
         WRITELN "Cpu time used [sec.] :", cpu_time;
         WRITELN;

         IF interactive THEN
            LOOP
            WRITELN "Please enter one of:";
            WRITELN "q : quit simulation";
            WRITELN "s : stop observing, continue simulation";
            WRITELN "c : keep current model time interval and continue observing";
            WRITELN "n : as c, but switch to non-interactive mode";
            WRITELN "<real value n.nnEnn> : set new interval, continue observing";

            lastitem;
            READ answer::20;
            READ TEXT answer, first;
            ok := TRUE;
```

```
                CASE first
                    WHEN 'q' : stop_evaluation ("YOU didn't want to continue!");
                    WHEN 's' : watch_interval   := maxreal;
                    WHEN 'n' : interactive      := FALSE;
                    WHEN 'c' : ;
                    ELSE  :
                        IF digit (first)
                        THEN READ TEXT answer, watch_interval;
                                WRITELN "> New interval is ", watch_interval;
                        ELSE WRITELN "> ILLEGAL input : ", first;
                                ok := FALSE;
                        END IF;
                    END CASE;
                END LOOP UNTIL ok;
            END IF;

        END LOOP;
    END TYPE watcher;

BEGIN
    CREATE 1 PROCESS watcher (obs_interval);
END TYPE observer;
```

To use the *observer* it has to be copied from the standard modelling base (via %COPY statement). An *observer* component should preferable be declared within the model type.

Example:

```
%COPY "OBSERVER"

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

Note:

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", or (on UNIX systems) a separate process to display the results should be started, e.g., *tail -f t.*.tab &*.

F.2. Standard Services

Currently there is only one standard service, besides the predefined services *spend* and *hold.*

F.2.1. Watcher

The only standard service HI-SLANG currently provides for simulation is the type *watcher* (S), which permits, when called, the computation and output of intermediate results (the current values of all results demanded), allowing the progress of a simulation to be observed and thus controlled to a certain degree. The parameter *watch_interval* defines the intervals of model time at which the body of the *watcher* is executed. Processes of the type *watcher* therefore are permanently in the model from the moment of their creation. For reasons of efficiency, only a single *watcher*-process should be created (by CREATE) at one point of the HI-SLANG program.

```
TYPE watcher SERVICE (watch_interval : REAL DEFAULT 50);
BEGIN
    LOOP
        hold (watch_interval);
        transfer_results;
    END LOOP;
END TYPE watcher;
```

Like all standard types, the service *watcher* has to be introduced to a HI-SLANG program by a %COPY statement. This has to take place within the model type or component type that is to contain the CREATE statement corresponding to the creation of the *watcher* :

Example:

```
TYPE ct COMPONENT ;

%COPY "WATCHER"
    …
BEGIN
    …
    CREATE 1 PROCESS watcher (LET watch_interval := 100);
END TYPE ct;
```

Note:

Since *watcher* is a normal service and exactly one *watcher* process permanently exists within the enclosing component, the performance value for its POPULATION is influenced by this process (increased by 1).

Note:

The *watcher* only exists for reasons of upward compatibility. Please use the new component type *observer* instead.

F.3. Standard Component Control Procedures

The following component control procedures are predefined; they are implemented in SIMULA and contained in the HIT standard mobase. Although they are no keywords their names are mostly written upper-case within this manual to stress their importance for modelling.

name		name of 1. par	type / default 1.par	name of 2. par	type / default 2. par	only for component type	kind of procedure
ALWAYS	*						
LIMITED	S	capacity	INT 1				ACCEPT
RESTRICT	M	capacity	INT 1				
CPRIO	S,M					counter	
CRANDOM	S,M					counter	
FCFS	*	capacity	INT 1				
IMMEDIATE	*						
LCFS	S	capacity	INT 1				
LCFSPR	*						
PRIONP	S,M					prioserver ftserver	SCHEDULE
PRIOPREP	S,M					prioserver ftserver	
PRIOPRES	S					prioserver	
RANDOM	S,M	capacity	INT 1				
SEMSCHED	S					semaphor	
TOKSCHED	S					tokenpool	
AGGRDISP	-						
EQUAL	*	speed	REAL 1				
SDEQUAL	*	sdspeeds	ARRAY	speed	REAL 1		DISPATCH
SDSHARED	*	sdspeeds	ARRAY	speed	REAL 1		
SHARED	*	speed	REAL 1				
ALL	*						OFFER

Some of the standard procedures for component control, described in the following, possess parameters. In this case the procedure heads are given below. Since these procedures are, for reasons of efficiency already implemented in SIMULA, standard procedures with parameters cannot be checked for the correctness of their actual parameters by the HI-SLANG compiler. Only the modeller is therefore responsible for the correctness of these expressions on HI-SLANG level. Thus, if for example the types of the actual parameters do not correspond with the types of the formal parameters, it is possible that compilation errors occur in the SIMULA compilation.

Calls of these procedures by the user (e.g., within services) are of no effect and are commented with warnings by HIT at run time (simulation).

For the meaning of the marks (S), (M), (D), (L) and (*) and combinations of them see the introduction at the beginning of this appendix.

F.3.1. ACCEPT Procedures

(*) **ALWAYS**

Acceptance without limitations.

(S) **LIMITED** (capacity : INTEGER DEFAULT 1)

Acceptance with capacity limitations. The INTEGER parameter specifies the limitation. The solution of the waiting situation occurs according to FCFS.

Example:

```
COMPONENT c : ct (LET accept := limited (5));
```

(M) **RESTRICT** (capacity : INTEGER DEFAULT 1)

In the analytic-numerical case, a *server*'s capacity can be restricted by using RESTRICT. The integer parameter specifies the limitation. When the capacity of a *server* is exhausted, acceptance of further services is not possible. Services requiring service are refused and repeat the service (or the last phase of service in COX-distributions with several phases) at the *server* visited last. In the case of simulation, the use of RESTRICT is not possible (also see Appendix E.2.).

Example:

```
COMPONENT c : server (LET accept := restrict (3));
```

F.3.2. OFFER Procedures

(*) **ALL**

Every process in the *exit area* is offered. There are no parameters.

F.3.3. SCHEDULE Procedures

(S,M) **CPRIO**

Choice of the service call with highest priority allowed to perform its change operation. This procedure is only usable for components of type *counter*. See section E.1.4.1. for further restrictions in the analytical-numerical case.

(S,M) **CRANDOM**

Random choice of one waiting process allowed to perform its change operation. This procedure is only usable for components of type *counter*.

(*) **FCFS** (capacity : INTEGER DEFAULT 1)

First-Come-First-Scheduled. The INTEGER parameter specifies the limitation of the *service area.* Thus *fcfs (3)* means, that a maximum of three processes is let into the *service area.*

Example:

```
COMPONENT c : ct (LET schedule := fcfs (3));
```

(*) **IMMEDIATE**

Immediate access to the *service area,* no preemption.

(S) **LCFS** (capacity : INTEGER DEFAULT 1)

Last-Come-First-Scheduled. The INTEGER parameter specifies the limitation of the *service area.* Thus *lcfs (3)* means, that a maximum of three processes is let into the *service area.*

(*) **LCFSPR**

Last-Come-First-Scheduled-Preemptive-Resume, without parameters.

(S,M) **PRIONP**

PRIOrity Non-Preemptive, usable only for components of type *prioserver* or *ftserver.* The capacity of the *service area* is limited by 1. In case of an empty *service area* one of the processes with highest priority is scheduled, each one having the same probability to be scheduled.

(S,M) **PRIOPREP**

PRIOrity Preemptive REPeat, usable only for components of type *prioserver* or *ftserver.* The capacity of the *service area* is limited by 1. A preempt occurs if a process with the same or higher priority than the actually served process has entered the *entry area.* In case of one or more processes with highest priority a possible selection is performed randomly. It has to be considered here that the semantics of PRIOPREP vary between the simulative and the numerical case (see Appendix E.2.1. and F.1.8.).

(S) **PRIOPRES**

PRIOrity Preemtive RESume, usable only for components of type *prioserver* or *ftserver.* The capacity of the *service area* is limited by 1. A preempt occurs if a process with the same or higher priority than the actually served process has entered the *entry area.* Reentering the *service area* after a preemption the service execution will be repeated completely with an identical amount. In case of one or more processes with highest priority in the *entry area* a possible selection is performed randomly.

(*) **RANDOM** (capacity : INTEGER DEFAULT 1)

When RANDOM is used, waiting processes are chosen randomly. The INTEGER parameter specifies the limitation of the *service area*. Thus, *random (2)* means that a maximum of two processes is let into the *service area*.
In the analytical case, RANDOM may only be used for the component types *server, prioserver* and *ftserver* and only RANDOM (1) is permitted.

Example:

```
COMPONENT c : ct (LET schedule := random (2));
```

(S) **SEMSCHED**

Required only for the component type *semaphor*, where it is the default. The scheduling strategy is First-Come-First-Scheduled.

(S) **TOKSCHED**

Required only for the component type *tokenpool*, where it is the default. The scheduling strategy is First-Come-First-Scheduled.

F.3.4. DISPATCH Procedures

(-) **AGGRDISP**

This *dispatch* procedure is internally required for aggregated component types in the simulative case. The procedure may not be specified by the HI-SLANG user.

(*) **EQUAL** (speed : REAL DEFAULT 1.0)

Each process of the *service area* gets the same "full" service speed, defined by parameter *speed*. The parameter *speed* also specifies the standard speed (stream UTILIZATION, see Section 5.1.2.).

(*) **SDEQUAL** (VALUE sdspeeds : ARRAY OF REAL;
speed : REAL DEFAULT 1.0)

State-dependent service speeds. Each process of the *service area* gets the same "full" service speed, defined by the product of parameter *speed* and the appropriate entry of the array *sdspeeds*. For the meaning of parameter *sdspeeds* and an example see SDSHARED.

(*) **SDSHARED** (VALUE sdspeeds : ARRAY OF REAL;
speed : REAL DEFAULT 1.0)

State-dependent service speeds, equal distribution of the service speed of the component (product of parameter *speed* and the appropriate entry of the array *sdspeeds*) on all processes in the *service area.*

The parameter *speed* specifies the standard speed (see also Section 5.1.2., stream UTILIZATION), *sdspeeds* names the populations and the corresponding service speeds of the component. *sdspeeds* is a two-dimensional REAL array, which cannot be modified by concurrently executing processes.

Example:

```
VARIABLE sp : ARRAY [1..2, 1..anz_pop] OF REAL;
```

The first line contains the populations (> 0) in ascending order, which are rounded to INTEGER. Gaps are allowed (for interpretation, see below). The second line contains the corresponding speed factors of the component (> 0.0).

For *speed* factor selection the actual number of processes in *entry area* and *service area* is used.

Example:

Let the following *speeds* array be given:

1	7	8	12	←¾¾	populations
1.2	1.1	0.7	0.4	←¾¾	corresponding service speeds of the component
↑	↑	↑	↑		
1-6	7	8-11	from 12	←¾¾	valid for area of populations

The corresponding array aggregate has the form:

```
[ [ 1, 7, 8, 12] , [ 1.2, 1.1, 0.7, 0.4 ] ]
```

Note:

The first specified population (the first element in the first line of the field) has to be equal to one to avoid errors.

(*) **SHARED** (speed : REAL DEFAULT 1.0)

Distribution of the service speed of the component (parameter *speed*) in equal parts on all processes in the *service area.* The parameter *speed* also specifies the standard speed (stream UTILIZATION, see Section 5.1.2.).

Example:

```
COMPONENT c : ct (LET dispatch := shared (1.5));
```

G. Description of Output Formats

This appendix describes all the compiler and analyzer outputs being of interest for the user. Using the HIT standard link names (see table in Section 8.1.) these are:

G.1.	"LISTING"	The listing generated by the HI-SLANG compiler as well as by all analyzers. The latter normally includes the solver information, while the compiler listing may contain an XREF listing.
G.2.	"SYSOUT"	The standard output (e.g., on terminal) of compiler or analyzer.
G.3.1.	"TABLE"	Analyzer results in form of a table.
G.3.2.	"DUMP"	Analyzer results in form of a dump file.
G.3.3.	"GRAPH"	Simple graphs producable from dump files.
G.3.4.	"HISTOGRAM"	Histograms producable from dump files.
G.4.	"PREANA"	An aggregated component type producable by the DOQ4 solver.
G.5.	"TRACE"	The trace of a simulation.
G.5.1.		Event trace.
G.5.2.		State trace.

The numbers prefixing the link names above denote the subsection of this appendix describing that output format. Outputs which do not contain information for the user but intermediate data sets of the tool are not explained here, as:

"CODE" the SIMULA code generated by the HI-SLANG compiler
"MATRIX" the matrix scheme of the MARKOV solver
"STATES" the state table of the MARKOV solver
"PRINTDS" for internal use only: displays the data structures of an analyzer.

Most of the output formats described here are illustrated by the example in Appendix I.

G.1. Format of the Listing

The compiler as well as every generated analyzer of the HIT system produces a listing via the standard link name "LISTING". It has record length 133 including a print control character. The listing has a default binding to a file named by the file name generator (see Section 8.2.6.). Normally the analyzer listing is appended to the compiler listing if "LISTING" is not rebound in the control file.

The compiler listing (G.1.2.) informs the user about the current translation. It consists of

G.1.2.1. a listing of those parts of the control file relevant for the compiler, if there are any,

G.1.2.2. a listing of the assembled HI-SLANG source,

G.1.2.3. an optional cross reference table (XREF).

The analyzer listing (G.1.3.) informs the user about all the evaluations performed. It also starts with a listing of those parts of the control file relevant for the analyzer. Moreover information why a certain algorithm was used to solve the model are given, if this facility is not suppressed in the control file.

G.1.1.Common Elements of the Listing

In both listings the parts listed above are

1. numerated twice (except XREF and solver information),
2. started or interrupted by a new page with a title line,
3. separated by completion messages which may be preceded by a list of error messages or warnings.

These common elements will be introduced first. A sample listing can be found in Appendix I.3.

G.1.1.1. Numeration within Listings

Every line listed is numbered twice: columns 1-5 contain the absolute line number unique within the assembled source, while columns 6-10 contain a file relative line number. The latter is very useful for error corrections since it permits a direct access to the relevant file or object and the relevant line number within it. The file or object is denoted by a code character immediately following the relative line number. Most code characters are letters (starting with 'a'); for the control file a blank is used and SYSIN is marked by a '*'.

Within the line follows a seperator "': '" and source code which may be indented. The colon may be substituted by the capital letters 'E' or 'W', this way marking lines containing errors or causing warnings. Because the source listing is completed after PASS 1 this facility is only implemented for messages produced by FAN or PASS 1.

G.1.1.2. Page Titles

Every page of the listing is started with a title having the following format:

column		meaning	format
1-20	:	version number of HIT	HIT Version n.n.nn
25-67	:	variable title text	43 arbitrary characters
70-80	:	date of compilation	installation dependent, e.g., jjjj/mm/dd
84-89	:	time of compilation	installation dependent e.g., hh:mm
93-102	:	page number	PAGE nnnn
106-128	:	place of HIT development	University of Dortmund

The variable text may be specified by %TITLE statements in the source. One empty line and 63 regular lines follow. This number may be overridden by using %PARM=LINES=n.

G.1.1.3. Completion Messages and Message Tables

The end of every compiler pass is indicated by one empty line and one completion message. The same holds for every evaluation. Completion messages have the following format:

column		meaning	format
1	:	message indication	>
3-13	:	pass name	FAN, PASS1, PASS2, PASS3, ACG, SCG, SIMUL, DOQ4, LIN2, MARK or T O T A L
16-42	:	pass status (4 possibilities)	Okay. Only nnnn Warnings. nnnn Errors detected. nnnn Errors, nnnn Warnings.
45-77	:	used cpu time (for this pass resp. evalutation)	Cpu Time used: nnnn.nnn Seconds.

In the following "pass" does not only mean a compiler pass, but also one evaluation (of a series).

Pass name and pass status are separated by a colon. If the status is not *Okay* the compiler resp. analyzer has detected errors and/or warnings for this pass. These are summarized in a table of messages preceeding the completion message. The title of this table is underlined:

Number Line : Description of Errors or Warnings detected by <pass name>.

All messages consist of the following elements, as indicated by the title line:

- The code number of the message (W.nnnn, E.nnnn, or A.nnnn for warnings, errors or severe errors (abort errors) respectively).
- The number of the source line causing the message. This number may be missing, especially for analyzer messages.
- A colon.
- The message text. It is contained in columns 16-132 and may be continued on the same columns of the next lines. All message texts are read from the HIT message library.

If the relative line number and the absolute line number are different, another message line is given in an analogous format:

- The number of the message is the same and therefore not given.
- The relative line number within that file object causing the message. This number is followed by the code character of that file object.
- No colon.
- The message text is "IN LINK=", followed by the link name of the file object.

If there was a message number starting with 'E', then only the running pass is contained. For message numbers starting with 'A', which indicates a severe error, the current pass is terminated at once. This means that the compiler resp. the current evaluation is also terminated after printing a completion message with pass name "T O T A L".

G.1.2.Compiler Listing

As already mentioned the compiler listing consists of a control file listing, a listing of the assembled HI-SLANG source and an optional XREF listing. If the control file contains a record %PARM=NOSOURCE, there will be no listing at all.

G.1.2.1. Control File Listing

The compiler listing starts with a listing of all records within the control file used which are interpreted by the FAN system of the compiler. The default title text gives the name of the control file. A completion message with pass name FAN completes this listing. If the control file does only contain HI-SLANG source text there will be no control file listing and no such completion message.

Please note that by numbering and indenting, a listed line will be cut if the line would become longer than 132 characters.

G.1.2.2. Assembled Source Listing

The main part of the compiler listing follows on a new page. If it does not start with a %TITLE statement the default title remains. The source listing is assembled by executing all %COPY statements within the source and/or within the files or objects being copied.

The listing may be suppressed totally by %PARM=NOSOURCE or partially by using %NOSOURCE at the beginning and %SOURCE at the end of the HI-SLANG portion not to be listed. Although these %-commands are not listed such listing gaps can be detected by gaps in the absolute and relative line numbering.

A %PAGE within the source causes the listing to be continued on a new page. %TITLE has the same effect, but the first 43 characters of its argument define the new title text.

Moreover the appearance of the listing is influenced by the %PARM parameters NORESWD and INDENT. Without NORESWD all keywords are printed upper-case and all user-defined identifiers are printed lower-case. With INDENT = *c n* a blockwise indenting of *n* positions may be specified, *c* being a character connecting block start and block end (see Section 8.2.1.). The keywords BEGIN and TO (for CONCURRENT) interrupt this connection to clearly separate declaration part and statement part or the different statement parts respectively.

By numbering and indenting, a listed line will be cut if the line was longer than 132 characters.

The source listing ends with a completion message with pass name PASS 1. An XREF listing may follow but may be preceded by a message

> XREF-Listing may be incomplete due to Errors in PASS 1.

G.1.2.3. XREF Listing

If %PARM=XREF was specified a cross reference table of all objects and types occuring in the assembled source is generated. These source elements are not only differentiated by their names, but also by their scopes. For example, variables with the same name, but declared within different blocks are presented separately.

The title line of XREF listing pages has standard format, but the title text already starts in column 1. It reads:

```
Identifier    S/: Type or ConstantLine Access Pairs    * XREF *
```

Consequently every XREF line has the following structure:

column		meaning	format
1-12	:	name of HI-SLANG element	up to 12 lowcase letters
14	:	marking of a system defined element	'S' or ':'
16-33	:	kind or value of the element	see below
35-132	:	line number / access-pairs	nnnnnccc

The HI-SLANG elements are listed in alphabetical order, the first element starting with a new first letter is preceded by an empty line.

The element name is separated from the following information by one colon, which is substituted by an 'S' for predefined standard elements.

The kind of the element is indicated next in the line (columns 16-33). There are the following possibilities.

kind of element		meaning or comments
Type	VARIABLE	*Type* missing for record objects
Type	CONSTANT	*Type* missing for record constants
Type	PROCEDURE	USE-, PROVIDE- and main-declarations of procedures and PROCEDURE ARRAYs. *Type* missing for zero- or multi-valued procedures.
Type	SERVICE	USE-, PROVIDE- and main-declarations of services resp. service arrays. *Type* missing for zero- or multi-valued services.
Type	ARRAY (nn)	*Type* missing for record arrays. *nn* is the array dimension.
PROCESS	ARRAY (1)	process array
COMPONENT	ARRAY (1)	component array
	PROCESS	declared process
	MODEL	model object
	COMPONENT	component object
	ENCLOSE	multiply used component object
TYPE	SERVICE	service, if the service is not provided
TYPE	COMPONENT	component type
TYPE	MODEL	model type
STREAM	Kind	Kind ::= STATE | EVENT | COUNT, user-defined stream

	COLLECT	COLLECT-block. If there is no AS name there is no element name in columns 1-12.
SIMULA	SIMULA	component control procedure
	EXPERIMENT	experiment
	EVAOBJECT	evaluation object
	HIERARCHY	load filtering hierarchy

For *Type* above the following possibilities exist:

Type ::= REAL | INTEGER | BOOLEAN| CHARACTER | TEXT | INFILE | OUTFILE | POINTER

For arrays also their dimension is given in brackets. For constants their value is given instead of their type. The format is

nnnnnnnnnEznn	for REAL
nnnnnnnn	for INTEGER
TRUE or FALSE	for BOOLEAN
'c'	for CHARACTER
"13 characters"	for TEXT. Text values longer than 13 characters are indicated by three dots appended.

Starting at column 35 the line/access-pairs are printed one after the other and may continue on the same column in the next lines. Every pair has the format "nnnnnccc". The first 5 digits denote the absolute line number in the assembled source listing where the source element occurs, while at most three following characters characterize this occurance. The alternatives are:

- d : Declaration of the element.
- r : Read access.
- w : Write access.
- p : Occurance as actual parameter, replaces *r* or *w* or both, depending on the parameter transmission mode. (Formal parameters are marked with *d*.)
- * : Multiple occurance within one line.

Example:

A line

```
13      2b    :    i := abs (i);
```

in the source listing causes these XREF entries:

```
abs             S    INTEGER PROCEDURE                13r
i               :    INTEGER VARIABLE          5d     13wp*
```

A further example can be found in Appendix I.3. The XREF listing is completed by a message giving the number of different elements and names occuring in the source:

> *nnnnn* different Objects, *nnnnn* different Identifiers.

Up to 4 completion messages follow (if there are no errors), having the pass names PASS 2, PASS 3, ACG or SCG (depending on the method) and "T O T A L ", the latter summarizing all completion messages of the single passes. Additionally the compile rate is given just below the cpu time used for the total compilation:

Compile Rate : *nnnn.nnn* Lines/Sec.

It is defined as the number of compiled source lines divided by the sum of the cpu times of all compiler passes. The latter is less than the cpu time used by the compiler, due to the compiler main program and especially to the initialization phase.

G.1.3.Analyzer Listing

The analyzer listing gives some relevant information about all evaluations (or aggregations) performed. It also starts with a listing of a section of the used control file. Here the section valid for the analyzer appears. If this section does not contain an explicit binding of the link name "LISTING" the analyzer listing is appended to the compiler listing.

The control file listing is followed by information concerning each single evaluation. It always starts on a new page with a table identifying that evaluation (the same information is found in the beginning of all tablular result outputs):

Control	*name*		
Experiment Name	*name*		
Model Type	*name*		
Model Name	*name*		
Model Parameters	Name	Type	Actual Value
	name	*type*	*value*
	…		
Used Method	*method*		
Date of Compile	*date*	Time of Compile	*time*
Start Date of Run	*date*	Start Time of Run	*time*

Next some relevant information is given, why a special evaluation algorithm was selected. They are called solver information. All analytical solvers (modules used by the generated analyzers) implement more than one algorithm for the class of models denoted by the method name given in the experiment specification. An appropriate algorithm is selected automatically when the analyzer is started.

By specifying %PARM=NOSOLVERINFO in the analyzer control file the generation of such solver information can be suppressed. Their format depends on the solver used, although the general structure is similar: It normally starts with a decision table:

SELECTION CRITERIA OF *solver_name*:

Model/Experiment Features	*algorithm_1*	*algorithm_2*	…	*algorithm_m*
condition_1	yes	no		no
condition_2	yes	yes		no
…				
condition_n	no	no		yes

A *no* in a column means that the algorithm cannot be selected if that condition holds. Thus the first algorithm from the left which only contains *yes* in all lines with fulfilled conditions for the model under study is selected, if any. If no condition is fulfilled then the leftmost algorithm is selected.

Note:

The table does not contain any information about the applicability of the corresponding solver: With the assumption that the solver can be applied it denotes the solution algorithm selected. For the restrictions of each solver see Appendix E.

Next the SELECTED ALGORITHM is given and the REASONS FOR SELECTION are listed in more detail, followed by some additional explanations and data, which concludes the solver information.

A completion message with the solver name as pass name (SIMUL, DOQ4, LIN2 or MARKOV) follows, which may be preceeded by a table of error messages for this evaluation. Next comes a footer of the following format:

Stop Date of Run *date* Stop Time of Run *time*
Used Cpu Time *value*

The above information (besides the control file listing) are given sequentially for every evaluation or aggregation performed. After finishing the last evaluation a completion message with pass name "T O T A L" is emitted.

%PARM=NOSOLVERINFO only suppresses those parts of the analyzer listing which are described in greater detail for all solvers in the following subsections.

G.1.3.1. Solver Information of DOQ4

The header information is followed by the SELECTION CRITERIA OF DOQ4:

Model/Experiment Features	Exact MVA	Convolution/ MVA	Approximate MVA	Response Time Preservation
Aggregation	no	yes	no	yes
Server with population dependent service speeds	no	yes	no	yes
Prioserver	no	no	yes	yes
Server with FCFS scheduling and non-identical service demands	no	no	yes	yes
Prioserver with PRIONP scheduling used by closed chains (services)	no	no	no	yes
Prioserver with PRIOPREP scheduling used by open and closed chains (services)	no	no	no	yes
Server with FCFS scheduling and non-exponential service demands	no	no	no	yes

The SELECTED ALGORITHM line inserts one of the above algorithm names into a message, which distinguishes between exact evaluation of separable queueing models from approximate evaluation of non-separable queueing models. The REASONS FOR SELECTION given next is one of the following:

For Exact MVA:
The model can be mapped to a product form queueing network. Service speeds of its servers are population independent.

For Convolution/MVA or RTP:
Exact and approximate MVA techniques cannot be applied:
- An aggregation is requested.
- Server *name* has population dependent service speeds.

For Approximate MVA or RTP:
Exact MVA and Convolution/MVA techniques cannot be applied:
- Server *name* has a priority scheduling discipline.
- Service demands at server *name* with FCFS scheduling are not identical.

For Response Time Preservation (RTP):
Exact MVA, Convolution/MVA and approximate MVA techniques cannot be applied:
- Server *name* has PRIONP scheduling and is used by closed chains (services).
- Server *name* has PRIOPREP scheduling and is used by open and closed chains (services).
- Service demands at server *name* with FCFS scheduling are not negative exponentially distributed.

The RTP algorithm additionally give the following INFORMATION ABOUT THE SOLUTION PROCESS: The number of RTP iterations has been *n*.
Moreover the exact algorithms (the first two above) yield a warning if more than 50 cpu seconds are estimated for the solution process.

G.1.3.2. Solver Information of LIN2

The LIN2 solver always applies the Linearizer algorithm for calculation of mean values. If performance bounds are additionally requested via estimator BOUNDS, one of two performance bounds algorithms runs in connection with the Linearizer. Thus the header information is followed by the following SELECTION CRITERIA OF LIN2:

Model/Experiment Features	Linearizer and PBH	Linearizer and Asymptotic Expansion	Linearizer
Speed values at state dependent stations are not monoton decreasing	no	yes	yes
A closed chain doesn't visit an infinite server station	yes	no	yes
0.75 < Mean alpha value < 1.0	no	yes	yes
Mean alpha value ≤ 0.0	yes	no	yes
No bounds requested	no	no	yes

The mean alpha value is necessary to verify the normal usage condition. The normal usage condition is, from the interpretation view, a indication for the utilization at infinite server stations. A simple explanation of the normal usage condition is not possible because the underlying theory is not easy to understand. We refer to /KnND89/ and /MiKe86/.

After the name of the SELECTED ALGORITHM one of the following REASONS FOR SELECTION is then given:

For Linearizer and Asymptotic Expansion:
- Accuracy is set to *n*.
- The mean alpha value (*n.nnnnnn*) is less than 1.0 and greather than 0.75. If the alpha value is in this range then the asymptotic expansion method will always be selected.
- The (mean) alpha value (*n.nnnnnn*) is less than 1.0 and greather than 0.0.
  The other implemented performance bounds method (based on MVA techniques) isn't applicable to the specified model because the sequence of speed values at *name* is not monoton increasing.

For Linearizer and PBH:
- Accuracy is set to *n*.
- The other implemented performance bounds method (named asymptotic expansion)
  - is also applicable but not selected because the alpha value (*n.nnnnnn*) is less or equal 0.75.
  - isn't applicable to the specified model. The closed chain (service)*name* doesn't visit an infinite server station.
  - isn't applicable to the specified model. The normal usage condition is violated because the alpha value at station *name* is *n.nnnnnn*.

Otherwise:
- Selection of a performance bounds method is not possible:
  - The PBH method isn't applicable because the sequence of speed values at *name* is not monoton increasing.
  - The asymptotic expansion isn't applicable because the closed chain (service) *name* doesn't visit an infinite server station.
  - The asymptotic expansion isn't applicable because the alpha value at station *name* is *n.nnnnnn* .
- No bounds are desired, since accuracy is not set or set to less than or equal 0.0.

G.1.3.3. Solver Information of MARKOV

Since the decision between the available algorithm is delayed until the state space is explored the standard evaluation identifying table is immediately followed by some information about this state space:

STRUCTURAL MODEL INFORMATION:

Number of states..... : *n*
Kind of state equivalence for partitioning..... : Chain population(s)
Number of components of a macro state..... : *n*
Number of macro states..... : *n*
Number of states of the largest macro state..... : *n*
Number of states of the smallest macro state..... : *n*
CPU-time used for state space generation..... : *n.nnnnnn* seconds
CPU-time used for matrix construction..... : *n.nnnnnn* seconds
Order of the matrix ..... : *n*
Bandwidth of the matrix..... : *n*
Fill-in of the matrix..... : *n*
Fill-in of the matrix..... : *n.nnnnnn* %

Besides "Chain population(s)" the texts "Degradation state of ft_server(s)" or "none" may appear in the second line of that block. Next MARKOV's decision table and the name of the selected algorithm (one of those found in the table) follows:

SELECTION CRITERIA OF MARKOV:

Model/Experiment Features	DIRECT	Gauss-Seidel with A/D steps	SOR
order ∗ bandwidth > *r*	no	yes	yes
no. macro states = 1	yes	no	yes
no. macro states > *s*	yes	no	yes

SELECTED ALGORITHM:

ANALYTICAL MARKOV: *algorithm* is used as linear equations solver.

Note that *r* and *s* in the table above are substituted by implementation-dependent values. Hereafter one of the following five sentences appears in the REASONS FOR SELECTION part:

- DIRECT solution method (Grassmann algorithm) selected, because the product of the order of the matrix and its bandwidth, which restricts the possible fill-in, is lower than *r*.

- SOR method is selected, because the system of linear equations is large and
  - the number of macro states is 1,
  - the number of macro states is greater than *s*.

- Gauss-Seidel iteration with aggregation/disaggregation steps is selected, because the system of linear equations is large and the number of macro states is in the range of 2 to *s*.

Next the MARKOV solver emits INFORMATION ABOUT THE SOLUTION PROCESS. This may start with:

Number of iterations …… :	*n*	
Number of aggregation/disaggregation steps..... :	*n*	
Used relaxation factor..... :	*n.nnnnnn*	
Reduction factor …… :	*n.nnnnnn*	
Estimated error for state probabilities …… :	*n.nnnnnn*	%

The above block is repeated periodically, until the estimated error for state probabilities is small enough. Finally (or sometimes only) the following three lines are given:

CPU-time used for matrix construction and equations solution..... :	*n.nnnnnn*	seconds
CPU-time used for computation of basic measures. :	*n.nnnnnn*	seconds

The following messages may occur in between in the case of numerical problems:
- Inaccurate results from direct method are improved by iterations.
- The solution method is changed to Gauss-Seidel iteration.
- A switch from direct to iterative solution method occured.

G.1.3.4. Solver Information of SIMUL

Since there are no different algorithms for the simulative solver, there is no decision table, and the REASONS FOR SELECTION do only contain a trivial information: "SIMUL requested.".

Hereafter a list of Start and Stop Conditions reached (titled EVALUATION TRACE) is given, ordered by model time: For user convenience such a message is written during simulation whenever a start or stop condition is reached relating to a MEASURE statement, or a basic stop condition in a CONTROL statement changes its value. This special kind of condition trace helps the user in understanding unexpected system behavior. For MEASURE statements it has the following examplary format:

EVENTS *n* DUE TO HIERARCHY *name*: START condition reached in MEASURE statement:
EVAULATIONOBJECT *name,* STREAM *name,* DUE TO HIERARCHY *name,* MODELTIME *n.nnnnnn*

or

START condition reached in MEASURE statement:
EVALUATIONOBJECT *name,* STREAM *name,* DUE TO HIERARCHY *name,* MODELTIME *n.nnnnnn*

The format for CONTROL statements is similar:

EVENTS *n* DUE TO HIERARCHY *name*:
STOP condition reached in CONTROL statement, MODELTIME *n.nnnnnn*

The next section is titled INFORMATION ABOUT ESTIMATORS. It contains information about estimators that yield the value *undefined* after an evaluation. The output has general format:

Estimator {MEAN | STANDARDDEVIATION | CONFIDENCE LEVEL} for *stream_type* stream *stream_name* at *evaluationobject* due to *hierarchy* is undefined because: *reason*

One of the following possible reasons is inserted:

Mean

Event	- No updates
Count	- Measure interval has length zero
State	- Measure interval has length zero

Standarddeviation

Event	- No updates
	- Mean value is too large
Count	- Mean value is nearly zero
State	- Measure interval has length zero
	- Mean value is too large

Confidence level (independent of stream type)

- Degree of autoregressive model is zero
- Mean value is nearly zero
- Time series is alternating
- Parameters of autoregressive model are too large
- Variance is negative
- Variance is too large
- Autoregressive model has less than one degree of freedom
- Confidence interval is wider than 50% of mean value
- Stream is in pre phase
- Not enough data to estimate confidence interval

Some of these reasons (e.g., 'Variance is negative') are only consequences of numerical problems, which may occur. For more information about estimation of confidence intervals using autoregressive models cf. /LiSS89/.

Moreover the autocorrelation values are given for every stream which has CONFIDENCE LEVEL as an estimator. The output has the following format :

Autocorrelation values for *stream* at *evaluationobject* due to *hierarchy* :

1: *n.nnnnnnEnn* 2: *n.nnnnnnEnn* 3: *n.nnnnnnEnn* 4: *n.nnnnnnEnn* 5: *n.nnnnnnEnn*
6: *n.nnnnnnEnn* …

The lines consist of the autocorrelation values (of values having the denoted distances), which are calculated in order to estimate the confidence interval. There are up to five values in one line. Each value lies in the interval [-1..1]. The first line consists of the values for the distances one to five, the next of the values for the distances six to ten and so on. The distance is written in front of each value. There may be less values than the degree given in the MEASURE statement. In some cases (e.g., numerical problems during computation) there may be no values at all. In such a case the message

No values available

will appear.

If the stream is still in its pre phase, there also will be no autocorrelation values, but then the message

Stream is in pre phase

will occour.

Note:

It may additionally be helpful to set %PARM=UPDATES: In this case the number of updates that have occured is displayed in result tables within all mean value fields.

G.2. Format of the Standard Output

During the running of the HI-SLANG compiler or analyzer (or even OMA), messages are yielded on standard output (e.g., terminal) to inform the user about the state of the compilation or evaluation and the errors detected. These are essentially the same messages as given in the listing (see Appendix G.1.), but reformatted to lines of no more than 80 characters. Detected by <pass name> does not appear in the title of the error message tables.

The following start message initially appears on the standard output:

line 1:	**columns**	**contents**
	1-21 :	HIT <kind> licence (kind: denotes the kind of the licence)
	27-53 :	<name of licence token>
	54-80 :	ends <date> (or: no termination date)

line 2:	**columns**	**contents**
	1-21 :	HIT Version n.n.nnn
	27-53 :	<program name> (HI-SLANG Compiler, HIT-Analyzer <Name>, HIT-OMA)
	54-80 :	of <date and time of program creation>

Queries of the FAN system (e.g., %BIND "CONTROL" TO ?) and the echo of the answer entered by the user follow next, furthermore completion messages of all passes, preceeded possibly by error message tables.

The outputs caused by HI-SLANG WRITE statements also occur on the analyzer standard output. See Appendix I.3. for an example.

G.3. Format of Results

The performance values of the specified model, computed by the analyzer, can be represented directly as

1) a TABLE and/or
2) a DUMP FILE.

With the help of PLOT statements, dump files can be converted to a

3) simple GRAPH or
4) HISTOGRAM.

These four result formats are now described in detail. The outputs of the individual evaluations are concatenated, each starting on a new page (except dump files) in the case of an evaluation series, provided the corresponding link name has an EXTEND binding (which is also the default).

G.3.1.Format of a Table

Tables are written via link name "TABLE" (as default) or the link name appearing in the optional OUTPUT part of MEASURE statements or EVALUATIONOBJECT declarations. "TABLE" has a default EXTEND binding to a file named by the file name generator (see Section 8.2.6.). The record length is 133 characters including print control characters.

Table files start with a title line containing the page number in column 123 (PAGE nnnn). An empty line follows, and the link name of the table appears in the first column of the third line.

Next is an identification block. Line by line, it has the following contents:

line		**columns 1-23**	**columns 25-80**
1	:	Control	<name of control file or object>
2	:		
3	:	Experiment name	<name of experiment>
4	:		
5	:	Model Type	<model type>
6	:	Model Name	<name of model object>
7	:		
8	:	Model Parameters	Name Type Actual Value
9	:		==========
10	:		<n+1 lines giving name, type and actual value
of .			all model parameters including *seed* (in case of
10+n	:		simulation)>
11+n	:		
12+n	:	Used Method	<method name>

For analytical experiments, the line giving the acutal value of the parameter *seed* is omitted.

The identification block is followed by one empty line and a date block of the format:

line		columns 1-23	columns 25-36
1	:	Date of Compile	<date value>
2	:	Start Date of Run	<date value>
3	:		
4	:		
5	:		
6	:		
7	:	Stop Date of Run	<date value>
8	:	Used Cpu Time	<fixed point REAL with 5 fraction digits>
9	:		
10	:	Reached Model Time	<fixed point REAL with 5 fraction digits>

Here as well as in the following, <*date value*> and <*time value*> stand for installation dependent output formats of dates and times.

Lines 9 and 10 are only given for simulative experiments. Lines 1, 2 and 7 have more contents:

line		columns 50-66	columns 70-...
1	:	Time of Compile	<time value>
2	:	Start Time of Run	<time value>
7	:	Stop Time of Run	<time value>

The tabular output of the performance values follows on a new page. There is a separate table for every evalution object. This is the design of such a table:

Evaluationobjectname : <evaobj>

Hierarchy	Esti	<stream 1>	. . .	<stream s>
<hier 1>	<esti 1>	<value 111>	. . .	<value s11>
	<esti 2>	<value 112>	. . .	<value s12>
	.	.	.	.
	.	.	.	.
	<esti m>	<value 11m>	. . .	<value s1m>
<hier 2>	<esti 1>	<value 121>	. . .	<value s21>
	.	.	.	.
	.	.	.	.
	<esti m>	<value 12m>	. . .	<value s2m>
	.	.	.	.
	.	.	.	.
	.	.	.	.
	.	.	.	.
<hier n>	<esti 1>	<value 1n1>	. . .	<value sn1>
	.	.	.	.
	.	.	.	.
	<esti m>	<value 1nm>	. . .	<value snm>

Such tables are split into several tables if they do not fit on a page concerning their length or width. For every load filtering hierarchy *<hier 1>* to *<hier n>* occuring in MESAURE statements and for every desired estimator *<esti 1>* to *<esti m>*, and finally for every predefined or user-defined stream *<stream 1>* to *<stream s>* the user was interested in, the corresponding field of the table contains the performance value *<value ijk>*. $1 \le i \le s$, $1 \le j \le n$, $1 \le k \le m$ related to the given specifications.

Every such table thus resembles the entire set of MESAURE statements

MEASURE	<stream 1>, ...,	<stream s>
AT	<evaobj>	
DUE TO	<hier 1>	, …,	<hier n>
ESTIMATOR	<esti 1>	, …,	<esti m>

referring to the same evaluationobject *<evaobj>*. The estimator *<esti k>* may already be given in the declaration of *<evaobj>* and may therefore not appear in the MEASURE statement.

Note:

Even if there are several MEASURE statements for the same evaluation object there will be only one table for that object. If you want to have several tables for measurements of the same component, please use different evaluation objects referring this component!

Load filtering hierarchy names are shortened to 12 characters if necessary, while stream names, centered in the table head, may assume a length of up to 25 characters. Names of predefined load filtering hierarchies or streams (e.g., ALL, POPULATION) are printed in upper-case letters, user-defined names are given in lower-case letters. The following abbreviations are used for estimator names:

Mean	:	mean value	(all solvers)
Lower	:	lower performance bound of mean value	(only LIN2)
Upper	:	upper performance bound of mean value	(only LIN2)
Stdev	:	standard deviation	(only simulative)
Con nn%	:	confidence interval of width *nn*%	(only simulative)
Freq	:	frequency intervals	(only simulative)

Within the following format descriptions each *n* denotes a single digit or a leading blank. Especially *nnnnn.nnnnnn* means additionally:

nnnnnn.nnnnn	for $10^{-4} \le x < 10^{6}$
-nnnnn.nnnnn	for $-10^{5} < x \le -10^{-4}$
n.nnnnnnEznn	otherwise (implementation dependent)

The estimators yield a different number of values:

- **Mean:** Up to 4 values may be written to such fields:

  ```
   nnnnn.nnnnnn                [nnn]
  [nnnnn.nnnnnn       nnnnn.nnnnnn]
  ```

  Here the first number is the mean value. The number behind the mean value gives the number of updates for that stream recorded during simulation, if %parm=updates was used. Correspondingly, the two numbers in the second line denote the minimum and maximum value recorded, if %parm=minmax was used in the control file.

- **Lower, Upper, Stdev:** The corresponding value *x* is given simply in the format

  nnnnn.nnnnnn

  in the first 13 columns of the field of the table which has the total length of 27 characters.

- **Con**: The format for *Con* is

  ```
  nnnnn.nnnnnn+- nn.nn %
  nn                   nn
  ```

  The subfields contain the mean value of the confidence interval, the width of this interval as percentage and, in the next line, the specified degree (of the autoregressive model) and the degree that seems to be sufficient (cf. /Litz85/). As before, the mean value may also be given in an exponential format.

- **Freq**: For the estimator *Freq* (only for EVENT or STATE streams), the frequency value is given in the first line and the interval bounds in the second line of the field. For every specified interval such a two-line field is given. For absolute occurrences in context with EVENT streams it has the following form:

  ```
  nnnnnn
  nnnnn.nnnnnn   nnnnn.nnnnnn
  ```

  In other cases the frequency value is a real.

  ```
   nnnnn.nnnnnn
  nnnnn.nnnnnn  nnnnn.nnnnnn
  ```

  The format depends on the analyzer option FREQUENCYFORMAT (see 8.2.1.3).

"Undefined" can appear instead of a value for any estimator (for the reasons see Section 5.1.4.). If there is more than one MEASURE statement for an evaluation object, some fields in the table may remain empty.

Example:

```
MEASURE POPULATION          AT evaobj DUE TO hier
ESTIMATOR MEAN;

MEASURE TURNAROUNDTIME      AT evaobj DUE TO hier
ESTIMATOR FREQUENCY ...;
```

In this case, the table generated has the following appearance:

Evaluationobjectname : evaobj

HIERARCHY	ESTI	POPULATION	TURNAROUNDTIME
hier	Mean	<value>	<no value>
	Freq	<no values>	<values>
	.	.	.
	.	.	.
	.	.	.
	Freq	<no values>	<values>

A table may contain performance values for up to 4 streams. If there are more than 4 streams, at least one further table of the same type is calculated. If the table becomes too long due to too many load filtering hierarchies, it is continued on the next page.

Every new page begins with the three-line heading described earlier. After three empty lines the table head is repeated, then the table continues. A new table begins after four empty lines.

An example for a table output can be found in Appendix I.5.

G.3.2.Format of a Dump File

Dump files are written via link name "DUMP" (as default) or the link name used in the optional OUTPUT part of MEASURE statements or EVALUATIONOBJECT declarations. "DUMP" has a default EXTEND binding to a file named by the file name generator (see Section 8.2.6.). The record length is 132 characters; there are no print control characters within dump files.

A dump file basically contains the same information as the tables described above, though it is presented in a compact, machine-readable format, which is not easy to survey for a human reader. Dump files are nevertheless described here because they serve as interface for the simple graphical result presentations described below. Moreover, dump files may be interpreted by user-written special purpose software.

Values contained in dump files can also be written or accessed by using the predefined procedures *put_result* and *get_result*, respectively, within a HI-SLANG source (see Appendix D.1.).

A dump file starts with an identification block and a date block similar to those of table outputs; however, they are prefixed by "% " in each line to make them easy to skip by programs. Lines 7 and 9 (see Appendix G.3.1.) are not used and the date block is split after "Start Date of Run" by inserting the link name of the dump file and all dump information records: In detail it has the following appearance:

line		**columns 1-23**	**columns 25-80**
1	%	Control	<name of control file or object>
2	%		
3	%	Experiment name	<name of experiment>
4	%		
5	%	Model Type	<name of model type>
6	%	Model Name	<name of model object>
7	%	Model Parameters	Name Type Actual Value
8	%		<n+1 lines giving name, type and actual value
.			of all model parameters including *seed* (in case
8+n	%		of simulation)>
9+n	%		
10+n	%	Used Method	<method name>
11+n	%		
12+n	%	Date of Compile	<date value>
13+n	%	Start Date of Run	<date value>
14+n	%	Header is	<dump file header>
15+n	%		
16+n	%		

In analytical experiments the line giving the actual value of parameter *seed* is omitted. The date values of lines 12+n and 13+n only extend to column 38. These same lines, as known already from tables, also contain time values:

line		**columns 52-68**	**columns 72-...**
12+n	:	Time of Compile	<time value>
13+n	:	Start Time of Run	<time value>

The format of <*date value*> and <*time value*> is installation dependent.

Line 16+n is followed by a title line and one percent-blank line. The title line and the format and meaning of the entries in each column (the dump records following) can be described as follows:

columns	title	format	meaning
1-12 :	% Ev. Object	t12	evaluation object name
14-38 :	Measure	t25	stream name
41-47 :	Degree	i7	degree of autoregressive model, if > 0 (only for CONFIDENCE, cf. /Litz85/)
56-67 :	Hierarchy	t12	load filtering hierarchy name
70-82 :	Abcissa	r13	abcissa value given in MEASURE statement (default 0) for graphical result presentation
85-101 :	Estimator	t17	one of the estimator names MEAN, UPPER BOUND, LOWER BOUND, STANDARDDEVIATION, CONFIDENCE or FREQUENCY
104-108 :	<none>	i5	for CONFIDENCE: the level for FREQUENCY: number of lines following (intervals)
110-122 :	Result	r13	performance value (for CONFIDENCE: interval width as percentage)
125-132 :	# / Degree	i8	for CONFIDENCE: actual degree of the autoregressive model

Each such line describes one performance value. The following format forms have been used in the table above:

tx : text with up to *x* characters
ix : right bordering integer in a field of length *x*
rx : real number with *x-6* mantissa digits and field width *x*.

Note that evaluation object names, stream names and hierarchy names are shortened automatically, if necessary (i.e., limit of format is exceeded).

For estimator CONFIDENCE each regular line is followed by a second line containing the means of the confidence interval using format r13 in the *Result*-row.

An additional line is used for every specified interval by the estimator FREQUENCY.

For EVENT streams, the additional lines have the following format:

columns	format	meaning
95-107 :	r13	lower interval bound
110-122 :	r13	upper interval bound
125-132 :	i8	Number of events with performance values occuring in the interval. The title of this row may thus be interpreted as number or degree (#/Degree).

For STATE streams, the additional lines have the following format:

columns	format	meaning
81- 93 :	r13	lower interval bound
95-107 :	r13	upper interval bound
110-122 :	r13	accumulated time, the trajectory value is within the interval, divided by the observation time

In dump files the value "Undefined" is represented by an installation-dependent large real value (maxlongreal/4). The HI-SLANG constant UNDEFINED yields this value.

Two lines containing only "% " terminate the list of dump records. Finally, the date block is completed by the following four lines:

columns 1-25	columns 27-38
% Stop Date of Run	<date value>
%	
% Used Cpu Time	<used cpu time>
% Reached Model Time	<fixed point number with 5 fraction digits>

The first line further contains:

columns 52-68	columns 72-...
Stop Date of Run	<time value>

The last two lines are only given for simulations. In case of evaluation series the dump output of the next evaluation is appended. The example of a dump file can be found in Appendix I.6.

G.3.3.Format of a Histogram

Histograms are written via link name "HISTOGRAM" (as default) or the link name occuring in the optional OUTPUT part of a HISTOGRAM statement. "HISTOGRAM" has a default EXTEND binding to a file named by the file name generator (see Section 8.2.6.). The record length is 133 characters including print control characters.

Every page of a histogram starts with a title in the standard format (G.1.1.2.) with the histogram link name in the title field. One empty line always follows the title line.

On the first page there is an identification block of this format:

line		columns 1-23	columns 25-80
1	:	HISTOGRAM	
2	:	==========	
3	:		
4	:	EVALUATION OBJECT	<evaobj>
5	:	MEASURE	<stream>
6	:	HIERARCHY	<hier>
7	:	ESTIMATOR	FREQUENCY
8	:		
9	:		
10	:		
11	:	<separator line consisting of 132 '-'-characters>	

The histogram itself follows on the next pages, beginning with a "starting mark" and continued by the sequence of bars. Every bar uses five lines, the fifth line of a bar being the first line of the next bar.

If there are more than 13 bars a "successor mark" is printed at the bottom of the page and the new page starts (with the title line and) with a "predecessor mark", followed by the next 13 bars, etc.

When all bars have been printed, the ending mark and inscription of the abscissa are printed. All marks have this format:

line		columns	meaning	format
1	:	1-132	ordinate inscription	right-adjusted
2	:			
3, 4	:	15-17	starting mark	--- !
56, 57	:	15-17	successor mark	! V
1, 2, 3	:	15-17	predecessor mark	^ ! !
n	:	15-17	ending mark	!
n+1	:			---
n+2	:	11-132	abscissa inscription	left-adjusted

A bar entry looks like this:

line		**columns 1-14**	**16**	**17-m** (m is length of the bar)
1	:	<lower bound>	!	%%%%%%%%%%%%%%%%
2	:		!	%%%%%% n.nnn E+-nnn %%
3	:	<upper bound>	!	%%%%%%%%%%%%%%%%
4	:		!	/
5	:		!	_____/

Line 2 contains the frequency value, which is also the bar length. If the bar is not long enough to hold this integer value the value is written to the right of the bar, separated by two blanks. The first line of the next bar (provided it exists) on the same page overwrites the fifth line of its predecessor.

The values <*lower bound*> and <*upper bound*> are given in a format depending on their value x:

$10^{-4} \le x < 10^{6}$	:	nnnnnn.nnnnnn
$-10^{5} < x \le -10^{-4}$	:	-nnnnn.nnnnnn
otherwise	:	zn.nnnnnnEznn

For an example see Appendix I.8.

G.3.4.Format of a Graph

Graphs are written via the standard link name "GRAPH" (as default) or the link name given in the optional OUTPUT part of a GRAPH statement. "GRAPH" has a default EXTEND binding to a file named by the file name generator (see Section 8.2.6.). The record length is 133 characters including print control characters.

Every page of a graph starts with a title in the standard format (G.1.1.2.) with the graph's link name in the title field. One empty line always follows the title line.

The graph output starts with one identification block for every curve:

line		**columns 1-23**	**columns 25-80**
1	:	CURVE (<symbol>)	
2	:	==========	
3	:		
4	:	EVALUATION OBJECT	<evobj>
5	:	MEASURE	<stream>
6	:	HIERARCHY	<hier>
7	:	ESTIMATOR	<esti>
8	:		
9	:		
10	:		
11	:	<separator line consisting of 132 '-'-characters>	

The actual graphs follow after the last identification block, each of them shown on a new page. Their appearence depends on the points to be plotted.

Every graph consists of

- the abcissa, partitioned by 13 marks, each fixed with a corresponding value,
- the ordinate, partitioned by 7 marks, each fixed with a corresponding value, and
- the scaling, two lines at the bottom of the page.

The page for a graph has an imaginative grid of 13 vertical and 7 horizontal lines. One horizontal line is the abscissa and one vertical line is the ordinate. The respective meanings of the axes are appended at their free ends as text values, followed by a scaling factor, a text "*& nn" (corresponding to multiplied by 10^n). The intersection of both axes is marked 0.

Curve symbols (the dots composing the different curves) have priority over the axis symbols and their inscriptions and may therefore overwrite these. If several curve symbols require the same "pixel", only the last (with identification block given last) can be seen. The curve symbols used are mentioned in the identification blocks. They are assigned in the following order:

* X # % § $. A B C D E F G H I J K L M N O P Q R S T U V W Y Z
  (32 symbols)

If there are more than 32 curves, the 33rd curve will again be assigned the first character listed, etc.

For a curve for estimator CONFIDENCE the confidence interval is also "plotted" by two '-'-characters appearing above and below the curve symbol at an appropriate distance. The '-'-characters may overwrite the curve symbol.

At the bottom of the page the scaling is given as follows:

```
ABSCISSA  SCALE   DIVISION:n.nnnnnnn PER '-'  <per column>
ORDINATE  SCALE   DIVISION:n.nnnnnnn PER '!'  <per line>
```

G.4. Format of an Aggregated Component Type

Experiments of METHOD ANALYTICAL "separable" (where "separable" may be substituted by one of its aliases, e.g., "DOQ4", see Section 4.2.10.) may contain AGGREGATE statements. In this case the generated (pre-)analyzer creates an aggregated component type (shortly called aggregate) for every AGGREGATE statement executed. Such aggregates may be used instead of the original HI-SLANG component type for reasons of efficiency if certain resticitions are satisfied (see Appendix E.1.).

Aggregates are written via the standard link name "PREANA" (as default) or the link name appearing in the optional OUTPUT part of an AGGREGATE statement. The standard link name has a default binding to a file named by the file name generator (see Section 8.2.6.). The record length is 132 characters.

Aggregates start with five lines of comments similar to those of a dump file (see Appendix G.3.2.), but only the experiment name, component type name and the date and time of the compilation and pre-analyzer run is contained, followed by the method name.

One empty line separates these comments from the HI-SLANG interface of the aggregate, i.e., the declaration of the component type including its PROVIDE declaration and service declarations of all provided services (dummy declarations with empty bodies). The component type body is also empty since all bodies are represented by the speeds table.

Another empty line separates the HI-SLANG interface from the speeds table starting with the control statement %SPEEDS, interpreted as %EOF by the compiler (see Section 8.4.). In contrast, an analyzer using such an aggregate, i.e., interpreting the speeds table, interprets %SPEEDS as start of data.

%SPEEDS is followed by D+1 lines giving the number of provided services D and for each such service its maximal population allowed in the component, as defined by CREATE statements within the AGGREGATE statement. This information appears as follows:

line		**columns 1-25**	**27-34**	**30-39**	**40-65**
1	:	NUMBER OF PROVIDES	<Integer>		
2	:	<Provide name 1>		POPULATION	<Integer>
…		…		…	…
D+1	:	<Provide name D>		POPULATION	<Integer>

The list of speed values follows next. It is terminated by two percent-blank lines, the second containing the stop date and time of the pre-analyzer run.

An AGGREGATE statement such as

```
AGGREGATE ct;
    CREATE N(1) PROCESS st1;
    …
    CREATE N(D)PROCESS stD;
END AGGREGATE;
```

creates a SPEEDS-table with

(1) $$D \left[\prod_{i=1..D} (N(i)+1) - 1 \right]$$

(D, N(i) are defined below)

real values of field length 13 and format n.nnnnnnE±nn, with 9 values to a line. Their meaning is defined as follows:

Let $D > 0$ be the number of provided services of the aggregate and $N(d)$ be the maximum population of service number d defined by CREATE statements, $1 \le d \le D$. Then the speeds table for service d contains the values

(2) $$s_d(\underline{n}) = \frac{D(I, d; \underline{n})}{n_d} = \frac{1}{T(I, d; \underline{n})}$$

where $D(...)$ signifies the throughput of service d within the aggregated system I containing $\underline{n}$ customers of D classes, $\underline{n} = (n_1, \ldots, n_D)$. On the other hand, $T(...)$ stands for the corresponding turnaround times, the reciprocal also yielding the speeds according to Little's law.

The following definitions are only given for users interested in how to obtain the speed values $s_d(\underline{n})$, $1 \le d \le D$, from the SPEEDS-table:

(3) $$back(D) = 1$$

(4) $$back(d) = back(d+1) \cdot (N(d+1)+1), \quad 1 \le d \le D-1.$$

Then the index for $s_d(\underline{n})$ within the speeds values for service d and a population vector $\underline{n}$ is

(5) $$\sum_{d=1}^{D} (n_d \cdot back(d))$$

and the index for the speeds table in total is

(6) $$(d - 1) \cdot \#speeds\_d + \sum_{d=1}^{D} (n_d \cdot back(d))$$

where

(7) $$\#speeds\_d = \text{number of speeds for every service } d$$

$$:= \left[\prod_{d=1}^{D} (N(d) + 1) \right] - 1$$

The speed values are arranged in the order defined by the CREATE statements of the AGGREGATE statement. This order is also given just after the %SPEEDS keyword within the aggregate.

G.5. Format of a Trace

G.5.1.Event Trace

If desired, a simulative event trace is written via the standard link name "TRACE". By default it has an EXTEND binding to a file named by the file name generator (see Section 8.2.6.). The record length is 115 characters; there are no print control characters within a trace. A trace file either contains all events of a simulative evaluation (CONTROL TRACEALL) or only those occuring in selected components (CONTROL AT <*evaobj*> TRACE) in an order of increasing model time. All area transitions of processes form events that can be subject to tracing. The events are caused by local processes, i.e., by process declarations or processes generated by CREATE SUBMIT statements, while service calls do not generate a new process. Moreover, every branch of a CONCURRENT statement creates a subprocess recorded in the trace.

The trace consists of a starting message (first line), a list of all actual model parameters, a (possibly very long) set of trace records and a stop message (last line). These lines of additional information are prefixed by "% ", distinguishing them from trace records. All user-defined identifiers utilized in the trace are printed lower-case with a maximum of 12 characters.

The starting message successively contains the experiment name, model type name, model object name and the date and time of simulator generation and simulator start, separated by " / ". The formats used for date and time are installation-dependent.

The following lines specify the name, type, and actual value for each formal parameter of the model type being simulated, concluded by one such line for the predefined model parameter *seed*.

An arbitrary number of trace records follows. Their format depends on the selected TRACEFORMAT (see Section 8.2.1.3.).

The last trace record is followed by a one-line stop message containing the used cpu time, reached model time and termination date and time of the simulation, in this order, separated by " / ".

Additional information can be written chronologically into the trace file by using WRITE FILE *tracefile* statements (see Appendix D.2.). An example trace can be found in Appendix I.7.

G.5.1.1. First Format of Trace Records

In the first format of event records, each line characterizes one event of the simulation under up to 8 aspects of information (a-h):

	columns		**meaning**
a)	1-13	:	Actual model time of the event (real value of format n.nnnnnnEznn)
b)	15-27	:	Record descriptor. The following events are registered:
			- BIRTH, DEATH : Creation and termination of a local process, respectively. Only given for TRACEALL.

- ASK : A service call occuring in a process. The process descends to the component providing that service, i.e., the next activities take place there. The process puts his announcement in the *announce queue*.
- >ENTRY : A process arrives in the *entry area* of the component providing the service called.
- ENTRY>SERVICE : Process transition to the *service area*, i.e., start or resumption of process execution.
- SERVICE>ENTRY : Preemption of a process.
- SERVICE>EXIT : Termination of a service execution.
- ENTRY>EXIT : Resumption of a process which has nothing more to do.
- FORK : Start of a CONCURRENT statement; generation of a new concurrent branch.
- JOIN : Termination of a concurrent branch (not the last one).
- LAST-JOIN : Termination of the last concurrent branch, i.e., termination of a CONCURRENT statement.

c) 29-40 : Type name of the local process registered.

d) 42-53 : PROVIDE name of the service called by its respective USE name.

e) 55-61 : Unambignous number of the local process or a concurrent branch (see FORK).

f) 63-80 : Name of the component providing the previously called service (for ASK and >ENTRY, compare g) or the service just called (otherwise). If this component belongs to an array, the array name appears here and the array index is given in columns 76-80.

g) 82-99 : Only given for ASK and >ENTRY: the name of the component providing the service just called. Again there may be a component array index in columns 95-99.

h) 100-112 : Remaining service request of the process; only given for *request* of a *server* and for *hold* and *spend* for the events ENTRY>SERVICE or SERVICE>ENTRY. In the case of an initial arrival in the *service area*, consequently the actual value of the *request* parameter appears here.

The information fields d and f-h do not appear for the events BIRTH and DEATH.

System-defined components of type *server* called *hold_server* and *spend_server* exist for calls of *hold* or *spend* within services, respectively. These names may be used in the trace (see f and g).

If CONTROL AT <evaobj> TRACE is used for the evaluation, the trace file contains only those trace records which contain the name of the component addressed by <*evaobj*> in columns f or g.

G.5.1.2. Second Format of Trace Records

The event trace of the second format starts with an empty line and a header line, both introduced by "% ". One line per event is given with the following fields:

a)	1- 13	:	Model time.
b)	15- 21	:	A unique number identifying the involved process.
c)	23- 35	:	The kind of event (BIRTH, DEATH, ANNOUNCE, >ENTRY, ENTRY>SERVICE, SERVICE>ENTRY, SERVICE>EXIT, ENTRY>EXIT, EXIT>, FORK, JOIN, LAST-JOIN).
d)	37- 54	:	The component of the event, in case of ANNOUNCE the component of the calling service, if any; columns 50-54 contain an array index, if necessary.
e)	56- 67	:	The involved service, in case of ANNOUNCE the calling service, if any.
f)	69- 80	:	The next component in case of an ANNOUNCE, column 76-80 index, if necessary.
g)	88- 99	:	The service at the next component in case of an ANNOUNCE.
h)	101-113	:	The remaining service request of the process at a server.

The meaning of events is as follows:

BIRTH	:	Creation of a process.
DEATH	:	Termination of a process.
ANNOUNCE	:	An announcement is put into an *announce queue.*
>ENTRY	:	A process enters the *entry area.*
ENTRY>SERVICE	:	A process is transfered from *entry* to *service area.*
SERVICE>ENTRY	:	A process is transfered from *service* to *entry area.*
SERVICE>EXIT	:	A process is transfered from *service* to *exit area.*
ENTRY>EXIT	:	A process is transfered from *entry* to *exit area.*
EXIT>	:	A process leaves an *exit area.*
FORK	:	A concurrent branch of a CONCURRENT statement starts.
JOIN	:	A concurrent branch (not the last one) of a CONCURRENT statement finishes.
LAST-JOIN	:	The last concurrent branch of a CONCURRENT statement finishes.

If not relevant, fields in a line can be left empty. Note, that each FORK generates a new process identification number for the concurrent branch. JOIN and LAST-JOIN mark the end of a concurrent branch respectively the last concurrent branch. System-defined components of type *server* called *hold_server* and *spend_server* appear in the trace.

The appearance of events is controlled by the TRACE and TRACEALL attributes of the CONTROL statement (cf. Section 5.8.2.) and dynamically by TRACE_ON and TRACE_OFF (cf. Appendix D.2.4.).

G.5.1.3. Third Format of Trace Records

The third format of trace records is an extension of the second format (cf. the previous section). Additionally, all calls of component control procedures of components to be traced appear. As event names, ACCEPT, SCHEDULE, DISPATCH and OFFER are used. Only the fields for model time and the component name are filled for these kind of events.

G.5.2.State Trace

The trace file may also contain the so-called state trace. The state trace gathers information about the locations of processes in the model. It is produced either by the user (with the procedure *trace_state*) or automaticly if a possible deadlock situation occurs. The state trace is always written to the trace file, even if you have not given TRACE or TRACEALL in the CONTROL statement. The procedures *trace_on* and *trace_off* have no effect on the state trace.

Each line of the state trace begins with a % in column 1. The state trace consists of three parts, the starting message, the actual information about the process locations and a stop message. The start message begins with an empty line, followed by a line with the format

```
%START OF STATE TRACE. MODELTIME       n.nnnnnnEznnn
                       TIME OF LAST EVENT  n.nnnnnnEznnn
```

The real values denote the model time of the trace output and, respectively, the time when the last event in the model has happened. The last line of the start message is again empty.

The stop message consists of two lines. The first line contains

```
%END OF STATE TRACE.
```

and the second is empty.

The information part of the state trace lists all components that contain at least one process in any of their areas. The order of the components represent the structure of the model. After each component all its subcomponents can be found. Components that contain no processes are not listed. Remember that the model itself is treated like a component and therefore may occur here, too.

Each component description starts with a line that has the format:

columns	meaning
2-10	The word 'COMPONENT'.
13-24	Name of the component.
26-32	If the component belongs to an array, the index of the array is listed here, otherwise this field is empty.

This line is followed by a list of lines with the information about all processes in the component. Each of these lines has the following format:

columns	meaning
6-19	Component area (ANNOUNCE QUEUE, ENTRY AREA, SERVICE AREA, EXIT AREA).
21-32	Name of the process in that area.
34-40	Number of the process. This is the same number as in the event trace.
42-54	Arrival time of the process in the area (real value of format n.nnnnnnEznnn).

If one area contains more than one process, each process information is written on a new line, but the area name is only given for the first process. The processes are listed in ascending order due to their arrival time in the specific area. Areas not containing any process are not listed.

Each component description ends with an empty line. If there are no processes at all in the model, you will get the message: %NO PROCESSES IN THE MODEL.

Example:

```
%
%START OF STATE TRACE.  MODELTIME 1.200000E+00   TIME OF LAST EVENT 1.175000E+00
%
%COMPONENT   c_1
%       ENTRY AREA              proc_1              1       9.000000E-001
%                               proc_2              4       1.100000E+000
%
%COMPONENT   c_2                    2
%       SERVICE AREA            proc_2              5       5.500000E-001
%
%END OF STATE TRACE.
%
```

H. Advice on Error-Identification

If errors emerge during HI-SLANG-compilation, it is advisable to study the corresponding section in the main part of the reference manual for correct treatment. This appendix starts with some remarks on error messages and warnings emitted by the HIT system and concentrates on giving advice for handling errors that result from the SIMULA compiler (this should never occur) or from executing an analyzer.

H.1. Error Messages and Warnings of HIT

Error messages and warnings can be delivered from all passes of the HI-SLANG compiler, from all analyzers and from OMA. Every error message and warning consists of a (error) number and accompanying line number that precedes a message or warning text. Errors are characterized by an 'E', warnings by a 'W' preceding that error number. Errors causing an immediate interruption of compiling or running a program, are identified by an 'A' (ABORT).

Error messages and warnings issued by the FAN system as well as those ones resulting from mobase accesses may appear in any compiler pass and any analyzer execution. Additionally the analyzers may cause errors of the SIMULA run time system (see below).

Upon detection of an error that does not cause an immediate interruption of the compilation the compiler recovers a position from which it can continue the compilation. Every rule of the HI-SLANG syntax disposes of such recovery positions.

Such information is yielded as a warning following the corresponding PASS1 error message in the following format:

W.0473 *line* : One token of the following set was expected: {*a b*}. Skipped source until terminal *c* was found within current line.

This means the tokens *a* and *b* are allowed at this position, and the following source is skipped up to the next recovery position, which is token *c* found in line *line*.

This accounts for the possibility of follow-up errors either not being detected at all or being assigned a wrong message. The additional warning will help you to understand what has happened.

H.2. Unexpected Analyzer Behaviour

If the generated analyzer does not behave as expected, e.g., if it produces false results or error messages, or should the SIMULA run time system even abort, the following steps should be gone through:

H.2.1.Additional Outputs

First, some additional control code should be inserted with WRITE statements into the section of HI-SLANG code presumably containing the error. This, however is only possible if having chosen the simulative method, or if only the experiment description part is concerned. The search for an error may therefore necessitate a temporary change of the solving method. This could be an appropriate application for conditional compilation using compiler directives (%IF, see Section 8.3.).

Additionally, the simulator can be activated to generate a trace (see Appendix G.5) that may deliver valuable information to analyze error states. The trace file, though, could grow very rapidly, but this can be controlled by tracing only at specific components (via CONTROL AT ... TRACE), or at specific time intervals (using *trace_off*, *trace_on*).

Besides the normal event trace a state trace can be additionally requested by calling procedure *trace_state*, and the control messages can be inserted into the trace using WRITE FILE tracefile, ...;

H.2.2.Code Inspection

If the error cannot be located that way, the generated code or rather the SIMULA listing can be investigated by using XREF (if provided by the Simula system used). This is advisable concerning SIMULA run time errors. Here are a few tips:

Run time errors such as division by zero, I/O-errors and the like are detected and announced by the SIMULA system in order to make the generated SIMULA code as short and efficient as possible. The SIMULA run time system has been principally adopted, so it has not been necessary to write a HI-SLANG run time system for basic computations.

These messages may contain line numbers which of course do not refer to the HI-SLANG but to the SIMULA source code. Please use the new compiler option %parm=DEBUG to add HI-SLANG line numbers to the code! It may be helpful to look for a SIMULA error in the SIMULA listing locating an error by line number. It is also possible to use a SIMULA cross reference listing (if available) since the names within the SIMULA source are nearly equal to the names of the HI-SLANG source. They only have an additional prefix to make them unambiguous. Note that in old SIMULA systems the 12 leading characters of a name are significant whereas in HI-SLANG up to 80 characters of a name are significant. The prefix has the following notation:

znnnmmmm

The *z* specifies the mode of the named object (SIMULA procedure or class):

c : service or local process
m : model type
p : PROVIDE name, this is a service

s : implicite state of a process (given by the service parameters)
x : component type, if SIMULA control procedures have been specified (see I.2.)
z : component type and any other variable

In addition there are the characters 'g', 'o', 'q', 'u', 'v' and 'y', which are, however, not important in this context.

The following three digits *nnn* distinctively enumerate all component types and model types. *mmmm* is a distinctive number for each object within a model type or component type.

Example:

```
z0010023hi_slang_name
```

Predefined procedures and constants (Appendix D.) as well as predefined elements of HI-SLANG such as *seed, popul...* and *request* (Chapter 7.) do not have such a prefix because they are already unequivocal.

The generated SIMULA source is very similar to the HI-SLANG source, especially regarding services. HI-SLANG control elements like, e.g., LOOP … END LOOP, LOOP ... UNTIL, AVERAGE ... TIMES, CASE, BRANCH and CONCURRENT, nonexistent in SIMULA, are an exception.

An independent SIMULA class is generated for each of the parallel branches of a CONCURRENT statement, and is assigned a name by the following mode:

```
conc<i>
```

where *i* is a numbering of all CONCURRENT statements in a HI-SLANG source.

A detailed description of the generated code for simulation is given in /BuSt90/.

If a logical error within a HI-SLANG program cannot be found in spite of an intensive search, there may actually be an error in the HIT system. In this case, please contact the University of Dortmund (the exact address can be found in the beginning of this Manual).

H.3. SIMULA Compile Errors

If errors appear while compiling a generated simulator or analyzer using the SIMULA compiler, three cases have to be regarded:

a) Error in standard component procedures written in SIMULA:

   Correct the error or better, write the control procedure in HI-SLANG, since SIMULA control procedures are no longer supported.

b) Constant arithmetic errors:

   Optimizing SIMULA compilers will evaluate constant expressions. This may cause compile time errors similar to run time errors just like "division by zero".

c) Other errors:

   These may be bugs within the HIT system. You should contact the Universität Dortmund if they occur.

I.An Example

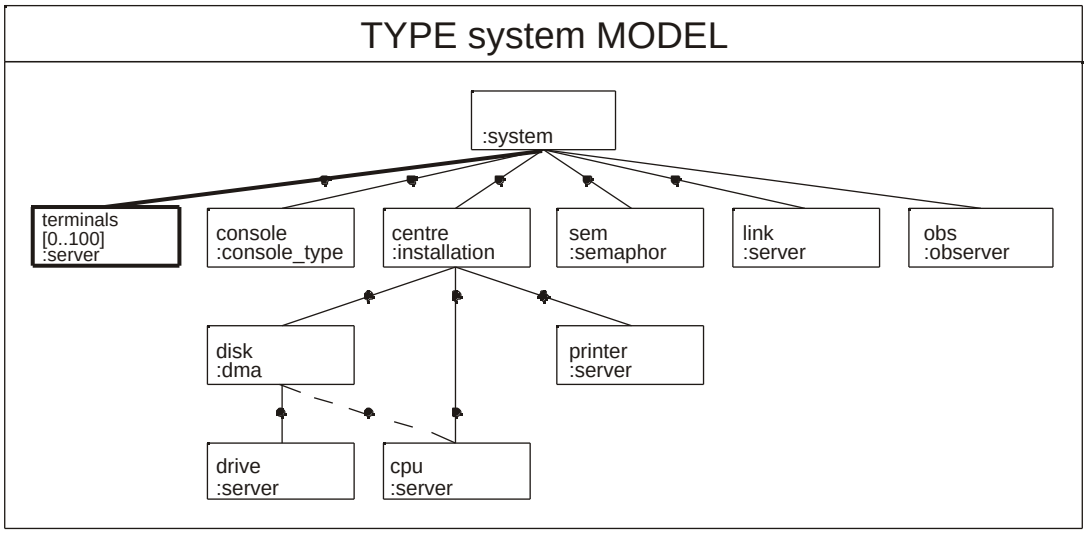
The cooperation of all the HI-SLANG constructs presented will be demonstrated by a complex but well structured and documented example. It is a model of a computing system, somewhat artificial, but uses most of the HI-SLANG constructs.

The first segment describes the model and the experiment step by step. The complete source listing, including XREF, is followed by listings of all the results produced, but compressed from record length 132 to 80 for editing reasons.

I.1. Description of the Model

The model description consists of four main parts: the definition of model type, *system*, and component types *console_type*, *installation* and *dms*. Moreover, the predefined component types *server* , *semaphor* and *observer* are used.

The structure of the model is given using the following HITGRAPHIC display (for HITGRAPHIC see /Sczi93/):



I.1.1. Model Type *system*

The computing system model is parameterized to allow being used in a series of evaluations with different actual parameters. One parameter denotes the number of terminals in the *system*, while the other gives the degree of parallelity of *batch* processes.

Two event streams, *wait_time* and *run_time* follow. The event streams are used for measuring mean time intervals between generation and activation as well as between activation and termination. They are written by UPDATE statements in the service body.

Many processes described by the services *batch* and *dialog* are generated in the model, but there is only one process of a third type, *watch_batch*, observing and sometimes starting *batch* jobs.

These three services of the model will now be described in detail:

batch:

In HI-SLANG the state of a process is modelled by the parameter set of the corresponding service (we didn't want to introduce an explicit state concept).

The state of *batch* jobs can be observed during their existence. It consists of the *owner* of the job, the time of generation, *submitted_at,* the time of activation, *started_at,* and a flag, *terminated,* set as the last action of *batch.*

All these parameters have a default value, so they need not be assigned when creating processes. The default value of *submitted_at* results from a call of the predefined procedure, *time,* at the time of process generation.

Declarations of all services used (provided by other components) follow.

The body starts with a semaphor, P-operation, limitting the number of *batch* jobs running in parallel, thus avoiding overloading the *system*. Storing of the start time, incrementing of *batch_count* and appending the sequence number of the batch job to *owner* for identifying *batch* jobs clearly within print statements, follow. All these statements are timeless (with regard to model time), so the *wait_time* stream can be updated subsequently.

Only the next two statements affect model time: Starting a batch job results in transferring it to and executing it on a computer. Then the *run_time* stream is updated, a termination message produced, flag *terminated* set and semphor V-operation executed.

dialog:

A *dialog* process is represented by a dialog cycle: run 500 times at mean. Thus some *dialog*s will run infinitely, that is as long as the current evaluation runs. It may also occur that one dialog process models more than one user dialog, the new user immediately starting when the other logs out.

One *dialog* cycle is modelled as follows: A mean thinking time of 20 seconds is assumed for each terminal in the array and a transmission time of approx. 2 seconds per command is supposed.

There are two classes of commands: either the user has started a *batch* job (probability 5%), then a message containing the number of the current *dialog* process is sent to the *console*, or he has entered another command which is executed and the result transmitted back to the terminal.

watch_batch:

The service *watch_batch* is to observe *batch* jobs via the process name *last_batch*, which is updated by every execution of a corresponding SUBMIT statement. Periodically, every 300 seconds, a new *batch* job is generated if currently no *batch* job is running (more exactly: *last_batch* terminated) because *system* utilization is low then, and jobs with lower priority can be started by the system. The *watch_batch* process terminates after two hours of model time.

Rest of *system*:

The services of the model use services of different components: an array of *terminals*, a hierarchically modelled computer of type *installation*, a communication channel *link*, a *console* of type *console_type* for message output and a *semaphor* to limit the number of batch jobs running in parallel. The predefined type *semaphor* therefore has to be copied from the standard mobase (modelling base), using %COPY. The behaviour of all these components is controlled by predefined or user written component control procedures given as actual parameters of the component types.

A declaration of six initial *batch* jobs *init_batch* as a process array and the process name *last_batch* follow. In the REFER part all used services of all services are bound to provided services of the above components. After the keyword BEGIN the load of the model is specified: Apart from all processes of the *init_batch* ARRAY (the number of which is written to the *tracefile*), it consists of the *watch_batch* process, one *dialog* process for each terminal and a further *batch* process. The last one has *owner* name "LOAD" and serves to initialize the process name *last_batch*.

I.1.2. Component Type *console_type*

This component type provides one timeless service, *print*, to output two texts on the console (represented by the standard output) in form of a table. The result value of the service is its first parameter, so *print* can be used within expressions.

Every message is prefixed by the current model time formatted as clock time hh:mm:ss. This is achieved by a local multi-valued procedure named *hms* to convert model time in seconds to hours, minutes and seconds. Blanks within clock times in the case of second or minute values less than 10 are avoided by the expressions in the WRITELN statement.

In the body of this component type the table head is printed. The declaration of a global procedure *to_text* follows to convert an integer to its textual representation without leading blanks.

I.1.3. Component Type *installation*

This component type models a computer providing two services of interest: *execute* and *search*. This hierarchical type is planned to be aggregated by the analytic algebraical method. Therefore it is stored in another file, copied here (in the listing this can be detected by finding a letter following the relative line number). Neither the component type nor its provided services are parameterized.

execute:

Processes of type *execute* perform the following cycle about 7 times: First the *cpu* is occupied for approx. 6 seconds, then the computed result may be printed, lasting 0.2, 2 or 20 seconds at mean with different probabilities.

search:

Processes of type *search* execute the following cycle 5 times at mean: First approx. 12 bytes are read from a disk, then the cpu is used for about 50 msec. to process these bytes.

Rest of *installation*:

The services used are bound to three different components: a *cpu*, a *printer* and a *disk*. The first two components are modelled as *servers*, while the *disk* of type *dms* is modelled more detailled.

The REFER part is followed by the statement part of the component body, periodically generating local processes of typ *search*, roughly modelling the overhead of *installation*.

I.1.4. Component Type *dms*

The disk management system, *dms*, provides one service, *transfer*, to read from a *disk*.

transfer:

This service models a read access on a disk using the *cpu* for about 20 msec. and the disk for about 1.2 msec. per byte to be read.

Rest of *dms*:

The body consists of component declarations and the REFER part as usual. The same *cpu* as within component type *installation* is used. Therefore the *cpu* is declared as ENCLOSE, that is virtual. The main declaration of the *cpu* can be found in the body of *installation*.

I.2. Description of the Experiment

The model type *system*, described above, is analyzed by a series of simulations. Type *installation* can be aggregated first, but is not done here. All performance values are written to a table with link name "TABLE" as well as to a dump file "DUMP".

The declaration part of the experiment consists of three variables, *num_term*, *cpu_time*, and *more* to control the series. The statement part follows, consisting of an UNTIL loop. Its body begins with a query on the number of terminals and the maximum cpu time to be used. The values entered for the current evaluation are protocolled, then the EVALUATE statement follows. It also has a declaration part and a statement part.

Declaration Part of EVALUATE:

A model, named *edv* of type *system*, parameterized with the number of terminals read from standard input and the initial value 3 for the semaphor is generated and six evaluation objects are declared. The evaluation objects specify those components as end points of a path in the component hierarchy, which have performance values of interest. Their estimators, kinds of result representation and measuring start times are predefined here to avoid repeating these specifications in every measure statement.

Declarations of load filtering hierarchies follow to allow differentiating the results due to initially causing local processes in higher model layers and the REFER-paths over which services are called transitively down to the evaluation object.

The hierarchies *dia* and *bat* serve to filter out the load portions induced by *dialog* resp. *batch* processes. They are used as abbreviations within the next load filtering hierarchies declared.

The hierarchies *cent_bat*, *link_bat* and *sema_bat* filter out *batch* load portions at the *centre*, the *link* and the *semaphor*, while *cent_dia* and *link_dia* filter out *dialog* load portions at the same components. The hierarchy *sema_bat_p* is more specialized than *sema_bat*: only the load induced by P_operations within *batch* processes is filtered. *Link_merge* is a merge of two hierarchies with the same end point.

Finally there are two hierarchies not starting at *edv*, but at *centre*. They filter loads in *cpu* induced by local processes within *centre*. These two hierarchies have identical start and end points, but describe different paths from start to end point.

Statement Part of EVALUATE:

The statement part is composed of five MEASURE and one simulation CONTROL statement. Only these kinds of statements are allowed within EVALUATE.

The first MEASURE statement concerns the user-defined streams of the model type, i.e., *wait_time* and *run_time* of *batch* processes. Beside mean value and standard deviation of these streams, the values for a non cumulative histogram are calculated.

At computer *centre* (addressed by *cent*) the number and turnaround time of processes caused by *batch* processes, *dialog* processes and any processes (hierarchy *all*) within the model are measured. At channel *link* its occupation and the probability for being utilized are measured separately for load portions caused by *dialog* and *batch* processes as well as both of them.

At the *semaphor* the throughput of all semaphor operations and especially P_operations done by *batch* processes is measured. As before, the estimators given for the evaluation object are used, but the standard deviation for UTILIZATION is not defined.

Finally at the *cpu* OCCUPATION and POPULATION are measured in total (hierarchy *all*) and especially for local processes in *centre*, i.e., the load portion caused by the statement part of *installation*, separately for both possible paths.

The simulation CONTROL statement stops the simulation after the given amount of cpu seconds or after six hours of model time, if POPULATION at *centre* is precise enough at that time. To save cpu time this test is only done at the component addressed via *cent.* Further, a trace is demanded for *semaphor* events.

At the end of all evaluations a histogram for one of the self-defined streams is plotted from the data written to the dump file.

I.3. Listing with XREF

The listing is a concatenation of the compiler listing and the listing produced by the generated analyzer.

I.3.1. Compiler Listing

The compiler listing starts with a formatted printing of the control file and the HI-SLANG source. Completion messages of the different compiler passes and an XREF listing follow.

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     1    University of Dortmund

    1    1a:  %COMMON
    2    2a:     %PARM=LINES=60
    3    4a:  %COMPILER
    4    5a:  |  %PARM=INDENT=|3,XREF,MAXERROR=20
    5    6a:  |  %BIND "INSTALL" TO RefMan.ins
    6   10a:  %END

> FAN        :      Okay.                  Cpu Time used :      0.010 Seconds.
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996
13:18    PAGE     2    University of Dortmund

    1   12a:  % Example as contained and described in
    2   13a:  % the HI-SLANG Reference Manual.
    3   14a:
    4   15a:
    5   16a:  CONSTANT hour : INTEGER DEFAULT 3600 {sec.};
    6   17a:
    7   18a:
    8   19a:
    9   20a:  TYPE system MODEL
   10   21a:  |  (number_of_terminals ,
   11   22a:  |   batch_parallel      : INTEGER);
   12   23a:  |
   13   24a:  |  STREAM   wait_time   : EVENT;
   14   25a:  |           run_time    : EVENT;
   15   26a:  |
   16   27a:  |  VARIABLE batch_count : INTEGER;
   17   28a:  |
   18   29a:  |
   19   30a:  |  TYPE batch SERVICE
   20   31a:  |  |  {process state}
   21   32a:  |  |  (owner        : TEXT    DEFAULT "INIT";
   22   33a:  |  |   submitted_at : REAL    DEFAULT time;
   23   34a:  |  |   started_at   : REAL    DEFAULT -1;
   24   35a:  |  |   terminated   : BOOLEAN DEFAULT FALSE);
   25   36a:  |  |
   26   37a:  |  |  USE SERVICE
   27   38a:  |  |  |  p; v;
   28   39a:  |  |  |  print     (t1, t2:TEXT) RESULT TEXT;
   29   40a:  |  |  |  transmit  (amount:REAL  DEFAULT 2.0);
   30   41a:  |  |  |  execute;
   31   42a:  |  |  END USE;
   32   43a:  |  |
   33   44a:  |  BEGIN
   34   45a:  |  |  p;
   35   46a:  |  |  started_at   := time;
   36   47a:  |  |  batch_count  := batch_count + 1;
   37   48a:  |  |
   38   49a:  |  |  WRITE TEXT owner,
   39   50a:  |  |        batch_count::3, '/', owner;
   40   51a:  |  |
   41   52a:  |  |  print(owner, "starting   after " &
   42   53a:  |  |  to_text(time-submitted_at) & " sec.");
   43   54a:  |  |
   44   55a:  |  |  UPDATE wait_time BY time-submitted_at;
   45   56a:  |  |  transmit;
   46   57a:  |  |  execute;
   47   58a:  |  |  UPDATE run_time BY time-started_at;
   48   59a:  |  |
   49   60a:  |  |  print(owner, "terminated after " &
   50   61a:  |  |        to_text(time-started_at) & " sec.");
   51   62a:  |  |
   52   63a:  |  |  terminated := TRUE;
   53   64a:  |  |  v;
   54   65a:  |  END TYPE batch;
   55   66a:  |
   56   67a:  |  TYPE dialog SERVICE (terminal_number:INTEGER);
   57   68a:  |  |  USE
   58   69a:  |  |  |  SERVICE
   59   70a:  |  |  |  transmit (amount:REAL);
   60   71a:  |  |  |  print    (t1, t2:TEXT) RESULT TEXT;
```

```
HIT   Version 3.6.000   File RefMan.hit                                1996-03-25
13:18    PAGE     3    University of Dortmund

   61   72a:  |  |  |  compute;
   62   73a:  |  |  |  SERVICE ARRAY
   63   74a:  |  |  |  think (time:REAL);
   64   75a:  |  |  END USE;
   65   76a:  |  |
   66   77a:  |  BEGIN
   67   78a:  |  |  AVERAGE 500 TIMES
   68   79a:  |  |  {mean: 500 iterations}
   69   80a:  |  |  LOOP
   70   81a:  |  |  |  think [terminal_number] (negexp(1/20));
   71   82a:  |  |  |  transmit (negexp(1/2));
   72   83a:  |  |  |
   73   84a:  |  |  |  IF draw(0.05)
   74   85a:  |  |  |  |  THEN  {enter batch job from dialog}
   75   86a:  |  |  |  |
   76   87a:  |  |  |  |  SUBMIT batch (print("DIALOG" &
   77   88a:  |  |  |  |         to_text(terminal_number),""))
   78   89a:  |  |  |  |         NAME last_batch AFTER negexp(1);
   79   90a:  |  |  |  |  ELSE
   80   91a:  |  |  |  |  compute;
   81   92a:  |  |  |  |  transmit (negexp(1/5));
   82   93a:  |  |  |  END IF;
   83   94a:  |  |  END LOOP;
   84   95a:  |  END TYPE dialog;
   85   96a:  |
   86   97a:  |
   87   98a:  |  TYPE watch_batch SERVICE;
   88   99a:  |  |  {uses process name last_batch to}
   89  100a:  |  |  {inspect batch jobs every 300 sec.}
   90  101a:  |  |  USE
   91  102a:  |  |  |  SERVICE
   92  103a:  |  |  |  print   (t1, t2:TEXT) RESULT TEXT;
   93  104a:  |  |  |  SERVICE ARRAY
   94  105a:  |  |  |  display (time:REAL);
   95  106a:  |  |  END USE;
   96  107a:  |  |
   97  108a:  |  BEGIN
   98  109a:  |  |  LOOP
   99  110a:  |  |  |  display[0](300);
  100  111a:  |  |  |
  101  112a:  |  |  |  IF last_batch.terminated
  102  113a:  |  |  |  |  {no batch running}
  103  114a:  |  |  |  |  THEN
  104  115a:  |  |  |  |  SUBMIT batch (print("SYSTEM",""))
  105  116a:  |  |  |  |    NAME last_batch;
  106  117a:  |  |  |  END IF;
  107  118a:  |  |  |
  108  119a:  |  |  END LOOP UNTIL time > 1*hour;
  109  120a:  |  |  trace_off;
  110  121a:  |  END TYPE watch_batch;
  111  122a:  |
  112  123a:  %COPY "SEMAPHOR"
  132  124a:  %COPY "INSTALL"
  133    1c:  |
  134    2c:  % component type installation included in Refman.hit
  135    3c:  |
  136    4c:  |  TYPE installation COMPONENT; {may be aggregated}
  137    5c:  |  |  PROVIDE
  138    6c:  |  |  |  SERVICE
  139    7c:  |  |  |  execute;
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     4    University of Dortmund

  140    8c:  |  |  |  search;
  141    9c:  |  |  END PROVIDE;
  142   10c:  |  |
  143   11c:  |  |
  144   12c:  |  |  TYPE dms COMPONENT; {--------------------------}
  145   13c:  |  |  |  PROVIDE
  146   14c:  |  |  |  |  SERVICE
  147   15c:  |  |  |  |  transfer (bytes:INTEGER);
  148   16c:  |  |  |  END PROVIDE;
  149   17c:  |  |  |
  150   18c:  |  |  |
  151   19c:  |  |  |  TYPE transfer SERVICE (bytes:INTEGER);
  152   20c:  |  |  |  |  USE
  153   21c:  |  |  |  |  |  SERVICE
  154   22c:  |  |  |  |  |  overhead     (time   :REAL);
  155   23c:  |  |  |  |  |  drive_access (amount :REAL);
  156   24c:  |  |  |  |  END USE;
  157   25c:  |  |  |  |
  158   26c:  |  |  |  BEGIN
  159   27c:  |  |  |  |  overhead     (negexp(1/0.02));
  160   28c:  |  |  |  |  drive_access (negexp(1/(bytes/1.2E3)));
  161   29c:  |  |  |  END TYPE transfer;
  162   30c:  |  |  |
  163   31c:  |  |  |
  164   32c:  |  |  |  ENCLOSE   cpu   :server;{cpu of installation}
  165   33c:  |  |  |  COMPONENT drive :server;
  166   34c:  |  |  |
  167   35c:  |  |  |
  168   36c:  |  |  |  REFER transfer TO cpu, drive
  169   37c:  |  |  |  EQUATING
  170   38c:  |  |  |  |  transfer.overhead      WITH cpu  .request;
  171   39c:  |  |  |  |  transfer.drive_access  WITH drive.request;
  172   40c:  |  |  |  END REFER;
  173   41c:  |  |  |
  174   42c:  |  |  END TYPE dms; {---------------------------------}
  175   43c:  |  |
  176   44c:  |  |  TYPE execute SERVICE;
  177   45c:  |  |  |  USE
  178   46c:  |  |  |  |  SERVICE
  179   47c:  |  |  |  |  calculate       (time  :REAL);
  180   48c:  |  |  |  |  print           (amount:REAL);
  181   49c:  |  |  |  END USE;
  182   50c:  |  |  |
  183   51c:  |  |  BEGIN
  184   52c:  |  |  |  WHILE draw(0.875)    {mean: 7 iterations}
  185   53c:  |  |  |  LOOP                            {1/(1-p)}
  186   54c:  |  |  |  |
  187   55c:  |  |  |  |  calculate (LET time := negexp(1/6));
  188   56c:  |  |  |  |
  189   57c:  |  |  |  |  BRANCH
  190   58c:  |  |  |  |  |  PROB 0.32 :  print (negexp(1/0.2));
  191   59c:  |  |  |  |  |  PROB 0.20 :  print (negexp(1/2));
  192   60c:  |  |  |  |  |  PROB 0.03 :  print (negexp(1/20));
  193   61c:  |  |  |  |  END BRANCH;
  194   62c:  |  |  |  END LOOP;
  195   63c:  |  |  END TYPE execute;
  196   64c:  |  |
  197   65c:  |  |
  198   66c:  |  |  TYPE search SERVICE;
  199   67c:  |  |  |  USE
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     5    University of Dortmund

  200   68c:  |  |  |  |  SERVICE
  201   69c:  |  |  |  |  disk_access  (much:INTEGER);
  202   70c:  |  |  |  |  cpu_access   (time:REAL);
  203   71c:  |  |  |  END USE;
  204   72c:  |  |  |
  205   73c:  |  |  BEGIN
  206   74c:  |  |  |  LOOP
  207   75c:  |  |  |  |  disk_access (12); {mean: read 12 bytes}
  208   76c:  |  |  |  |  cpu_access  (negexp(1/0.05));
  209   77c:  |  |  |  |
  210   78c:  |  |  |  END LOOP UNTIL draw(0.2);{mean: 5 iterations}
  211   79c:  |  |  END TYPE search;
  212   80c:  |  |
  213   81c:  |  |
  214   82c:  |  |  COMPONENT
  215   83c:  |  |  cpu     : server (LET dispatch:=shared);
  216   84c:  |  |  printer : server;
  217   85c:  |  |  disk    : dms;
  218   86c:  |  |
  219   87c:  |  |
  220   88c:  |  |  REFER execute, search TO cpu, printer, disk
  221   89c:  |  |  EQUATING
  222   90c:  |  |  |  execute.calculate   WITH cpu     .request;
  223   91c:  |  |  |  execute.print       WITH printer .request;
  224   92c:  |  |  |
  225   93c:  |  |  |  search .disk_access WITH disk    .transfer;
  226   94c:  |  |  |  search .cpu_access  WITH cpu     .request;
  227   95c:  |  |  END REFER;
  228   96c:  |  |
  229   97c:  |  BEGIN
  230   98c:  |  |
  231   99c:  |  |  CREATE 1 PROCESS search EVERY negexp(1/15);
  232  100c:  |  |  {modelling of an overhead}
  233  101c:  |  |
  234  102c:  |  END TYPE installation;
  235  125a:  %COPY "OBSERVER"
  292  126a:  |
  293  127a:  |  COMPONENT
  294  128a:  |  terminals : ARRAY [0..100] OF server;
  295  129a:  |  centre    : installation (LET schedule:=fcfs);
  296  130a:  |  link      : server (LET dispatch:= sdequal
  297  131a:  |                     ([[1, 3, 7], [1.5, 1, 0.5]]));
  298  132a:  |
  299  133a:  |  console   : console_type;
  300  134a:  |  sem       : semaphor (batch_parallel);
  301  135a:  |  obs       : observer (2000, TRUE);
  302  136a:  |
  303  137a:  |  PROCESS
  304  138a:  |  init_batch: ARRAY [1..6] OF batch;
  305  139a:  |  last_batch: NAME FOR batch;
  306  140a:  |
  307  141a:  |
  308  142a:  |  REFER   dialog,  batch, watch_batch TO
  309  143a:  |        terminals, link,  centre, console, sem
  310  144a:  |  EQUATING
  311  145a:  |  |  dialog .think         WITH terminals .request;
  312  146a:  |  |  dialog .transmit      WITH link      .request;
  313  147a:  |  |  dialog .compute       WITH centre    .search;
  314  148a:  |  |  dialog .print         WITH console   .print;
  315  149a:  |  |
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     6    University of Dortmund

  316  150a:  |  |  batch  .p            WITH sem       .p;
  317  151a:  |  |  batch  .v            WITH sem       .v;
  318  152a:  |  |  batch  .transmit     WITH link      .request;
  319  153a:  |  |  batch  .execute      WITH centre    .execute;
  320  154a:  |  |  batch  .print        WITH console   .print;
  321  155a:  |  |
  322  156a:  |  |  watch_batch.print    WITH console   .print;
  323  157a:  |  |  watch_batch.display  WITH terminals .request;
  324  158a:  |  END REFER;
  325  159a:  |
  326  160a:  BEGIN {load of the model}
  327  161a:  |
  328  162a:  |  WRITELN FILE tracefile, "initial batch load:",
  329  163a:  |                    init_batch.upper_bounds[1] ;
  330  164a:  |
  331  165a:  |  CREATE 1 PROCESS watch_batch;
  332  166a:  |
  333  167a:  |  {one dialog for every terminal}
  334  168a:  |  BLOCK
  335  169a:  |  |  VARIABLE run : INTEGER;
  336  170a:  |  BEGIN
  337  171a:  |  |  FOR run:=1 STEP 1 UNTIL number_of_terminals
  338  172a:  |  |  LOOP
  339  173a:  |  |  |  CREATE 1 PROCESS dialog(run)
  340  174a:  |  |  |  AT normal(15*run, 5);
  341  175a:  |  |  END LOOP;
  342  176a:  |  END BLOCK;
  343  177a:  |
  344  178a:  |  SUBMIT batch("LOAD") NAME last_batch;
  345  179a:  |
  346  180a:  END TYPE system;
  347  181a:
  348  182a:
  349  183a:  TYPE console_type COMPONENT;
  350  184a:  |  PROVIDE
  351  185a:  |  |  SERVICE
  352  186a:  |  |  print (t1, t2:TEXT) RESULT TEXT;
  353  187a:  |  END PROVIDE;
  354  188a:  |
  355  189a:  |
  356  190a:  |  TYPE print SERVICE (t1, t2:TEXT) RESULT TEXT;
  357  191a:  |  |
  358  192a:  |  |  PROCEDURE hms RESULT INTEGER,INTEGER,INTEGER;
  359  193a:  |  |  BEGIN
  360  194a:  |  |  |  RESULT time //  hour MOD 24,   {hours}
  361  195a:  |  |  |         time MOD hour //  60,   {min. }
  362  196a:  |  |  |         time MOD 60;           {sec. }
  363  197a:  |  |  END PROCEDURE hms;
  364  198a:  |  |
  365  199a:  |  |  VARIABLE h, m, s : INTEGER;
  366  200a:  |  BEGIN
  367  201a:  |  |
  368  202a:  |  |  (h, m, s) := hms;
  369  203a:  |  |  RESULT t1;
  370  204a:  |  |
  371  205a:  |  |  IF t2="" THEN t2 := "submitted"; END IF;
  372  206a:  |  |
  373  207a:  |  |  WRITELN  h::2,  ':', m//10::1, m MOD 10::1,
  374  208a:  |  |           ':', s//10::1, s MOD 10::1,
  375  209a:  |  |           " |", t1::12, "| ", t2;
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     7    University of Dortmund

  376  210a:  |  END TYPE;
  377  211a:  |
  378  212a:  |
  379  213a:  BEGIN
  380  214a:  |
  381  215a:  |  WRITELN " TIME      NO / OWNER   STATE OF BATCH";
  382  216a:  |  WRITELN "---------+------------+-----------------------";
  383  217a:  END TYPE console_type;
  384  218a:
  385  219a:
  386  220a:
  387  221a:  PROCEDURE to_text (i:INTEGER) RESULT TEXT;
  388  222a:  |
  389  223a:  |  VARIABLE t:TEXT;
  390  224a:  BEGIN
  391  225a:  |  IF i<>0
  392  226a:  |  |  THEN
  393  227a:  |  |  WRITE TEXT t,  i:: entier(log(abs(i)))+1;
  394  228a:  |  |  ELSE
  395  229a:  |  |  WRITE TEXT t, 'O';
  396  230a:  |  END IF;
  397  231a:  |
  398  232a:  |  RESULT t;
  399  233a:  END PROCEDURE to_text;
  400  234a:
  401  235a:
  402  236a:  EXPERIMENT analysis METHOD SIMULATIVE;
  403  237a:  |
  404  238a:  |  VARIABLE
  405  239a:  |    cpu_time  : REAL;
  406  240a:  |    num_term  : INTEGER DEFAULT 1;
  407  241a:  |    more      : CHARACTER;
  408  242a:  |
  409  243a:  BEGIN
  410  244a:  |  LOOP
  411  245a:  |  |  WRITELN;
  412  246a:  |  |  WRITELN "Please enter cpu time and #terminals";
  413  247a:  |  |  READLN          cpu_time,          num_term;
  414  248a:  |  |  WRITELN         cpu_time::3::21,   num_term;
  415  249a:  |  |  WRITELN;
  416  250a:  |  |
  417  251a:  |  |  EVALUATE
  418  252a:  |  |  |  MODEL edv : system (num_term, 3);
  419  253a:  |  |  |
  420  254a:  |  |  |  EVALUATIONOBJECT
  421  255a:  |  |  |     syst        VIA edv,
  422  256a:  |  |  |     cent        VIA edv.centre,
  423  257a:  |  |  |     link        VIA edv.link,
  424  258a:  |  |  |     sem         VIA edv.sem
  425  259a:  |  |  |  DEFAULT
  426  260a:  |  |  |     ESTIMATOR   MEAN, STANDARDDEVIATION
  427  261a:  |  |  |     OUTPUT      TABLE "TABLE",
  428  262a:  |  |  |              DUMPFILE "DUMP";
  429  263a:  |  |  |
  430  264a:  |  |  |  EVALUATIONOBJECT
  431  265a:  |  |  |     cpu         VIA edv.centre.cpu
  432  266a:  |  |  |  DEFAULT
  433  267a:  |  |  |     ESTIMATOR   CONFIDENCE LEVEL 90
  434  268a:  |  |  |     START       MODELTIME 300;
  435  269a:  |  |  |
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     8    University of Dortmund

  436  270a:  |  |  |
  437  271a:  |  |  |  HIERARCHY
  438  272a:  |  |  |     bat         DEFAULT (edv, batch);
  439  273a:  |  |  |     dia         DEFAULT (edv, dialog);
  440  274a:  |  |  |
  441  275a:  |  |  |     cent_bat    DEFAULT  bat.(centre);
  442  276a:  |  |  |     link_bat    DEFAULT  bat.(link);
  443  277a:  |  |  |
  444  278a:  |  |  |     sema_bat    DEFAULT  bat.(sem);
  445  279a:  |  |  |     sema_bat_p  DEFAULT  (edv,batch,p).(sem);
  446  280a:  |  |  |
  447  281a:  |  |  |     cent_dia    DEFAULT  dia.(centre);
  448  282a:  |  |  |     link_dia    DEFAULT  dia.(link);
  449  283a:  |  |  |
  450  284a:  |  |  |     link_merge  MERGE    link_dia, link_bat;
  451  285a:  |  |  |
  452  286a:  |  |  |     cpu1_own DEFAULT    (edv.centre,search).(cpu);
  453  287a:  |  |  |     cpu2_own DEFAULT    (edv.centre).(disk).(cpu);
  454  288a:  |  |  |
  455  289a:  |  |  BEGIN
  456  290a:  |  |  |
  457  291a:  |  |  |  MEASURE wait_time, run_time
  458  292a:  |  |  |     AT      syst
  459  293a:  |  |  |     ESTIMATOR MEAN, STANDARDDEVIATION,
  460  294a:  |  |  |     FREQUENCY INTERVAL
  461  295a:  |  |  |               [ 0..1,1..2,2..5,5..10,
  462  296a:  |  |  |                 10..50,50..1000];
  463  297a:  |  |  |
  464  298a:  |  |  |  MEASURE POPULATION, TURNAROUNDTIME
  465  299a:  |  |  |     AT      cent
  466  300a:  |  |  |     DUE TO  cent_bat, cent_dia, all
  467  301a:  |  |  |     OUTPUT  TABLE "SYSOUT", TABLE "TABLE"
  468  302a:  |  |  |     START   EVENTS 2 DUE TO cent_bat;
  469  303a:  |  |  |
  470  304a:  |  |  |  MEASURE OCCUPATION, UTILIZATION
  471  305a:  |  |  |     AT      link
  472  306a:  |  |  |     DUE TO  link_dia, link_bat, link_merge;
  473  307a:  |  |  |
  474  308a:  |  |  |  MEASURE THROUGHPUT
  475  309a:  |  |  |     AT      sem
  476  310a:  |  |  |     DUE TO  sema_bat, sema_bat_p
  477  311a:  |  |  |     OUTPUT  TABLE "TABLE";
  478  312a:  |  |  |
  479  313a:  |  |  |  MEASURE OCCUPATION, POPULATION
  480  314a:  |  |  |     AT      cpu
  481  315a:  |  |  |     DUE TO  cpu1_own, cpu2_own, all;
  482  316a:  |  |  |
  483  317a:  |  |  |
  484  318a:  |  |  |  CONTROL
  485  319a:  |  |  |     AT cent
  486  320a:  |  |  |        STOP  CPUTIME cpu_time
  487  321a:  |  |  |         OR   MODELTIME 6*hour
  488  322a:  |  |  |         OR   CONFIDENCE LEVEL 91 WIDTH 20
  489  323a:  |  |  |              MEASURE POPULATION
  490  324a:  |  |  |     AT sem TRACE;
  491  325a:  |  |  |
  492  326a:  |  |  END EVALUATE;
  493  327a:  |  |
  494  328a:  |  |  WRITELN;
  495  329a:  |  |  WRITELN "Another experiment ? (y,n)";
```

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:18    PAGE     9    University of Dortmund

  496  330a:  |  |  READLN   more;
  497  331a:  |  |
  498  332a:  |  END LOOP UNTIL more # 'y';
  499  333a:  |
  500  334a:  |  HISTOGRAM
  501  335a:  |     PLOT MEASURE          "wait_time"
  502  336a:  |          EVALUATIONOBJECT "syst"
  503  337a:  |          HIERARCHY        "all"
  504  338a:  |          INPUT            "DUMP";
  505  339a:  |
  506  340a:  END EXPERIMENT analysis;

> PASS 1     :      Okay.                  Cpu Time used :      1.060 Seconds.
```

```
Identifier  S/: Type or Constant   Line Access Pairs    * XREF *     1996-03-25
13:18    PAGE    10    University of Dortmund


             :      COLLECT          13 d
             :      COLLECT          14 d

abs          S REAL PROCEDURE       393 p
amount       : REAL VARIABLE         29 d
amount       : REAL VARIABLE         59 d
amount       : REAL VARIABLE        155 d
amount       : REAL VARIABLE        180 d
analysis     :      EXPERIMENT      402 d
answer       : TEXT VARIABLE        241 d    265 w    266 r    276 r

bat          :      HIERARCHY       438 d    441 r    442 r    444 r
batch        : TYPE SERVICE          19 d     76 r    104 r    304 r    305 r
                                    308 r    316 r    317 r    318 r    319 r
                                    320 r    344 r    438 r    445 r
batch_count  : INTEGER VARIABLE      16 d     36 rw*   39 r
batch_parall : INTEGER VARIABLE      11 d    300 p
bytes        : INTEGER VARIABLE     147 d    151 d    160 r

calculate    : TYPE SERVICE         179 d    187 r    222 r
cent         :      EVAOBJECT       422 d    464 r    484 r
cent_bat     :      HIERARCHY       441 d    464 r    468 r
cent_dia     :      HIERARCHY       447 d    464 r
centre       :      COMPONENT       295 d    309 r    313 r    319 r    422 r
                                    431 r    441 r    447 r    452 r    453 r
compute      : TYPE SERVICE          61 d     80 r    313 r
console      :      COMPONENT       299 d    309 r    314 r    320 r    322 r
console_type : TYPE COMPONENT       299 r    349 d
cpu          :      ENCLOSE         164 d    168 r    170 r    453 r
cpu          :      COMPONENT       215 d    220 r    222 r    226 r    431 r
                                    452 r
cpu          :      EVAOBJECT       431 d    479 r
cpu1_own     :      HIERARCHY       452 d    479 r
cpu2_own     :      HIERARCHY       453 d    479 r
cpu_access   : TYPE SERVICE         202 d    208 r    226 r
cpu_time     : REAL VARIABLE        405 d    413 w    414 r    486 r
cpu_time     S REAL PROCEDURE       251 r

dia          :      HIERARCHY       439 d    447 r    448 r
dialog       : TYPE SERVICE          56 d    308 r    311 r    312 r    313 r
                                    314 r    339 r    439 r
digit        S BOOLEAN PROCEDURE    275 r
disk         :      COMPONENT       217 d    220 r    225 r    453 r
disk_access  : TYPE SERVICE         201 d    207 r    225 r
dispatch     : SIMULA SIMULA        215 r    296 r
display      : TYPE SERVICE          94 d     99 r    323 r
dms          : TYPE COMPONENT       144 d    217 r
draw         S BOOLEAN PROCEDURE     73 r    184 r    210 r
drive        :      COMPONENT       165 d    168 r    171 r
drive_access : TYPE SERVICE         155 d    160 r    171 r

edv          :      MODEL           418 d    421 r    422 r    423 r    424 r
                                    431 r    438 r    439 r    445 r    452 r
                                    453 r
entier       S INTEGER PROCEDURE    393 r
execute      : TYPE SERVICE          30 d     46 r    319 r
execute      : TYPE SERVICE         139 d    176 d    220 r    222 r    223 r
                                    319 r

fcfs         S      PROCEDURE       295 p
first        : CHARACTER VARIABLE   242 d    266 w    269 r    275 p    278 r

h            : INTEGER VARIABLE     365 d    368 w    373 r
hms          :      PROCEDURE       358 d    368 r
hold         S      PROCEDURE       247 r
```

```
Identifier  S/: Type or Constant   Line Access Pairs    * XREF *     1996-03-25
13:18    PAGE    11    University of Dortmund

hour         : 3600                   5 d    108 r    360 r    361 r    487 r

i            : INTEGER VARIABLE     387 d    391 r    393 rp*
init_batch   : PROCESS ARRAY( 1)    304 d    329 r
installation : TYPE COMPONENT       136 d    295 r
interactive  : BOOLEAN VARIABLE     238 d    254 r    272 w

last_batch   :      PROCESS          78 w    101 r    105 w    305 d    344 w
lastitem     S BOOLEAN PROCEDURE    264 r
link         :      COMPONENT       296 d    309 r    312 r    318 r    423 r
                                    442 r    448 r
link         :      EVAOBJECT       423 d    470 r
link_bat     :      HIERARCHY       442 d    450 r    470 r
link_dia     :      HIERARCHY       448 d    450 r    470 r
link_merge   :      HIERARCHY       450 d    470 r
log          S REAL PROCEDURE       393 p

m            : INTEGER VARIABLE     365 d    368 w    373 r*
maxreal      S  1.6342665E+308      271 r
more         : CHARACTER VARIABLE   407 d    496 w    498 r
much         : INTEGER VARIABLE     201 d

negexp       S REAL PROCEDURE        70 p     71 p     78 r     81 p    159 p
                                    160 p    187 p    190 p    191 p    192 p
                                    208 p    231 r
normal       S REAL PROCEDURE       340 r
num_term     : INTEGER VARIABLE     406 d    413 w    414 r    418 p
number_of_te : INTEGER VARIABLE      10 d    337 r

obs          :      COMPONENT       301 d
obs_interval : REAL VARIABLE        237 d    289 p
observer     : TYPE COMPONENT       237 d    301 r
ok           : BOOLEAN VARIABLE     243 d    267 w    279 w    282 r
overhead     : TYPE SERVICE         154 d    159 r    170 r
owner        : TEXT VARIABLE         21 d     38 r     39 r     41 p     49 p

p            : TYPE SERVICE          27 d     34 r    316 r    445 r
p            : TYPE SERVICE         116 d    121 d    316 r
print        : TEXT SERVICE          28 d     41 r     49 r    320 r
print        : TEXT SERVICE          60 d     76 p    314 r
print        : TEXT SERVICE          92 d    104 p    322 r
print        : TYPE SERVICE         180 d    190 r    191 r    192 r    223 r
print        : TEXT SERVICE         314 r    320 r    322 r    352 d    356 d
printer      :      COMPONENT       216 d    220 r    223 r

request      S TYPE SERVICE         170 r    171 r    222 r    223 r    226 r
                                    311 r    312 r    318 r    323 r
run          : INTEGER VARIABLE     335 d    337 w    339 p    340 r
run_time     : STREAM EVENT          14 dr    47 w    457 r

s            : INTEGER VARIABLE     365 d    368 w    374 r*
schedule     : SIMULA SIMULA        295 r
sdequal      S      PROCEDURE       296 p
search       : TYPE SERVICE         140 d    198 d    220 r    225 r    226 r
                                    231 r    313 r    452 r
sem          : INTEGER VARIABLE     119 d    123 rw*  128 rw*
sem          :      COMPONENT       300 d    309 r    316 r    317 r    424 r
                                    444 r    445 r
sem          :      EVAOBJECT       424 d    474 r    484 r
sem_init     : INTEGER VARIABLE     114 d    119 r
sema_bat     :      HIERARCHY       444 d    474 r
sema_bat_p   :      HIERARCHY       445 d    474 r
semaphor     : TYPE COMPONENT       114 d    300 r
server       S TYPE COMPONENT       164 r    165 r    215 r    216 r    294 r
                                    296 r
shared       S      PROCEDURE       215 p
```

```
Identifier  S/: Type or Constant   Line Access Pairs    * XREF *     1996-03-25
13:18    PAGE    12    University of Dortmund

started_at   : REAL VARIABLE         23 d     35 w     47 r     50 r
stop_evaluat S      PROCEDURE       270 r
submitted_at : REAL VARIABLE         22 d     42 r     44 r
syst         :      EVAOBJECT       421 d    457 r
system       : TYPE MODEL            9 d    418 r

t            : TEXT VARIABLE        389 d    393 r    395 r    398 r
t1           : TEXT VARIABLE         28 d
t1           : TEXT VARIABLE         60 d
t1           : TEXT VARIABLE         92 d
t1           : TEXT VARIABLE        352 d    356 d    369 r    375 r
t2           : TEXT VARIABLE         28 d
t2           : TEXT VARIABLE         60 d
t2           : TEXT VARIABLE         92 d
t2           : TEXT VARIABLE        352 d    356 d    371 rw*  375 r
terminal_num : INTEGER VARIABLE      56 d     70 r     77 p
terminals    : COMPONENT ARRAY( 1)  294 d    309 r    311 r    323 r
terminated   : BOOLEAN VARIABLE      24 d     52 w    101 r
think        : TYPE SERVICE          63 d     70 r    311 r
time         : REAL VARIABLE         63 d
time         : REAL VARIABLE         94 d
time         : REAL VARIABLE        154 d
time         : REAL VARIABLE        179 d    187 r
time         : REAL VARIABLE        202 d
time         S REAL PROCEDURE        22 r     35 r     42 r     44 r     47 r
                                     50 r    108 r    250 r    360 r    361 r
                                    362 r
to_text      : TEXT PROCEDURE        42 r     50 r     77 r    387 d
trace_off    S      PROCEDURE       109 r
tracefile    S OUTFILE CONSTANT     328 r
transfer     : TYPE SERVICE         147 d    151 d    168 r    170 r    171 r
                                    225 r
transfer_res S      PROCEDURE       248 r
transmit     : TYPE SERVICE          29 d     45 r    318 r
transmit     : TYPE SERVICE          59 d     71 r     81 r    312 r

upper_bounds S INTEGER ARRAY( 1)    329 r

v            : TYPE SERVICE          27 d     53 r    317 r
v            : TYPE SERVICE         116 d    126 d    317 r

wait_time    : STREAM EVENT          13 dr    44 w    457 r
watch_batch  : TYPE SERVICE          87 d    308 r    322 r    323 r    331 r
watch_interv : REAL VARIABLE        240 d    247 p    271 w    276 w    277 r
watcher      : TYPE SERVICE         240 d    289 r


>   139 different Objects,   110 different Identifiers.

Number  Line : Description of Errors or Warnings detected by PASS 2
-------------+------------------------------------------------------------------
W.0526    42 : 'to_text' is a non-local access in 'system'.
          53a  in LINK=CONTROL
W.0526    50 : 'to_text' is a non-local access in 'system'.
          61a  in LINK=CONTROL
W.0526    77 : 'to_text' is a non-local access in 'system'.
          88a  in LINK=CONTROL
W.0526   108 : 'hour' is a non-local access in 'system'.
         119a  in LINK=CONTROL
W.0526   360 : 'hour' is a non-local access in 'hms'.
         194a  in LINK=CONTROL
W.0526   361 : 'hour' is a non-local access in 'hms'.
```

```
HIT   Version 3.6.000   COMPILER MESSAGES                          1996-03-25
13:18    PAGE    13    University of Dortmund

         195a  in LINK=CONTROL

> PASS 2     :      Only       6 Warnings.  Cpu Time used :      0.470 Seconds.

> PASS 3     :      Okay.                   Cpu Time used :      0.100 Seconds.

> SCG        :      Okay.                   Cpu Time used :      0.910 Seconds.


> T O T A L  :      Only       6 Warnings.  Cpu Time used :      2.770 Seconds.

                  Compile  Rate :  203.968 Lines/Sec.
```

The warnings are caused by non-local access of the prodedure ´to_text´ and the constant ´hour´.

I.3.2. Analyzer Listing

The analyzer listing starts with a control file listing. Then for every evaluation performed some identifying data, called header information is given, followed by the solver information generated. When the evaluation is finished, a completion message and a footer is generated.

```
HIT   Version 3.6.000   File RefMan.hit                              1996-03-25
13:19    PAGE     1    University of Dortmund

    1    1a:  %COMMON
    2    2a:     %PARM=LINES=60
    3    7a:
    4    8a:  %ANALYZER
    5    9a:     %PARM=UPDATES,MINMAX
    6   10a:  %END

> FAN        :      Okay.                  Cpu Time used :      0.010 Seconds.



Control                 File RefMan.hit

Experiment Name         analysis

Model Type              system
Model Name              edv

Model Parameters        Name                Type                Actual Value
                        ========================================================
                        number_of_te        INTEGER                        5
                        batch_parall        INTEGER                        3
                        seed                INTEGER                       13

Used Method             SIMULATIVE

Date of Compile         1996-03-25              Time of Compile     13:18
Start Date of Run       1996-03-25              Start Time of Run   13:20



REASONS FOR SELECTION:

SIMUL requested.



EVALUATION TRACE:

EVENTS 2 DUE TO HIERARCHY cent_bat: Start condition reached in MEASURE statement:
EVALUATIONOBJECT cent, STREAM POPULATION, DUE TO HIERARCHY cent_bat, MODELTIME
98.884071

...

Start condition reached in MEASURE statement:
EVALUATIONOBJECT cpu, STREAM POPULATION, DUE TO HIERARCHY ALL, MODELTIME 300.000000
```

```
INFORMATION ABOUT ESTIMATORS:

Autocorrelation values for STATE stream OCCUPATION at cpu due to ALL :

 1:  8.959583E-001        2:  8.231860E-001        3:  7.919517E-001
 4:  7.694346E-001        5:  7.483061E-001
 6:  7.307758E-001        7:  7.149036E-001        8:  6.990188E-001
 9:  6.897857E-001       10:  6.849659E-001


Autocorrelation values for STATE stream POPULATION at cpu due to ALL :

 1:  8.959583E-001        2:  8.231860E-001        3:  7.919517E-001
 4:  7.694346E-001        5:  7.483061E-001
 6:  7.307758E-001        7:  7.149036E-001        8:  6.990188E-001
 9:  6.897857E-001       10:  6.849659E-001

...

Autocorrelation values for STATE stream POPULATION at cpu due to cpu2_own :

 1:  2.409507E-001        2: -1.889751E-002        3: -1.415398E-002
 4: -1.953010E-002        5:  2.375447E-002
 6:  1.035470E-001        7: -1.226179E-002        8: -7.633794E-003
 9: -9.042022E-003       10:  3.196087E-002

Number  Line : Description of Errors or Warnings detected by SIMULATIVE
-------------+------------------------------------------------------------------
W.0485       : Names have been truncated in the trace file.

> SIMULATIVE :      Only       1 Warning .  Cpu Time used :      7.070 Seconds.


Stop  Date of Run       1996-03-25              Stop  Time of Run   13:21


Reached Model Time      2000.00000


> T O T A L  :      Only       1 Warning .  Cpu Time used :      7.230 Seconds.
```

I.4. Terminal Output

On the terminal the HI-SLANG compiler start messages for each pass are given. Outputs of the Simula compiler and linker follow (not listed here). When the analyzer starts, additional all WRITE statement outputs occur on the terminal, and if OUTPUT TABLE "SYSOUT" was used, intermediate results may occur here as well.

```
HIT   Our Own Licence     Uni Dortmund, Inf.IV     ends  96-11-30
HIT   Version 3.6.000     HI-SLANG Compiler          of  96-03-18  14:58

Please enter name of Compiler SOURCE or CONTROL file:
RefMan.hit

> FAN        :      Okay.                   Cpu Time used :      0.010 Seconds.

> PASS 1     :      Okay.                   Cpu Time used :      1.060 Seconds.

Number  Line : Description of Errors or Warnings
-------------+------------------------------------------------------------------
W.0526    42 : 'to_text' is a non-local access in 'system'.
          53a  in LINK=CONTROL
W.0526    50 : 'to_text' is a non-local access in 'system'.
          61a  in LINK=CONTROL
W.0526    77 : 'to_text' is a non-local access in 'system'.
          88a  in LINK=CONTROL
W.0526   108 : 'hour' is a non-local access in 'system'.
         119a  in LINK=CONTROL
W.0526   360 : 'hour' is a non-local access in 'hms'.
         194a  in LINK=CONTROL
W.0526   361 : 'hour' is a non-local access in 'hms'.
         195a  in LINK=CONTROL

> PASS 2     :      Only       6 Warnings.  Cpu Time used :      0.470 Seconds.

> PASS 3     :      Okay.                   Cpu Time used :      0.100 Seconds.

> SCG        :      Okay.                   Cpu Time used :      0.910 Seconds.

> T O T A L  :      Only       6 Warnings.  Cpu Time used :      2.770 Seconds.

                                Compile  Rate :  203.968 Lines/Sec.
```

```
HIT   Our Own Licence     Uni Dortmund, Inf.IV     ends  96-11-30
HIT   Version 3.6.000     HIT-Analyzer               of  96-03-18  14:58

Please enter name of Analyzer CONTROL file:
RefMan.hit

> FAN        :      Okay.                   Cpu Time used :      0.010 Seconds.

Please enter cpu time and #terminals
>>               100.000          5


Number  Line : Description of Errors or Warnings
-------------+------------------------------------------------------------------
W.0485       : Names have been truncated in the trace file.
 TIME      NO / OWNER   STATE OF BATCH
---------+------------+-----------------------
 0:00:00 |  1/INIT    | starting   after 0 sec.
 0:00:00 |  2/INIT    | starting   after 0 sec.
 0:00:00 |  3/INIT    | starting   after 0 sec.
 0:00:18 |  1/INIT    | terminated after 18 sec.
 0:00:18 |  4/INIT    | starting   after 18 sec.
 0:01:39 |  2/INIT    | terminated after 99 sec.
 0:01:39 |  5/INIT    | starting   after 99 sec.
 0:03:27 |  3/INIT    | terminated after 207 sec.
 0:03:27 |  6/INIT    | starting   after 207 sec.
 0:04:25 |  4/INIT    | terminated after 248 sec.
 0:04:25 |  7/LOAD    | starting   after 265 sec.
 0:04:44 |  5/INIT    | terminated after 185 sec.
 0:05:11 |  6/INIT    | terminated after 104 sec.
 0:06:06 |  7/LOAD    | terminated after 101 sec.
 0:06:41 |DIALOG2     | submitted
 0:06:44 |  8/DIALOG2 | starting   after 3 sec.
 0:06:48 |DIALOG4     | submitted
 0:06:48 |  9/DIALOG4 | starting   after 0 sec.
 0:06:52 |  8/DIALOG2 | terminated after 8 sec.
 0:08:03 |  9/DIALOG4 | terminated after 75 sec.
 0:10:00 |SYSTEM      | submitted
 0:10:00 | 10/SYSTEM  | starting   after 0 sec.
 0:10:20 | 10/SYSTEM  | terminated after 20 sec.
 0:12:24 |DIALOG2     | submitted
 0:12:25 | 11/DIALOG2 | starting   after 1 sec.
 0:12:42 | 11/DIALOG2 | terminated after 17 sec.
 0:15:00 |SYSTEM      | submitted
 0:15:00 | 12/SYSTEM  | starting   after 0 sec.
 0:16:32 | 12/SYSTEM  | terminated after 92 sec.
 0:18:58 |DIALOG1     | submitted
 0:18:59 | 13/DIALOG1 | starting   after 1 sec.
 0:19:25 | 13/DIALOG1 | terminated after 26 sec.
 0:20:00 |SYSTEM      | submitted
 0:20:00 | 14/SYSTEM  | starting   after 0 sec.
 0:20:40 | 14/SYSTEM  | terminated after 40 sec.
 0:24:00 |DIALOG2     | submitted
 0:24:01 | 15/DIALOG2 | starting   after 1 sec.
 0:27:48 | 15/DIALOG2 | terminated after 228 sec.
 0:30:00 |SYSTEM      | submitted
 0:30:00 | 16/SYSTEM  | starting   after 0 sec.
 0:31:30 | 16/SYSTEM  | terminated after 90 sec.
```

{now the observer becomes active, and writes those tables which were bound to SYSOUT}

```
HIT   Version 3.6.000    TABLE                                    1996-03-25
13:20    PAGE     1    University of Dortmund

Control                File RefMan.hit

Experiment Name        analysis

Model Type             system
Model Name             edv

Model Parameters       Name                Type                Actual Value
                       ========================================================
                       number_of_te        INTEGER                        5
                       batch_parall        INTEGER                        3
                       seed                INTEGER                       13

Used Method            SIMULATIVE

Date of Compile        1996-03-25              Time of Compile     13:18
Start Date of Run      1996-03-25              Start Time of Run   13:20



Stop  Date of Run                              Stop  Time of Run
Used CPU Time          6.78000

Reached Model Time     2000.00000
```

```
HIT   Version 3.6.000    TABLE                                        1996-03-25
13:20    PAGE     2    University of Dortmund

Evaluationobjectname : cent
--------------------------


Hierarchy   |Esti |        POPULATION         |      TURNAROUNDTIME       |
============+=====+===========================+===========================+
ALL         |Mean |      4.740834          680|     28.351454          346|
            |     |     0.000000     22.000000|     0.020875    246.215286|
            +-----+---------------------------+---------------------------+
            |Stdev|      6.347350             |     56.485613             |
            +-----+---------------------------+---------------------------+
cent_bat    |Mean |      0.654784           28|    101.523782           14|
            |     |     0.000000      3.000000|     6.992692    246.215286|
            +-----+---------------------------+---------------------------+
            |Stdev|      0.912515             |     79.348869             |
            +-----+---------------------------+---------------------------+
cent_dia    |Mean |      1.582061          395|     15.973440          199|
            |     |     0.000000      5.000000|     0.023110    232.187229|
            +-----+---------------------------+---------------------------+
            |Stdev|      1.920860             |     45.714139             |
            +-----+---------------------------+---------------------------+

Current model time   : 2.000000E+003
Cpu time used [sec.] : 6.970000E+000

Please enter one of:
q : quit simulation
s : stop observing, continue simulation
c : keep current model time interval and continue observing
n : as c, but switch to non-interactive mode
<real value n.nnEnn> : set new interval, continue observing
>>
YOU didn't want to continue!

...


> SIMULATIVE :      Only       1 Warning .  Cpu Time used :      7.070 Seconds.

Another experiment ? (y,n)
>>
> T O T A L  :      Only       1 Warning .  Cpu Time used :      7.230 Seconds.
```

I.5. Table Output

For every evaluation performed header information of about a half page size and the corresponding tables are given.

In contrast to the format presented here the tables generated use '=', '-', '+' and '|'-characters to draw the table borders. For the example the following tables are generated:

Note that the results may differ on your installation even vastly, since the model does not reach a steady state.

```
HIT   Version 3.6.000    TABLE                                        1996-03-25
13:20    PAGE     1    University of Dortmund

Control                 File RefMan.hit

Experiment Name         analysis

Model Type              system
Model Name              edv

Model Parameters        Name                 Type                 Actual Value
                        ==========================================================
                        number_of_te         INTEGER                         5
                        batch_parall         INTEGER                         3
                        seed                 INTEGER                        13

Used Method             SIMULATIVE

Date of Compile         1996-03-25              Time of Compile     13:18
Start Date of Run       1996-03-25              Start Time of Run   13:20




Stop  Date of Run                               Stop  Time of Run
Used CPU Time           6.79000

Reached Model Time      2000.00000
```

```
HIT   Version 3.6.000    TABLE                                        1996-03-25
13:20    PAGE     2    University of Dortmund

Evaluationobjectname : cent
--------------------------


Hierarchy   |Esti |        POPULATION         |      TURNAROUNDTIME       |
============+=====+===========================+===========================+
ALL         |Mean |      4.740834          680|     28.351454          346|
            |     |     0.000000     22.000000|     0.020875    246.215286|
            +-----+---------------------------+---------------------------+
            |Stdev|      6.347350              |     56.485613             |
            +-----+---------------------------+---------------------------+
cent_bat    |Mean |      0.654784           28|    101.523782           14|
            |     |     0.000000      3.000000|     6.992692    246.215286|
            +-----+---------------------------+---------------------------+
            |Stdev|      0.912515              |     79.348869             |
            +-----+---------------------------+---------------------------+
cent_dia    |Mean |      1.582061          395|     15.973440          199|
            |     |     0.000000      5.000000|     0.023110    232.187229|
            +-----+---------------------------+---------------------------+
            |Stdev|      1.920860              |     45.714139             |
            +-----+---------------------------+---------------------------+



Evaluationobjectname : cpu
--------------------------


Hierarchy   |Esti |        OCCUPATION         |        POPULATION         |
============+=====+===========================+===========================+
ALL         |Mean |      0.417139         6948|      0.417139         6948|
            |     |     0.000000      1.000000|     0.000000      1.000000|
            +-----+---------------------------+---------------------------+
            |Stdev|      0.493086              |      0.493086             |
            +-----+---------------------------+---------------------------+
            |Con  |      0.417139 +-    29.74%|      0.417139 +-    29.74%|
            | 90% |           10            10|           10            10|
            +-----+---------------------------+---------------------------+
cpu1_own    |Mean |      0.017912         1223|      0.017912         1223|
            |     |     0.000000      1.000000|     0.000000      1.000000|
            +-----+---------------------------+---------------------------+
            |Stdev|      0.132633              |      0.132633             |
            +-----+---------------------------+---------------------------+
            |Con  |      0.017912 +-    35.89%|      0.017912 +-    35.89%|
            | 90% |           10             6|           10             6|
            +-----+---------------------------+---------------------------+
cpu2_own    |Mean |      0.007283         1223|      0.007283         1223|
            |     |     0.000000      1.000000|     0.000000      1.000000|
            +-----+---------------------------+---------------------------+
            |Stdev|      0.085028              |      0.085028             |
            +-----+---------------------------+---------------------------+
            |Con  |      0.007283 +-    33.36%|      0.007283 +-    33.36%|
            | 90% |           10             6|           10             6|
            +-----+---------------------------+---------------------------+
```

```
HIT   Version 3.6.000    TABLE                                        1996-03-25
13:20    PAGE     3    University of Dortmund

Evaluationobjectname : link
--------------------------
```

Hierarchy	Esti	OCCUPATION		UTILIZATION	
link_bat	Mean	0.010128	29	0.016000	50
		0.000000	1.000000	0.000000	3.000000
	Stdev	0.100128		0.165987	
link_dia	Mean	0.351626	448	0.700764	1393
		0.000000	1.000000	0.000000	4.000000
	Stdev	0.477478		1.044163	
link_merge	Mean	0.357259	460	0.716764	1442
		0.000000	1.000000	0.000000	4.000000
	Stdev	0.479192		1.056000	

```
Evaluationobjectname : sem
--------------------------
```

Hierarchy	Esti	THROUGHPUT	
sema_bat	Mean	0.016500	32
		0.000000	227.737979
	Stdev	61.185248	
sema_bat_p	Mean	0.008500	16
		0.000000	359.460463
	Stdev	101.680547	

```
HIT   Version 3.6.000    TABLE                                        1996-03-25
13:20    PAGE     4    University of Dortmund

Evaluationobjectname : syst
--------------------------
```

Hierarchy	Esti	run_time	wait_time
ALL	Mean	97.399720 16	37.207069 16
		8.326025 247.548619	0.000000 265.470695
	Stdev	77.045182	79.606540
	Freq	0	11
		0.000000 1.000000	0.000000 1.000000
	Freq	0	0
		1.000000 2.000000	1.000000 2.000000
	Freq	0	1
		2.000000 5.000000	2.000000 5.000000
	Freq	1	0
		5.000000 10.000000	5.000000 10.000000
	Freq	5	1
		10.000000 50.000000	10.000000 50.000000
	Freq	10	3
		50.000000 1000.000000	50.000000 1000.000000

...

I.6. Dump File Output

The header information of a dumpfile is similar to that of a table, but commented out and split into header and footer by the dump file records for each evaluation performed. Since the dump output has a record length of 132 it is displayed in a compressed format here.

```
% Control               File RefMan.hit
%
% Experiment Name       analysis
%
% Model Type            system
% Model Name            edv
% Model Parameters      Name                Type                Actual Value
%                       number_of_te        INTEGER                        5
%                       batch_parall        INTEGER                        3
%                       seed                INTEGER                       13
%
% Used Method           SIMULATIVE
%
% Date of Compile       1996-03-25              Time of Compile     13:18
% Start Date of Run     1996-03-25              Start Time of Run   13:20
% Header is             DUMP
%
%
% Ev. Object  Measure      Hierarchy    Estimator      Result              #/Degree
%
syst         wait_time     ALL          FREQUENCY      6
                                              0.000000E+000  1.000000E+000       11
                                              1.000000E+000  2.000000E+000        0
                                              2.000000E+000  5.000000E+000        1
                                              5.000000E+000  1.000000E+001        0
                                              1.000000E+001  5.000000E+001        1
                                              5.000000E+001  1.000000E+003        3
syst         wait_time     ALL          MEAN               3.720707E+001
syst         wait_time                  STANDARDDEVIATION  7.960654E+001
syst         run_time      ALL          FREQUENCY      6
                                              0.000000E+000  1.000000E+000        0
                                              1.000000E+000  2.000000E+000        0
                                              2.000000E+000  5.000000E+000        0
                                              5.000000E+000  1.000000E+001        1
                                              1.000000E+001  5.000000E+001        5
                                              5.000000E+001  1.000000E+003       10
syst         run_time      ALL          MEAN               9.739972E+001
syst         run_time      ALL          STANDARDDEVIATION  7.704518E+001
link         OCCUPATION    link_bat     MEAN               1.012821E-002
link         OCCUPATION    link_bat     STANDARDDEVIATION  1.001280E-001
link         UTILIZATION   link_bat     MEAN               1.600000E-002
link         UTILIZATION   link_bat     STANDARDDEVIATION  1.659870E-001
link         OCCUPATION    link_merge   MEAN               3.572588E-001
link         OCCUPATION    link_merge   STANDARDDEVIATION  4.791920E-001
link         UTILIZATION   link_merge   MEAN               7.167639E-001
link         UTILIZATION   link_merge   STANDARDDEVIATION  1.056000E+000
link         OCCUPATION    link_dia     MEAN               3.516258E-001
link         OCCUPATION    link_dia     STANDARDDEVIATION  4.774778E-001
link         UTILIZATION   link_dia     MEAN               7.007639E-001
link         UTILIZATION   link_dia     STANDARDDEVIATION  1.044163E+000
%
%
% Stop  Date of Run                          Stop  Time of Run
%
% Used CPU Time         6.86000
% Reached Model Time    2000.00000

...
```

I.7. Trace Output

The simulative trace normally becomes very long and therefore it is shortened here, indicated by "...".

```
% analysis / system / edv / 1996-03-25 / 13:18 / 1996-03-25 / 13:20
% number_of_te INTEGER                      5
% batch_parall INTEGER                      3
% seed         INTEGER                     13
initial batch load:          6
0.000000E+000 ASK           batch       p              3 system         sem
0.000000E+000 >ENTRY        batch       p              3 system         sem
0.000000E+000 ENTRY>SERVICE batch       p              3 sem
0.000000E+000 SERVICE>EXIT  batch       p              3 sem
0.000000E+000 ASK           batch       print          3 sem            console
0.000000E+000 >ENTRY        batch       print          3 sem            console
0.000000E+000 ASK           batch       p              4 system         sem
0.000000E+000 >ENTRY        batch       p              4 system         sem
0.000000E+000 ENTRY>SERVICE batch       p              4 sem
0.000000E+000 SERVICE>EXIT  batch       p              4 sem
0.000000E+000 ASK           batch       print          4 sem            console
0.000000E+000 >ENTRY        batch       print          4 sem            console
0.000000E+000 ASK           batch       p              5 system         sem
0.000000E+000 >ENTRY        batch       p              5 system         sem
0.000000E+000 ENTRY>SERVICE batch       p              5 sem
0.000000E+000 SERVICE>EXIT  batch       p              5 sem
0.000000E+000 ASK           batch       print          5 sem            console
0.000000E+000 >ENTRY        batch       print          5 sem            console
0.000000E+000 ASK           batch       p              6 system         sem
0.000000E+000 >ENTRY        batch       p              6 system         sem
0.000000E+000 ASK           batch       p              7 system         sem
0.000000E+000 >ENTRY        batch       p              7 system         sem
0.000000E+000 ASK           batch       p              8 system         sem
0.000000E+000 >ENTRY        batch       p              8 system         sem
0.000000E+000 ASK           batch       p             10 system         sem
0.000000E+000 >ENTRY        batch       p             10 system         sem
1.792208E+001 ASK           batch       v              3 console        sem
1.792208E+001 >ENTRY        batch       v              3 console        sem
1.792208E+001 ENTRY>SERVICE batch       v              3 sem
1.792208E+001 SERVICE>EXIT  batch       v              3 sem
1.792208E+001 ENTRY>SERVICE batch       p              6 sem
1.792208E+001 SERVICE>EXIT  batch       p              6 sem
1.792208E+001 ASK           batch       print          6 sem            console
1.792208E+001 >ENTRY        batch       print          6 sem            console
9.888407E+001 ASK           batch       v              4 console        sem
9.888407E+001 >ENTRY        batch       v              4 console        sem
9.888407E+001 ENTRY>SERVICE batch       v              4 sem
9.888407E+001 SERVICE>EXIT  batch       v              4 sem
9.888407E+001 ENTRY>SERVICE batch       p              7 sem
9.888407E+001 SERVICE>EXIT  batch       p              7 sem
9.888407E+001 ASK           batch       print          7 sem            console
9.888407E+001 >ENTRY        batch       print          7 sem            console
2.072674E+002 ASK           batch       v              5 console        sem
2.072674E+002 >ENTRY        batch       v              5 console        sem
2.072674E+002 ENTRY>SERVICE batch       v              5 sem
2.072674E+002 SERVICE>EXIT  batch       v              5 sem
2.072674E+002 ENTRY>SERVICE batch       p              8 sem
2.072674E+002 SERVICE>EXIT  batch       p              8 sem
2.072674E+002 ASK           batch       print          8 sem            console
2.072674E+002 >ENTRY        batch       print          8 sem            console
2.654707E+002 ASK           batch       v              6 console        sem
2.654707E+002 >ENTRY        batch       v              6 console        sem
2.654707E+002 ENTRY>SERVICE batch       v              6 sem
2.654707E+002 SERVICE>EXIT  batch       v              6 sem
2.654707E+002 ENTRY>SERVICE batch       p             10 sem
2.654707E+002 SERVICE>EXIT  batch       p             10 sem
2.654707E+002 ASK           batch       print         10 sem            console
2.654707E+002 >ENTRY        batch       print         10 sem            console
```

```
.   .   .

7.619930E+002 SERVICE>EXIT  batch       v             64 sem
9.000000E+002 ASK           batch       p             79 system         sem
9.000000E+002 >ENTRY        batch       p             79 system         sem
9.000000E+002 ENTRY>SERVICE batch       p             79 sem
9.000000E+002 SERVICE>EXIT  batch       p             79 sem
9.000000E+002 ASK           batch       print         79 sem            console
9.000000E+002 >ENTRY        batch       print         79 sem            console
9.918273E+002 ASK           batch       v             79 console        sem
9.918273E+002 >ENTRY        batch       v             79 console        sem
9.918273E+002 ENTRY>SERVICE batch       v             79 sem
9.918273E+002 SERVICE>EXIT  batch       v             79 sem
1.139011E+003 ASK           batch       p            102 system         sem
1.139011E+003 >ENTRY        batch       p            102 system         sem
1.139011E+003 ENTRY>SERVICE batch       p            102 sem
1.139011E+003 SERVICE>EXIT  batch       p            102 sem
1.139011E+003 ASK           batch       print        102 sem            console
1.139011E+003 >ENTRY        batch       print        102 sem            console
1.165310E+003 ASK           batch       v            102 console        sem
1.165310E+003 >ENTRY        batch       v            102 console        sem
1.165310E+003 ENTRY>SERVICE batch       v            102 sem
1.165310E+003 SERVICE>EXIT  batch       v            102 sem
1.200000E+003 ASK           batch       p            104 system         sem
1.200000E+003 >ENTRY        batch       p            104 system         sem
1.200000E+003 ENTRY>SERVICE batch       p            104 sem
1.200000E+003 SERVICE>EXIT  batch       p            104 sem
1.200000E+003 ASK           batch       print        104 sem            console
1.200000E+003 >ENTRY        batch       print        104 sem            console
1.240414E+003 ASK           batch       v            104 console        sem
1.240414E+003 >ENTRY        batch       v            104 console        sem
1.240414E+003 ENTRY>SERVICE batch       v            104 sem
1.240414E+003 SERVICE>EXIT  batch       v            104 sem
1.440540E+003 ASK           batch       p            124 system         sem
1.440540E+003 >ENTRY        batch       p            124 system         sem
1.440540E+003 ENTRY>SERVICE batch       p            124 sem
1.440540E+003 SERVICE>EXIT  batch       p            124 sem
1.440540E+003 ASK           batch       print        124 sem            console
1.440540E+003 >ENTRY        batch       print        124 sem            console
1.668278E+003 ASK           batch       v            124 console        sem
1.668278E+003 >ENTRY        batch       v            124 console        sem
1.668278E+003 ENTRY>SERVICE batch       v            124 sem
1.668278E+003 SERVICE>EXIT  batch       v            124 sem
1.800000E+003 ASK           batch       p            145 system         sem
1.800000E+003 >ENTRY        batch       p            145 system         sem
1.800000E+003 ENTRY>SERVICE batch       p            145 sem
1.800000E+003 SERVICE>EXIT  batch       p            145 sem
1.800000E+003 ASK           batch       print        145 sem            console
1.800000E+003 >ENTRY        batch       print        145 sem            console
1.890130E+003 ASK           batch       v            145 console        sem
1.890130E+003 >ENTRY        batch       v            145 console        sem
1.890130E+003 ENTRY>SERVICE batch       v            145 sem
1.890130E+003 SERVICE>EXIT  batch       v            145 sem
%         6.84000 /      2000.00000 / 1996-03-25 / 13:21
```

I.8. Histogram Output

As an example of the simple builtin graphical facilities the histogram generated for the example will be displayed:

```
HIT   Version 3.6.000   WELCOME TO HIT                              1996-03-25
13:21    PAGE     1    University of Dortmund

Histogram
=========

Evaluation Object       syst
Measure                 wait_time
Hierarchy               all
Estimator               FREQUENCY

HIT   Version 3.6.000   WELCOME TO HIT                              1996-03-25
13:21    PAGE     2    University of Dortmund

ORDINATE
           ---
            |
 0.000000E+000 |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
               |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 1.100E+001 %%
      1.000000 |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
               |                                                                                             /
      1.000000 |____________________________________________________________________________________________/
               |  0.000E+000
      2.000000 |
               |
      2.000000 |%%%%%%%%%%%
               |%%%%%%%%%%%  1.000E+000
      5.000000 |%%%%%%%%%%%
               |          /
      5.000000 |_________/
               |  0.000E+000
     10.000000 |
               |
     10.000000 |%%%%%%%%%%%
               |%%%%%%%%%%%  1.000E+000
     50.000000 |%%%%%%%%%%%
               |          /
     50.000000 |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
               |%%%%%%%%%%%%%%%%%% 3.000E+000 %%
   1000.000000 |%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
               |                              /
               |_____________________________/
               |
              ---

            ABSCISSA
```

J. Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X