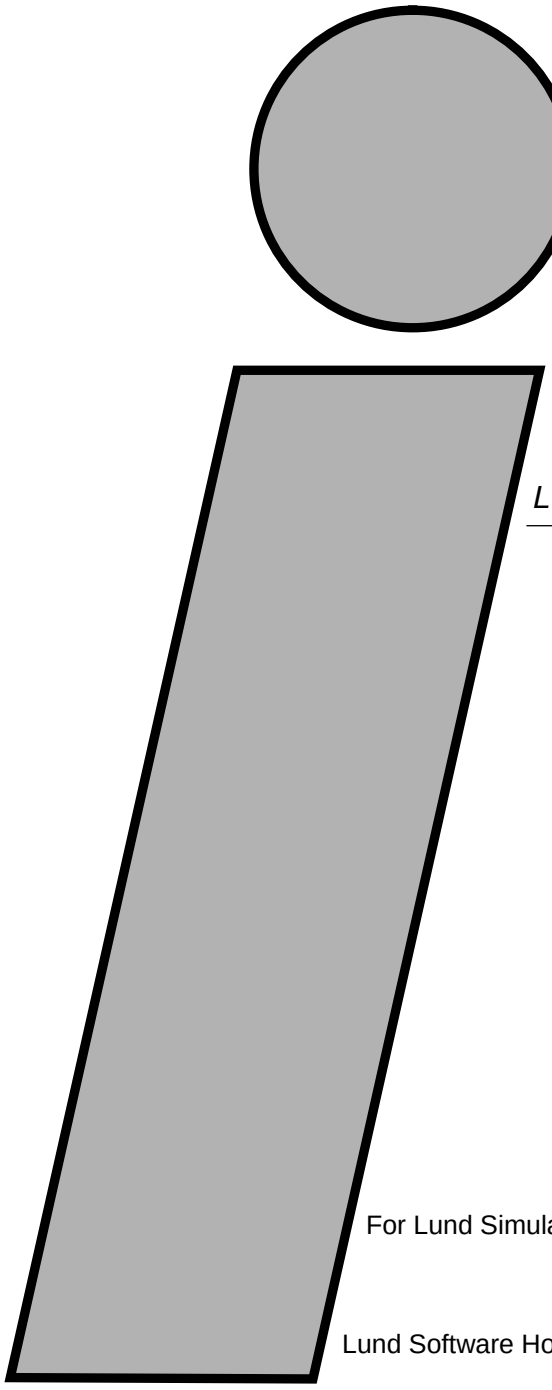

Lund Simula Documentation

Using Lund Simula on Unix Systems

For Lund Simula version 4.15 or later

Lund Software House AB, Sweden

Lund Simula Documentation

Using Lund Simula on Unix Systems
Version 4.15
by Boris Magnusson and Per Holm

Printed at: 7 December 1995 4:13 pm

Lund Software House AB
P.O.Box 7056
S-220 07 Lund, Sweden

Table of Contents

1 Introduction

Simula is a general purpose programming language. It is the origin of object oriented programming. The concepts of generalization and specialization is very well supported by the language constructs class and virtual procedure. In Simula inheritance of actions are just as natural as inheritance of data attributes. Constructs for modular programming and implementation of abstract datatypes make the language well suited for solving complex problems. In addition Simula is a well standardized language, which means that Simula programs are easy to port from one computer to another. A standard application package, class simulation, turns the language into a powerful tool for process-oriented simulation.

Simula is an extension of Algol 60 and is defined in the Simula Standard [1]. Extensions and changes to the language standard are made by the Simula Standards Group, SSG, where the implementors of the language are members. Many users are members of the Association of Simula Users, ASU. The quarterly ASU publication, Simula Newsletter, is a valuable source of information about matters concerning Simula.

This document is a guide on how to use the Lund Simula system, it is not a guide to Simula itself. It is written with the assumption that the reader has a fair knowledge of Simula. For an introduction to the language, Simula begin [2], An introduction to programming in Simula [3] and Objektorienterad Programmering i Simula [4] (in Swedish), Object-Oriented Programming with Simula [5] and (Serafim) [6] are recommended.

Throughout the document RTS is used as an abbreviation for Run-Time System. In the examples the font helvetica is used for computer written text and helvetica italics for text typed by the user. % is the UNIX command prompt.

The Simula system for UNIX[1]™ has been developed by Lund Software House, Sweden. It is a sub-implementation based on the Simula system developed by Lund Software, Sweden. It conforms to the Simula Standard. The system features a powerful symbolic debugger, SIMDEB. SIMDEB is documented in a manual of its own [7].

Addresses

Lund Software House
P.O.Box 7056
S-220 07 Lund
Sweden

ASU Secretary, Henry Islo
The Royal Institute of Technology
Department of Manufacturing Systems
S-100 44 Stockholm
Sweden

1. ™ UNIX is a trademark of AT&T

2 An Introductory Example

Before diving into the details of the system we show an example of a Simula program and how to compile and run it at your installation.

Create the following Simula program with the help of your favorite text editor. Give your program the name stay.sim:

```
begin
   Outtext("Simula is here to stay.");
   Outimage;
end
```

The program is compiled with the simcomp command:

```
%simcomp  stay
Lund Software STANDARD Simula Compiler revision 4.15.
Executing at Lund Software House 12-OCT-95 17:16:00
End of pass 1 0.090 seconds
End of pass 2 0.090 seconds
End of pass 3 0.160 seconds
End of pass 4 0.020 seconds
End of pass 5 0.000 seconds
End of pass 6 0.280 seconds

   No errors and no warnings.
   End of Simula compilation.
```

The simcomp command will produce two files: stay.o and stay.atr. The file stay.o contains the relocatable object code. The stay.atr file contains information for the debugger and for separate compilation, it is of no concern here. The program is linked with the simld command.

```
%simld   stay
```

The simld command will produce the file stay, containing executable code. You can now run the program by entering:

```
%stay
Simula is here to stay.
%
```

You will ordinarily use Simula through shell scripts: simcomp, simmake and simld. Use *'man simula'* or *'simman simula'* for on-line documentation.

The next chapter describes a few useful scripts distributed with the system. The compiler itself is described in chapter 4, and appendix A gives the complete syntax and semantics for the command which invokes the compiler. Compiler error messages are listed in appendix D. Chapter 5 describes how Simula programs are linked. The Run-Time System is described in Chapter 6. Appendix C gives some specific hardware information.

Chapter 7 gives detailed information on separate compilation of Simula classes and procedures. Chapter 8 explains how to call external C procedures.

3 Useful scripts

You will usually invoke Simula through shell scripts. This chapter describes a few useful scripts distributed with the system. They are normally to be found in /usr/local/simulabin with links installed to them in /usr/local/bin

- simcomp script used to run the Simula compiler (and the 'as' assembler when needed) to translate the program into an object file (see chapter 4).
- simld script used to run the ld linker to combine a Simula main program, possibly separately compiled Simula classes and procedures, and library routines into an executable program (see chapter 5).
- simmake script used to selectively compile a set of interdependent files as needed (see chapter 7.4).
- simman script to access the simula on-line 'man' pages (see chapter 13).

These shell scripts take the name of one or more Simula program files as parameters. You may give options before the file name. An example:

```
%simcomp –h stay
%simld stay
%stay –p
```

The source program file should have the extension '.sim'. By extension we mean the last dot of the file name and any characters following it. You need not supply the '.sim' file name extension, if not, the compiler will extend the supplied name with this default extension.

Various files are produced when running the scripts. The names of these files are made up from the base name of the source program file and an appropriate extension. I. e. any '.sim' extension is stripped from the source file name before the name of a produced file is made up. Using the above example, let us show the names of the files involved:

stay.sim	– Simula source program
stay.atr	– attribute information
stay.o	– object code
stay	– executable machine code
stay.s	– optional symbolic assembly code
stay.lis	– optional list file
stay.err	– optional error message file
stay.zr	– optional symbolic intermediate code

A program may consist of separately compiled classes and procedures in addition to the main program. These has to be compiled bottom up in the order they import each other. When linking such a program the names of the separately compiled components must also be supplied. The same goes when a Sim-

ula program calls routines written in other languages. Then the names of the files containing the object code for these routines must be supplied to the linker. An example:

```
%simld myProg classA procedureP Croutine - lX
```

4 Compiling

The command which invokes the Simula compiler has the following general structure:

```
simcomp [-option] ... file
```

The normal action of the Simula compiler is to read the input file and translate it into an object-file and an attribute information file. This normal action can be changed by the use of options. For a complete list of all the options, see Appendix A.

The compiler runs in six passes, and each pass ends with a message. All produced files have the same name as the input file, but with different extensions as described in the previous section. The main results are the object code in a file with extension '.o', and the attribute information file with the extension '.atr'. This file contains information used by the debugger and for separate compilation. Systems of dependent separately compiled Simula classes or procedures must be compiled 'bottom up' with independent files first and main programs last. See chapter 7 for more details on separate compilation. If errors are detected by the compiler, they are sorted and listed at the end of the compilation. Errors are by default listed on standard error. The input file must be present, if not the compiler complains.

The Simula compiler behaves slightly differently in different UNIX implementations. In some implementations it produces binary object code directly (in a '.o' file). In other implementations it produces symbolic assembly code (in a '.s' file), which is then fed through the assembler. This difference is hidden by the simcomp script which invokes the assembler in case the compiler produce assembly code.

4.1 Error Handling

At the end of the fourth pass error messages are written to the standard error file, usually the terminal. The compiler is designed to recover from an error and to detect all errors in a program. Even if a program contains a number of errors, code is usually produced. When an error is found a call to a special run-time error routine is inserted in the produced code. This makes it possible to execute and debug a program even if it does not compile correctly. The following error types are defined:

Warning - does not affect the produced code in any way.

Error - due to a violation of the Simula rules. Code is produced to call a run-time routine which issues an error message.

Fatal - aborts the compilation immediately, no code is produced.
OS - due to violation of some operating system rule, e.g. ”illegal file name”.
Internal - probably due to compiler malfunction. Often, an internal error will have its root in a user error so ingenious that the compiler constructors have never thought of it.

If the compiler reports Warnings and/or Errors the resulting program is executable, if, however any of the other error types are reported, these have to be corrected and the program recompiled before the program can be executed. Error codes and error messages are listed in appendix D. If for some reason the compilation is terminated before the error listing is produced, consult Appendix D for instructions on how to obtain and interpret an intermediate error listing.

Please report all compiler malfunctions, using the error form of Appendix E.

4.2 Line Numbers

The compiler generates code to keep track of the line number where the program is currently executing. This makes it possible to produce informative error messages during run-time, and to use the debugger facilities.

Line number code is generated at the first executable statement of each line and at certain other selected places, such as at inner and end. If you write expressions which range over several source lines, the line number will not be updated at the new lines inside the expression. This has the consequence that an error that occurs during the calculation of such an expression may lead to a somewhat misleading error message.

Example:

```
1 begin
2   integer array A (1:10);
3   A(1) := A(2)+
4       A(12);
5 end
```

The error message will in this case be ”Subscript out of range at line 3", because that is the line at which the statement started.

4.3 Compiler Directives

A source program line that has a '%' (ASCII 37) as the first character of the line is treated as a compiler directive. Note that the only compiler directive which presently is standardized is %<space>, the line comment.

The following directives are currently implemented:

%<white space> text
A directive line with a space or horizontal tab directly following the %-character is regarded as a comment line.

%INCLUDE filename
The source file with the name filename is included in place of the INCLUDE directive line. INCLUDE directives may be nested, i.e. the included file may contain another INCLUDE directive, but only to a level of 5.

%LINENUMBERS OFF
Inhibit generation of code to keep track of the source program line numbers during run-time. This will lead to reductions in code size and execution time for the affected program sections, but will also make the use of several of the debugger commands impossible. (See also '–h' option to simcomp, Appendix A.)

%LINENUMBERS ON
Turn on the generation of line number code. This is the default. This option is ignored if the -h option is set upon invocation of the compiler.

%PAGE
Eject to the top of a new page in the listing file. If no program listing is being produced this directive has no effect.

%LIST OFF
Turn off listing of the following lines of the source program, until a%LIST ON command is encountered.

%LIST ON
Turn on source program listing. This is the default. It has effect only if the l option is set upon invocation of the compiler.

%TITLE text
Print a heading on each page of the program listing. The heading text is separated from the directive TITLE by one space character.

%PAGELENGTH n
Set lines per page to n for the program listing. n must be a positive number. The default value is 60.

4.4 General Description

For the interested reader the six passes of the compiler are briefly described.

- Pass 1 This pass performs the lexical analysis.
- Pass 2 Declarations and block structure are analyzed. The program is divided into blocks. A block is the input unit to pass 3.
- Pass 3 Syntax and semantic analysis is done, and intermediate code is produced. The intermediate code can be regarded as machine code for a fictitious computer, with instructions on a slightly higher level than that of the target computer.
- Pass 4 This pass produces a sorted list of the source code with line numbers and error messages if errors are present. It also writes a list of the encountered errors to the standard error file, usually the termi-

nal. Most of the actions of this pass are optional and governed by options.

Pass 5 Prints the intermediate code in symbolic form, if requested by the z option. This facility is mainly for use by the compiler implementors.

Pass 6 Translates the intermediate code to machine code.

5 Linking

A compiled Simula program must be linked with Simula's Run-Time System. The linking is done by the standard UNIX linker ld. The linker resolves all references from your program to the Run-Time System. The following (simplified) linker command is usually given:

```
%ld –o program /lib/crt0.o program.o –lsimula –lc
```

This gives as output executable machine code on the file program, as specified by the -o option. The file /lib/crt0.o contains the entry point where the program will start executing, it must occur before all other object code files in the command. Not all Unix system keep crt0.o in this location but the file is placed in some other location, sometimes named by the person installing the software. In such cases the simld-script has to be modified to match the local installation. This is normally done when Simula is installed. If this is a problem at your site it will show as an error message from the linker when using simld. The Run-Time System is usually found in the library file /usr/local/simulabin/lib/libsimula.a. To link a Simula program, the C library (usually in /usr/libc.a) must also be available. The -l options tells the linker to search these two libraries, in the correct order.

Most modern UNIX versions have more advanced linking scheme which means that a more elaborate linking command is needed. It is recommended to use the script simld in which these details have been hidden. It may be different for different UNIX implementations. simld takes the base name of the main program source file as argument, you can see an example of this in section 2. If the program refers to separately compiled classes or procedures, the names of their source files must also be supplied. See an example of such use in section 3 and 7.1. For a detailed decription of simld and options, see Appendix A.

If you want to use the Simula debugger during execution of the program you must add the -d option to simld. For more details on the use of the debugger, please refer to the SIMDEB User's Guide[5].

An important design goal of the Run-Time System, and indeed of the entire Simula system, has been to keep down the size of executable programs. This goal has resulted in the Run-Time System being fragmented into several small routines. When a program is linked as above, you will get only the RTS routines actually referred to by your program.

6 Simula Program Execution

This chapter describes how to run your Simula program when it has been compiled and linked. To run a program, just type the name of the file containing the executable code, possibly followed by one or more run-time options. Example:

```
%simulaprogram –p –k=512
```

In this case '–p' will make the runtime system print a 'log on' message including the current version number and a 'log off' message including time consumed and time spent on garbage collection. The '–k=' option is used to control the amount of logical memory used by the Simula runtime system. For a complete list of all the options see Appendix B. To start the program in the Debugger add the '–d' option to the program execution line. For more details on how to use the debugger, please refer to the SIMDEB User's Guide[5].

7 External Simula classes and procedures

The Lund Simula system fully implements separate compilation as described in [1]. In addition there are facilities for calling procedures written in other programming languages (see chapter 8). A source module is contained in one file and may be compiled separately. A source module is either:

- a main program
- a class
- a procedure
- several classes and/or procedures in the same file[1]

When a class or a procedure contained in the file source-file is compiled separately, an attribute file with name source-file.atr is created. The attribute file contains information about the declaration structure of the module. This information is used by the compiler for checking purposes when another module makes use of the class or procedure in source-file. In addition to information necessary for separate compilation, the attribute file contains information which is used by the debugger. Therefore an attribute file is created also for a main program.

7.1 Naming a separately compiled module

It is often practical to put separately compiled classes or procedures in files with the same name as the class or procedure (plus the '.sim' extension). When a programmer is making use of a separately compiled module, this is done by declaring the class or procedure in question as external.

---

1. This is a slight extension (or very liberal interpretation) of the Standard, originally a bug, which was explored by users and has turned out to be very useful.

Example:

```
external class  Aclass ;
```

The name of the attribute file can be communicated to the compiler in one of two ways, either implicitly or explicitly. In the above example implicit naming is used. The attribute file is then supposed to have the name aclass.atr. In general the attribute file is supposed to have the name external-id.atr, where the file name external-id.atr is always lower case, regardless of the actual spelling of the class or procedure name in the program text.

Implicit naming is often convenient to use, but when this is not the case, the attribute file can be named explicitly.

Example:

```
external class  Aclass = "Attribute-File" ;
```

Here the compiler assumes that the attribute file is named Attribute-File. If no such file exists, the compiler will automatically supply an '.atr' extension to the file name, and try Attribute-File.atr instead. When using the extended capability of putting several classes or procedures in the same file, the external declaration should match the *first* declaration in the file. The rest of the declarations in the file are brought in as a side effect of bringing in this first declaration.

7.2 Finding attribute files

When a source module is compiled, all the attribute files named implicitly or explicitly by external declarations must be available and contain up-to-date information. The compiler always look in the current working directory first for attribute files (and include files) it needs. If the file is not found there, the compiler uses the *library searchlist*, a list of directories, where it looks for the attribute files[1]. The searchlist can be defined in two parts on the command line, using the –I= and –L= options. The –I= option defines a set of libraries (implemented as directories) to search for attribute files. The compiler first tries to find these libraries locally (or directly if they are absolute filenames), if not found the compiler uses the *directory searchlist* to look for the library directories.

The *directory searchlist* is defined on the command line with the –L= option and by the environment variable, SIM_LIBRARY_PATH. In both cases they define a string with directories separated by colons (':'). The compiler searches for libraries first in the directories given by the –L= option on the command line, the directories taken in order from left to right. If the needed file is still not found the directories given in SIM_LIBRARY_PATH are searched, again from left to right. The result of the search is to establish a set of directories – matching the library names – where to search for attribute files. The search order for libraries is thus:

---

1. The mechanism is modelled after the options to 'ld', –L and –l respectively

1. '.', the current directory
2. directories given explicitly with the '–L=' option
3. directories given with the environment variable SIM_LIBRARY_PATH

The *library searchlist* is defined on the command line with the '–I=' option, which again defines a string with directories separated by colons (':'). The compiler searches the directory searchlist for these libraries and defines the library searchlist from these names, again from left to right. This list is finally used to find files searched for. The search order for libraries is thus:

1. '.', the current directory
2. directories matching the libraries given with the -I= option, from left to right.
3.
4.

With this mechanism it is thus not necessary to supply the full pathname of a file in the source, but it can be put together at compile time from three parts: the implicit or explicit filename, one of the directories given in the library searchlist and one of the directories given in the directory search path. The following examples are equivalent:

```
simcomp -I=/usr/local/simulabin/simlib:/usr/local/simulabin/libsim <file>
simcomp -L=/usr/local/simulabin -I=simlib:libsim <file>
```

Assuming libsim and simlib are directories installed in the default location /usr/local/simulabin, the compiler will look for attribute files and include files in:

1. '.', the current directory
2. /usr/local/simulabin/simlib
3. /usr/local/simulabin/libsim

There is furthermore an convenient option to simcomp, '–l', which makes simcomp append the list of Simula library directories in /usr/local/simulabin, supplied with the Simula system, to the end of directories given with the '–I=' option, and /usr/local/simulabin to the end of the string given by the '–L=' option. A similar option is available with the simld script. The above example could thus be written simply:

```
simcomp -l <file>
```

The search list mechanism is also useful to manage locally developed libraries. When handling versions of such libraries it is only necessary to change a name in the '–I='- string (or the '–L='- string) in order to switch between library versions.

Sometimes it can be unclear which attribute file is actually consulted by the compiler. This may particularly be the case if several versions of libraries are kept in different directories at the same time. In the case there is confusion of where an attribute file is found the –f option instructs the compiler to print the full filename of all files it reads.

7.3 Interface checking

The compiler is only able to check that the attribute file is up to date in some special cases. This is when an external file is read, and an external file it is using in its turn, is younger. If so, the compiler will issue a message saying that the intermediate file needs recompilation.

In general, however, the check that the source modules have been compiled in the correct order is delayed until link time. This check is performed as follows. The object code for a separately compiled module will contain globally visible labels of the form <name><compilation-time><sequence-number>. The compiler uses the <compilation-time>, which is saved in the attribute file, to rebuild the label names. If a separate module is recompiled, the label names will change. If a depending module is not recompiled, it will use the old label names. This will cause the linker to produce error messages, complaining about undefined names. The <name> part gives a hint on which module to recompile.

Example: example.sim contains:

```
class Example(x); integer x;;
```

main.sim contains

```
begin
   external class Example;
   ref(Example) P;
   P :- new Example(2);
end
```

The command sequence:

```
%simcomp example
%simcomp main
%simld main example
```

should run with no problem.

Changing example.sim and recompile it, but not recompiling main should result in a link time error:

```
%simcomp example
%simld main example
Undefined: example95101309201344001
```

Recompiling main and then running simld again fixes the problem.

When recompiling an external module, the time-stamp is updated only when the signature information in the attribute file is changed. This means that recompilation of an external module after minor changes to the statement part can be done without need to recompile the depending modules (which might be many since the change ripples through the all the depending modules). The compiler issues a warning when it recompiles a module and it needs to update the attribute file. There is an option, '–r', which makes the compiler terminate the recompilation in case it needs to update the attribute file. This option might

be useful when doing changes to modules that are used by a large volume of software or by many users. In such cases it might be desirable to coordinate changes that require re-compilation.

7.4 Maintaining sets of related files with simmake

It is recommended that Simula external classes and procedures are kept in a directory with a name that reflects the library functionality. The standard Unix way of maintaining the compilation consistency of a set of files is to use the make facility. In order to use make the dependencies among related files are collected in a file, a Makefile. make will automatically order necessary recompilations, depending on the modification dates on the involved files.

Lund Simula offers the simmake utility to manage this task. Simmake reads Simula source-files and analyzes the dependencies between Simula files (by looking for external declarations and %include directives). This information can be used create a Makefile automatically. This works well, but there are some slight drawbacks: the Makefile needs to be updated when the dependencies change. Also, since make is looking at time-stamps, it will recompile files unnecessarily when a recompilation has resulted in an unchanged interface.

As an alternative, simmake can be instructed to run the compiler directly. Simmake understands the information in the attribute files and can do a better job than make when recompiling files after a change.

In the simplest case, a set of Simula files in a directory can be recompiled with the single command:

```
%simmake *.sim –c
```

It is, however, often a good idea to put this command, and the command to build an object library into a simple Makefile:

```
FILES=filea.sim fileb.sim filec.sim
LIB=libuseful.a
USEDLIBS=-L=/usr/local/simulabin -I=simlib:libsim

TODO: COMPILE $(LIB)
COMPILE:      ;simmake $(USEDLIBS) $(FILES) -c
$(LIB): $(FILES:.sim=.o)
                ar c $(LIB) $(FILES:.sim=.o)
```

In this example you need to update the macros FILES and LIB to reflect the contents and name of your own library and USEDLIBS to reflect dependencies on separately compiled classes/procedures in other Simula libraries. Simmake takes the same '–L=' , '–I=' and '–l' options as described above for simcomp.

Simmake can also be used more conservatively. It can print the commands needed to update a set of files, rather than executing them (option –s), or automatically create the dependency part of a Makefile (option –m). Details of simmake and its options are found in Appendix A.

8 Calling external C-routines

Routines written in C may be called from a Simula program. The C-routines must be declared in the Simula program according to one of the following two forms:

```
external C procedure <Identifier> is <procedure-declaration>;
external C procedure <Identifier>="String" is <procedure-declaration>;
```

The purpose of the <procedure-declaration> is to describe the procedure and its parameters to the compiler. The declaration must have an empty procedure body, written as ';' or 'begin end;'. Examples:

```
external C procedure put is
   procedure put(x); integer x;;
external C procedure get is
   integer procedure get;  begin end;
external C procedure xxx="XDisplay" is
   integer procedure InitDisplay;;
```

The name of the C procedure to call can be given in two different forms as shown above. In the first alternative, <Identifier> is converted into lower case, in the second case <Identifier> is ignored and 'String' is used (exactly as given with regards to upper/lower case). In the object code this name is used to identify the C procedure (and in both cases prefixed with an underscore on Unix BSD implementations to match C-compiler conventions). The name given in the <procedure-declaration> is used in the Simula program to call the C procedure and can thus be used to rename it.

The Simula standard describes an alternative syntax for the declaration of non-Simula procedures, without the 'is <procedure-declaration>' clause. This possibility is not supported.

8.1 Simula memory management and C-routines

Simula data structures, i.e. objects, procedure activation records, arrays and texts, are allocated in a separate memory area. This memory area is managed by a garbage collector which means that data structures can be moved in memory, in which case it updates all references kept in the Simula memory area accordingly. Addresses into the Simula memory area which have been created and used when calling a C routine can naturally not be found or updated by the Simula garbage collector. Such addresses are therefore very short-lived and risk to be invalidated when the garbage collector gets control the next time. They are only guaranteed to be valid for the duration of the call where created and should thus not be stored or copied in the called C-routine. C-routines that violate this rule are inherently unsafe and should be avoided or encapsulated within a C-routine with a proper interface.

In a couple of situations, addresses to Simula data structures must be to exposed to C-routines. Calling C-routines that have text strings, arrays, or C-

structs as input parameters is one situation. Here the C-routine only accesses data in the Simula memory area. If the Simula interface specification, or the call, is erroneous or does not meet the expectations of the C-routine (say a text string is shorter than expected) this means that the C-routine will fail or calculate bad results, but it will not likely harm Simula data structures. A more complex situation is created by C-routines expecting output parameters in the form of simple variables, strings, arrays or C-structs. In this case the C-routine is actually storing values in the Simula memory area. If the Simula interface or the call itself is erroneous (say a text buffer or an array is shorter than expected) this might result in the C-routine overwriting essential information in the Simula memory area. As a consequence the Simula program might fail, but often much later (possibly during the next garbage collection). This is a situation well known to C programmers but this is not supposed to happen in Simula (without calls to C-routines).

External C-routines must be called with simple parameters, not involving function calls or expressions. This restriction is introduced in order to not risk a garbage collection being triggered while parameters are calculated.

It is recommended that interface specifications to C-routines are written and tested with extra care since this is information that the Simula compiler can not check and has to trust. It is often also a good idea to encapsulate the C-routine call within a Simula procedure that checks the validity of actual parameters, and maybe also provides a slightly nicer and more intuitive interface to the C-routine. Example (see 8.5 for details on 'text' parameters):

```
procedure XXX(T); text T;
begin
   external C procedure XXX is
      procedure C_XXX(T,L); name T; text T; integer L;;
   if T=/=notext then
   begin
      C_XXX(T, T.lenght);
   end;
end --- XXX ---;
```

In the C-routine interface specifications, given in more detail below. In summary, the parameter specification 'name' means that a Simula internal address is transferred to the C-routine. This might cause consistency problems for the Simula runtime system if the address is saved by the C-routine over garbage collection, or if it does not match what the C-routine expects. The specification 'value' means that a memory area is allocated i C memory space (by calling 'malloc'). This might cause 'memory leak' problems if the called C-routine does not call 'free' in order to de-allocate the data (array or string).

8.2 Simple parameters to C-routines

Simple parameters can be transmitted to a C-routine using 'call by value' or 'call by name', matching C declarations of the form 'int and 'int *' respectively.

C parameter declaration	*Simula interface specification*
int I; short s; short b;	integer I; integer s; Boolean b;
char c;	character c;
float x;	long real x;
double y;	long real y;
(float x;	real x; -- rarely, see below)

By C conventions parameters are passed on the C call-stack. Integer values are expanded to full word (32-bit) quantities for alignment. They should thus always be specified as 'integer' (or 'boolean' when applicable). Similarly floating point numbers are by most C compilers converted to 'double' ('long real') when passed as parameters. Lund Simula also support specification of 'real' parameters for use with the rare C compilers that do not follow this convention.

C procedures needing addresses of locations when called are matched by Simula interface specifications using 'call by name'. This is typically used for C-routines to return values as a result of the call. When called in the Simula program the parameter must thus be matched by an assignable Simula variable.

C parameter declaration	*Simula interface specification*
int *I;	name I; integer I;
short *s;	*** Not supported ***
char *c	*** see handling of 'text' below
float *x;	name x; real x;
double *y;	name y; long real y;

Address are passed as full word (32-bit) entries on the C call-stack, and conversion of 'real' to 'long real' does thus *not* take place in these situations. Please notice that 'call by name' expose a Simula internal address which might be valid only during the call, since the garbage collector might possibly re-arrange Simula internal memory as soon as control is returned to Simula. C procedures that stores addresses obtained as parameters are inherently unsafe to use from Simula and should be avoided or encapsulated within a C-routine with a proper interface. Also notice that there is in C no syntactic difference between the address of a single value (as above) and the address of an array of values (as discussed in 8.3 below). In Simula these specifications come out differently.

8.3 Arrays of simple values as parameters

Simula supports two ways to specify arrays as parameters to C-routines, 'call-by-name' and 'call-by-value'. Multi-dimensional array are stored in column-major order.

C routine declaration	*Simula interface specification*
int *ArrI;	value ArrI; integer array ArrI;
int *ArrI;	name ArrI; integer array ArrI;
short *ArrS;	*** not supported ***
float *ArrX;	value ArrX; integer array ArrX;
float *ArrX;	name ArrX; integer array ArrX;
double *ArrY;	value ArrY; integer array ArrY;
double *ArrY;	name ArrY; integer array ArrY;

When 'call-by-value' is specified, Simula makes a copy of the actual array provided at runtime, allocates space for it in C address space (using 'malloc'), and pass the address of the copy to the C-routine. This call method is safe since it does not hand out any address into Simula address space to the C-routine, but it might be inefficient due to the copying, and it might leak memory if the C-routine does not call 'free' to deallocate the area after use. Also, if the C-routine returns results by filling in the array with values, this specification is not meaningful. Arrays of C 'short' are not supported since Simula currently implements both 'short integer' and 'integer' as 32-bit integers (see 8.10 for a discussion on how values can be converted).

When 'call-by-name' is specified, Simula use the address of the first element in the array as the parameter to the C-routine. When using this mode please observe that the address is only valid during the call, and make sure that the length of the provided array is what the C-routine expects, so it will not distort the Simula internal data that happens to be allocated after the array.

8.4 Simple results from C-routines

Simple function results are specified as returned as integer, real or long real

C routine declaration	Simula interface specification
int	integer procedure
short	integer procedure (or boolean procedure)
char	character procedure
float	real procedure
double	long real procedure

C-functions that return addresses of C structures can be specified in Simula as 'integer procedure'. The Simula program can store such addresses and use them as parameters to future C-routine calls, but it can not access the C data structures directly (see section 8.9 for how this can be done through a set of library functions).

8.5 Text parameters to C-routines

Strings in C are often stored using the convention with a NULL character as end marker. Simula provides two alternatives for specifying how a Simula text should be transferred to a C call, call-by-value and call-by-name.

C routine declaration	Simula interface specification
char *S;	value S; text S; -- copies to C-sting
char *S;	name S; text S; -- uses Simula text

Specifying the parameter as call-by-value means that Simula allocates a string in C address space, copies the actual parameter, extends it with a NULL character and uses the result as the parameter to the C routine. It is thus safe in that it gives the C routine a C string to work with. It is assumed that the called C routine will free the string when no longer used, if not, it will 'leak

memory'. This mechanism also means that any changes or results left in the string by the called routine are lost.

Specifying the parameter as call-by-name means that Simula simply will use the address of the first byte of the text-string in the Simula address space as parameter to the C routine. In this case the Simula *caller* of the routine must make sure that the supplied parameter is properly NULL terminated. In this case there is no risk for memory leaks, but the called C-routine should not save the supplied address longer than the duration of the call. There is as always a risk that the called C routine might make severe damage to the Simula data structures, in cases such as not proper NULL termination of the string or if it is not of the length expected by the C-routine. Example of typical usage of external C routine with a text parameter:

```
begin
   external C procedure unlink is
      integer procedure Delete_file(File_name);
         name File_name;
         text File_name;;
   text Filename;
   Outtext("File to delete: "); Breakoutimage;
   Inimage;
   Filename:-Sysin.Image.Strip & "!0!";
   if Delete_file(Filename) <> 0 then
   begin  Outtext("... failed");  Outimage;  end;
end
```

The C routine unlink is documented in section 2 of the UNIX manual. A similar procedure is included in class FileUtil of simlib.

8.6 Text results from C-routines

C-routines that return NULL-terminated strings are in Simula specified as text.

C routine declaration	Simula interface specification
char *	text procedure

Simula will create a Simula text and copy the contents of the returned NULL-terminated string. The result returned to the Simula program is thus an ordinary Simula 'text'. Potentially this mechanism might leak memory since Simula does not 'free' the returned C-string (it is not obvious that this should be done in all cases).

8.7 Text arrays as parameters to C-routines

Arrays of strings is a particular data-structure expected by some C-routines. In this case the C convention is to have all the strings NULL-terminated and also the array terminated by a NULL element.

C routine declaration	Simula interface specification
char **ArrS	value ArrS; text array ArrS;

At the call Simula allocates space for the NULL-terminated array and all the strings (again NULL-terminated) in C space. It is up to the called C-routine to free these data-structures after use, or the call will leak memory. In this case call-by-name is not supported.

8.8 C-routines as parameters to C-routines

C-routines that require a routine as a parameter can be specified, but the actual parameter can only be another C-routine. The parameter *must* be specified 'call-by-name'.

C routine declaration	Simula interface specification
*Cfunc()	name Cfunc; <type> procedure Cfunc;

Example:

```
external C procedure foo is
   long real procedure foo(x,y); long real z,y;;
external C procedure Integrate is
   long real procedure Integrate(a,b,step,f);
      name f; ! a C-function must be used at call.;
      long real a,b,step;
      long real procedure f;
Use:
   long real Result;
   Result:=Integrate(1, 10, 0.001, foo);
```

8.9 C-struct as input and output parameters to C-routines

Many C-routines expect an address of a C-struct as one of its parameters. A struct is an area of memory with a specified layout. Such a structure can be emulated in Simula by using an integer array of the proper size or a text. Depending on the layout of the actual C struct one or the other might be more convenient to use. We are not using Simula class objects for this purpose since the mapping from the declaration of a class, to the actual memory layout is non-trivial.

C routine declaration	Simula interface specification
rec1 *Struct1	name rec1; integer array rec1;
rec2 *Struct2	name rec2; text rec2;

When using an array, integer elements of the struct can be accessed as normal. Elements of other types in the struct (character, 16-bit short, real, long real) can be mapped to 'integer' by the routines (such as ItoSS, ItoR, IItoL) in class UnsafeConversion in simlib.

When using the text method the elements (when not characters) can be accessed using the routines in BitpackClass in simlib.

In order the access the right element of the array, or the right byte of the text,, one has to know the exact layout of the C-struct in memory. Unfortunately the layout can change with machine, version of Unix and C-compiler.

Again it is very important to get this kind of interfaces right, since the consequences of errors are severe, as any errors in the C language. It is therefore highly recommended that interfaces to routines with 'struct' parameters are embedded in Simula procedures or classes that also do some consistency checking. If such classes (or procedures) are implemented as external Simula classes this tricky interface work needs only to be done once.

8.10 C-struct as result from a C-routine call

In most cases C-routines are written to return data by filling in a struct supplied by the caller. This is the case we have covered above in section 8.9 where thus the (emulated) struct is allocated in Simula memory. In rare cases C-routines return results in structures they have allocated themselves. They thus end up being allocated in the part of memory managed by C and only returning an address to the struct to the Simula program. The C-routine can be specified as an integer procedure as described above (section 8.4), and the remaining problem is to access the fields of the struct.

The class MemoryAccess in simlib has operations for the purpose (GetIntAt, GetRealAt, etc.) and operations in UncheckedConversion can be useful also in this situation. In order to supply the right parameters to these routines, one has to know the exact layout of the struct in memory. Unfortunately the layout can change with machine, version of Unix, and C-compiler.

9 External Library Procedures

External library procedures are using a special protocol for calling and parameter passing. The parameter passing mechanism is more general than that for C procedures and can for instance give access to the internal structure of arrays and class objects. The use of library procedures therefore demands intimate knowledge of the Simula run-time structures and is mainly intended for extended functionality in form of library routines, such as those in simlib, provided by LSH.

Syntax:

```
external library procedure <call-name> is <procedure-declaration>;
```

Examples

```
external library procedure argv  is integer procedure argv ; ;
external library procedure argc  is integer procedure argc ; ;
external library procedure envp is integer procedure envp ; ;
```

These three procedures are included in the Run-Time System. They give access to the parameters of the command line which invoked the executing Sim-

ula program. The values returned are exactly the same bit pattern as was passed by the shell to the program. argv, argc and envp are documented in the UNIX manual, see for instance execl in section 3. The external class CmdLineClass in simlib give more convenient access to the command line using the Simula style of command line switches. As an alternative, UnixCmdLineClass gives unrestricted access to the command line and also environment variables. See documentation of the library Simlib for details.

10 Restrictions and Extensions

This chapter briefly describes the restrictions and extensions that apply to all LUND Simula implementations.

Implementation Extensions
The following language extensions may cause problems when moving a Simula program to another computer.

- Calling procedures written in other programming languages.
- The compiler directive %INCLUDE.
- Accepting several classes in the same file

Implementation Restrictions
The following restrictions apply:

- A class must be defined textually before all its subclasses.
- System defined procedures may not be transmitted as parameters.
- Bytesize [1, sect 10.8]. A bytesize of 8 is always returned.
- Locking and unlocking of files is not supported under Unix.

The following Simula feature is ignored:

- The setaccess option APPEND [1, sect 10.1.1] is just ignored if present. In UNIX, APPEND means that all output goes to the end of the file, which is not what we want for Directfiles.

Compiler Capacity Limitations
The following table gives a list of the most important capacity limitations of the compiler (a few other limitations exist, but will probably never be exceeded).

- Number of array subscripts:15
- Length of prefix chain: 16
- Number of declarations in one block: 32767
- Max significant characters in an identifier: 80

11 Implementation Notes

11.1 Input-Output

This section describes aspects specific to LUND Simula under UNIX.

Unix streams (files with buffering) are used to implement Simula IO. The implementation of files are compatible with the UNIX Standard IO library (section 3S routines), but do not actually call upon any of the section 3 routines, for reasons of efficiency and correctness.

All random access files are unbuffered, in the sense that they have no buffer of their own, seen from UNIX's viewpoint. I.e. there is no buffer associated with the FILE data structure of section 3S Standard IO. Instead, the Simula image is used by directfiles.

All other files have a large buffer which is allocated by the RTS. This buffer is used for efficiency reasons, its size is BUFSIZ as defined by <stdio.h> (a common size is 4K bytes). Thus large chunks can be moved to and from disk at a time.

All terminal IO is line buffered. Files connected to a terminal have a buffer of their own in UNIX Standard IO, just like other buffered files. This buffer is however used by Simula files only when reading, and then it contains at most one line at a time. When writing, the bytes are transferred directly from the Simula image to the terminal. In other words, terminal files output everything there is to output at once.

Beware that a UNIX system behaves as follows when reading from a terminal (in cooked mode, see tty(4) and read(3) in the UNIX manual):

- At most one line, terminated by the user typing a carriage return, is transferred.
- At least one character is transferred, i.e. a program will hang until a carriage return is typed.

This implementation assumes that only infiles and outfiles are used as terminals. I.e. a terminal is never opened as a bytefile, nor as a directfile. Table 1 shows which files use which kind of buffering.

file	*Large buffer*	*line buffered*	*unbuffered*
infile	X	X	
outfile	X	X	
directfile			X
inbytefile	X	X	
outbytefile	X	X	
directbytefile			X

Table 1 *Buffering technique used for Simula files*

All legality checks (legal file name etc.) are performed by the operating system. Note that a file object is created without any of these checks; the object is not connected to a UNIX file until the call to 'open'.

All normal text files can be read from a Simula program. When reading from an imagefile, the NewLine character which terminates a line is not transferred to the Simula image.

When writing to an imagefile, each call of outimage will result in the current image contents being output with a NewLine appended as terminator. This means that an imagefile can be treated as any other text file. In the case of an outfile, the image will be stripped of trailing blanks.

Printfiles are treated in the same way as outfiles. Page skips caused by eject are output as FormFeed characters. A terminal is, however, treated specially in that FormFeeds are not transmitted. To do this would cause inconvenient screen erasures to occur on many video terminals. The standard page length, linesperpage, of a printfile is 60 lines.

A directfile is implemented with fixed record length equal to the image length plus one. After each record of a directfile a NewLine is appended, which makes it possible to use a directfile as any other text file.

At the start of execution sysin and sysout are connected to the standard input and output, with an image length of 80 characters. They may be freely closed and reopened. Also note that the image of input and output files are not protected and can be assigned to longer buffers. This is particularly convenient for sysin and sysout that is opened by the RTS with80 characters long buffers.

Input/Output and External procedures
C routines can be called from Simula and perform IO on files opened by Simula. Beware however that the Simula file image should be emptied before calling C routines, because the C routines will work on the UNIX file (and buffers), and NOT on the Simula image. Also, C programs should flush buffers before returning to Simula.

11.2 Simulation

The sequencing set of the system class simulation is implemented as a singly linked list. No separate event notices are generated, all sequencing information is kept in the process objects themselves. The event time attribute of a process object is a long real quantity.

11.3 Garbage Collection

The algorithm for memory allocation is designed to avoid creating garbage. Often a block instance can be removed from the run-time stack when it is left. Data objects such as texts and arrays cannot be removed; neither can class instances. Due to these circumstances, normal Algol blocks and objects of this type are separated in memory. Blocks dynamically enclosed by a class object are allocated among the data objects, but are also removed when they are left, if possible.

The time for a garbage collection is highly dependent on the structure of the program, and even more on the amount of data which has to be moved by the garbage collector. The runtime option '-p' can be used to see how much of overall execution time is consumed by the garbage collector. With runtime option '–g' amessage are printed after each garbage collection, giving the consumed time, and the amount of free memory available after the collection. Memory utilization can be tuned by giving the Simula program more or less memory (pool) to work in using the options '–k=' or '–m='. In general the fraction of the overall time goes down if Simula is given a large pool to work in as long as it can be kept in main memory on the computer used. Unfortunately also the time to perform one collection goes up with memory size. For interactive programs it can therefore be of interest to give a Simula program a smaller pool to work in to achieve shorter collection pauses, although more frequent. As rule of thumb, 10-15% of the overall time consumed by the garbage collection, or leaving 1/3 to 1/2 of memory free after each collection, are usually reasonable compromises.

12 Managing versions of Lund Simula

With version 4.15 of Lund Simula all the scripts mentioned in this manual (simcomp, simld, simmake, simman) are dependent on the environment variable SIMULAHOME. The value of this variable controls where these scripts look for Simula related files. If not defined the value "/usr/local/simulabin" is used as default (this is the default installation directory for Lund Simula).

SIMULAHOME affect the selection of compiler to run, versions of runtime system and other libraries distributed with Lund Simula and other processors such as simmake.

It might be meaningful to define SIMULAHOME to the directory where a new version of Lund Simula is installed in order to check it out before making it publicly available. SIMULAHOME can also be convenient to use on machines where Simula, for some reason, have not been installed in the default location.

13 On-line documentation and additional libraries

This document, as well as the SIMDEB User's Guide, are found in /usr/local/simulabin in postscript form, and can be printed as needed. simman is used to access on-line 'man' pages.

simman simula – gives an overview and pointers to other man pages

simman simcomp/simmake/simld – give details for each script.

simlib

Simlib contains a set of classes with operation that are hard to write directly in Simula. One group of classes contain functionality on the 'bit-level'. Here are operations to manipulate bits of integers, change type of a bit-pattern and access memory locations directly, given the memory address. Another group of classes interface to Unix facilities that are frequently used. Here are routines to read directories, find out the status of a file and to parse the command line.

A users manual is stored in /usr/local/simulabin/simlib in postscript form.

simman simlib – lists the classes in the library with a one-line description

simman <classname> – provide interface descriptions for classes and their operations in the library. Notice that names are spelled with all lower case characters to avoid confusion.

libsim

On-line documentation for libsim follows the pattern for simlib with a postscript manual and man-pages available through 'simman'. This library contains a set of convenient procedures that are written in Simula. The procedures fall into several groups:

- Text utilities, such as Front, Rest, FrontStrip, Search
- Edit/de-edit utilities such as CheckReal, GetRadix, PutIntAtPos, and routines for scanning Simula text.
- Some mathematical and statistical utilities.
- Routines for sorting and searching.
- Utilities for dealing with Swedish national characters.

libsim was first developed at QZ, Stockholm, for the DEC-10 Simula implementation. The parts of libsim that are available here are those that were portable. Parts that have been removed are procedures that has been accepted as part of the standard Simula library or were closely dependent on the DEC environment and might be done differently in a Unix implementation.

simioprocess

This library contains a set of classes implementing a process concept useful when dealing with external events. These are seen as 'files' in Unix, such as 'stdin', connections to other programs such the X11 window manager, or Clients and Servers. This library makes it possible to write Simula processes that work in response to events arriving from such files. On-line documentation for simioprocess follows the pattern for simlib with a postscript manual and man-pages available through 'simman'.

simsocket

This library contains classes that makes it simple to write Client/Server applications in Simula communicating over Unix sockets on TCP/IP, possibly executing on different machines. The classes work well together with the simioprocess library. On-line documentation for simsocket follows the pattern for simlib with a postscript manual and man-pages available through 'simman'.

simxlib

This is library of interface routines to the popular X11 windowing system. The many operations has been grouped into classes representing abstractions and concepts found meaningful to understand the system. There is almost a one-to-one relation between operations on the xlib level and in our interface. Simxlib is designed to work together with the simioprocess library. This library is not yet complete, and the documentation is to a large extent missing. For the details we refer to X11 documentation. Often the classes in this library match chapters in tutorials on X programming. For a person with some experience in X-programming we hope, however, it to be possible to figure out how to use this library at least for simple cases. There is a very preliminary, and not completed, postscript

manual provided. The library and the documentation will be improved in future releases.

Other libraries

There may be more libraries distributed with Lund Simula, not mentioned in this short overview. Use the 'simman simula' on-line help to find out about which additional libraries that are also installed. Each library follows the same pattern as described for simlib, or, if some of the documentation is missing, will be provided in a future release.

References

[1] Simula Standard. Data processing–Programming languages–Simula. SS 63 61 14, SIS, Box 3295, Stockholm, Sweden, 1987. ISBN 91-7162-234-9.

[2] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug and K. Nygaard: Simula begin. Petrocelli/Charter. New York 1975. ISBN 0-88405-340-7.

[3] R. J. Pooley: An introduction to programming in Simula. Blackwell Scientific Publications, Great Britain. ISBN 0-632-01611-6. ISBN 0-632-01422-9 Pbk.

[4] P. Holm: Objektorienterad Programmering i Simula (In Swedish). Lund Institute of Technology. Computer Science Department. Box 118, S-22100 Lund, Sweden.

[5] B. Kirekrud: Object-Oriented Programming with Simula. Addison-Wesley 1989. ISBN 0-201-17574-6.

[6] S. Dahl & K. Lindqvist: Objektorienterad programmering och algoritmer i Simula (in Swedish), Studentlitteratur, Lund 1993 ISBN 91-44-37591-3

[7] SIMDEB User´s Guide. Lund Software House AB.

APPENDIX A: The simcomp, simmake and simld commands

Syntax: simcomp [-option] ... file

The normal action of the Simula compiler is to read the input file and translate it into an object code file and an attribute information file. This normal action can be changed by the use of options.

All produced files have the same name as the input file, but with different extensions. The object code file is given the extension '.o'. The attribute information file is given extension '.atr'. This file contains information used by the debugger and for separate compilation. If errors are detected, they are listed at the end of the compilation. Errors are by default listed on standard error.

The filename can be givien with or without the extension '.sim'. It must be present.

Compiler Options

-L=<dir1:dir2:...> Colon separated list of directories of where to search for libraries, specified by the -I= option. Search order is: '.', dir1, dir2, etc.

SIM_LIBRARY_PATH=<dir3:dir4:..> The content of this environment variable is concatenated to the content of the -L= option. Useful for a default path to be used in a series of compilations.

-I=<lib1:lib2:...> Colon separated list of libraries of where to search for Simula attribute files and include files. The full pathname of the libraries are identified by searching the directory searchlist defined by the -L= option and the SIM_LIBRARY_PATH environment variable. The result is the library searchpath with one element for each item in the -I= parameter: '.', <dirn>/<lib1>, <dirn>/<lib2>, ...

-l Include a search of the libraries installed with the Lund Simula system. Shorthand for -L=/usr/local/simulabin -I=simlib:libsim:...

-f Report all files read during compilation. Useful to sort out binding of dependencies on external files.

-r Specifies recompilation, the compilation is aborted if the interface part of the attribute file would need to be changed.

-h Line numbers inhibited. Do not produce code to take care of line numbers during run time. This option should be used only in fully tested programs, since it makes use of several of the debugger commands impossible. The benefits resulting from use of the h option are savings in object code size and in execution time.

-list Listing. Produce a list file with the same name as the input file, but with extension '.lis' It contains the source program text, line numbers and error messages. file.lis will be produced, even if the compiler was stopped by a fatal error.

-e Error listing. Produce an error list on file with extension '.err'. This is really the same as redirecting standard error. The error list normally appearing on the terminal is written to the file with extension '.err' instead.

-x Direct error messages. Error messages will be written on the console as soon as the errors are detected. The message will give less information than that at the end of pass five. This option is mainly intended for implementor

use, but can be useful if the compiler for some reason can not complete. See Appendix D for matching error numbers with error messages.

-w Warning messages inhibited. Normally at most 64 warnings and errors will be produced, then the compilation is aborted. With the w option all warnings will be inhibited and not counted in the sum of errors and warnings. Even when warning messages are inhibited the number of warnings will be printed at the end of the compilation.

-z Symbolic intermediate code. Produce symbolic intermediate code on the file with extension '.zr', This option is mainly intended for implementor use.

The simula compiler

Syntax: simula [-option] ... program

Options:

The same options as for simula, with the following additions:

-p Print messages. The compiler will print a short message after each pass (default for simcomp)

-s Symbolic assembly code. Produce symbolic assembly code on the file with extension '.s'.

-n No object code. Produce no relocatable object code. Normally object code is produced on the file with extension '.o'. Default for compilers that can not produce object code.

The simmake command

Simmake reads Simula source-files and analyzes the dependencies between Simula files (by looking for external declarations and %include directives). This information can be used create a Makefile or, as an alterative, simmake can run simcomp to recompile all files affected by a change. Simmake understands the information in the attribute files and can do a better job than make when recompiling files after a change since make is only looking at file systems time-stamps. The possibility to create a makefile is usefull also to get an overview of dependencies in a system of separately compiled Simula classes and procedures.

Syntax: simmake -option... program...

Options:

-c Compile the programs which need compilation by calling simcomp

-c=<script> Compile programs needing compilation by using <script>

-s Print the commands to compile the programs which need compilation (using a pessimistic assumption that all compiles result in updated attribute files).

-s=<script> same as '-s', but use <script> rather than simcomp for compilation.

-m Produce a Makefile to recompile programs using simcomp.

-m=<script> same as '-m', but use <script> rather than simcomp

-L=<dir1>:<dir2>:... directory searchlist, with same meaning as for simcomp. simmake also interprets the environment variable SIM_LIBRARY_PATH in the same as simcomp does.

-I=<lib1>:<lib2>:... library search list with the same meaning as for simcomp.

-C=<lib3>:<lib4>:... library search list searched before the '-I' list. Simula files found using this path that needs compilation are compiled (while files found using the '-I' list that are assumed to be compiled up to date).
-l include a search of the libraries installed with the Lund Simula system. Shorthand for "-L=/usr/local/simulabin -I=simlib:libsim:..."

The simld command

Syntax: simld main [<-option>] [<separatemodules>]... [<libraries>]...

simld is an interface script to the linker, 'ld'. Options not recognized by simld are passed on to ld.

The name of the Simula main program must appear first.

Separatemodules are separately compiled Simula procedures or classes where the location of the corresponding '.o' files are given explicitly.

Libraries are names of binary libraries of translated Simula or C code to search for matching routines. Note that 'ld' searches these libraries one by one, left to right, so the relative order between libraries that depend on each other are important. Libraries can be explicitly named, or given with the '-l<library> options as of below. Note that the option '-L<path> in such cases becomes important since it defines the directories of where to search for libraries.

Options:

-d include debugger support with the executable
-l<library> include the file lib<file>.a for search for object code to include.
-L<directory> include <directory> for search for libraries ('.a' files).
-l include a search of the libraries installed with the Lund Simula system. Shorthand for "-L/usr/local/simulabin/lib -lsimlib -llibsim" etc.

Notice that options for simld matches these for simcomp (and simmake). Specifying -L=dir1:dir2 should be matched with -Ldir1 -Ldir2 to simld, and specifying -I=liba:libb should be matched with -lliba -llibb. All three scripts takes the short hand option '-l' for including all libraries installed with the Lund Simula system. Also definition of the environment SIM_LIBRARY_PATH, interpreted by the simcomp command, can be matched with a definition of LD_LIBRARY_PATH, interpreted by 'ld', to directories where the corresponding object libraries are kept.

The simman command

Syntax: simman [<-option>] filename

simman is an interface to the 'man' utility. It searches for man-pages in /usr/local/simulabin/man only, which makes it easy to install and update Simula

related man-pages. Possible options given to simman are passed on to man. See man man for their meaning.

Environment variable SIMULAHOME

The environment variable SIMULAHOME is understod by all the script. When not defined it is defaulted to /usr/local/simulabin. When SIMULAHOME is defined to something else the above scripts will interpret that as defining a directory where Simula is installed. This can be used to manage different versions of Lund Simula (for example checking out a new version before made publically available) or for single user systems installed in a private directory.

APPENDIX B: Run-Time Options

Syntax: program [-option] ...

Options:

-d Start execution in the Simula debugger SIMDEB (only possible if the program has been linked with the SIMDEB library, see [7]}.
-g Print a message each time the garbage collector is called.
-k=n Set the memory pool size used for dynamic allocation to n Kbytes.
-m=n Set the memory pool size used for dynamic allocation to n Mbytes.
-p Print a message indicating the size of the storage pool at the start of execution. At the end of execution cpu time as well as the number of garbage collections is printed.
-input=file Connect sysin to file rather than the standard input channel.
-output=file Connect sysout to file rather than the standard output channel.

Note that the input and output of the debugger is not affected by these two options, but is still connected to the standard input and output channels. These two options make it possible to separate debug I/O from program I/O. This is particularly useful when debugging programs designed to work with piped I/O.

To run a program, just type the name of the file containing the executable code, possibly followed by one or more run-time options.
Example:

%simulaprogram –p –k=512

APPENDIX C: Hardware specific details

This appendix provides some hardware dependent information related to the Lund Simula implementation which is valid for 32-bit computers with representation of floating-point numbers that match ANSI/EIII 754-1985 standards.

Table 2 *Sizes in bytes of simple variables on 32-bit computers*

item	*allocated*	*used*	*comment*
short integer	4	4	-2147483648 – 2147483647
integer	4	4	-2147483648 – 2147483647
real	4	4	+-3.4E+38 – +- 1.1E-38
long real	8	8	+-1.8E+308 – +-2.3E-308
character	4	1	0 – 255, acharacter variable is initially assigned the value CHAR(0).
boolean	4	1	0=false, 1=true
ref	4	4	none is represented as zero
text	16	16	plus storage for the characters, maximum length 65535.

APPENDIX D: Compiler error messages

This listing might be particular useful with compiler option '-x', which makes the compiler report errors as line and error number when found. Normally errors are listed and printed with source-line and the error text below after the three first passes. If the compiler does not complete, for example after an unsuccessful recovery after an error, the error messages are never printed. In such a situation it might be useful to re-run the compilation with option '-x'. The last reported error often gives a hint of the location of the error in the source code. In such cases the user is kindly asked to submit a trouble report as of Appendix E.

no.	Message text
1	Too many formal procedures in the program
2	The program contains too many/too long identifiers
3	The program starts badly
4	Rubbish after program end
5	Too few 'end's - extra end inserted
6	Illegal construction before begin
7	Declaration in program text
8	Too deep begin-end nesting
9	Integer constant expression out of range
10	Illegal real constant format
11	Integer constant out of range
12	Real constant out of range
13	Real constant expression out of range
14	The program contains too many/too long literals
15	Illegal label format
16	This label is already declared as an identifier:
17	Multiply defined label:
18	Formal parameter already defined:
19	
20	Illegal construction
21	Parameter transmission mode already defined:
22	Specification of non-existent formal parameter:
23	More than one name or value list
24	Too many nested inspect statements
25	'short' not followed by 'integer' - accepted as short integer
26	'long' not followed by 'real' - accepted as long real
27	
28	Proc/switch/label cannot be formal parameter to class
29	Value is illegal mode for proc/switch/label parameter:
30	Missing specification of formal parameter:
31	Multiple declaration:
32	Missing 'do' in for statement
33	The program contains too many virtual quantities
34	Missing right parenthesis - inserted
35	Too many right parentheses
36	The declaration/specification list contains a non-identifier
37	Missing comma in declaration/specification list
38	Too many nested blocks
39	Too many errors stopped the compilation
40	Too many virtual declarations in class
41	Non-numeric value as array index
42	Inconsistent number of parameters to formal array:
43	Formal parameter already specified:
44	Name is illegal mode for formal parameter to class:
45	Misplaced name/value list - accepted
46	Misplaced operator in expression
47	Non-boolean expression follows if
48	Incompatible expressions around else
49	Incompatible types of formal and actual parameters
50	Too few actual parameters

51 Too many actual parameters
52 Illegal expression type follows then in conditional expression
53 := used instead of :-
54 Incompatible types in assignment
55 Misplaced if in expression, '(' inserted
56 Expression (or subexpression) starts badly
57 Identifier or literal misplaced, probably missing ';' or operator
58 Misplaced '(', 'if', 'new' or 'this'
59 Non-arithmetic expression follows + or -
60 Non-ref types in ref assignment
61 Boolean operator operates on non-boolean expression
62 Arithmetic, boolean or relational operator misused
63 Conditional expression ends badly
64 Conditional expression starts badly
65 Expression before dot is not ref or text
66 There is no text attribute called:
67 Non-boolean expression follows while
68 Bad parameter list to block prefix
69 No else-part in conditional expression
70 Duplicated comma in declaration/specification list
71 Too many nested 'include' directives
72 Class not defined:
73 Prefix not a class:
74 Prefix not defined:
75 Too many complicated nested blocks
76 Missing ')' after parameters to class:
77 Qualifier not a class:
78 Too few indices to array:
79 Too many indices to array:
80 More than one 'inner' in class body
81 'inner' not at outermost level of class body - ignored
82 Function value not used
83 Non-reference expression precedes is, in or qua
84 'is' or 'in' is not followed by a class name
85 Class name after 'is' or 'in' is not in the same prefix chain as the object
86 Qua is not followed by a class name
87 Class name after qua is not in the same prefix chain as the object
88 'this' is not followed by an allowed class name:
89 Bad construction containing qua
90 Wrong qualification around :-
91 Wrong qualification around 'else'
92 Assign not allowed
93 Standard procedure name not followed by '('
94 Actual parameter to standard procedure not followed by ',' or ')'
95 Illegal operation on booleans
96 Illegal operation on characters
97 Incompatible types around operator
98 Illegal types around == or =/=
99 Objects around '==' or '=/=' are not in the same prefix chain
100 Not declared:
101 Too complicated expression - simplify it
102 Bad construction after '==' or '=/='
103 There is no attribute called:
104 := or :- misplaced
105 Non-compatible qualifications of formal and actual parameters
106 Inconsistent number of parameters to formal procedure:
107 Inconsistent parameter type to formal procedure:
108 Missing index to array:
109 Misplaced Simula-word
110 Bad actual parameter called by name
111 Bad parameter to formal procedure
112 Missing '(' in procedure call
113 Bad ending of procedure call
114 Missing repetition variable
115 Repetition variable not declared:
116 Too many formal (or virtual) procedures in the current block

117 This class has no parameters:
118 Too few parameters to prefixed block
119 Too many parameters to prefixed block
120 Missing ')' or ',' in array expression
121 Illegal character
122 This prefix class has no parameters:
123 Conditional statement ends badly
124 If may not follow then in conditional statement
125 Else may not follow for-statement
126 Else may not follow while-statement
127 Missing then in conditional statement
128 Expression after (re)activate is not ref(process)
129 Expression after before/after is not ref(process)
130 (re)activate in block not prefixed by simulation
131 Expression after at/delay is not of type real
132 Too many nested for/inspect statements
133 Goto non-label
134 This label cannot be reached:
135 Actual procedure type not converted:
136 Non-reference expression follows inspect
137 Identifier after 'when' is not declared
138 'when' is not followed by class name
139 More than 255 declarations (textually) in a block
140 The program contains too many blocks
141 Undefined compiler directive - ignored
142 Illegal operations on texts
143 Too many visible declarations
144 Syntax error in for statement list
145 Illegal repetition variable:
146 Missing or misplaced ':-' or ':=' in for statement
147 Missing 'until' after 'step' in for-statement
148 Variable called by name may not be used as repetition variable:
149 Ref-array parameter: wrong qualification
150 Not a class:
151 Missing parameter list to class:
152 Real operand in integer division
153 Else may not follow inspect-statement
154
155 Stand-alone class name, probably missing new:
156 'new' is not followed by an identifier
157 'when' is not followed by an identifier
158 String not closed at end-of-line, " inserted
159 Character constant not closed, ' inserted
160 Name-parameter: wrong qualification
161 Illegal directive parameter
162 Missing directive parameter
163 := and :- in the same statement
164 A statement starts badly
165 Wrong kind of actual parameter to procedure
166 Ref-procedure parameter: wrong qualification
167 Dot notation not allowed for object with class attributes
168 Statement ends badly, probably missing ';'
169 Cannot reach class declaration via dot notation
170 Procedure/class name is not identifier
171 Procedure/class name is not followed by '(' or ';' :
172 Missing ')' after formal parameter list in declaration of:
173 Transmission mode specification of non-existent formal parameter:
174 Missing do or when in inspect statement
175 Short real is treated as real
176 Long integer is treated as integer
177 Missing '(' in ref declaration/specification
178 Qualification in ref declaration is not identifier
179 Missing ')' in ref declaration
180 Declaration/specification list does not start with identifier
181 Missing semicolon
182 Erroneous array limit specifications:

183 Limits not specified for array:
184 Declarations follow specifications - treated as empty body
185 Switch name is not identifier
186 Switch name is not followed by ':=' :
187 Switch element is not of type label
188 Syntax error in switch element
189 'virtual' not followed by ':'
190 Name/value list follows virtual list
191 Hidden/protected list follows virtual list
192 More than one virtual list
193 Virtual already defined:
194 Illegal type or kind in virtual specification
195
196 Cannot assign to text expression
197 Virtual procedure declaration does not match specification:
198 'virtual' in specifications for procedure:
199 No match in this class for virtual:
200 Incompatible type/kind/qualification for virtual match:
201 Illegal delimiter after 'inner'
202 Virtual proc name is not the same as in specification:
203 _ is illegal outside identifiers
204 ISO code is > 255
205 Source program line too long
206
207 Wrong qualification for actual parameter
208 External item is not procedure or class
209 External identification is not text literal:
210 External mismatch, wrong name/type/kind/qualification:
211 Wrong internal format of attribute file for:
212 The attribute file is of the wrong version for:
213 External module is too old, recompile:
214 Unknown language in external "language" procedure:
215 External declaration of Simula procedure contains "is ...":
216 Too many declarations in the current block
217 Statement instead of expression
218 Prefix declared at wrong block level:
219 Constant may not be of type ref:
220 This constant is not yet evaluated:
221 Constant definition contains non-constant element:
222 Protected-specification of undefined attribute:
223 Protected, but virtual specification not in same class:
224 Attribute is already specified as hidden/protected:
225 Too many hidden/protected specifications
226 Hidden-specification of undefined attribute:
227 Hidden-specification of non-protected attribute:
228 Negative value raised to real value
229 Zero value raised to non-positive value
230 "is ..." must be followed by procedure specification:
231 "is <procedure-declaration>", procedure body must be empty:
232 Non-Simula procedure cannot be actual parameter:
233 Non-Simula procedure cannot be virtual match:
234 Function call in parameter to external non-Simula procedure

APPENDIX E: Error reports

We will be very grateful if you report all compiler or Run-Time System malfunctions to us. Also, please report any suggestions regarding enhancements of the Simula system or related documentation.

When submitting an error report, please follow these guidelines:

- Use the standard error report form (see the next page).
- Describe the problem as exhaustively as you can.
- Whenever possible, enclose a program listing and any other relevant listings.
- Try to isolate the problem. (Avoid submitting a 1000 lines program with the message "Does not work".)

Error reports should be mailed to:
Lund Software House., POBox 7056, S-220 07 Lund, Sweden,
or
by email to: boris@dna.lth.se.

Please make sure to include the indicated information on the form (next page).

Simula error report

User identification:
Reported by:_____
Company:_____
Phone: _____ Telefax: _____
email: _____
Address:_____
_____
_____
_____

System identification:
Compiler version (as printed with '-p'): _____
Machine and OS: _____ OS version _____

_____

Reason for report
|__Compiler error |__Run-Time System error |__ Debugger error
|__Documentation error |__Suggested enhancement
Problem description:

Enclosures: (if possible a small example program that show the problem)

_____ Mail to: _____
Lund Software House, Box 7056, S-220 07 Lund, Sweden or boris@dna.lth.se.

_____

For use by Lund Software House

Date received:_____Registration no:_____ Classification: _____
Corrective action: