Universiteit Gent
Faculteit van de Toegepaste Wetenschappen

HGPSS:
Hierarchical General Purpose Simulation System

User Manual

Filip Claeys
Supervisor: Hans Vangheluwe

1992

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The discrete-event simulation language GPSS (or General Purpose Simulation System) was originally developed by Gordon at IBM in 1962. The language has been used extensively and different commercial implementations were developed ever since its conception. Despite its expressive power and flexibility, the language suffers from some weaknesses. Three major deficiencies can be distinguished:

- Lack of conceptual support for hierarchical modeling
- Poor interfacing facilities with other software components
- No possibility of interfering with the way the GPSS processor works

By means of the HGPSS project, an effort has been made to resolve these three problems. The resulting HGPSS language can be regarded as a superset of the GPSS/360 dialect, introduced in 1967. In the following sections, no attempt will be made to clarify the basics of the GPSS/360 language. In depth information can be found in [Schriber 1974], [Neelamkavil 1987] and [Gordon 1978].

All three major GPSS problems have been resolved by the implementation of a two-level system.

- The bottom level consists of a host language object-oriented programming environment. A discrete-event simulation kernel has been developed in this host language. The kernel consists of a number of interrelated functions and classes for the support of GPSS-like process-interaction simulation. Calls to this kernel are implemented by a multitude of interface functions. This interface language has been denominated HGPSS++. It should be noted that performance issues were not taken into account in this stage of the development of the HGPSS++ kernel.
- The top level is made up of a compiler that converts HGPSS program statements into a set of corresponding HGPSS++ kernel calls.

C++ has been chosen as the host language for the first level of the system. The C++ language is especially well suited for discrete-event simulation. The object-oriented philosophy was conceived with the development of the discrete-event simulation language SIMULA, the early forerunner of C++. An elaborate description of the C++ language can be found in [Stroustrup 1987].

The HGPSS to HGPSS++ compiler has been implemented with the use of the lexical analyser generator LEX and the parser generator YACC [Mason & Brown 1990].

The bottom level simulation kernel and the HGPSS++ language will not be discussed any further. Additional information can be found in [Claeys 1992]. The top level however will be discussed in detail.

One might wonder why GPSS, despite its disadvantages, is still interesting as a basis for a new discrete-event simulation system. The set of additional features offered by GPSS in respect to other simulation languages may serve as an answer to this question. The most important advantages of GPSS can be categorised as follows:

- The process-interaction world view.
- The fact that the system is very well-know and widely used.

- The high-level character of the language constructs.

In fact, GPSS is a full programming language [Boehm & Jacopini 1966], because it offers

- Basic actions,
- Action sequences,
- Conditional structures,
- Iterative structures.

Chapter 2

A solution to the GPSS weaknesses

The first goal of the HGPSS project, namely to provide some conceptual support for hierarchical modeling, has been accomplished by the introduction of the submodel concept, reflected in the introduction and implementation of four additional basic simulation blocks:

- ENTERMODEL
- LEAVEMODEL
- INPUT
- OUTPUT

A submodel is considered as an entity and has to be declared by the newly implemented `SUBMODEL` declaration statement. The idea is to break up the model of the system to be simulated into a number of submodels. Each submodel can in turn be subdivided into a number of submodels. By applying this procedure recursively, a tree-like model hierarchy will evolve. In the end, the submodels represented by the leaves of the tree should represent the basic processes within the system. Furthermore, such a *hierarchical decomposition* approach supports *stepwise refinement*, in which atomic blocks, in particular tree leaves, can be arbitrarily decomposed to reflect more detailed, microscopic knowledge about the system.

The second goal, the creation of a method of interfacing with other software components, has been implemented in a number of ways. Constructs of the C++ host language can be interlaced with HGPSS statements in many ways. Two additional basic simulation blocks also provide interfacing support. These blocks are:

- INTERN
- EXTERN

By letting the modeler interfere with the way the HGPSS processor works, the final goal of the project is accomplished. The HGPSS processor is designed as an open system that can be expanded and modified in many ways. See [Claeys 1992] for further details on this subject.

Chapter 3

The HGPSS language

3.1 General form of a HGPSS program

A HPGSS program, unlike a GPSS program, consists of a number of separate sections. There are two kinds of sections: *model sections* and *command sections*. Each program can only contain one command section. There is no limitation to the number of model sections. The presence of the command section is not mandatory. The number of model sections can be zero as well. The model sections can be regarded as the *model frame* of the system, whereas the command section represents the *experimental frame* [Zeigler 1976]. Each of the model sections represents one of the submodels the system is made out of. The system is modeled entirely by the collection of models and their interrelations. The command section depicts the environment the system is in. In general, the command section will specify what the inputs to the system are and what kind of output the modeler is interested in.

The following simple example will illustrate the concept of model and command section. A single-server system, more specifically a self-service shop with one cash desk, will be considered. Customers enter the shop, take the goods they want, then join the queue of customers waiting to check out and finally leave the queue and pay the bill. Figure 3.1 shows the HPGSS program for simulation of this system. The line numbers are not part of the language and are simply used for referencing. The HGPSS program consists of one model section (lines (1)-(10)) and one command section (lines (12)-(16)).

A model section always has to be encapsulated between a `MODEL` and an `ENDMODEL` statement. The `MODEL` keyword has to be followed by the name of the model. The command section begins with `COMMAND` and ends with `ENDCOMMAND`. Each HGPSS statement takes exactly one line. There are four kinds of statements:

- Block declaration statements
- Entity declaration statements

```
 (1) MODEL Shop
 (2)         GENERATE            10,7              Enter the shop
 (3)         ADVANCE             20,8              Take goods
 (4)         QUEUE               Wait              Join queue
 (5)         SEIZE               CashDesk          Arrive at cash desk
 (6)         DEPART              Wait              Leave the queue
 (7)         ADVANCE             8,2               Pay the bill
 (8)         RELEASE             CashDesk          Leave the cash desk
 (9)         TERMINATE           1                 Leave the shop
(10) ENDMODEL
(11)
(12) COMMAND
(13)         SIMULATE            Shop              Install model
(14)         START               1000              Simulate 1000 customers
(15)         PRINT               /                 Print all
(16)         END                                   Remove model
(17) ENDCOMMAND
```

Figure 3.1: A self-service shopping system

- Command statements
- Section marker statements

The last category includes only four statements:

- MODEL
- ENDMODEL
- COMMAND
- ENDCOMMAND

The command section consists entirely of command statements. The model sections are constructed by mixing block declaration statements with entity declaration statements. A complete summary of all HGPSS statements except the last category can be found in Tables 7.4, 7.5, 7.6, 7.7 and 7.8. Each statement can be prepended with white space and does not have to start in the first position of the line as in GPSS. Contrary to GPSS, HGPSS is no longer FORTRAN-based. Block and entity declarations can optionally be labeled. A label must precede the statement keyword and must be separated from the rest of the statement by white space. An exact definition of white space can be found in the extended Backus Naur Form description of the HGPSS language. Each HGPSS statement consists of a keyword followed by a list of parameters. The parameters must be separated from the keyword by white space. The parameters are separated from one another by a comma. There is no white space allowed between parameters and commas. The number and type of the parameters depends upon the statement. The parameter list can optionally be followed by a comment, separated from this parameter list by white space. Consider for instance line (2). The line contains a block declaration statement. The keyword is `GENERATE` and the parameters are respectively `10` and `7`. `''Enter the shop''` is a comment.

3.2 Labels

In GPSS, labels where restricted to capitals and a maximum of 5 characters. In HGPSS, labels can be up to 50 characters long. They may contain any letter or digit, and also underscores and dollar-signs. Labels with a digit as a leading character are not allowed. One must take care not to use labels that clash with HGPSS reserved words. Since HGPSS reserved words always consist of nothing but capitals, the clashing problem will certainly not arise when only lower case labels are used. Lower case labels will also improve program readability.

3.3 Comments

In Figure 3.1, one way of writing comments is illustrated. In fact there is another way to write comments in addition to putting them behind the statement parameters. Every source line that has a `*` as a first non-white space character, is considered as a line of comments.

3.4 Embedded C++ code

One of the major advantages of the HGPSS language is the possibility of writing C++ code between the HGPSS statements. The C++ code in a HGPSS program is simply copied into the file containing the generated HGPSS++ kernel calls, to be processed by the C++ compiler.

Figure 3.2 is an extension to the shopping system and illustrates the ways of putting C++ code into a HGPSS program. Line (1) shows the first way to put C++ code into a program. This method is very much like prepending a line of comments with a `*`. In the case of C++ code, the line has to be prepended with a `-`. Anything following the `-` until the end of the line will be passed literally to the HGPSS++ translator. Lines (5)-(9) show another way of putting C++ code into a program. A block of C++ code has been encapsulated within a `%{-%}` pair. The `%{` has to be the first non-white space character sequence on the line. The `%}` sequence on the other hand has to be the very last non-blank token on the line. It is obvious that the combination of `%{` and `%}` will be the best way to declare a large block of C++ code, while the `-` will be better in case there is just one line to be declared as such.

```
 (1) -#include <iostream.h>
 (2)
 (3) MODEL Shop
 (4)
 (5) %{   /* The following line of C++ code will output a message
 (6)         to the standard output to indicate that the model
 (7)         section is being processed simulation starts */
 (8)
 (9)         cout << "Model section being processed" << "\n";   }%
(10)
(11)         GENERATE            10,7               Enter the shop
(12)         ADVANCE             20,8               Take goods
(13)         QUEUE               Wait               Join queue
(14)         SEIZE               CashDesk           Arrive at cash desk
(15)         DEPART              Wait               Leave the queue
(16)         ADVANCE             8,2                Pay the bill
(17)         RELEASE             CashDesk           Leave the cash desk
(18)         TERMINATE           1                  Leave the shop
(19) ENDMODEL
(20)
(21) COMMAND
(22)         SIMULATE            Shop               Install model
(23)         START               1000               Simulate 1000 customers
(24)         PRINT               /                  Print all
(25)         END                                    Remove model
(26) ENDCOMMAND
```

Figure 3.2: The shopping system extended with C++ code

3.5 Referencing C++ variables and functions in HGPSS

Since C++ code is allowed in between HGPSS statements in a program there must also be a way of referencing C++ variables and functions in a pure HGPSS statement. This can be accomplished by double quoting calls to C++ variables and functions in HGPSS statements. This is illustrated in Figure 3.3. In line (11) there is a reference being made to the variable `spread`. In fact not only variables and function calls but also entire C++ expressions can be put between double quotes. Lines (4) and (5) could for instance be deleted from the previous example and line (11) replaced by `ADVANCE 8,"6+sqrt(2)"`.

3.6 Functions and variables

The GPSS `FUNCTION`, `VARIABLE`, `BVARIABLE` and `FVARIABLE` statements are supported by HGPSS, but these statements have been extended. The extension makes it possible to let the entities declared by the previous statements refer to C++ functions. These C++ functions must have `value_type` (in the case of the `FUNCTION`, `VARIABLE` and `FVARIABLE` statements) or `boolean_type` (in the case of the `BVARIABLE` statement) as the type of the returned values. These types are known to the HGPSS++ translator.

In Figure 3.4, the GPSS and HGPSS way of function and variable declaration are illustrated. In line (18) the C++ function `CppVar` is assigned to the HGPSS variable `HGPSSVar`. In line (30), the C++ function `CppFun` is assigned to the HGPSS function `HGPSSFun`. In both cases the name of the C++ function is double-quoted. The slash in front of the name of the function is mandatory to avoid confusion with the usual way of declaring variables and functions. Especially when functions are concerned, a lot can be gained by applying the HGPSS way. It makes the use of vast function lookup tables obsolete.

In Figure 3.4, another interesting feature is displayed. In lines (5) and (8), a random number generator other than the built-in HGPSS random number generators is used, i.e. the function `rand()`, which is part of the C++ standard library.

3.7 Referencing HGPSS entities in C++ code

Referencing C++ variables and functions in HGPSS statements is easy as illustrated before. One can also do the opposite: reference HGPSS entities in C++ code. This however requires the use of HGPSS++ kernel calls. Refer to [Claeys 1992] for more details.

```
 (1) -#include <math.h>
 (2)
 (3) MODEL Shop
 (4) -       value_type spread;
 (5) -       spread=6+sqrt(2);
 (6)         GENERATE            10,7               Enter the shop
 (7)         ADVANCE             20,8               Take goods
 (8)         QUEUE               Wait               Join queue
 (9)         SEIZE               CashDesk           Arrive at cash desk
(10)         DEPART              Wait               Leave the queue
(11)         ADVANCE             8,"spread"         Pay the bill
(12)         RELEASE             CashDesk           Leave the cash desk
(13)         TERMINATE           1                  Leave the shop
(14) ENDMODEL
(15)
(16) COMMAND
(17)         SIMULATE            Shop               Install model
(18)         START               1000               Simulate 1000 customers
(19)         PRINT               /                  Print all
(20)         END                                    Remove model
(21) ENDCOMMAND
```

Figure 3.3: The shopping system with reference to a C++ variable

```
 (1) -#include <stdlib.h>
 (2) -#include <math.h>
 (3)
 (4) -value_type CppVar(void)
 (5) -   { return(10+rand()*2); }
 (6)
 (7) -value_type CppFun(void)
 (8) -   { return(-log(1-rand())); }
 (9)
(10) MODEL VarFun
(11)
(12) **** GPSS VARIABLE ****
(13)
(14) GPSSVar  VARIABLE            10+RN1*2
(15)
(16) **** HGPSS VARIABLE ****
(17)
(18) HGPSSVar VARIABLE            /"CppVar"
(19)
(20) **** GPSS FUNCTION ****
(21)
(22) GPSSFun  FUNCTION            RN1,C24
(23)         0,0/.1,.104/.2,.222/.3,.355/.4,.509/.5,.69/.6,.915/.7,1.2-
(24)         .75,1.38/.8,1.6/.84,1.83/.88,2.12/.9,2.3/.92,2.52/.94,2.81-
(25)         .95,2.99/.96,3.2/.97,3.5/.98,3.9/.99,4.6/.995,5.3/.998,6.2-
(26)         .999,7/.9998,8
(27)
(28) **** HGPSS FUNCTION ****
(29)
(30) HGPSSFun FUNCTION            /"CppFun"
(31)
(32) ENDMODEL
```

Figure 3.4: The GPSS and HGPSS way of function and variable declaration

```
 (1) MODEL Model1
 (2) A       SUBMODEL            Model2
 (3) B       SUBMODEL            Model3
 (4)         ...
 (5)         ENTERMODEL          A,In
 (6)         LEAVEMODEL          A,Out
 (7)         ENTERMODEL          B,In
 (8)         LEAVEMODEL          B,Out
 (9)         ...
(10) ENDMODEL
(11)
(12) MODEL Model2
(13)         INPUT               In
(14)         ...
(15)         OUTPUT              Out
(16) ENDMODEL
(17)
(18) MODEL Model3
(19)         INPUT               In
(20)         ...
(21)         OUTPUT              Out
(22) ENDMODEL
(23)
(24) COMMAND
(25)         ...
(26)         SIMULATE            Model1
(27)         ...
(28) ENDCOMMAND
```

Figure 3.5: A multi-model system

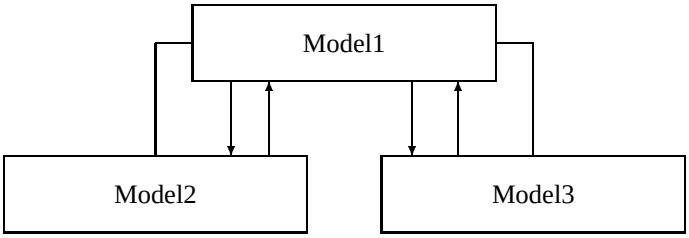

Figure 3.6: Model hierarchy of previous figure

3.8 Hierarchical modeling

Perhaps the most interesting extension to the GPSS language is the provision of some hierarchical modeling constructs. In all previous examples, only single-model programs where considered. A program can however contain several model sections, each describing a submodel of the complete model of the system. One of the model sections must describe the model at the root of the hierarchy tree. The name of this model must be notified to the system by means of the `SIMULATE` command. Each submodel used in another model must be declared by the `SUBMODEL` statement. A transaction currently in a certain model can be routed to a submodel relative to the first model by the use of the `ENTERMODEL` block. This block takes two parameters. The first parameter refers to the submodel the transaction is to be routed to. The second parameter refers to the location in the submodel where the transaction is to enter. These locations can be declared by the `INPUT` blocks. The `INPUT` block takes as a parameter the name of the input. The locations where a transaction can return to the model it is originating from, are marked by means of a `OUTPUT` block. This block takes as a parameter the name of the output. Lastly there is the `LEAVEMODEL` block. This block marks the location where the transaction is to return in the model it is originating from. A `LEAVEMODEL` block takes two parameters. The first one refers to the submodel and the second one to the output of the submodel.

Figure 3.5 will clarify the mechanism. The goal is to construct a program that describes a system consisting of two subsystems. The second subsystem is to be activated after the first subsystem has ended its task. The situation of Figure 3.5 can also be depicted by Figure 3.6. The lines in the figure without an arrow refer to the actual hierarchy, whereas the arrowed lines refer to the transaction flow. It is important to distinguish hierarchy from transaction flow

```
 (1) MODEL Model1
 (2) A       SUBMODEL            Model2
 (3) B       SUBMODEL            Model3
 (4) ENDMODEL
 (5)
 (6) MODEL Model2
 (7)         ...
 (8) ENDMODEL
 (9)
(10) MODEL Model3
(11)         ...
(12) ENDMODEL
(13)
(14) COMMAND
(15)         ...
(16)         SIMULATE            Model1
(17)         ...
(18) ENDCOMMAND
```

Figure 3.7: A multi-model system without transaction transfer

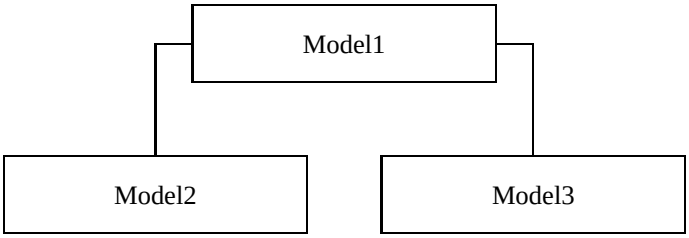

Figure 3.8: Model hierarchy of previous figure

because transactions need not necessarily flow from model to submodel and back.

In Figure 3.7, the situation is described where a model consists of two submodels without transfer of transactions. In fact this program models a system consisting of two subsystems that operate *concurrently* and without interaction. This situation can as in the previous example also be depicted by a diagram as in Figure 3.8.

All kinds of combinations of the previous two situations are possible. In principle there is no limitation to the depth of the hierarchy tree.

Each model has its own context. All entities used in a model are local. The only means of interchanging information between models are:

- via the transfer of transactions,
- via global C++ variables.

Entities can only be referenced from within the command section one model at a time, this because there is only one model active in the command section at a time. Initially the active model is the root model specified in the `SIMULATE` command. The active model can be changed to one of its submodels by using the `DOWN` command. This command allows descending the hierarchy tree and takes as a parameter the name of the submodel to go to. Only when the active model is the root model, identifier-labels can be used. In the other cases number-labels must be used. One can go up the hierarchy tree by the `UP` command. This command takes no parameters.

Figure 3.9 presents some examples of the use of the `UP` and `DOWN` commands. The command in line (22) will cause all storages of the root model to be printed, whereas the command in line (24) will produce some output on the storages of the root model's submodel named `ModelA`. In line (25), the use of a numeric label is mandatory.

When using identifier-labels as respective input name and output name parameter in the `ENTERMODEL` and `LEAVEMODEL` statements, it is necessary to declare the submodel by the `SUBMODEL` statement *before the first occurrence* of the `ENTERMODEL` or `LEAVEMODEL` statement. The compiler must know what kind of submodel is being referred to in these two statements.

All submodels used in a model need not be described in the same program file. It is possible to compile submodels separately and link them with other submodels and the command section. In Figure 3.10, an overview of the HGPSS

```
 (1) MODEL Model1
 (2) ModelA  SUBMODEL            Model2
 (3) ...
 (4) StorA   STORAGE             2
 (5) ...
 (6) ENDMODEL
 (7)
 (8) MODEL Model2
 (9) 1       SUBMODEL            Model3
(10) ...
(11) StorA   STORAGE             4
(12) ...
(13) ENDMODEL
(14)
(15) MODEL Model3
(16) ...
(17) ENDMODEL
(18)
(19) COMMAND
(20)         ...
(21)         SIMULATE            Model1              Model1 becomes active
(22)         PRINT               ,,STORAGE
(23)         DOWN                ModelA              Model2 becomes active
(24)         PRINT               ,,STORAGE
(25)         DOWN                1                   Model3 becomes active
(26)         ...
(27)         UP                                      Model2 becomes active
(28)         ...
(29)         UP                                      Model1 becomes active
(30)         ...
(31) ENDCOMMAND
```

Figure 3.9: The UP and DOWN commands

system structure is shown to clarify the possibilities.

3.9 Parameterised models

Models can be used as templates that will be adapted to a certain situation by the use of parameters. The way of assigning parameters to models works the same way as using function parameters in C++.

Figure 3.11 shows how a parameterised model can be constructed and used. The shopping system is considered again. The goal is not to simulate a thousand customers as in the previous cases but 24 hours. A clock model is therefore constructed that will signal the expiring of the time span. At line (14) the `MODEL` reserved word is followed by the name of the parameterised model which is on its turn followed by the formal parameter list. This starts with a `(` and ends with a `)`. Anything between these two characters will be regarded as the C++ description of the parameter list. At line (2) the `Clock` model is declared as a submodel relative to the `Shop` model. The name of the submodel is followed by the actual parameter list.

It is important to note that there is only one termination counter in the whole HGPSS system. Each `TERMINATE` block on whatever level of the model hierarchy will consequently refer to the same termination counter.

3.10 Interfacing and extending the language

A way of extending the HGPSS language and interfacing with other software components is by making use of the `INTERN` and `EXTERN` blocks.

The `INTERN` block takes as a parameter the name of a C++ function that will be triggered when a transaction enters the block.

The `EXTERN` block takes two parameters. The first parameter is the name of the C++ function that will be triggered when a transaction enters the block. The second parameter is the name of a C++ function that will be triggered on every active simulation time point. It is clear that the `EXTERN` block provides a way of interfacing with other programs or routines that are *asynchronous* to the discrete-event simulation taking place.

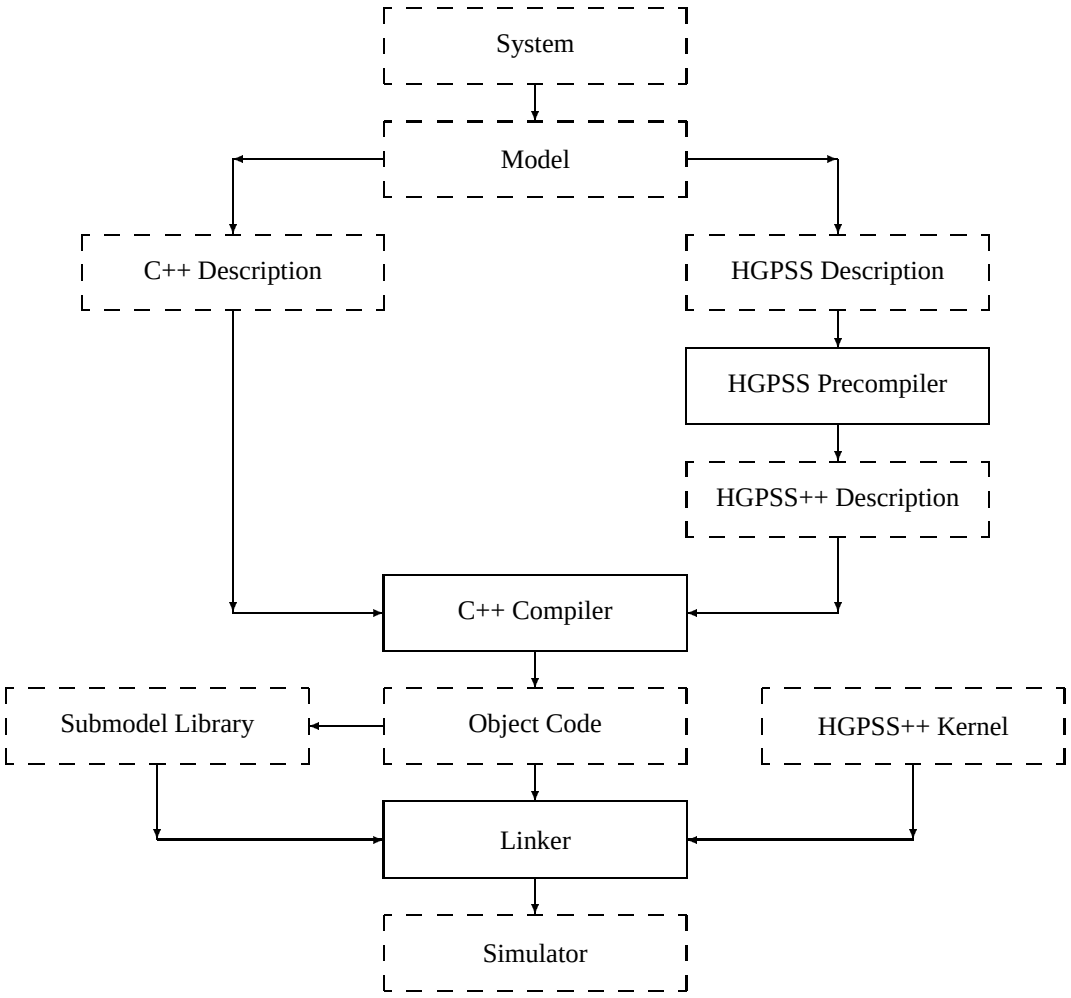

Figure 3.10: Process model of the HGPSS system

```
 (1) MODEL Shop
 (2) 1       SUBMODEL            Clock(24)           Initialize clock
 (3)
 (4)         GENERATE            10,7                Enter the shop
 (5)         ADVANCE             20,8                Take goods
 (6)         QUEUE               Wait                Join queue
 (7)         SEIZE               CashDesk            Arrive at cash desk
 (8)         DEPART              Wait                Leave the queue
 (9)         ADVANCE             8,2                 Pay the bill
(10)         RELEASE             CashDesk            Leave the cash desk
(11)         TERMINATE           /                   Leave the shop
(12) ENDMODEL
(13)
(14) MODEL Clock(long time)
(15)         GENERATE            "time*60"
(16)         TERMINATE           1
(17) ENDMODEL
(18)
(19) COMMAND
(20)         SIMULATE            Shop                Install model
(21)         START               1                   Simulate model
(22)         PRINT               /                   Print all
(23)         END                                     Remove model
(24) ENDCOMMAND
```

Figure 3.11: The shopping system with a clock

Figure 3.12 presents an example of the use of an `INTERN` block. Each time a transaction reaches the block, a message will be printed on the standard output.

In Figure 3.13, a message is printed at every active simulation time point. The `EXTERN` block is put outside the transaction flow because there is no relevant action to be performed if a transaction were to enter the block. Consequently the `NULL` refers to the fact that there is no function to be called upon arrival of a transaction.

Figure 3.14 illustrates the use of an `INTERN` block to perform low-level actions on the transactions entering the block. The low-level actions are carried out by HGPSS++ kernel calls.

3.11 Multiple analogous declarations

Suppose several entities of the same kind have to be declared. Instead of writing the desired number of declaration statements, a procedure as illustrated in Figure 3.15 can be used. Multiple copies of a storage with capacity 2 are created by putting the `STORAGE` command inside a loop. A C++ variable is used as a label. Note that the use of C++ expressions as labels is only allowed in the context of the declaration of entities and not blocks. This method does not work when the entities have to be identifier-labeled.

3.12 Output

A GPSS simulation automatically generates an output file containing all information gathered during the simulation process. The HGPSS system on the other hand does not output information unless explicitly asked for. The information relevant for a certain simulation can be selected. In fact, there are two ways of extracting information from the system. First of all there is the `PRINT` block. This block also exists in GPSS. The parameters of the HGPSS block however are somewhat different from those of the GPSS block. The first and second parameter are the same as in GPSS. They are also optional. The third parameter is as in GPSS an entity mnemonic. The mnemonics are not the same as in GPSS. Table 7.3 lists all HGPSS entity mnemonics. When this parameter is unspecified, information about *all* entities will be output. The optional fourth HGPSS parameter is a C++ pointer to a file. When this parameter is unspecified, information will be sent to the default output file. When the parameter is specified, information can be redirected to the file pointed to by the C++ file pointer.

Another way to generate output is by using the `PRINT` command. This command works entirely the same way as the block with the same name.

By default, the name of the output file is `hgpsspp.out`. The default file name can be changed by giving the new name as a second parameter to the `SIMULATE` command.

```
(1) -#include <iostream.h>
(2)
(3) -void Message(void)
(4) -   { cout << "A customer pays the bill\n"; }
(5)
(6) MODEL Shop
(7)         GENERATE            10,7                Enter the shop
(8)         ADVANCE             20,8                Take goods
(9)         QUEUE               Wait                Join queue
(10)        SEIZE               CashDesk            Arrive at cash desk
(11)        DEPART              Wait                Leave the queue
(12)        INTERN              "Message"           Print message
(13)        ADVANCE             8,2                 Pay the bill
(14)        RELEASE             CashDesk            Leave the cash desk
(15)        TERMINATE           1                   Leave the shop
(16) ENDMODEL
(17)
(18) COMMAND
(19)        SIMULATE            Shop                Install model
(20)        START               1000                Simulate 1000 customers
(21)        PRINT               /                   Print all
(22)        END                                     Remove model
(23) ENDCOMMAND
```

Figure 3.12: The shopping system using an INTERN block

```
(1) -#include <iostream.h>
(2)
(3) -void Message(void)
(4) -   { cout << "Active time point\n"; }
(5)
(6) MODEL Shop
(7)         EXTERN              "NULL","Message"
(8) ...
```

Figure 3.13: The shopping system using a EXTERN block

```
(1) %{
(2) // The following function copies the number of parameters of
(3) // the active transaction into its first parameter.
(4)
(5) void Copy(void)
(6)    {
(7)      TransactionClass *transaction;
(8)      transaction=Processor.Transaction();
(9)      transaction->SetParameter(1,transaction->NbrOfParameters());
(10)   }
(11) %}
(12)
(13) MODEL Model
(14) ...
(15)        INTERN              "Copy"
(16) ...
(17) ENDMODEL
```

Figure 3.14: Low-level actions using an INTERN block

```
 (1) MODEL Model
 (2)
 (3) *Writing the same statement 5 times in GPSS,
 (4)
 (5) 1        STORAGE             2
 (6) 2        STORAGE             2
 (7) 3        STORAGE             2
 (8) 4        STORAGE             2
 (9) 5        STORAGE             2
(10)
(11) *reduces to this in HGPSS
(12)
(13) -     for (int i=1; i<=5; i++)
(14) "i"      STORAGE             2
(15)
(16) ENDMODEL
```

Figure 3.15: Multiple analogous declarations

```
 (1) -#include <fstream.h>
 (2)
 (3) COMMAND
 (4)
 (5) -       ofstream file_pointer;
 (6) -       file_pointer=fopen("output.txt");
 (7)
 (8) *Output info on all facilities to default output file
 (9)
(10)         PRINT               ,,FACILITY
(11)
(12) *Output info on facility A to default output file
(13)
(14)         PRINT               A,,FACILITY
(15)
(16) *Output info on absolute clock to file pointed to by "file_pointer"
(17)
(18)         PRINT               ,,ABSOLUTECLOCK,"file_pointer"
(19)
(20) *Change name of default output file to "output.txt"
(21)
(22)         SIMULATE            ModelName,"output.txt"
(23)
(24) ENDCOMMAND
```

Figure 3.16: Some output related commands

Figure 3.16 presents some examples of output related commands.

Chapter 4

HGPSS versus GPSS

4.1 Introduction

In this section, an overview of the syntactic and semantic differences between GPSS/360 and HGPSS will be presented. Also, the new block declarations, entity declarations, and commands will be discussed.

4.2 Block declarations

4.2.1 GPSS block declarations

1. ADVANCE:
   - Syntax: When all parameters are omitted, a slash is needed to eliminate some parser ambiguities.
   - Semantics: No differences.
2. ASSEMBLE: No differences.
3. ASSIGN: No differences.
4. BUFFER: No differences.
5. DEPART: No differences.
6. ENTER: No differences.
7. GATE: No differences.
8. GATHER: No differences.
9. GENERATE:
   - Syntax: When all parameters are omitted, a slash is needed to eliminate some parser ambiguities.
   - Semantics:
     – The optional parameter which indicates the type of the parameters of the transactions generated, has no effect. Whether this parameter is supplied or not, the type of the parameters will always be `value_type`.
     – The number of parameters a transaction will be supplied with has to be specified explicitly. Defaulting the number of parameters is allowed, but results in the generation of a transaction with zero parameters.
10. LEAVE: No differences.
11. LINK: No differences.

12. LOGIC: No differences.
13. LOOP: No differences.
14. MARK:
    - Syntax: When all parameters are omitted, a slash is needed to eliminate some parser ambiguities.
    - Semantics: No differences.
15. MATCH: No differences.
16. MSAVEVALUE:
    - Syntax: No differences.
    - Semantics: The optional matrix type parameter has no effect. Whether this parameter is supplied or not, the type of the elements of the matrix will always be `value_type`.
17. PREEMPT:
    - Syntax: No differences.
    - Semantics: In GPSS there are three alternatives to the way a transaction that has been preempted from a facility, behaves:
      (a) The preempted transaction is put on an interrupt chain and will be restored to the facility when the facility is returned to it.
      (b) The preempted transaction is routed to some other block in the model, and is removed from contention for use of the facility. That is, it will not later be automatically reinstated by the processor as the capturer of the facility.
      (c) The preempted transaction is routed to some other block in the model, but is not removed from contention for later automatic reinstatement as the capturer of the facility.

      In HGPSS only the first alternative is implemented. The parameters of the `PREEMPT` statement that refer to the two other alternatives can however be supplied but will have no effect.
18. PRINT:
    - Syntax:
      – When all parameters are omitted, a slash is needed to eliminate some parser ambiguities.
      – The third parameter still refers to an entity mnemonic but the HGPSS entity mnemonics are different to those of GPSS.
      – There is an optional fourth parameter. This parameter refers to a C++ file pointer. The output generated by the block will be sent to the file pointed to by the pointer, and not to the default output file.
    - Semantics: No differences.
19. PRIORITY: No differences.
20. QUEUE: No differences.
21. RELEASE: No differences.
22. RETURN: No differences.
23. SAVEVALUE:
    - Syntax: No differences.
    - Semantics: The optional savevalue type parameter has no effect. Whether this parameter is supplied or not, the type of the savevalue will always be `value_type`.

24. SEIZE: No differences.
25. SELECT: No differences.
26. SPLIT: No differences.
27. TABULATE: No differences.
28. TERMINATE:
    - Syntax: When all parameters are omitted, a slash is needed to eliminate some parser ambiguities.
    - Semantics: No differences.
29. TEST: No differences.
30. TRANSFER:
    - Syntax: In the case of a statistical block and a Standard Numerical Attribute as the selection mode parameter, the selection mode parameter had to be prepended with a dot in GPSS. This dot was required to indicate the fact that the integer value defined by the selection mode parameter was considered as the fractional part of a real with zero as the integer part. In HGPSS however this parameter should by itself represent a real between zero and one. The dot is consequently not allowed in HGPSS.
    - Semantics: No differences.
31. UNLINK: No differences.

4.2.2 HGPSS block declarations

1. ENTERMODEL: When a transaction enters this block, it will be removed from the model it is currently in, and will be injected in the submodel specified by the first parameter. The second parameter is the number of the input of the submodel the transaction is injected into.
2. EXTERN: This block allows interfacing with other programs or routines that are asynchronous to the discrete-event simulation. The block takes two parameters. The first parameter is the name of a C++ function that will be called when a transaction enters the block. The second parameter is the name of a function that will be called on every active simulation time point. To indicate that there is no function to be called, a `NULL` may be entered as a parameter.
3. INPUT: This block indicates an input to a model. Via this input, transactions originating from a higher level in the model hierarchy can be injected in the model. The parameter specifies the name of the input.
4. INTERN: This block allows interfacing with other routines or programs. The block takes one parameter. This parameter is the name of a C++ function that will be called when a transaction enters the block.
5. LEAVEMODEL: This block indicates the location in the model where a transaction that has previously been injected in a submodel, can return into the first model. The first parameter specifies the submodel and the second parameter the number of the output of the submodel.
6. OUTPUT: This block indicates an output to the model. Via this output, transactions originating from a higher level in the model hierarchy can return to the model they originated from. The parameter specifies the number or name of the output.

4.3 Entity declarations

4.3.1 GPSS entity declarations

1. BVARIABLE:
   - Syntax: Besides the original GPSS form of the statement, there is another newly implemented form. The HGPSS boolean variable can be assigned to a C++ function. Each time the boolean variable is referenced, the C++ function will be called. The name of the C++ function has to be double-quoted and prepended with a slash.
   - Semantics: No differences.
2. FUNCTION:
   - Syntax:
     - In case the list of function points is spread over more than one line, this has to be indicated with a `-` at the end of all lines but the last.
     - Besides the original GPSS form of the statement, there is another newly implemented form. The HGPSS function can be assigned to a C++ function. Each time the function is referenced, the C++ function will be called. The name of the C++ function has to be double-quoted and prepended with a slash.
     - As opposed to GPSS, matrix Standard Numerical Attributes are allowed as function arguments.
   - Semantics: No differences.
3. MATRIX:
   - Syntax: No differences.
   - Semantics: The matrix type parameter has no effect. Whether this parameter is set to `H` or `F`, the type of the elements of the matrix will always be `value_type`.
4. QTABLE: No differences.
5. STORAGE:
   - Syntax: Only one of the two GPSS forms of storage declaration is allowed.
   - Semantics: No differences.
6. TABLE: No differences.
7. VARIABLE:
   - Syntax: Besides the original GPSS form of the statement, there is another newly implemented form. The HGPSS variable can be assigned to a C++ function. Each time the variable is referenced, the C++ function will be called. The name of the C++ function has to be double-quoted and prepended with a slash.
   - Semantics: No differences.
8. FVARIABLE:
   - Syntax: Besides the original GPSS form of the statement, there is another newly implemented form. The HGPSS fullword variable can be assigned to a C++ function. Each time the fullword variable is referenced, the C++ function will be called. The name of the C++ function has to be double-quoted and prepended with a slash.
   - Semantics: There is no difference between variables and fullword variables. Both will have to return a `value_type` as a result.

4.3.2 HGPSS entity declarations

1. SUBMODEL: This statement declares a model as a submodel relative to another model. The declaration takes as a parameter the type of the submodel. It is favourable always to put the `SUBMODEL` statement at the top of a model section. By doing this, the HGPSS system will always know the type of the submodels used.

4.4 Commands

4.4.1 GPSS commands

1. CLEAR:
   - Syntax: There are no parameters allowed to this statement, contrary to GPSS. As a result there is no way to retain the contents of savevalues after a `CLEAR` command is carried out. C++ variables can however always be used to retain information.
   - Semantics: No differences.
2. END: No differences.
3. INITIAL:
   - Syntax:
     - Multiple entries separated by a slash are not allowed.
     - The use of `-` to indicate subranges is not allowed.
   - Semantics: No differences.
4. JOB: No differences.
5. RESET: No differences.
6. SIMULATE:
   - Syntax: This command has the same name as the GPSS `SIMULATE` command but has nothing in common with this command. The command is used to tell the system by means of the first parameter which model is located at the root of the hierarchy tree. The second parameter is optional and allows the default output file name to be overridden.
   - Semantics: See syntax.
7. START:
   - Syntax: The printout suppression, initial value of snap interval counter and signal for chain printouts parameters are not supported.
   - Semantics: No differences.

4.4.2 HGPSS commands

1. DOWN: This command allows descending the hierarchy tree. The only parameter refers to the submodel that has to activated.
2. PRINT: This command works entirely the same way as the `PRINT` block.
3. UP: This command allows climbing the hierarchy tree. The command uses no parameters.

Chapter 5

How to create a simulator

In this section, all the necessary steps to create a simulator will be presented. First of all, the files the HGPSS system consists of are listed.

5.1 HGPSS system files

- HGPSS to HGPSS++ compiler:
  - *hgpss.l*: LEX description of the lexical analyser for use on UNIX systems.
  - *hgpss.lxi*: LEX description of the lexical analyser for use on MSDOS systems.
  - *hgpss.y*: YACC description of the parser. There are two versions of this file: one for use on UNIX systems and one for use on MSDOS systems.
  - *initial.y*: YACC description of auxiliary initial parameter parser.
  - *logical.y*: YACC description of auxiliary logical parameter parser.
  - *param.y*: YACC description of auxiliary parameter parser.
  - *tokens.h*: List of all tokens for use with *hgpss.l* of *hgpss.lxi*.
  - *hgpss.exe*: Executable HGPSS to HGPSS++ compiler for use on MSDOS systems.
  - *hgpss*: Executable HGPSS to HGPSS++ compiler for use on UNIX systems.
- HGPSS++ kernel:
  - Access files:
    - *hgpsspp.h*: Header file granting partial access to the HGPSS++ kernel. This file will be included in compiled HGPSS programs.
    - *hgpssppa.h*: Header file granting complete access to the HGPSS++ kernel. This file will be included upon request in compiled HGPSS programs.
  - System files:
    - *header.h*: Header file to be included in all HGPSS++ system files.
    - *funcsupp.C*: Support functions.
    - *funcuser.C*: User functions.
    - *classsup.C*: Support classes.
    - *classsys.C*: System classes.
    - *classent.C*: Entity classes.
    - *classenc.C*: Entity chain classes.
    - *classblo.C*: Block classes.
    - *classpro.C*: Processor classes.

5.2 The HGPSS to HGPSS++ compiler

Once a HGPSS program is written, it should be compiled to a HGPSS++ program with the HGPSS to HGPSS++ compiler. This program generates three files from the input file:

1. A so called *output file*, which is the actual HGPSS++ program.
2. A so called *header file*, containing a class declaration and a constant structure containing the resolved labels for all the models in the input file.
3. A so called *variable file*, containing C++ functions automatically generated from GPSS-like `VARIABLE`, `BVARIABLE` and `FVARIABLE` statements.

The two last files will be automatically included in the first one when the HGPSS++ program is compiled with a C++ compiler.

The HGPSS to HGPSS++ compiler needs some command line arguments. There are four possible forms these command line arguments can take:

1. -e *name*
   - The file hgpsspp.h will be included in the output file.
   - The input file *name*.gps will be processed.
   - The output file will be denominated *name*.C.
   - The header file will be denominated *name*.h.
   - The variable file will be denominated *name*.var.
2. -f [*name1* [*name2* [*name3* [*name4*]]]]
   - The file hgpsspp.h will be included in the output file.
   - The input file *name1* will be processed. The default input file name is hgpsspp.gps.
   - The output file will be denominated *name2*. The default output file is hgpsspp.C.
   - The header file will be denominated *name3*. The default header file is hgpsspp.h.
   - The variable file will be denominated *name4*. The default variable file is hgpsspp.var.
3. -E *name*
   - The file hgpssppa.h will be included in the output file.
   - The input file *name*.gps will be processed.
   - The output file will be denominated *name*.C.
   - The header file will be denominated *name*.h.
   - The variable file will be denominated *name*.var.
4. -F [*name1* [*name2* [*name3* [*name4*]]]]
   - The file hgpssppa.h will be included in the output file.
   - The input file *name1* will be processed. The default input file name is hgpsspp.gps.
   - The output file will be denominated *name2*. The default output file is hgpsspp.C.
   - The header file will be denominated *name3*. The default header file is hgpsspp.h.
   - The variable file will be denominated *name4*. The default variable file is hgpsspp.var.

5.3 Compiling and linking the simulator

After the HGPSS++ output, header and variable file have been generated out of the HGPSS input file by the HGPSS to HGPSS++ compiler, the output file should be compiled with a C++ compiler. The object file resulting from the output file should then be linked with the object files resulting from the compilation of the HGPSS++ system files.

Chapter 6

Extended Backus Naur Form description

This section contains an Extended Backus-Naur Form description of the HGPSS-language. The notation consists of metalinguistic variables, metalinguistic symbols, HGPSS reserved words and HGPSS symbols. The grammar itself is presented as a sequence of rules, where each rule consists of a left-hand side and a right-hand side connected by the metasymbol ::=. The left-hand side of the rule is a metavariable. The right-hand side of the rule is an expression consisting of metavariables, metasymbols, HGPSS reserved words and HGPSS symbols. Wherever a metavariable appears in the right-hand side of a rule, the right-hand side of any rule having that variable as its left-hand side can be unambiguously substituted for the variable. This is the distinguishing characteristic of context-free grammars. A metavariable is a character string consisting of letters and underscores. The variable name is chosen to be meaningful in its context. The metasymbols to be used are the following:

- `::=` connects left- and right-hand side of a rule.
- `|` separates alternative right-hand side items.
- `[]` encloses an optional right-hand side item.
- `{}` encloses a repeated right-hand side item that can appear zero or more times.
- `()` is used for grouping alternative right-hand side items.
- `-` indicates a class of characters.

HGPSS reserved words are distinguished from metavariables by consisting only out of capitals. Exceptions to this rule are single quoted. HGPSS symbols are not quoted, except when there might be confusion with metasymbols. Other symbols used:

- `\t` tabulate character.
- `\r` line feed character.
- `\n` new line character.

```
program           ::= model_sections command_section
model_sections    ::= { model_section | start text }
model_section     ::= start MODEL white_space identifier [ '(' anything_except_) ')' ] end
                      decls
                      start ENDMODEL end
decls             ::= { start decl }
decl              ::=                        block_decl_body  end
                  |   identifier white_space block_decl_body  end
                  |   integer white_space entity_decl_body end
                  |   expression white_space entity_decl_body end
                  |   identifier white_space entity_decl_body end
                  |   text
command_section   ::= [ start COMMAND end
                      commands
                      start ENDCOMMAND end
                      texts ]
```

```
commands          ::= { start command }
command           ::= command_body end
                  |   text
texts             ::= { start text }
text              ::= %{ anything_except_%} %} new_line
                  |   - anything_except_new_line new_line
                  |   * anything_except_new_line new_line
                  |   new_line
block_decl_body   ::= ADVANCE    white_space (  /
                                              | parameter
                                              | [ parameter ] , parameter )
                  |   ASSEMBLE   white_space parameter
                  |   ASSIGN     white_space parameter [ + | - ] , parameter [ , parameter ]
                  |   BUFFER
                  |   DEPART     white_space parameter [ , parameter ]
                  |   ENTER      white_space parameter [ , parameter ]
                  |   ENTERMODEL white_space ( expression , ( integer | expression )
                                              | ( integer | identifier ) , entity_name )
                  |   EXTERN     white_space expression , expression
                  |   GATE       white_space gate_kind white_space parameter [ , parameter ]
                  |   GATHER     white_space parameter
                  |   GENERATE   white_space ( /
                                              | parameter
                                              | [ parameter ] , parameter
                                              | [ parameter ] , [ parameter ] , parameter
                                              | [ parameter ] , [ parameter ] , [ parameter ] , parameter
                                              | [ parameter ] , [ parameter ] , [ parameter ] , [ parameter ] ,
                                                parameter
                                              | [ parameter ] , [ parameter ] , [ parameter ] , [ parameter ] ,
                                                [ parameter ] , parameter
                                              | [ parameter ] , [ parameter ] , [ parameter ] , [ parameter ] ,
                                                [ parameter ] , parameter , F )
                  |   INPUT      white_space entity_name
                  |   INTERN     white_space expression
                  |   LEAVE      white_space parameter [ , parameter ]
                  |   LEAVEMODEL white_space ( expression , ( integer | expression )
                                              | ( integer | identifier ) , entity_name )
                  |   LINK       white_space parameter , ( FIFO
                                                         | LIFO
                                                         | P integer
                                                         | P expression ) [ , parameter ]
                  |   LOGIC      white_space logic_kind white_space parameter
                  |   LOOP       white_space parameter , parameter
                  |   MARK       white_space ( /
                                              | parameter )
                  |   MATCH      white_space parameter
                  |   MSAVEVALUE white_space parameter [ + | - ] , parameter , parameter , parameter [ , H ]
                  |   OUTPUT     white_space entity_name
                  |   PREEMPT    white_space parameter [ , PR
                                                       | , [ PR ] , parameter
                                                       | , [ PR ] , [ parameter ] , parameter
                                                       | , [ PR ] , [ parameter ] , [ parameter ] , RE ]
                  |   PRINT      white_space ( /
                                              | parameter
                                              | [ parameter ] , parameter
                                              | [ parameter ] , [ parameter ] , print_kind
                                              | [ parameter ] , [ parameter ] , [ print_kind ] , expression )
                  |   PRIORITY   white_space parameter , [ BUFFER ]
                  |   QUEUE      white_space parameter [ , parameter ]
                  |   RELEASE    white_space parameter
                  |   RETURN     white_space parameter
                  |   SAVEVALUE  white_space parameter [ + | - ] , parameter [ , H ]
                  |   SEIZE      white_space parameter
                  |   SELECT     white_space ( logical_select_kind white_space parameter , parameter ,
                                                parameter [ , , , parameter ]
                                              | min_max_select_kind white_space parameter , parameter ,
                                                parameter , , parameter
                                              | relational_select_kind white_space parameter , parameter ,
                                                parameter , parameter , parameter [ , parameter ] )
                  |   SPLIT      white_space parameter , parameter [ , parameter
                                                                   | , [ parameter ] , parameter ]
                  |   TABULATE   white_space parameter [ , parameter ]
                  |   TERMINATE  white_space ( /
                                              | parameter )
                  |   TEST       white_space test_kind white_space parameter , parameter [ , parameter ]
                  |   TRANSFER   white_space ( BOTH , [ parameter ] , parameter
                                              | parameter , [ parameter ] , parameter
                                              | , parameter )
                  |   UNLINK     white_space parameter , parameter , ( parameter | ALL )
```

```
                                        [ , ( parameter | BACK )
                                        | , [ parameter | BACK ] , parameter
                                        | , [ parameter | BACK ] , [ parameter ] , parameter ]
entity_decl_body   ::= BVARIABLE  white_space ( / expression
                                        | bvariable_expression )
                   |   FUNCTION   white_space ( / expression
                                        | parameter , function_kind integer end function_point
                                          { / function_point | - end function_point }
                   |   MATRIX     white_space ( X | H ) , ( integer | expression ) , ( integer | expression )
                   |   SUBMODEL   white_space identifier [ ( anything_except_) ) ]
                   |   QTABLE     white_space entity_name , value , value , ( integer | expression )
                   |   STORAGE    white_space ( integer | expression )
                   |   TABLE      white_space ( parameter | IA | RT ) , value , value ,
                                        [ W ] ( integer | expression ) [ , value ]
                   |   VARIABLE   white_space ( / expression
                                        | variable_expression )
                   |   FVARIABLE  white_space ( / expression
                                        | variable_expression )
command_body       ::= CLEAR
                   |   DOWN       white_space entity_name
                   |   END
                   |   INITIAL    white_space ( LS ( integer | expression | $ identifier  )
                                        | ( MX | MH ) ( integer | expression | $ identifier )
                                          '(' ( integer | expression ) , ( integer | expression ) ')'
                                          , value
                                        | ( X | XH ) ( integer | expression | $ identifier ) , value )
                   |   JOB
                   |   PRINT      white_space ( /
                                          | parameter
                                          | [ parameter ] , parameter
                                          | [ parameter ] , [ parameter ] , print_kind
                                          | [ parameter ] , [ parameter ] , [ print_kind ] , expression )
                   |   RESET
                   |   SIMULATE   white_space identifier [ '(' anything_except_) ')' ] [ , expression ]
                   |   START      white_space ( integer | expression )
                   |   UP
```

Auxiliary rules:

```
white_space            ::= ( '\t' | '\r' | ' ' ) { '\t' | '\r' | ' ' }
new_line               ::= '\n'
digit                  ::= 0-9
letter                 ::= a-z | A-Z | _ | $
integer                ::= digit { digit }
exponent               ::= ( E | 'e' ) [ + | - ] integer
real                   ::= integer . [ integer ] [ exponent ]
                       |   [ integer ] . integer [ exponent ]
                       |   integer exponent
identifier             ::= letter { letter | digit }
expression             ::= " anything_except_" "
value                  ::= [ + | - ] integer
                       |   [ + | - ] real
                       |   expression
anything_except_"      ::= string of zero or more characters not containing the " character
anything_except_\n     ::= string of zero or more characters not containing the new_line character
anything_except_)      ::= string of zero or more characters not containing the ) character
anything_except_%}     ::= string of zero or more characters not containing the %} substring
function_kind          ::= C
                       |   D
                       |   E
                       |   L
                       |   M
gate_kind              ::= LS
                       |   LR
                       |   M
                       |   NM
                       |   NI
                       |   I
                       |   NU
                       |   U
                       |   SE
                       |   SF
                       |   SNE
                       |   SNF
logic_kind             ::= I
                       |   R
                       |   S
print_kind             ::= BLOCK
                       |   BVARIABLE
```

```
                        |   EXTERN
                        |   FACILITY
                        |   FUNCTION
                        |   INPUT
                        |   LOGICSWITCH
                        |   MATCHINGCHAINCHAIN
                        |   MSAVEVALUE
                        |   OUTPUT
                        |   QUEUE
                        |   RANDOMNBRGENERATOR
                        |   SAVEVALUE
                        |   STORAGE
                        |   SUBMODEL
                        |   TABLE
                        |   TRANSACTION
                        |   USERCHAIN
                        |   VARIABLE
                        |   ABSOLUTECLOCK
                        |   RELATIVECLOCK
                        |   TERMINATIONCOUNTER
                        |   CURRENTEVENTCHAIN
                        |   FUTUREEVENTCHAIN
logical_select_kind     ::= U
                        |   NU
                        |   I
                        |   NI
                        |   SE
                        |   SNE
                        |   SF
                        |   SNF
                        |   LR
                        |   LS
min_max_select_kind     ::= MIN
                        |   MAX
relational_select_kind  ::= G
                        |   GE
                        |   E
                        |   NE
                        |   LE
                        |   L
test_kind               ::= G
                        |   GE
                        |   E
                        |   NE
                        |   LE
                        |   L
logical_attribute       ::= FU
                        |   FNU
                        |   FI
                        |   FNI
                        |   LS
                        |   LR
                        |   SE
                        |   SNE
                        |   SF
                        |   SNF
relational_operator     ::= ''E''
                        |   ''G''
                        |   ''GE''
                        |   ''L''
                        |   ''LE''
                        |   ''NE''
single_sna              ::= C1
                        |   PR
                        |   M1
plain_sna               ::= N
                        |   W
                        |   F
                        |   FC
                        |   FR
                        |   FT
                        |   FN
                        |   Q
                        |   QA
                        |   QC
                        |   QM
                        |   QT
                        |   QX
                        |   QZ
```

```
                        |   RN
                        |   X
                        |   XH
                        |   R
                        |   S
                        |   SA
                        |   SC
                        |   SR
                        |   SM
                        |   ST
                        |   TB
                        |   TC
                        |   TD
                        |   P
                        |   MP
                        |   CA
                        |   CC
                        |   CH
                        |   CM
                        |   CT
                        |   BV
                        |   V
matrix_sna              ::= MH
                        |   MX
start                   ::= [ white_space ]
end                     ::= new_line
                        |   white_space anything_except_\n new_line
_parameter              ::= single_sna
                        |   plain_sna ( integer | expression | $ identifier | * integer | * expression )
                        |   matrix_sna ( integer | expression | $ identifier | * integer | * expression )
                            '(' ( integer | expression ) , ( integer | expression ) ')'
parameter               ::= value
                        |   identifier
                        |   _parameter
entity_name             ::= integer
                        |   expression
                        |   identifier
function_point          ::= start value , ( value | identifier | _parameter )
term                    ::= integer
                        |   real
                        |   expression
                        |   _parameter
variable_expression     ::= term
                        |   + variable_expression
                        |   - variable_expression
                        |   variable_expression + variable_expression
                        |   variable_expression - variable_expression
                        |   variable_expression * variable_expression
                        |   variable_expression / variable_expression
                        |   variable_expression @ variable_expression
                        |   '(' variable_expression ')'
b_term                  ::= term
                        |   term relational_operator term
                        |   logical_attribute ( integer | expression | $ identifier | * integer | * expression )
bvariable_expression    ::= b_term
                        |   bvariable_expression + bvariable_expression
                        |   bvariable_expression * bvariable_expression
                        |   '(' bvariable_expression ')'
```

Chapter 7

Statement summary

In Tables 7.4, 7.5 and 7.6, a complete list of all HGPSS block declaration statements and their parameters is presented. Tables 7.7 and 7.8 list all entity declaration statements and all command statements respectively. Table 7.2 gives an overview of all Standard Numerical Attributes. There is no difference between the HGPSS Standard Numerical Attributes and those of GPSS. Contrary to the Standard Numerical Attributes, the HGPSS entity mnemonics are entirely different from those of GPSS. The HGPSS entity mnemonics are listed in Table 7.3. Table 7.1 lists and explains all metalinguistic variables used in the other tables.

The following items are relevant to a proper understanding of the tables:

- Parameters listed in adjacent columns have to be separated by commas in the program source.
- The metavariables indicating the type of the parameters are listed beneath a textual description of the parameter.
- A metavariable or literal encapsulated between `[` and `]` refers to an optional parameter.
- In certain cases, alternative combinations of parameters are listed top to bottom.
- Upper case letters and words are to be taken literally.
- Commas separate alternatives on the same line.
- `(...)` refers to a parameter list.
- All occurrences of `/`, `(` and `)` are to be taken literally.
- $\pm$ means `+` *or* `-`.

Metalinguistic Variables		
VARIABLE		
integer	Unsigned Integer Value	
identifier	Identifier containing Letters, Digits, _ and $ The First Character is a Letter, _ or $	
expression	Double-quoted C++-Expression	
value	Signed integer, Real or expression	
entity_name	integer, expression or identifier	
row	integer or expression	
column	integer or expression	
parameter	Single SNA	
	Plain SNA followed by	integer expression $identifier *integer *expression
	Matrix SNA followed by	integer(row,column) expression(row,column) $identifier(row,column) *integer(row,column) *expression(row,column)
mnemomic	Entity Mnemonic	

Table 7.1: Metalinguistic variables

HGPSS Standard Numerical Attributes		
Single SNA		
SNA	ENTITY	
C1	Processor	Value of Relative Clock
PR	Transaction	Priority Level
M1	Transaction	Residence Time in Model
Plain SNA		
SNA	ENTITY	
N	Block	Total Count
W	Block	Current Count
F	Facility	Availability Status
FC	Facility	Capture Count
FR	Facility	Fractional Utilisation
FT	Facility	Average Holding Time per Capture
FN	Function	Current Value
Q	Queue	Current Content
QA	Queue	Average Content
QC	Queue	Entry Count
QM	Queue	Maximum Content
QT	Queue	Average Residence Time
QX	Queue	Non-zero Average Residence Time
QZ	Queue	Non-zero Entry Count
RN	Random Number Generator	Random value between 0.0 inclusive and 1.0 exclusive
X	Savevalue	Value of Fullword Savevalue
XH	Savevalue	Value of Halfword Savevalue
R	Storage	Remaining Capacity
S	Storage	Current Content
SA	Storage	Average Content
SC	Storage	Entry Count
SR	Storage	Fractional Utilisation
SM	Storage	Maximum Content
ST	Storage	Average Holding Time per Unit
TB	Table	Average Value of Non-weighted Entries
TC	Table	Number of Non-weighted Entries
TD	Table	Standard Deviation of Non-weighted Entries
P	Transaction	Parameter Value
MP	Transaction	Time since move into MARK block
CA	User Chain	Average Content
CC	User Chain	Total Entries
CH	User Chain	Current Content
CM	User Chain	Maximum Content
CT	User Chain	Average Holding Time per Entry
BV	Boolean Variable	Current Value
V	Variable	Current Value
Matrix SNA		
SNA	ENTITY	
MH	Matrix Savevalue	Halfword Value of Element in specified Row and Column
MX	Matrix Savevalue	Fullword Value of Element in specified Row and Column

Table 7.2: HGPSS Standard Numerical Attributes

HGPSS Entity Mnemonics	
MNEMONIC	
BLOCK	Block
BVARIABLE	Boolean Variable
EXTERN	Extern
FACILITY	Facility
FUNCTION	Function
INPUT	Input
LOGICSWITCH	Logic Switch
MATCHINGCHAINCHAIN	Chain of Matching Chains
MSAVEVALUE	Matrix Savevalue
OUTPUT	Output
QUEUE	Queue
RANDOMNBRGENERATOR	Random Number Generator
SAVEVALUE	Savevalue
STORAGE	Storage
SUBMODEL	Submodel
TABLE	Table
TRANSACTION	Transaction
USERCHAIN	User Chain
VARIABLE	Variable
ABSOLUTECLOCK	Absolute Clock
RELATIVECLOCK	Relative Clock
TERMINATIONCOUNTER	Termination Counter
CURRENTEVENTCHAIN	Current Event Chain
FUTUREEVENTCHAIN	Future Event Chain

Table 7.3: HGPSS entity mnemonics

HGPSS Block Declaration Summary, Part 1							
BLOCK	A	B	C	D	E	F	G
ADVANCE	Mean Time	Spread Modifier					
	[parameter]	[parameter]					
ASSEMBLE	Assembly Count						
	parameter						
ASSIGN	Parameter No.	Value to be Assigned	No. of Function Modifier				
	parameter[±]	parameter	[parameter]				
BUFFER							
DEPART	Queue Name	No. of Units					
	parameter	[parameter]					
ENTER	Storage Name	No. of Units					
	parameter	[parameter]					
ENTERMODEL	Model Name	Input Name					
	expression expression integer identifier	integer expression entity_name entity_name					
EXTERN	Function Called upon Arrival	Function Called upon Polling					
	expression	expression					
GATE { *LS* / *LR* }	Logic Switch Name	Next Block if Condition is False					
	parameter	[parameter]					
GATE { *M* / *NM* }	Name of ASSEMBLE, GATHER, or MATCH Block	Next Block if Condition is False					
	parameter	[parameter]					
GATE { *NI* / *I* / *NU* / *U* }	Facility Name	Next Block if Condition is False					
	parameter	[parameter]					
GATE { *SE* / *SF* / *SNE* / *SNF* }	Storage Name	Next Block if Condition is False					
	parameter	[parameter]					
GATHER	Gather Count						
	parameter						
GENERATE	Mean Time	Spread Modifier	Offset Interval	Limit Count	Priority Level	No. of Parameters	Type of Parameters
	[parameter]	[parameter]	[parameter]	[parameter]	[parameter]	[parameter]	[F]
INPUT	Input Name						
	entity_name						

Table 7.4: HGPSS block declaration summary, part 1

HGPSS Block Declaration Summary, Part 2							
BLOCK	A	B	C	D	E	F	G
INTERN	Function Called upon Arrival						
	expression						
LEAVE	Storage Name	No. of Units					
	parameter	[parameter]					
LEAVEMODEL	Model Name	Output Name					
	expression expression integer identifier	integer expression entity_name entity_name					
LINK	Name of User Chain	Merge Criterion	Alternate Block				
	parameter parameter parameter	FIFO LIFO expression	[parameter] [parameter] [parameter]				
LOGIC $\left\{ \begin{matrix} I \\ R \\ S \end{matrix} \right\}$	Logic Switch Name						
	parameter						
LOOP	Parameter No.	Next Block if Parameter $\neq 0$					
	parameter	parameter					
MARK	Parameter No.						
	[parameter]						
MATCH	Name of Conjugate MATCH Block						
	parameter						
MSAVEVALUE	Matrix Name	Row No.	Column No.	Value to be Saved	Matrix Type		
	parameter[±]	parameter	parameter	parameter	[H]		
OUTPUT	Output name						
	entity_name						
PREEMPT	Facility Name	Priority Option	Next Block for Preempted Transaction	Parameter No. of Preempted Transaction	Remove Option		
	parameter	[PR]	[parameter]	[parameter]	[RE]		
PRINT	Lower Limit	Upper Limit	Entity Mnemonic	File Name			
	[parameter]	[parameter]	[mnemomic]	[expression]			
PRIORITY	New PR Value	Buffer Option					
	parameter	[BUFFER]					
QUEUE	Queue Name	No. of Units					
	parameter	[parameter]					
RELEASE	Facility Name						
	parameter						
RETURN	Facility Name						
	parameter						
SAVEVALUE	Savevalue Name	Value to be Saved	Savevalue Type				
	parameter[±]	parameter	[H]				
SEIZE	Facility Name						
	parameter						

Table 7.5: HGPSS block declaration summary, part 2

HGPSS Block Declaration Summary, Part 3							
BLOCK	A	B	C	D	E	F	G
SELECT { *U*,*NU* / *I*,*NI* / *SE*,*SNE* / *SF*,*SNF* / *LR*,*LS* }	Parameter in which to Place Entity No.	Lower Limit	Upper Limit			Alternate Exit	
	parameter	parameter	parameter			[parameter]	
SELECT { *MIN* / *MAX* }	Parameter in which to Place Entity No.	Lower Limit	Upper Limit		Attribute to Examine		
	parameter	parameter	parameter		parameter		
SELECT { *G* / *GE* / *E* / *NE* / *LE* / *L* }	Parameter in which to Place Entity No.	Lower Limit	Upper Limit	Comparison Value	Attribute to Examine	Alternate Exit	
	parameter	parameter	parameter	parameter	parameter	[parameter]	
SPLIT	No. of Offspring	Next Block for Offspring	Serialization Parameter	No. of Parameters for each Offspring			
	parameter	parameter	[parameter]	[parameter]			
TABULATE	Table Name	Weighting Factor					
	parameter	[parameter]					
TERMINATE	Termination Counter Decrement						
	[parameter]						
TEST { *G* / *GE* / *E* / *NE* / *LE* / *L* }	First Value	Second Value	Next Block if Condition is False				
	parameter	parameter	[parameter]				
TRANSFER	Selection Mode	First Block Examined	Second Block Examined				
	BOTH	[parameter]	parameter				
TRANSFER	Selection Mode	First Block	Second Block				
	parameter	[parameter]	parameter				
TRANSFER	Selection Mode	Next Block Entered					
		parameter					
UNLINK	Name of User Chain	Next Block for Unlinked Transactions	Unlink Count	Parameter No.	Match Argument	Alternate Exit	
	parameter parameter parameter parameter	parameter parameter parameter parameter	parameter ALL parameter ALL	[parameter] [parameter] [BACK] [BACK]	[parameter] [parameter] [parameter] [parameter]	[parameter] [parameter] [parameter] [parameter]	

Table 7.6: HGPSS block declaration summary, part 3

HGPSS Entity Declaration Summary					
ENTITY	A	B	C	D	E
BVARIABLE	Combination of Numeric Data Specifications, Relational Operators, Logical Attributes and Boolean Operators				
	Numeric Data Specifications	Relational Operators	Logical Attributes	Boolean Operators	
	parameter	'G', 'GE' 'L', 'LE' 'E', 'NE'	FU, FNU FI, FNI SE, SNE SF, SNF	+, *	
BVARIABLE	Function Name				
	/expression				
FUNCTION	Function Argument	Function Type and No. of Points	Data points separated by / or - + new line		
	parameter parameter parameter parameter parameter	Cinteger Dinteger Einteger Linteger Minteger	value,value value,identifier value,parameter		
FUNCTION	Function Name				
	/expresion				
MATRIX	Matrix Type	Number of Rows	Number of Columns		
	X, H	integer, expression	integer, expression		
SUBMODEL	Submodel Type and Arguments				
	identifier[(...)]				
QTABLE	Name of Queue	Inclusive Upper Limit of Lowest Frequency Class	Width of Intermediate Frequency Classes	Number of Frequency Classes	
	entity_name	value	value	integer, expression	
STORAGE	Storage Capacity				
	integer, expression				
TABLE	Table Arguments	Inclusive Upper Limit of Lowest Frequency Class	Width of Intermediate Frequency Classes	Number of Frequency Classes	Time Interval for RT-Table
	parameter, IA, RT parameter, IA, RT	value value	value value	[W]integer [W]expression	[value] [value]
VARIABLE	Combination of Numeric Data Specifications and Arithmetic Operators				
	Numeric Data Specifications	Arithmetic Operators			
	parameter	+, -, /, *, @			
VARIABLE	Function Name				
	/expression				
FVARIABLE	Combination of Numeric Data Specifications and Arithmetic Operators				
	Numeric Data Specifications	Arithmetic Operators			
	parameter	+, -, /, *, @			
FVARIABLE	Function Name				
	/expression				

Table 7.7: HGPSS entity declaration summary

HGPSS Command Summary				
COMMAND	A	B	C	D
CLEAR				
DOWN	Submodel Name			
	entity_name			
END				
INITIAL	Logic Switch to be Set			
	LSinteger LSexpression LS$identifier			
INITIAL	Matrix Savevalue	Initial Value		
	MXinteger(row',column) MXexpression(row',column) MX$identifier(row',column) MHinteger(row',column) MHexpression(row',column) MH$identifier(row',column)	value value value value value value		
INITIAL	Savevalue	Initial Value		
	Xinteger Xexpression X$identifier XHinteger XHexpression XH$identifier	value value value value value value		
JOB				
PRINT	Lower Limit	Upper Limit	Entity Mnemonic	File Name
	[parameter]	[parameter]	[mnemonic]	[expression]
RESET				
SIMULATE	Model Type and Arguments	File Name		
	identifier[(...)]	[expression]		
START	Initial Termination Counter Value			
	integer, expression			
UP				

Table 7.8: HGPSS command summary

Bibliography

[Boehm & Jacopini 1966]
Boehm, Corrado
Jacopini, Giuseppi
Flow diagrams, Turing machines, and languages with only two formation rules
Communications of the ACM, vol. 9, no. 5
1966

[Claeys 1992]
Claeys, Filip
HGPSS: Object-georiënteerde "process-interaction" simulatie
M.Sc. in Computer Science thesis
Universiteit Gent
Ghent, Belgium 1992

[Gordon 1978]
Gordon, G.
System Simulation
Prentice-Hall
Englewood Cliffs, New Jersey, U.S.A. 1978

[Mason & Brown 1990]
Mason, Tony
Brown, Doug
LEX & YACC
O'Reilly & Associates, Inc.
Sebastopol, California, U.S.A. 1990

[Neelamkavil 1987]
Neelamkavil, Francis
Computer simulation and modelling
John Wiley & Sons
1987

[Schriber 1974]
Schriber, Thomas J.
Simulation Using GPSS
John Wiley & Sons
New York, New Jersey, U.S.A. 1974

[Stroustrup 1987]
Stroustrup, Bjarne
The C++ Programming Language
Addison-Wesley
1987

[Zeigler 1976]
Zeigler, B.P.
Theory of Modelling and Simulation
John Wiley & Sons
New York, New Jersey, U.S.A 1976