The Esterel v5_91 System Manual

G. Berry and the Esterel Team

Centre de Mathématiques Appliquées
Ecole des Mines de Paris
2004 Route des Lucioles
06565 Sophia-Antipolis

INRIA
2004 Route des Lucioles
06565 Sophia-Antipolis

esterel-request@sophia.inria.fr
http://www.inria.fr/meije/esterel

June 5, 2000

Contents

Chapter 1

Introduction

This manual describes how to use the Esterel v5_91 compiler, how to interface the generated code in embedded applications, and how to simulate Esterel programs using either the graphical simulator `xes` or the textual simulator `csimul`.

The Esterel v5_91 compiler is an improvement over the former Esterel v5_21 compiler. The language has been extended by adding the `pre` signal and value operators, see the Esterel Primer [1]. Some bugs have been fixed. Automaton code generation has been entirely rebuilt. It is now fully based on constructive causality, just as all the other compiling techniques. The session recorder of the `xes` simulator has been improved. The generated code is fully compatible with that of Esterel v5_21. As the name suggests, this should be the last release of the Esterel v5 series. The next compiler Esterel v6 will handle modular compiling.

We assume basic knowledge of the Esterel language, which is presented in details in [1], and of issues such as program constructiveness, which are fully studied in [2]. The last reference also explains how Esterel programs are translated into Boolean circuits.

Chapter 2, *Getting Started*, is a quick introduction. Chapter 3, *Using the Esterel Compiler*, tells how to compile programs using the `esterel` command. Chapter 4, *The Esterel to C Interface*, explains how to embed the generated code. Chapter 5, *Building Esterel Simulators*, explains how to build a simulator in a C environment. Chapter 6, *The* `xes` *Graphical Simulator*, explains how to perform graphical simulations, while Chapter 7, *Simulation with* `csimul`, presents the stream-based interactive or batch simulator. Chapter 8, *Simulation Examples*, presents a few examples. Finally, Chapter 9, *Constructive Cyclic Programs in Esterel v5_91*, explains how the

Esterel v5_91 compiler checks program for constructiveness and generates sequential code from cyclic programs. Since there are some limitations, it is important to read this chapter if you have to deal with cyclic valued programs.

Please signal any bug or bad explanation and suggest any improvement by sending mail to `esterel-users@sophia.inria.fr`. We appreciate your feedback.

> **WARNING:** *Chapter 6,* The `xes` Graphical Simulator*, should be printed in color if possible.*

Chapter 2

Getting Started

2.1 Installing and Uninstalling the Esterel v5_91 System

2.1.1 Unix Systems

The installation procedure is described in the `README.txt` file of the Esterel v5_91 tar file. We recall it here:

1. Extract the tar file directory and place it where you want it to stay. Be careful: this directory MUST remain permanently at the same location in your machine or network, since the installation procedure only builds symbolic links to the files it contains.

2. Edit the `Makefile` file and set the definitions of the following 9 variables `ESTEREL_DISTRIB_DIR`, `ESTEREL_COMMAND`, `XES_COMMAND`, `XESTEREL_COMMAND`, `BIN_DIR`, `LIB_DIR`, `INCLUDE_DIR`, `MAN1_DIR`, and `MAN3_DIR`. They control the location where the Esterel components will be accessible. The `ESTEREL_DISTRIB_DIR` defines the installation directory location. The Esterel automatic installation procedure can build links into appropriate places. For instance, assuming that Esterel v5_91 is installed in `/opt/esterelv5_91`, one may want to install the `esterel` command in `/usr/local/bin`, by creating a link:

   ```
   /usr/local/bin/esterel -> /opt/esterelv5_91/bin/esterel
   ```

   To do this, set the `BIN_DIR` variable to `/usr/local/bin`. No link is created if the variable is left empty. The `ESTEREL_COMMAND`, `XES_COMMAND`

and XESTEREL_COMMAND variables are used to change the name of the links. Links to libraries, include files and manual entries can be created in a similar way using the LIB_DIR, INCLUDE_DIR, MAN1_DIR, and MAN3_DIR variables. Please, perform the following checks:

- Check that ESTEREL_DISTRIB_DIR is defined in the Makefile and that it defines a path known from other machines that may access it by network.
- Check that the values of BIN_DIR (resp. LIB_DIR, etc.) are different from those of ESTEREL_DISTRIB_DIR/bin, (resp. /lib). It is impossible to link a file onto itself!
- If a previous Esterel distribution exists with links, uninstall it beforehand (see below). If you want the new and old systems to coexist, you can reinstall the old one by changing the command names (e.g. changing esterel into esterelv5_21) and installing the old libraries in specific directories (e.g. libv5_21).

3. Type

```
make install
```

The Esterel automatic installation procedure sets the access rights of the installed files. If needed, check that they suit your own system management policy.

The Esterel compiler and xes simulator should be ready to run. To check that the Esterel software is installed and operational and to identify the software's version, type the commands:

```
esterel -version
xes -version
```

To uninstall the Esterel compiler installed by the Makefile, i.e. all the symbolic links created by the installation procedure above, type

```
make uninstall
```

2.1.2 Windows NT

The Esterel installation is automatically performed by the Setup.exe program. The installation does the following:

1. it creates a new group named Esterel in the Programs menu. The group gives access to:

- the Esterel GUI (graphical user interface) `xesterel`,
- the Esterel installation directory,
- the manual,
- the manual pages in HTML format,
- the Esterel Primer,
- the uninstallation program.

2. it adds the `ESTEREL` variable to the user environment and modifies the path.

Note: the compiling of the C code generated by Esterel must be compatible with Visual Studio to run simulations. The `xes` simulator uses `CL.EXE` and `LINK.EXE` from the Microsoft DevStudio 6 development environment. These tools use the `INCLUDE` and `LIB` environment variables, which have to be set correctly to make the tools work outside of DevStudio. Please refer to the file `Vc\bin\Vcvars32.bat` in the Visual C++ distribution.

2.2 Handling a Simple Example

To try Esterel compiling, enter the following program in file `foo.strl` using your favorite editor:

```
module Foo :
input I, J;
output O;
await I;
await J;
emit O
end module
```

To compile the `Foo` program into an executable C code, just type

```
esterel foo.strl
```

This command generates an executable C file `foo.c` ready to be embedded in a complete application using the interface conventions described in Chapter 4. However, this is not the simplest thing to do. Beforehand, it is recommended to simulate the program and check its correct behavior using the `xes` simulator and symbolic debugger. For this, recompile the program with the `-simul` option:

```
esterel -simul foo.strl
```

The `foo.c` generated file now contains additional code for simulation and symbolic debugging. Compile the `foo.c` file and call the `xes` command with argument the object file.

```
cc -c foo.c
xes foo.o
```

Two windows appear on the screen: a main panel that provides interaction with the simulator, and a source code panel. Click on the `tick` button to start the simulation with an empty event. The first `await` keyword turns red, indicating that the program is now waiting for the input `I`. Enter a second event with `I` present by double-clicking on the `I` button in the main panel. The first `await` turns back to blue and the second `await` turns red, indicating that the program is now waiting for `J`. Double-click on the `J` input button. The `O` output button turns red, telling that the output signal `O` is emitted. An orange warning window pops up since the module execution is terminated. Click on the `Reset` button to restart the simulation or on the `Quit` button to quit.

A richer animation is provided if the Esterel program is compiled with the `-I` interpretation option, as in

```
esterel -I -simul foo.strl
cc -c foo.c
xes foo.o
```

Then, in each reaction, the statements effectively executed appear on green background. The simulation is slightly slower, but is should be fast enough even for very large programs.

To see the generated code in a readable form, type

```
esterel -Ldebug foo.strl
```

This generates a file `foo.debug` that contains a human-readable description list of equations that encodes the generated control circuit [2].

2.3 Code Generation Options

The Esterel v5_91 compiler is able to generate software or hardware code in different ways, which we briefly present. All the software codes have the same run-time interface and they are fully interchangeable in applications.

2.3.1 Sorted Circuit Code

The default is to generate a *sorted circuit code*, i.e. a sorted sequence of Boolean equations that implements the behavior of the Esterel program. With no option, this sequence is printed out in C. This is what you got by executing the previous command "`esterel foo.strl`". This default mode only applies to *statically acyclic programs*, as described in the Esterel language primer [1]. Sorted circuit code can also be generated for *cyclic constructive programs* [1, 2] using the `-causal` option of the Esterel command, as in

```
esterel -causal prog.strl
```

For programs that do have static cycles, the `-causal` option invokes a compiling algorithm based on Binary Decision Diagrams (BDDs), which can use a large and somewhat unpredictable amount of space and time, see [7].

2.3.2 Interpreted Unsorted Circuit Code

One can also generate an *unsorted circuit code* using option `-I`:

```
esterel -I -simul foo.strl
```

The `-I` option has important advantages and drawbacks studied in Section 3.3.2. As we mentioned before, it provides better source code debugging information. When running the `xes` simulation for code compiled with option `-I`, a small C interpreter interprets the circuit equations. The statements executed in the current reaction appear on a green background in the source-code window. This visualizes the exact execution path. We recommend to use option `-I` when building a simulator with the `-simul` option.

2.3.3 Automaton Code

A third possibility for software code generation is to generate an automaton-based code. For this, use the `-A` option:

```
esterel -A [-simul] foo.strl
```

The generated `foo.c` file is fully interchangeable with the previous ones, but the circuit is replaced by an explicit finite-state machine transition table. The advantages and drawbacks of automata are described in Section 3.3.3[1].

[1]The `v3` technology available up to Esterel v5_21 is no more available. It is completely superseded by the new technology, thanks to Yannis Bres. The single-state automaton `-S` option has also been suppressed.

As for sorted circuit code, a human-readable form of the automaton is printed in file `foo.debug` when typing the following command:

```
esterel -Adebug foo.strl
```

For more information about the emitted signals, type

```
esterel -Adebug:"-emitted -names" foo.strl
```

The '`:`' symbol is used to pass a list of additional option to the final code generator.

2.3.4 Hardware Circuit Generation

Pure Esterel programs are programs that only handle pure signals, i.e. that involve no types, constants, functions, procedures, tasks, valued signals, or variables. Only simple counters are allowed in Pure Esterel, as in "`await 3 S`" or "`repeat 5 times`" or "`repeat N times`" provided that `N` is an initialized integer constant. For Pure Esterel programs, one can also generate hardware circuit netlists. For example, to build a circuit from `foo.strl`, type

```
esterel -Lblif foo.strl
```

The `-L` option defines the target language, here the `blif` circuit description format (`blif` stands for *Berkeley Logic Interchange Format*). An actual circuit can be build from the `foo.blif` file by using logic optimizers and technology mappers not detailed here. Option `-Ablif` is also available to generate circuits based on explicit automata, although this is generally not a good idea.

2.4 The xesterel GUI

If you prefer a graphical interface to a command line interface, use the `xesterel` GUI (Graphical User Interface). It can be used to generate and execute an `esterel` or `xes` command line, setting the various options by clicking on graphical objects. See the on-line documentation for more details.

2.5 Examples

Chapter 8 contains basic examples. Their Esterel and C codes are in the documentation directory of the Esterel tar file together with this manual.

The distribution tar file also contains a non-trivial complete example: a wristwatch with timekeeper, stopwatch, and alarm. Go to the `wristwatch` directory, and read the `README` file. Print the file `paper.ps` that contains a description of the program (it has been rewritten for v5_91 using `pre`, please have a look even if you know it already). Edit the Makefile appropriately, and try at least running the `tkww` fullscreen simulation and the `xww` xes-based simulation. When using `xes`, Click on whatever is red or blue in the source code windows and observe the result.

Try different code generation options by changing the `ESTEREL_FLAGS` and `ESTEREL_SIMUL_FLAGS` variable in the `Makefile`. Try in particular option `-A` for automaton code generation.

Chapter 3

Using the Esterel Compiler

This chapter presents the compiler structure in a detailed way. Its reading is useful for making the best usage of it.

3.1 The Esterel v5_91 Compiler Structure

The Esterel v5_91 compiler is made out of several internal processors controlled by a single `esterel` command. Normally, the user should only call this command and should never call directly the internal processors. However, it is important to know the names and functions of the processors. For this, it is necessary to understand how the compiler handles source and intermediate codes using simple file suffix rules.

`strl` A file `foo.strl` contains *source* Esterel code. It can contain several modules.

`ic` A file `foo.ic` contains *intermediate* Esterel code. A file `foo.ic` is obtained from a file `foo.strl` by running the `strlic` processor. Each source module produces an intermediate code module, which starts by a set of tables and continues by a set of statements written in an imperative parallel *kernel code* roughly equivalent to the kernel Esterel calculus [2] but presented in a graph form, which is more efficient for compilation purposes. The `strlic` processor type-checks the source code and generates the `ic` code only if no errors are found.

If a source module contains calls to other modules, the same calls appear in the `ic` code.

The current version of `ic` code is `ic8`.

The `iclc` processor resolves open calls and links together several `ic` modules by recursively expanding submodule calls. If all submodules are provided, one obtains a *fully linked* `ic` module that contains no submodule calls. Fully linked `ic` code is also called `lc` code for compatibility with previous versions. The `iclc` processor also performs type-checking of submodule calls.

The `ic` format can conveniently replaces source code in libraries. It can also be used when source code communication is undesirable. It is always equivalent to input `foo.strl` or `foo.ic` into the `esterel` command.

`sc` A file `foo.sc` contains *unsorted circuit code* obtained by the compiling process described in [2].

Unsorted circuit code contains the same tables as `ic` code, but kernel statements are replaced by equations of a Boolean circuit. The circuit may be combinationally cyclic, and the equations are written in no particular order. The `lcsc` processor transforms fully linked `lc` files into `sc` files. It is the heart of the compiler.

For technical reasons, the Esterel v5_91 compiler uses two versions of the `sc` code. Version `sc6` is used by default; it is the same as for Esterel v5_21. The new `scoc` automaton generator uses the newer version `sc8`.

`ssc` A file `foo.ssc` contains *sorted circuit* code. This code code contains the same tables as `ic` and `sc` codes, but the equations are sorted and printed in topological order: any Boolean variable that appears on the right-hand-side of an equation is previously defined by another equation. The `scssc` processor transforms `sc` files into `ssc` files. It accepts only acyclic `sc` circuit code. The much more elaborate `sccausal` processor performs a full constructiveness (or causality) analysis of `sc` code and generates a `ssc` code if the program is constructive [1, 2]. The analysis can be expensive if the program is cyclic.

The current version of `ssc` code is `ssc6`

`oc` A file `foo.oc` contains *automaton* code, common to the Lustre and Esterel compilers. Automaton code contains the same tables as `ic` code, but the kernel statement table are replaced by an

automaton state / transition table. The current version is `oc5`. The `scoc` processor transforms `ssc` files (version `sc8`)into `oc` files. Unlike unsorted or sorted circuit codes, automaton code can be generated only for comparatively small programs since state explosion can occur. When it can be generated, automaton code is very efficient.

`c` A file `foo.c` contains executable C code or ANSI C code. Such code can be generated in three different ways: from `sc` code, using the `scc` processor; from `ssc` code, using the `sscc` processor; from `oc` code, using the `occ` processor. These three processors share the same executable, which is actually `occ`. The `scc` and `sscc` commands are simply shell scripts that call `occ` with different options.

`debug` A file `foo.debug` contains a human-readable form of the `ssc` or `oc` codes, respectively generated by the `sscdebug` and `ocdebug` processors (`sscdebug` is a shell script that calls `ocdebug` with an appropriate option).

`blif` A file `foo.blif` contains synchronous circuit descriptions written in the `blif` format (Berkeley Logical Interchange Format). Such a file is generated from a `ssc` file by the `sscblif` processor. For Pure Esterel programs, i.e. programs that handle no data (only pure signals and counters), the circuit is behaviorally equivalent to the source Esterel program and it can be directly implemented in hardware, preferably after combinational and sequential optimizations described in other documents. For general Esterel programs and for Pure Esterel programs to be implemented in software, the `sscblif` processor must be used with option `-soft`, using the command

```
esterel -Lblif:-soft foo.strl
```

The compiler then extracts the *control part* of the source program and prints it as a synchronous `blif` circuit. This circuit can be optimized using techniques and tools described in separate documents [5, 6] and the optimized version can be used to rebuild an optimized `ssc` code using a processor called `blifssc` not presented here.

3.2 Usage of the esterel Command

The `esterel` command receives as arguments a list of options and a list of files to be compiled. Options and files can appear in any order. The files can be either source Esterel files (suffix `.strl`) or intermediate code files (suffixes `.ic`, `.lc`, `.sc`, `.ssc`, and `.oc`).

Since the general `esterel` command has numerous options, we have developed an experimental `xesterel` graphical user interface to help building an `esterel` command line. The options can either be written by hand, for example in a Makefile, or be generated by `xesterel`.

First, to be sure to run Esterel v5_91, execute the following command line:

```
esterel -version
```

Replacing `-version` by `-info` will print extended information about the current version.

3.2.1 Basic Usage

The simplest use of the `esterel` command is to compile a single file `foo.strl` with no option, by executing the following command line:

```
esterel foo.strl
```

This generates a C code file `foo.c` by calling successively the processors `strlic`, `iclc`, `lcsc`, `scssc`, and `sscc`. The intermediate code files are removed (see option `-K` below to keep them). The generated C code is suited for being embedded in applications. The interface conventions are described in Chapter 4.

To run simulations, use option `-simul`:

```
esterel -simul foo.strl
```

Then, compile the resulting C program and call the `xes` simulator

```
cc -c foo.c
xes foo.o
```

This works that simply only if the Esterel code does not refer to external objects such as types, constants, functions, procedures, or tasks. Otherwise, one must define these objects in the target language (e.g. C) and link them with `foo.o` as explained in Chapter 5.

3.2.2 Compiling Multi–Files Programs

If the source code is in several files, say `main.strl` and `aux.strl`, put both file names in the command line and tell what should be the basename of the result using option -B:

```
esterel main.strl aux.strl -B foo
```

This command line also generates a file `foo.c`. If one forgets to specify the basename, it will be `esterel` by default and the object file will be called `esterel.c`.

All the Esterel modules that are not themselves included in other modules are compiled. When including Esterel module libraries, one may need to specify which is the main module to compile as a program. This is done using the `-main` option followed by the main module name:

```
esterel -main WRISTWATCH watch.strl alarm.strl
stopwatch.strl -B wristwatch
```

3.2.3 Performing Sanity Checks

Some options perform extensive sanity checks of the Esterel source code: constructiveness, single emission of single signals, etc. Before embedding any generated code, it is useful to perform the checks as follows:

```
esterel -W -Icheck foo.strl
```

The `-W` option prints various warnings that are not printed by default. Read them carefully. The `-Icheck` option performs checks for constructiveness and performs other checks on signals. It is detailed in Section 3.5.

3.2.4 Keeping Intermediate Files

To keep the intermediate files for further use, type

```
esterel -simul -K main.strl aux.strl -B foo
```

The `ic` files will be `main.ic` and `aux.ic`. The other intermediate code files will be `foo.lc` for the linked `ic` code, `foo.sc` for the `sc` code, and `foo.ssc` for the `ssc` code.

To perform the compilation in several steps, for example with an intermediate stop at the `sc` code level, type

```
esterel -sc main.strl aux.strl -B foo
esterel foo.sc
```

The available options are `-ic`, `-lc`, `-sc`, `-ssc`, and `-oc`.

This illustrate the facts that one never needs to directly call the compiler's internal processors and that any intermediate code can be passed to the `esterel` command.

Since the Esterel v5_91 compiler internally uses two different versions of the `sc` code, there is a slight difficulty when using partial compilation for automaton generation. By default, i.e. when not using the `-A` option the version of `sc` is `sc6`. When using the `-A` option, the version is `sc8`. Therefore, using both `-A` and `-sc` will generate `sc8` code, which can later only be processed by option `-A`.

3.2.5 Printing Details about Compiling

To know which processors the `esterel` command is calling, use the `-v` verbose option:

```
esterel -v -simul main.strl aux.strl -B foo
```

Add option `-stat` if you want to know how much resources the processors use.

To know which processors the `esterel` command would call without performing the compilation, type

```
esterel -n main.strl aux.strl -B foo
```

3.3 Controlling Code Generation

There are three ways to generate code from Esterel programs: from the `ssc` sorted circuit code, which is the default, directly from the `sc` unsorted circuit code using option `-I`, which makes it possible to compile all constructively correct Esterel programs, and from the `oc` automaton code. We present them in turn.

3.3.1 Sorted Circuit Code Generation

By default, the C code is generated from the `ssc` sorted circuit code using the processor chain `strlic`, `iclc`, `lcsc`, `scssc`, and `sscc`. The `scssc` processor performs a simple topological sort of the equations, which is fast but limited. Only *statically acyclic* programs [1, 2] can be handled in this way. In such programs, there must be no *potential* cyclic instantaneous dependency between signals. When cycles are found, the `scssc` processor

prints a textual error message that is usually hard to read. The cycles (more precisely, the strongly connected components) can be visualized using the `-cycles` option:

```
esterel -cycles foo.strl
```

This command calls the `xes` symbolic debugger described in Chapter 6 to visualize the cycles.

For general programs, one must use the `-causal` option to generate sorted circuit code:

```
esterel -causal foo.strl
```

The following cyclic programs will now compile:

```
module Cyclic1:
output X, Y;
present X then emit Y end;
pause;
present Y then emit X end
end module

module Cyclic2:
input I;
output X, Y;
present I then
   present X then emit Y end
else
   present Y then emit X end
end present
end module
```

In both programs, there is a static instantaneous dependency from `X` to `Y` due to "`present X then emit Y`" and a reverse static instantaneous dependency from `Y` to `X` due to "`present Y then emit X`". In `Cyclic1`, because of the "`pause`" statement, the potential cycle is not an actual cycle since the dependencies are not active at the same instant. In `Cyclic2`, the potential cycle is not an actual cycle since the first dependency is meaningful only if `I` is present while the second dependency is meaningful only if `I` is absent. (Try `esterel -cycles` to visualize the static cycles). These facts cannot be discovered by simple equation sorting. They require the much more elaborate constructive causality analysis performed by the `sccausal` processor when option `-causal` is set. Beware: the `-causal` option can be expensive

in compiling time and space. More details on code generation from cyclic programs can be found in Chapter 9.

The `-L` target language definition option can be used to change the target language. The `L` letter is followed by a language name. For example,

```
esterel -Ldebug foo.strl
esterel -causal -Ldebug cycle.strl
```

generate a readable code using `sscdebug` instead of `sscc`. For Pure Esterel programs, one can use the `-Lblif` option to generate a `blif` hardware circuit:

```
esterel -Lblif foo.strl
```

Then, the `sscblif` processor replaces `sscc`. To extract the control of an Esterel program for optimized software code generation or verification purposes, one must pass the option `-soft` to `sscblif`. This is done as follows:

```
esterel -Lblif:-soft bigprogram.strl
```

The Esterel v5_91 compiler is open, and other code generators may be added to it. For example, typing option `-LLego` will call the `sscLego` code generator, which generates code for the Lego Minstorms^{TM} robots.

3.3.2 Unsorted Circuit Code Generation

Object C code is directly generated from the `sc` unsorted circuit code when option `-I` is used, as in

```
esterel -I -simul main.strl aux.strl -B foo
```

The processor chain is `strlic`, `iclc`, `lcsc`, and `scc`.

Unsorted circuit code has major advantages but also some drawbacks. Let us start with the advantages.

First, the `-I` option makes it possible to compile *cyclic* programs that contain instantaneous cyclic dependencies between signals, such as the `Cyclic` program above. Compiling is always very fast. However, cyclic programs may or may not have a meaning, according to the *constructive semantics* of Esterel described in [2]. With option `-I`, all Esterel programs generate C code, be them meaningful or not. At run-time, for each input event, the C code performs a fast *interpretation* of the circuit that succeeds and produces the right output and state change if and only if the program-input pair is meaningful, i.e. constructive in the sense of [2]. If the program-input pair is non-constructive, execution returns an error code that must be checked by the caller. Examples of constructive and non-constructive cyclic programs are given in [2].

Second, the `-I` option is especially useful for simulation. Running

```
esterel -I -simul foo.strl
```

and invoking the `xes` simulator as before, one can simulate any Esterel program. Constructiveness errors will provoke simulation errors and an explanation of the error will be given directly on the source code using the `xes` source code debugging interface, see Section 6.4.8. Furthermore, symbolic debugging is enhanced since the *exact control path* followed during the transition is shown on the source code using a green background. Therefore, we strongly recommend to use option `-I` when running `xes` (try it on the wristwatch example).

The drawback is of course that causality errors are *not* reported at compile-time when the `-I` option is used, which may be misleading. Fortunately, the Esterel v5_91 compiler provides a way to check that a program is constructive. Just type

```
esterel -W -Icheck foo.strl
```

This calls the aforementioned `sccausal` processor, performs the full constructiveness analysis, but does not generate code. If option `-Icheck` succeeds, the code generated with the `-I` option will never encounter a causality error and it is safe to embed it. Other properties such as single emission of single signals are also checked on the way. If errors are found during the check, they are reported graphically using the `xes` symbolic debugging interface.

> **WARNING:** *Option* `-Icheck` *should always be used before embedding code generated with option* `-I`*. Since this option also checks single emission of single signals, which is not checked by default, the option should also be used before embedding any Esterel generated code.*

3.3.3 Automaton Code Generation

A third possibility for software code generation is to generate an automaton-based code. The main advantage is speed: reactions using automata are usually faster than reactions using sorted or unsorted circuit code. The main drawback is size: in the worst case, the automaton table can be exponentially bigger than the source code, while the circuit's size is most often linear and (rare) worst-case square. Automata are usually preferred for small applications (man-machine interface drivers, communication protocols, etc). Try both automata and circuits for a given application, and choose the best. There is no general choice rule.

Automaton code generation is done when using the `-A` option:

```
esterel -A [-simul] foo.strl
```

The processor chain is `strlic`, `iclc`, `lcsc`, `scoc`, and `occ`. This process generates an automaton from any constructive Esterel program, but it can be quite expensive. (Compared to v5_21, there is no more need to use the `-causal` option in conjunction with the `-A` option when generating automaton code. The `scoc` processor independently performs full constructiveness analysis).

Inline-code can be generated instead of tables and indirect function calls for the automaton using the `-inline` option of the `occ` processor. This option generates faster, and sometimes smaller, objects for small automatons. This option is **NOT** compatible with the -simul, -sc or -ssc options. The command line becomes:

```
esterel -A:-inline foo.strl
```

A different target language can be specified as for option `-L`. If option `-Abar` is used, the back-end `ocbar` replaces `occ`. For instance, `-Adebug` generates a readable description of the automaton by calling `ocdebug` and `-Afc2` generates an automaton in the `fc2` format suited for the Esterel verification tools, calling `ocfc2`.

3.3.4 ANSI C Code Generation

ANSI C code is available for any kind of code generation. One just has to specify the `-ansi` option to the code generator commands are:

- Sorted circuit code generation:

  ```
  esterel -Lc:-ansi [-causal] [-simul] foo.strl
  ```

- Unsorted circuit code generation:

  ```
  esterel -Ic:-ansi [-simul] foo.strl
  ```

- Automaton code generation:

  ```
  esterel -Ac:-ansi [-causal] [-simul] foo.strl
  ```

3.4 Why Compilation May Fail

Compilation may fail for a variety of reasons.

- The program may have syntax errors, type-checking errors, or logically inconsistent exclusion or implication relations. The error messages are printed by `strlic`.
- The program may contain several modules calling each other in an inconsistent ways (cyclic structure, type clashes, etc.). The error messages are printed by `iclc`.
- The program may contain a statically instantaneous loop, i.e. a loop with a potentially instantaneous path from `loop` to `end loop`. The error message is printed by `lcsc`. The current compiler Esterel v5_91 can only handle *statically loop-free programs* defined in [2].
- The program may have a combinational cycle. In this case, if neither option `-I` nor option `-causal` is specified, the program is rejected by the `scssc` processor. A graphical error message can be visualized by typing

  ```
  esterel -cycles foo.strl
  ```

  Use the `-I` or `-causal` options to compile cyclic programs.

All Esterel textual error messages have the same format. They can be directly used by programming tools, such as the Emacs text editor, to point to the location of the errors (the Emacs `<CTRL>-X ‘` "*next error*" command is used to jump from one error location to another). All Esterel graphical error messages are reported using a variant of the `xes` simulator.

Internal error messages denote compiler bugs and should be reported to `esterel-bugs@sophia.inria.fr` together with the result of the command

```
esterel -info.
```

and the Esterel source code if possible.

3.5 Sanity Checks

Three options of the `esterel` commands performs sanity checks on the program. These checks should be performed before embedding a program.

- Option `-W` prints various warnings that can be useful to detect potential problems in the program.
- Option `-Icheck` performs a complete constructiveness analysis and checks for single emission of single signals. No code is generated.
- Option `-single` checks for single emission of single signals and code is generated normally.

The checks performed by the `-Icheck` and `-single` options can be expensive since they involve a state reachability analysis.

Constructiveness analysis is detailed in Chapter 9. We only study single signals here.

3.5.1 Multiple Emission of Single Signals

In Esterel, a signal that is not declared to be combined (also called multiple) should not be emitted more than once in any reaction. That fact is checked at run-time in simulation mode but not in embeddable generated C code. The `-Icheck` option check this property *at compile time* for all program state and inputs, only for programs previously found to be constructive.

The following program does not respect the single signal emission condition:

```
module SingleError :
input I0, I1, I2;
output O1: integer, O2: integer;
present I0 then
   emit O1(0);
   emit O2(0)
end present;
present I1 then
   emit O1(1)
end present;
present I2 then
   emit O2(2)
end present
end module
```

If the input signals `I0` and `I1` are both present, then the output signal `O1` is emitted twice. Similarly, if `I0` and `I2` are simultaneously present, then the signal `O2` is emitted twice. To get the error messages, type

```
esterel -Icheck SingleError.strl
```

Two counter-examples are displayed, one for each signal that violates the single emission property. Each counter-example is composed of an input sequence and of a graphical presentation of the execution path that leads to the error on the source code.

One can make this program correct by adding the relations

```
relation I0 # I1, I0 # I2;
```

to make the input signals exclusive. This is done in program `SingleOk.strl` to be found in the distribution tape. The `-Icheck` option then stops complaining.

If the program is statically cyclic, running the `-Icheck` option followed by the `-causal` option will perform twice the constructiveness check. One can save time by checking constructiveness, checking single emission of single signals, and generating code with one command line only using option `-single`:

```
esterel -causal -single constructive.strl
```

3.6 Options of the esterel command

3.6.1 Version Identification

The first options are useful to print the compiler version.

`-version` *Print the version of the compiler and auxiliary processors.*

No compiling is performed.

`-info` *Print extended information about the version compiler and auxiliary processors.*

No compiling is performed. Please join the output of the command `esterel -info` to any bug report.

3.6.2 Verbose Compilation

The next option tells the `esterel` command to report facts about how compiling goes.

`-v` *Verbose mode.*

Print the calls to the compiler processors.

`-n` *Explain what compiling will do.*

Just as `-v`, but no compiling is performed.

`-stat` *Compilation statistics.*

Print the time and memory used by the auxiliary processors.

`-size` *Print code size.*

Print information about the size of intermediate and object codes.

`-show` *Trace automata states.*

The `-show` option is meaningful only in conjunction with the `-A` automaton generation option. It displays the race between explored states and created states [3].

`-W,-w` *Print or hide compilation warnings.*

Only harmful warnings are printed by default. The `-W` option prints all compilation warnings. Use it before claiming confidence in a program. On the contrary, the `-w` option suppresses all warnings for silent compilation.

3.6.3 Code Generation Options

`-main Foo` *Choose the main module to compile.*

A set of `strl` or `ic` files can contain several modules. By default, the `iclc` linker generates code for all root modules, i.e. all modules that are not themselves called by other modules. This may cause the presence of undesired modules in the `lc` linked intermediate code, such as unused library modules. The `-main Foo` option specifies that linked intermediate code should only be generated for the `Foo` module.

`-L` *Sorted circuit code generation.*

The `-L` option specifies that the target is sorted circuit code and can also specify the target language, as in `-Lc`, `-Ldebug`, or `-Lblif`. The default target language is C, and the default option is `-Lc`. In this case, the output file has suffix `.c`. If another language is chosen, say by `-Ldebug`, then the output file has the language name for suffix, here `.debug`; The `esterel` command calls the back-end processor `sscdebug` instead of `sscc`.

`-causal` *Sorted circuit code generation from cyclic programs.*

The `-causal` option triggers full constructive causality analysis of the program using the `sccausal` processor. It is compatible with option `-L`, which can be used in conjunction to define the target language, as in `-causal -Ldebug`.

`-I` *Unsorted circuit code generation.*

The `-I` option specifies that the target is unsorted circuit code. When using this option, remember that constructive causality is not tested at compile time and that all programs generate code. Constructiveness errors are reported only at run-time. Use option `-Icheck` to perform a full constructive causality check of the program.

`-A` *Automaton code generation.*

The `-A` option specifies that the target is automaton code. The target language is changed as before: using `-Adebug` will call `ocdebug` instead of `occ` and generate a readable automaton description if a file with suffix `.debug`.

`-simul` *Simulation code generation*

The `-simul` option must be used in conjunction with any of `-L`, `-I`, or `-A` to generate code suited for simulation using `xes` or `libcsimul.a`. This works only for C code generation.

`-s` *Do not generate code*

Compiling is performed, but no output file is generated.

3.6.4 Constructiveness Analysis and Other Verifications

`-Icheck` *Check for constructiveness and perform other verifications*

Calls the `sccheck` processor, which is a variant of the `sccausal` causality analysis processor. This will check that the program is actually constructive, without generating code. Other properties are verified, such as single emission of single signals. In case of errors, the messages are displayed graphically using the `xes` symbolic debugger, see Chapter 6. We strongly recommend to use option `-Icheck` before embedding any Esterel code.

`-cycles` *Graphically display static cycles* To be used for statically cyclic programs. Executing the command line

```
esterel -cycles foo.strl
```

will pop up an error dialog box and source code windows where the strongly connected components of the program are shown using a pink background.

`-single` *Graphically display single signals emitted twice* This option checks single emission of single signals (i.e. valued signal without a combination function). Executing the command line

```
esterel -single foo.strl
```

will generate code normally unless there is a multiple emission of a single signal. In that case, the compiler pops an error dialog box and source code windows where the single signals emitted more than once are shown. Note: `-single` implies `-causal`.

3.6.5 Controlling File Names

`-B` *Define the basename of the generated files*

The `esterel` command takes as arguments a number of files suffixed by `.strl`, `.ic`, `.lc`, `.sc`, `.ssc`, or `.oc` and processes them. Normally, it directly outputs C code. The basename of the output C file is the basename of the argument file if there is only one argument, as in

```
esterel foo.strl
```

Here, the generated C file is called `foo.c`. If there is more than one argument, the default basename is `esterel`. Since this is often inconvenient, the `-B` option allows to define the basename. For example, the call

```
esterel foo.strl bar.strl -B foo
```

puts its result in file `foo.c` in the current directory.

`-D` *Define the directory in which to place the generated files*

The default directory in which the output file is generated is the current working directory. The `-D` option modifies this directory. For example, the call

```
esterel foo.strl -D /tmp
```

puts its result in file `foo.c` in `/tmp`.

3.6.6 Partial Compilation

It is sometimes convenient to stop compiling at some intermediate code level. The following options determine at which level to stop:

`-ic` *Stop the compilation after the production of the* `ic` *intermediate code files*. This option is especially useful to store intermediate code in libraries. For example, the following command puts the `ic` code of its argument files in `foo.ic`, `bar.ic` and `foobar.ic`:

```
esterel -ic foo.strl bar.strl foobar.strl
```

Notice that each `.strl` file generates a `.ic` file.

`-lc` *Stop the compilation after the production of the* `lc` *linked intermediate code file*.

This option stops the compilation after the `iclc` processor has been run. Unlike for the `-ic` command, there is only one output file whose name is determined either by the basename of the argument file if there is only one file argument or by the `-B` and `-D` options (default `./esterel.lc`). For example, the command

```
esterel -lc foo.ic
```

generates a single linked file `foo.lc`, the command

```
esterel -lc foo.strl bar.strl foobar.strl -D ../bar
```

generates a single linked file `../bar/esterel.lc`, and the command

```
esterel -lc foo.strl bar.strl -B foo
```

generates a linked file `foo.lc`.

`-sc` *Stop the compilation after the production of the* `sc` *unsorted circuit code file*.

This option stops the compilation after the `lcsc` processor has been run. There is only one output `.sc` file whose basename is determined as for the `-lc` option. The `sc` file is in format `sc6`, unless `-A` option is used, in which case it is in format `sc8`.

`-ssc` *Stop the compilation after the production of the* `ssc` *sorted circuit code file.*

This option stops the compilation after either the `scssc` processor or the `sccausal` processor has been run. There is only one output `.ssc` file whose basename is determined as for the `-lc` option.

`-oc` *Stop the compilation after the production of the* `oc` *automaton code file.*

This option stops the compilation after the `scoc` processor has been run. There is only one output `.oc` file whose basename is determined as for the `-lc` option.

Any intermediate code file can be processed further by calling the `esterel` command again. For example, to translate circuit code in `foo.sc` into sorted circuit code in `foo.ssc`, simply type

```
esterel -oc foo.sc
```

This is the way to call the `scssc` processor using only the `esterel` command.

3.6.7 Keeping Intermediate Files

By default, when compiling a set of files, the `esterel` command removes all the intermediate `ic`, `lc`, `sc`, `ssc`, and `oc` files it has created on the way. It is sometimes useful to keep the intermediate code files. Using the option `-K`, all intermediate files are kept. The option can be specialized to save only some particular format, using the forms `-Kic`, `-Klc`, `-Ksc`, `-Kssc`, and `-Koc`.

For example, the command

```
esterel -Klc -Kssc foo.strl bar.strl -B foo
```

generates `foo.c` that contains sorted circuit C code, and, in addition, keeps the linked intermediate `ic` code in file `foo.lc` and the sorted circuit code in file `foo.ssc`. The intermediate code files `foo.ic`, `bar.ic`, and `foo.sc` are deleted. The above command is equivalent to the following command sequence:

```
esterel -lc foo.strl bar.strl -B foo
esterel -ssc foo.lc
esterel foo.ssc
```

Remember that Esterel v5_91 uses two versions of the `sc` code: version `sc8` for automaton code generation, version `sc6` for the other cases. If you use intermediate `sc` files, always use consistently option `-A` when calling the `esterel` command to write and read the files.

3.6.8 Passing Options to Processors

Processor-specific options can be passed to processors using an additional ':' symbol. This is especially useful for code generators. For example,

```
esterel -Lblif:-soft foo.strl
```

calls `sscblif` with option `-soft` to extract the control circuit, and

```
esterel -Adebug:"-emitted -names" foo.strl
```

calls `ocdebug` with options `-emitted -names` for a more verbose debug printout. Notice the need for quotes when several options are passed to the back-end processor.

Chapter 4

The Esterel to C Interface

4.1 Introduction

The Esterel v5 system generates a C code file from a source Esterel program. The C code can be generated either for direct execution, which is the default, or for interactive simulation if Esterel is called with the `-simul` option.

The main object in the generated code is the *reaction function*, which inherits the name of the compiled Esterel module. Inputs and outputs are performed using auxiliary input and output functions whose names are automatically computed from the module and signal names. The run-time interface is purely procedural. No assumption is made on the operating system that supports the execution. The decisions of when an input event occurs and when signals should be considered as simultaneous are left to the user. This makes it possible to execute the generated code in arbitrary execution environments.

The run-time interface is independent of the code generation style: the sorted equation code generated by the default option or by the `-causal` option, the unsorted equation code generated by the `-I` option, and the automaton code generated by the `-A` option have exactly the same interface.

The generated C code may require some auxiliary code to define the types, constants, functions, procedures, and tasks used in the module's body. This auxiliary code is called the *data-handling code*. Some *master code* is also needed to realize the execution interface with the outside world, i.e. detect input events, call the reaction function, and perform output actions.

We start in Section 4.2 by an overview of the data handling and execution interface. In Section 4.3, we present the details of the data-handling interface. Section 4.4 is devoted to the reaction interface. Tasks are handled

in Section 4.5.

To simplify the presentation, we assume that the C code of an Esterel program `PROG` is in a file `prog.c`. This file is typically generated from `prog.strl` by the command

```
esterel prog.strl
```

4.2 Overview

If the Esterel source program refers to user-defined types, constants, functions, procedures, or tasks, the user must link the generated code with some *data-handling* code that defines the implementation of these objects.

The actual definition of the user-defined types must be known to compile the generated file `prog.c`. The type definitions must appear in a file `prog.h`, which is automatically included by `prog.c`. In addition to type definitions, the `prog.h` file can contain inline definitions of constants, functions, and procedures by `#define` directives. The constants, functions and procedures not defined in this way can appear in any other C file; we suggest to write them in a file `prog_data.c` than can be used both in execution and in simulation mode.

For actual execution, the generated code must also be linked with some *master code* that realizes the interface with the outside world (i.e. detects input events and realizes output events).

Assume that the user decides that the module `PROG` should react to an input event, composed for example of two simultaneous input signals `I1` and `I2`, where `I2` is a valued signal having as input value the value of the C expression `exp`. The user first calls two automatically generated *input C functions* `PROG_I_I1` and `PROG_I_I2`, the order between the calls being irrelevant:

```
PROG_I_I1();
PROG_I_I2(exp);
```

The user then calls the reaction function by executing the C code

```
PROG();
```

> **WARNING:** *Reactions are not reentrant and must be executed in an atomic way. During the execution of the reaction function, neither user input C functions nor the reaction function itself can be called.*

During its execution, the reaction function can call the user-supplied C *data-handling functions* that implement the functions, procedures, and tasks declared in the Esterel program. It can also read a sensor `S1` by calling the user-supplied *sensor C function* `PROG_S_S1` that should return the sensor's value.

If the Esterel program emits the output signal `O1`, the generated C code calls the user-supplied *output C function* `PROG_O_O1` with the appropriate output value as argument if the signal `O1` is valued.

To summarize, the user must write functions to read sensors, named `PROG_S_xx`, and output functions, named `PROG_O_xx`. The reaction function `PROG` and the input functions `PROG_I_xx` are defined by the Esterel compiler in the generated code. Notice that all functions related to program input, output, or execution are prefixed with the program name. On the contrary, the data-handling function names and the module names are not prefixed. Therefore, it is unwise to use a name of the predefined C functions as esterel function name (`itoa` for example) and a C reserved word as module name (`switch` or `main` for example).

4.3 C Code for Data Handling

4.3.1 Where to Define the Data-handling Objects

Assume as before that the basename used by Esterel is `prog`. Then a directive of the form

```
#include "prog.h"
```

is generated in the `prog.c` file if the Esterel input file declares a type, a constant, a function, a procedure, or a task.

The `prog.h` include file must contain the code needed to separately compile the generated `prog.c` C file. Therefore, `prog.h` must at least contain the C definition of the user-defined types used in the source program. It can also contain inline constant, function, and procedure definitions by `#define` directives.

The constants, functions, and procedures used in the Esterel source program and not `#define`d in `prog.h` are automatically declared to be `extern` in the generated file `prog.c`. They can be defined in any other C file. As said before, it is often convenient to write them in a file `prog_data.c`.

In all cases, the types, constants, functions, and procedures must be defined in C with the same name as in the Esterel.

4.3.2 Predefined Types

The basic type `integer` is implemented as `int`. The basic type `boolean` is also implemented as `int`, with constants `false = 0` and `true = 1`. The basic types `float` and `double` are respectively implemented by the C types `float` and `double`.

There are some peculiarities for the basic type `string`, since there is no real string type in C. It is implemented as follows: the type itself is declared as `char*`; a variable `VAR` of type `string` is declared as an array of characters *and is allocated* in the generated code, by the declaration

```
char __PROG_Vxx [STRLEN];
```

where `xx` is some allocation number. The constant `STRLEN` is set to 81 in file `prog.c`. You must edit this file to change the length of strings, or use compiling command of the form

```
cc -DSTRLEN=125 -c prog.c
```

String assignment is done by copy using the C `strcpy` function. If you really want variable length strings, then you must define your own user type!

4.3.3 User-defined Types

The file `prog.h` must contain a type definition for each user type declared in the source program. If a type is called `T` in the source program, it must also be called `T` in `prog.h`. Any declaration of a variable or signal of type `T` in the source program generates a C declaration of the form

```
T __PROG_Vxx;
```

Therefore, it is preferable to use the `typedef` C construct to declare the types. This is compulsory for structures. Here is an example:

```
typedef struct
{
    int hours;
    int minutes;
    int seconds;
} TIME;
```

If `T` is a user type defined in a module, the module can refer to the assignment procedure for `T` and to the Boolean equality and inequality functions for objects of type `T`. Such references are generated only if necessary.

Assignment Function

A reference to the assignment procedure is generated in the following cases:

- There is an explicit assignment or variable initialization of type `T`, i.e. the assignment symbol "`:=`" is used somewhere for a variable of type `T`.
- There is a valued signal of type `T`, either in the main module or in one of its submodules. In the case of valued signals, assignment is necessary to handle signal emissions.
- An `exec` statement calls a task with a reference parameter of type `T`.

For assignments to be correctly generated, the user must write an *assignment function* for each user type `T`. This C function will be automatically called to execute assignments. It takes two arguments, the first one of type `T*`, the second one of type `T`. Since the result of an assignment function is never used, the result type should be `void`. For example, the source Esterel assignment

```
X := exp
```

where both `X` and `exp` are of type `T` generates the following C code:

```
_T(&__PROG_Vxx, exp)
```

> **WARNING:** *Assignment must always be done by full copy or be equivalent to a full copy (or bitwise copy). Arbitrarily strange behaviors can appear otherwise.*

If C supports assignment by bitwise copy on type `T`, then one can just define `_T` in the following way:

```
#define _T(x,y) (*(x)=y)
```

The parentheses are needed to avoid priority conflicts. Notice that a source assignment `X := exp` generates the C assignment

```
(*(&__PROG_Vxx) = exp)
```

which is immediately translated back into `__PROG_Vxx = exp` by the C compiler. There is no loss of efficiency.

Equality Function

A reference to the equality function of type T is generated if and only if a comparison "=" between objects of type T appears in the Esterel code.

In this case, the user must write an equality function named _eq_T. This C function takes two arguments of type T and returns a Boolean, i.e. an int in C. For example, the definition of equality for type TIME could be:

```
int _eq_TIME (t1, t2)
   TIME t1, t2;
{
   return (   t1.hours   == t2.hours
           && t1.minutes == t2.minutes
           && t1.seconds == t2.seconds);
}
```

In the same way, if unequality is used for type T in the source program, an unequality C function named _ne_T must be defined. This C function has the same type as _eq_T; it can be defined as the negation of _eq_type, but sometimes a more clever implementation can also be used.

A reference to the unequality function of type T is generated if and only if a comparison "<>" between objects of type T appears in the module or one of its submodules.

An Example with Arrays

Let us consider an example that involves arrays. Since there is no array type declaration in C, arrays have to be encapsulated in structures. Notice the use of the assignment and inequality C functions of the underlying TIME type.

```
typedef struct {TIME array[10];} ARRAY_10_OF_TIME;
_ARRAY_10_OF_TIME (pa, a) /* assignment */
    ARRAY_10_OF_TIME *pa, a;
{
   int i;
   for (i = 0; i < 10; i++)
      _TIME(&(pa->array[i]), a.array[i]);
}
int _eq_ARRAY_10_OF_TIME (a1, a2) /* equality */
    ARRAY_10_OF_TIME a1, a2;
{
   int i;
```

```
   for (i = 0; i < 10; i++)
      if (_ne_TIME(a1.array[i], a2.array[i])) return 0;
   return 1;
}
int _ne_ARRAY_10_OF_TIME (a1, a2) /* inequality */
    ARRAY_10_OF_TIME a1, a2;
{
   int i;
   for (i = 0; i < 10; i++)
      if (_ne_TIME(a1.array[i], a2.array[i])) return 1;
   return 0;
}
```

4.3.4 Conversion To and From Strings

When the Esterel C code is generated for interactive simulation, objects of arbitrary types must be printed by the simulators and possibly entered as input signal values. For this, the user must define a string representation for each user-defined type and provide three *conversion functions*: a function that converts an object into a string for printing, a function that converts a string to an object for reading, and a function that checks the syntax of a string to be read.

The printing function for type `T` must be called `_T_to_text`. It must take a type `T` object as argument and returns a pointer to characters.

The reading function for type `T` must be called `_text_to_T` and it must return `void`. It must take as arguments a pointer to characters and a pointer to `T`. The function must convert a string into a `T` object by performing side-effects on its pointer argument, the `T` object being allocated in the generated code.

The checking function must be called `_check_TYPE`. It must take a pointer to character as argument and return an `int`. The function must returns 0 if the string is not valid, i.e. if the string is not accepted by the `_text_to_T` conversion function as a `T` object representation.

For example, if the Esterel program uses a type `TIME` declared as a C `struct` compound of three integer fields called `hours`, `minutes`, and `seconds`, the conversion functions could be defined as:

```
void _text_to_TIME (time_ptr, str)
    TIME* time_ptr;
    char* str;
{
   sscanf(str, "%d:%d:%d",
          &(time_ptr->hours),
          &(time_ptr->minutes),
          &(time_ptr->seconds));
}
char* _TIME_to_text (time)
    TIME time;
{
   static char buf[9]="";
   sprintf(buf, "%02d:%02d:%02d",
           time.hours,
           time.minutes,
           time.seconds);
   return (buf);
}
```

The above functions convert a *dd:dd:dd* string representation of time to a type `TIME` object, and conversely.

Notice that no `TIME` object is allocated by the `_text_to_TIME` conversion function; it only performs side-effects using its `time_ptr` pointer argument. Notice also that the handling of string representations is up to the user. In the above `_TIME_to_text` function, our choice is to use a static character array `buf`. This is correct since the `xes` and `csimul` simulators guarantee that the conversion function is never called twice before printing the string representation.

A possible implementation of the `_check_TIME` function is

```
int _check_TIME (string)
    char* string;
{
   int hours, minutes, seconds;
   return((  sscanf(string, "%d:%d:%d",
                    &hours, &minutes, &seconds) == 3)
           ? 1 : 0);
}
```

This is of course a partial check. One could also checks for the bounds of the numerical fields.

Remark: *It is convenient to write the string conversion functions together with the other data-handling functions. Their definitions can be enclosed in a "*`#ifdef SIMUL`*" directive, making them defined only in simulation mode.*

4.3.5 Constants

Each constant used in the Esterel program must be defined in C with the same name, unless it is initialized in the Esterel code. A constant can be defined either by a `#define` directive in `prog.h` or by a standard C variable definition. If not `#define`d, a constant is automatically declared to be `extern` in `prog.c`. It can therefore be defined in any other file. Consider the example:

```
constant NUMBER_OF_PERSONS: integer,
         LUNCH_TIME: TIME;
```

Then `prog.h` can contain

```
#define NUMBER_OF_PERSONS 45
```

and `prog_data.c` can contain:

```
TIME LUNCH_TIME = {12, 0, 0};
```

4.3.6 Functions

Each function used in the Esterel program must have a C definition. The definition can be given either by a `#define` directive in `prog.h` or by a classical C function definition. If not `#define`d, the C function is automatically declared to be `extern` in the generated file `prog.c`.

A C function definition must match the source function type declaration. For example, let us define the function declared in the `PROG` module by the following Esterel declaration:

```
function FETCH (ARRAY_10_OF_TIME, integer) : TIME;
```

A possible definition is:

```
TIME FETCH (a, i)
   ARRAY_10_OF_TIME a;
   int i;
{
   return(a.array[i]);
}
```

4.3.7 Procedures

Each procedure used in the Esterel program must have a C definition, using either a `#define` directive in `prog.h` or a standard C function definition in some other file. If not `#defined`, the C function is automatically declared to be `extern` in `prog.c`.

An Esterel procedure has two argument lists: the first one is the list of reference arguments, the second one the list of value arguments. In C, the two lists are concatenated into a single list; the reference arguments are passed by pointers and the value arguments are passed by value. For example, consider the Esterel procedure declaration:

```
procedure PROC (T1) (T2);
```

The corresponding C function `PROC` has two arguments, the first one of type pointer to `T1`, the second one of type `T2`. Therefore, the correct C declaration is:

```
void PROC (pt1, t2)
   T1 *pt1;
   T2 t2;
{
   ...
}
```

Here is another example:

```
procedure STORE (ARRAY_10_OF_TIME) (integer, TIME);
```

A correct C implementation is:

```
void STORE (pa, i, t)
   ARRAY_10_OF_TIME *pa;
   int i;
   TIME t;
{
   _TIME(&(pa->array[i]), t);
}
```

Notice the use of the assignment function `_TIME` to perform the time update.

There is an exception for strings, which are already pointers. No additional pointer is generated for strings appearing in the reference argument list. For example:

```
procedure STORE_CHAR (string) (integer, CHAR);
```

can be implemented by

```
void STORE_CHAR (s, i, c)
   char *s;
   int   i;
   CHAR  c;
{
   s[i] = c;
}
```

where `CHAR` is a user type implemented by `char`. The declaration "`char *s`" is used instead of "`string *s`" as would be the case for user types.

4.4 The Reaction Interface

The reaction function provided by the Esterel compiler for a program `PROG` is also called `PROG`. In addition, for input and output, the generated code provides a function for each input signal, and the user must provide an output function per output signal and a sensor function per sensor.

4.4.1 Input Signals

For each input signal `IS`, the Esterel compiler generates an *input C function* called `PROG_I_IS`, which takes an argument of the appropriate type if the signal `IS` conveys a value. For example, from the Esterel declarations

```
input WATCH_MODE_COMMAND;
input WATCH_TIME (WATCH_TIME_TYPE);
return R;
```

appearing in a module named `DISPLAY`, the compiler generates the following functions:

```
void DISPLAY_I_WATCH_MODE_COMMAND () {...}
void DISPLAY_I_WATCH_TIME (__V) WATCH_TIME_TYPE __V; {...}
void DISPLAY_I_R () {...}
```

When a program `PROG` should react to an input event composed of one or more simultaneous input signals, the associated input C function(s) should be called before calling the main execution function `PROG`.

If several input functions are called before calling `PROG`, the corresponding input signals are considered as forming the current input event of the reaction. The input signals are considered as being simultaneous. Therefore, the notion of "simultaneous signals" is a purely logical one at the C

level. Two signals are considered as simultaneous at that level as long as their input C functions are both called before calling the reaction function. Which signals are to be considered as simultaneous and when to call the automaton is entirely left to the user.

For single signals, if the same input function is called twice, only the last call matters. For combined signals, the values of successive calls to the input function are combined using the signal's combination function.

> **WARNING:** *In the C code, the ordering between the input function calls forming an input event is irrelevant.*

For example, assume that some signals have arrived from the external world, say a pure signal `IS1` and an integer-valued signal `IS2` conveying the integer value 3. To perform the corresponding program reaction, one must first call the two automatically generated input functions `PROG_I_IS1` and `PROG_I_IS2` and then call the C function `PROG`. One can execute the following sequence:

```
PROG_I_IS1 ();
PROG_I_IS2 (3);
PROG ();
```

Assume now that `IS` is a combined integer signal with addition as combination function and that the following sequence is executed

```
PROG_I_IS (1);
PROG_I_IS (2);
PROG ();
```

then the current value of `IS` is $1 + 2 = 3$ just before the call to the reaction function. This sequence is exactly equivalent to the following one:

```
PROG_I_IS (3);
PROG ();
```

4.4.2 Return Signals

Return signals are particular input signals used to signal the completion of external tasks, see [1]. In the C generated code, return signals are handled exactly as standard input signals.

4.4.3 Output Signals

For each output signal `OS`, the user must write a void *output C function* `PROG_O_OS` that takes an argument of the appropriate type if the signal `OS` conveys a value. This function is automatically called by the reaction function if the signal is emitted.

> **WARNING:** *The order of the output function calls performed by the reaction function is arbitrary and unspecified*[1].

Assume that a reaction causes the output of a pure signal `OS1` and of an integer signal `OS2` with value 4. Then `PROG` calls the user-defined C functions `PROG_O_OS1` and `PROG_O_OS2` with the appropriate arguments; the following calls will be executed (in arbitrary order) in the body of `PROG`:

```
PROG_O_OS1 ();
PROG_O_OS2 (4);
```

The C functions `PROG_O_OS1` and `PROG_O_OS2` must do whatever is necessary to communicate with the actual environment.

4.4.4 Inputoutput Signals

For an inputoutput signal `IOS`, an input C function `PROG_I_IOS` is automatically generated as for an input signal, and the user must write an output C function `PROG_O_IOS` as for an output signal.

An inputoutput signal `IOS` received by the reactive program behaves as if it was internally emitted by it, and is therefore re-emitted outside whenever received in a reaction:

- if `IOS` is a pure inputoutput signal, then `PROG_O_IOS` is called if `IOS` is received or emitted by the program. Therefore, `PROG_O_IOS` is always called by the reaction function if `PROG_I_IOS` was called before.
- If `IOS` is a single signal, then there are two cases:
  - If `IOS` is received by the program, it cannot be emitted by it in the same reaction, which is detected by the Esterel compiler when option `-Icheck` is set. The function `PROG_O_IOS` is called with argument the value received by `PROG_I_IOS`.

[1] According to the Esterel semantics, this ordering *cannot* be specified

  - If IOS is not received, then PROG_O_IOS is called if and only if IOS is emitted by the program; the argument of PROG_O_IOS is the emitted value.

- If IOS is a combined signal, then all the emitted or received values are combined using the signal's combination function; the output function PROG_O_IOS is called if and only if IOS is received or emitted, with argument the combined value.

4.4.5 Sensors

A program reaction can access the current values of sensors. Let SE be a sensor of type T. If the program needs the current value of SE to perform its reaction, it calls an argumentless user-supplied *sensor C function* PROG_S_SE. This function must return a value of type T, which is the sensor current value. To ensure sensor value consistency, the program calls each sensor C function at most once in a reaction. Here is an example:

```
int PROG_S_TEMPERATURE () {...}
```

4.4.6 Reaction and Reset

For the Esterel main module PROG, the Esterel compiler generates a void argumentless C reaction function PROG and a void argumentless reset function PROG_reset that resets the program by performing the following actions:

- resetting the program to its initial state.
- resetting the valued interface signals for which an initial value was provided in Esterel source code.

The reset function should be called before any reaction is performed if there are initialized interface signals in PROG. To perform a program reaction, one calls the input C functions and then call the reaction function, as in:

```
PROG_I_IS1 ();
PROG_I_IS2 (3);
PROG ();
```

Programs often contain instantaneous initial statements, such as signal emissions or variable initializations, to be performed during the first reaction. To perform them, it is often useful (but not mandatory) to generate a "blank" initial event by calling the reaction function before calling any input C function. (This "boot transition" is different from the automaton reset).

4.4.7 Warnings and Advises

- Since assignments of initial values to valued interface signals are performed by the reset function, it is highly mandatory to *always call the reset function before initialization* if there are initialized signals.

- The relations between input signals specified in the source program are *not* checked when the automaton is called in direct embedded execution. The code may behave strangely if called with inputs which do not satisfy the relations. However, the simulation interface does enforce relations.

- The combination functions associated with combined signals must be commutative and associative. Otherwise, the results of signal combinations can be arbitrary since the combination order depends on the action schedule chosen by the compiler.

- The automaton code is *not reentrant* and its execution must be atomic. Therefore, during a call to the automaton, it is forbidden to call it again and to call input C functions. Arbitrarily strange behaviors can arise otherwise. In particular, *interrupt handling routines should never call directly input C functions or the automaton.* They should instead fill event queues to be read when the automaton call terminates. One can also use interruption masking during the automaton execution.

- Access to uninitialized variables and uninitialized signal values are *not* checked when the reaction function is called in direct execution. But the simulation engine does enforce this check.

4.5 Task Handling

We now describe the interface of the C code generated by the Esterel compiler for an `exec` statement. This code reflects concretely the way tasks are handled abstractly in Esterel. It is organized in two layers. The low-level layer is a direct interface to run-time C data structures that contain all the required information about the status of `exec` statements. The optional higher-level layer provides the user with a *functional interface*.

Our design might look heavy at first glance, but it intends to provide the user with maximal flexibility with respect to actual task handling. The functional interface is reasonably simple, but not fully general since other ways of interfacing `exec` statements can be thought of, in particular as

far as task suspension is concerned. The low-level interface is meant to be convenient for any user who wants to design its own fine-grain task handling.

Notice that we do not provide the user with an actual asynchronous task handling interface, because this is highly dependent on particular operating systems.

4.5.1 Low-level Layer: the ExecStatus Interface

We assume that the main module is called `PROG`. The following C function returns the number of `exec` statements in the compiled program:

```
int PROG_number_of_execs ();
```

The following C function returns the number of `exec` statements associated with a task of name `TASK` in the compiled program:

```
int PROG_number_of_execs_of_TASK ();
```

The ExecStatus Structure

Each `exec` statement, which is uniquely identified by its return signal, is associated with a C structure of type `__ExecStatus` that contains all relevant information about the `exec` status just after a reaction. This structure can be recovered in three ways:

by name: for each `exec` of return signal `R`, the generated C code contains a variable `PROG_exec_status_R` declared by:

```
__ExecStatus PROG_exec_status_R = ... ;
```

by absolute number: the generated code declares an array of pointers to the `__ExecStatus` variables, of size `PROG_number_of_execs()`, which has one entry for each `exec` statement:

```
__ExecStatus *PROG_exec_status_array [] = ... ;
```

by relative number: for each task `TASK`, the generated code contains an array of pointers to the `__ExecStatus` variables, with one entry for each `exec` of that task. The size of the array is given by the function `PROG_number_of_execs_of_TASK()`:

```
__ExecStatus *PROG_exec_status_array_of_TASK[] =
... ;
```

Here is the definition of the __ExecStatus structure:

```
typedef struct
{
    unsigned int start          : 1 ;
    unsigned int kill           : 1 ;
    unsigned int active         : 1 ;
    unsigned int suspended      : 1 ;
    unsigned int prev_active    : 1 ;
    unsigned int prev_suspended : 1 ;
    unsigned int exec_index;
    unsigned int task_exec_index;
    void (*pStart)(); /* takes a function as argument */
    int (*pRet)();    /* may take a value as argument */
} __ExecStatus;
```

The meaning of these fields is as follows:

`start` has value 1 if and only if the `exec` statement starts and is not immediately killed. In that case, a new instance of the task code should be started in the current instant. See below for how to recover the actual parameter values using the `pStart` field.

`kill` has value 1 if and only if the `exec` statement is killed in the current instant. Then, the currently running instance of the task should be killed; notice that `kill` can only be 1 if there is such a running instance.

`active` has value 1 if and only if the `exec` statement is active in the current instant; this means that the `exec` is started in the current instant or has been started before, has not yet been killed, and that the task code has not yet returned.

`suspended` has value 1 if and only if the `exec` statement is active and suspended in the current instant by an enclosing `suspend` statement.

`prev_active` has value 1 if the `exec` was already active in the previous Esterel instant.

`prev_suspended` has value 1 if the `exec` was suspended in the previous instant.

`exec_index` an integer index identifying uniquely the `exec` statement. This index ranges between 0 and $n-1$ if the Esterel program contains n `exec` statements after full submodule instantiation.

`task_exec_index` an integer index identifying uniquely the `exec` statement among those referring to the same task. This index ranges between 0 and $p-1$ if the Esterel program contains p `exec` statements for this task after full submodule instantiation.

`pStart` an auxiliary function pointer to be used at start time, i.e. when `start` is 1. See details below.

`pRet` a pointer to the return function `PROG_I_R` associated with the return signal, if the name of the main module is `PROG` and the name of the return signal is `R` (remember that a return signal is just a particular input signal). See details below.

The function pointed to by `pStart` takes a user-provided function as argument, and the reference and value arguments are passed to this user function with the same convention as for a procedure (reference arguments as pointers, value arguments as values). A typical use is

```
if (exec_status.start)
      (*exec_status.pStart) (my_start);
```

This will call the user-provided function `my_start` with arguments the arguments of the task at start time.

The user-provided function `my_start` should perform two actions: effectively starting the task in the environment, and saving the pointers to the reference arguments for their update at return time, see below.

Calling the `(*pRet)` or `PROG_I_R` function in the master code amounts to emitting `R`, hence to signal to Esterel that the task is completed. The return function takes a value if and only if the return signal carries a value; then the value passed becomes that of the return signal. The return function can be called either directly using its full name `PROG_I_R` or indirectly through the `pRet` pointer.

When the return function is called, the locations pointed by the pointers passed at start time for reference arguments are supposed to contain the values updated by the task.

Notice that there are redundancies between the fields of __ExecStatus. For example, prev_active and prev_suspended could be computed directly by the user. However, we chose to include these informations since they are very easy to compute from within Esterel and very handy for the user.

Reincarnation of exec Statements

Notice that an exec statement can be killed and restarted in the same instant, for example by executing the following:

```
loop
   exec T(...)(...) return R
each I
```

In this case, when I occurs, there may be two active occurrences of the task code that the user has to manage properly. The first one is the one being killed, the second one is the one being started. There can be no more that these two occurrences.

Handling Reference Arguments

Let us give more details on the handling of reference arguments. Consider an Esterel variable X implemented as a C variable of location X, and assume that X is passed by reference in an exec statement.

At starting time, the contents of X are copied into another location L whose address is passed to the user starting function my_start. During task execution, the user may freely modify the contents of L. At return time, i.e. when PROG_I_R or equivalently (*pRet) and then PROG are called, the contents of L are automatically copied back to location X.

This copy-restore mechanism is made necessary by the possibility of killing exec statements: if reference arguments could be modified in place at location X before an exec gets killed, the value of X would change in the Esterel program, which is forbidden by the Esterel semantics.

Update of reference arguments must be performed in place in the location L passed to the user starting function my_start; this is why these pointers should be saved by my_start. Actual update of X by L is triggered only when the automaton PROG is called with return signal R present (and of course only if the exec statement is not killed by an enclosing abortion statement).

4.5.2 The Functional Interface to Tasks

We now describe the much simpler functional interface. The user should provide four C functions:

- A *user start* function to start the task. This function receives the reference and value parameters plus a pointer to the `__ExecStatus` record of the `exec` statement as the last parameter; this is useful to index process-id tables associated with asynchronously running operating systems tasks, using the exec index fields.
- A *kill function* that is called when a task is killed, with a pointer to the `__ExecStatus` structure as argument.
- A *suspend function* that is called when the task becomes suspended, i.e. is now suspended but was not suspended in the previous instant (`suspended=1`, `prev_suspended=0`). This function also receives a pointer to the `__ExecStatus` structure as argument.
- A *resume function* that is called when the task should resume, i.e. when it was suspended at previous instance and it is neither suspended nor killed in the current instant. This function also receives a pointer to the `__ExecStatus` structure as argument.

To use the functional interface, one simply has to write a call to a specific `STD_EXEC` library macro with arguments the return signal name and the user functions, this for each `exec` and right after each call to the automaton:

```
#include "exec_status.h"
my_start () { ... }
my_kill () { ... }
my_suspend () { ... }
my_resume () { ... }
...
PROG();  /* perform a transition */
STD_EXEC (R1, PROG,
          my_start_1, my_kill_1,
          my_suspend_1, my_resume_1);
STD_EXEC (R2, PROG,
          my_start_2, my_kill_2,
          my_suspend_2, my_resume_2);
```

A special `__DUMMY__` function can be used if a user function is not necessary, for instance if there is no `suspend` statement in the Esterel program:

```
STD_EXEC (R2, PROG,
          my_start_2, my_kill_2, __DUMMY__, __DUMMY__);
```

Finally, one can also write

```
STD_EXEC_FOR_TASK (TASK, PROG,
                   my_start, my_kill,
                   my_suspend, my_resume);
```

This calls `STD_EXEC` for all return signals of task `TASK`.

To get familiarity with the functional and low-level `exec` generated code interfaces, we recommend reading `exec_status.h`.

4.6 The sametype Utility

In the context of a simulation, it may be convenient to define a user-type either as a predefined type or as an already defined user-type. For that purpose, a tool named `sametype` and a C header file `csimul.h` are provided in the Esterel distribution tape. The `sametype` tool is used to define a new type as a synonym of a previously defined one; it automatically generates all the C definitions required by the simulator. The `csimul.h` header file contains all the predefined type definitions; it must be included by C files using Esterel predefined types. These tools allow a quick implementation of a C simulator to ensure that the final program works well.

The executable file `sametype` must be called with the following syntax:

```
sametype NEW_TYPE OLD_TYPE
```

where `NEW_TYPE` and `OLD_TYPE` respectively denote the user-type to be defined and the target type used as actual definition. For example, if type `TEMPERATURE` is to be defined as a synonym for the predefined type `integer`, all necessary C definitions for type `TEMPERATURE` will be automatically deduced from those of `integer`, using the shell command

```
sametype TEMPERATURE integer
```

This command generates on the standard output stream the C definition of `NEW_TYPE` and the C definitions of all the functions associated to this type and required for the simulation: the assignment function, the equality and inequality test functions and the string conversion functions.

If the Esterel program uses no external object (type, constant, function, or procedure), the output of the `sametype` command can be added to the header file `prog.h`,

```
sametype  TEMPERATURE integer >> prog.h
```

The definitions generated by `sametype` use the definitions of `OLD_TYPE` and of its associated functions. If `OLD_TYPE` is a user-defined type, its definition must be provided in the file `prog.h` before the definition of `NEW_TYPE`, otherwise `OLD_TYPE` must be one of the following types: `integer`, `boolean`, `float`, `double`, or `symbolic`. The type `symbolic` must be used instead of `string`. The C definitions of the types `integer`, `boolean`, and `symbolic` are provided in the header file `csimul.h` that must be included in the file `prog.h`. Provided that `csimul.h` has been correctly installed, it should be included by using the following directive:

```
#include <csimul.h>
```

The functions associated to the types `integer`, `boolean`, `float`, `double`, and `symbolic` are provided in the simulation toplevels `csimul` and `xes`.

If the Esterel program declares constants of type `NEW_TYPE`, the actual C definitions for these constants can be provided in the file `prog.h` after the definitions generated by `sametype`.

As a summary, in the simple case where no procedure or function is declared in the Esterel program, the file `prog.h` has the following contents (assuming `OLD_TYPE` is not a user-type):

prog.h

```
#include <csimul.h>
/*         Definitions associated to NEW_TYPE
 *         and generated by sametype  */
typedef OLD_TYPE NEW_TYPE;
_NEW_TYPE(x,y) NEW_TYPE* x; NEW_TYPE y; {_OLD_TYPE(x,y);}
boolean _eq_NEW_TYPE(x,y) NEW_TYPE x; NEW_TYPE y; { ...
boolean _ne_NEW_TYPE(x,y) NEW_TYPE x; NEW_TYPE y; { ...
NEW_TYPE _cond_NEW_TYPE(x,y,z) boolean x; NEW_TYPE y; ...
char* _NEW_TYPE_to_text(x) NEW_TYPE x; { ...
boolean _check_NEW_TYPE(x) char* x; { ...
_text_to_NEW_TYPE(x,y) NEW_TYPE* x; char* y; { ...
/* Other type and constant definitions */
...
```

If the Esterel program declares procedures or functions, the definition of `NEW_TYPE` generated by `sametype` should be added in the header file `prog.h` while the definitions of the functions generated by `sametype` should be added in a C file, `prog_data.c`, that also includes the header `prog.h` (using a `#include` directive). For that purpose, the following commands can be entered:

```
sametype NEW_TYPE OLD_TYPE | head -1 >> prog.h
smetype NEW_TYPE OLD_TYPE | tail -7 >> prog_data.c
```

Then the C definitions of the Esterel procedures and functions must be added in the file `prog_data.c`.

Then, the files `prog.h` and `prog_data.c` have the following contents:

`prog.h`

```
#include <csimul.h>

/* Definition of NEW_TYPE generated by sametype */
typedef OLD_TYPE NEW_TYPE;

/* Other type and constant definitions */
 ...
```

`prog_data.c`

```
#include "prog.h"

/* Functions associated to NEW_TYPE and generated
 * by sametype */
_NEW_TYPE(x,y) ...
boolean _eq_NEW_TYPE(x,y) ...
boolean _ne_NEW_TYPE(x,y) ...
NEW_TYPE _cond_NEW_TYPE(x,y,z) ...
char* _NEW_TYPE_to_text(x) ...
boolean _check_NEW_TYPE(x) ...
_text_to_NEW_TYPE(x,y) ...

/* Other hand-written functions associated to Esterel
   procedures and functions */
...
```

WARNING: *The type* `string` *cannot be used as* `OLD_TYPE` *for variable allocation problems. It is conveniently replaced by the type* `symbolic` *which behaves like the string type. The type* `symbolic` *is useful for Esterel user-types which do not have any real contents like protocol messages.*

Chapter 5

Building Esterel Simulators

5.1 Introduction

The Esterel v5 system contains two tools for interactive or batch simulation of Esterel programs: the `xes` graphical simulator, which includes source-code debugging facilities, and the `csimul` stream-based interactive or batch simulator.

The `xes` simulator requires a graphical workstation (X-window based for Unix versions, or PC NT). Events are entered by clicking on buttons, and additional debugging information is displayed directly on source code.

The `csimul` simulation toplevel provides the user with a purely tty or file stream-based interface. In interactive mode, the user repeatedly enters an input event and the simulator replies by printing the associated output event and possibly additional informations about the internal state, signal and variable values, and external tasks current state. In batch mode, events and commands are read from files.

Using its session recorder/player, the `xes` simulator can also write and read simulation files in the same format as `csimul`.

This chapter explains how to build Esterel simulators. Chapter 6 explains how to use the `xes` simulator, while Chapter 7 is devoted to the `csimul` simulator

For simplicity, we assume that the Esterel program is in a single file `prog.strl`.

5.2 Building a Simulator

For both tools, the simulated code is the C code obtained by compiling the Esterel program with the `-simul` option:

```
esterel -simul prog.strl
```

The simulation code can be generated with any of the code generation options of the `esterel` command. To have full source code debugging in `xes`, we advise to use the `-I` option:

```
esterel -I -simul prog.strl
```

However, constructiveness will not be checked beforehand since it is only checked in each reaction, see Section 3.2.3. Use `esterel -Icheck` to check it beforehand.

Compared to the normal embeddable generated code, the simulation code in `prog.c` has handles to be used by simulator toplevels to pass events and to retrieve debugging information such as values of signals and variables. This code must be linked to a simulation toplevel library and possibly to data-handling code written as explained in Chapter 4. Executable simulation files are called *simulators*.

> **WARNING:** *Only C code generated with the* `-simul` *option can be used for Esterel simulation. The linking phase will fail otherwise.*

5.3 Simulating Simple Programs

A source Esterel program that involves no user-defined types, constant, functions, or procedures can be simulated right away without writing any other C code. This is true even if the Esterel program executes external tasks, since task execution is interactively simulated under user control. The simplest way to run a simulation is to compile the C file and to call the `xes` command:

```
cc -c prog.c
xes prog.o
```

The graphical simulation process proper is described in Chapter 6.

To build a `csimul` simulator, link the generated C file with the `libcsimul.a` simulation library to be found in the Esterel tape:

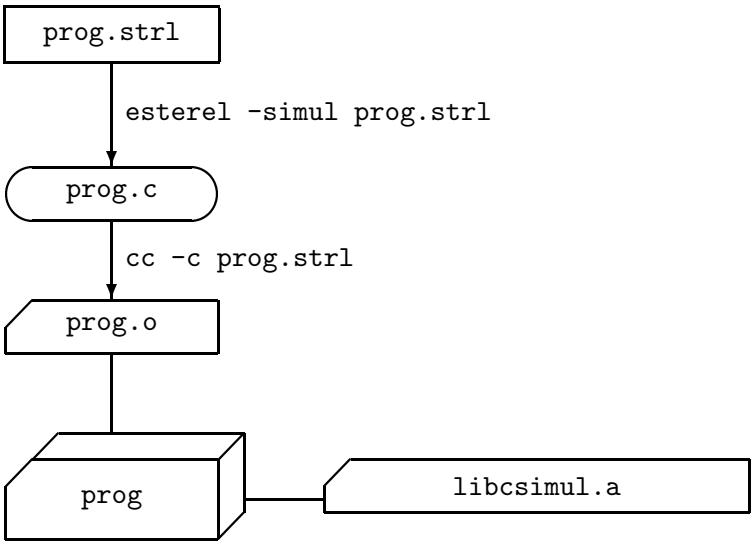


Figure 5.1: Building a simple Esterel simulator

```
cc -o prog prog.o -lcsimul
```

provided that the library is in your library load path. The library file can also be passed explicitly:

```
cc -o prog prog.o /usr/local/lib/esterel/libcsimul.a
```

Figure 5.1 summarizes the building of a simulator from the `prog.strl` file using the library. Only `prog.strl` is written by the user. Arrows denote compilations and lines denote the UNIX loader `ld`.

5.4 Simulating Programs with User-Defined Data

For programs that involve user-defined data objects, the data-handling C code described in Chapter 4 must be provided by the user. If the program involves user-defined types, these types must be defined in a `prog.h` file. The data operation must be defined in one or more C files. In particular, the conversion functions from types to string must be provided to print values, and the reverse conversion and checking functions from string to types must be provided for types for which values can be entered as values of input signals or sensors. These conversions are described in Section 4.3.4. Here

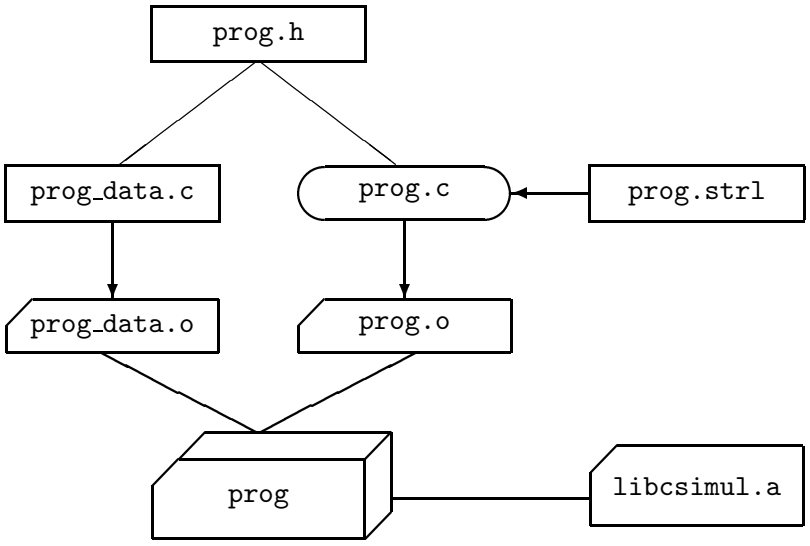


Figure 5.2: Building an esterel simulator using external data types

we assume that all data operations are in `prog_data.c`. The `xes` simulator is started as follows:

```
cc -c prog.c
cc -c prog_data.c
xes prog.o prog_data.o
```

To avoid typing a long `xes` command, one can save the graphical simulator in a fast-loadable file and feed this file later on into `xes` to run the simulation:

```
xes -o xprog prog.o prog_data.o
xes xprog.exe
```

Notice the `exe` suffix, necessary for compatibility between Windows NT and Unix.

The `csimul` simulator is built as follows:

```
cc -c prog.c
cc -c prog_data.c
cc -o prog prog.o prog_data.o -lcsimul
```

provided that the library is in your library load path. The library files can also be passed explicitly:

```
cc -o prog prog.c prog_data.c \
   /usr/local/lib/esterel/libcsimul.a
```

Figure 5.2 illustrate this case. As in Figure 5.1, only files of which the names are enclosed in rectangular boxes are written by the user. In our example, there are three such files: the Esterel source file `prog.strl`, the C header file `prog.h` that contains the type definitions and the C file `prog_data.c` that contains the data-handling code. Thin lines illustrate C `#include` directives. Thick arrows denote compilations and thick lines denote the UNIX loader.

5.5 Multi-Module Files

The generated code file can contain several Esterel root modules, i.e. modules that are not instantiated in other modules. At present, only the `csimul` simulation toplevel is able to operate on all the root modules contained in the C code file. The `xes` simulation toplevel only operates on the first root module found in the code file. Use option `-main` of `esterel` to guarantee that there is only one compiled module, see Section 3.6.3.

Chapter 6

The xes Graphical Simulator

The `xes` graphical simulator is the main tool to exercise and debug Esterel programs. It offers source code animation and displays a lot of information on the source code using colors. It makes it possible to record simulation sessions, and to set breakpoints. However, it works in read-only mode. It is impossible to dynamically change the value of an object without using the reactive interface, unlike in conventional C code debuggers. This would indeed be incompatible with the Esterel semantics and would be very uncontrollable since Esterel is a concurrent language.

6.1 Starting an xes Simulation

Let us consider the `WATCH` example explained later in Section 8.3 (in this section, there is no need to understand what the watch is doing). The watch is made of the `watch.strl` file containing the Esterel source code, the `watch.h` file that defines the user-type `TIME`, and the `watch_data.c` file that contains the data-handling C code. As explained in Chapter 5, the simulator is started by executing the following shell commands:

```
esterel -I -simul watch.strl
cc -c watch.c watch_data.c
xes watch.o watch_data.o
```

The `xes` simulator starts by displaying two windows, as shown on Figure 6.1. The *main panel* is used to drive the simulation. It is always present on the screen. The main panel consists of several parts, one for each of the Esterel interface objects: input signals, sensors, output signals, and tasks. The input and output parts in turn split into a pure subpart and a valued subpart (not all Esterel programs have all the interface object kinds; therefore, the main

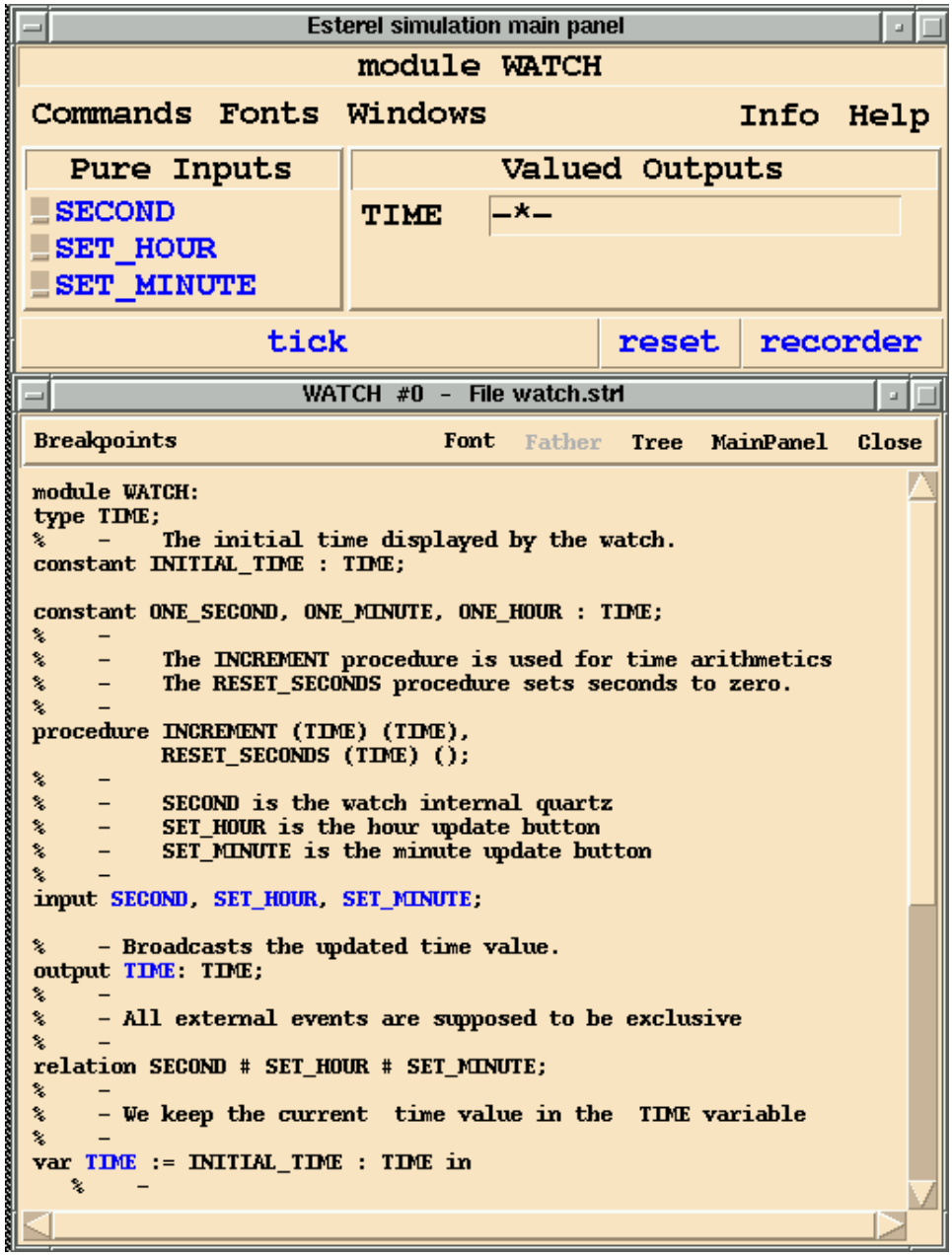


Figure 6.1: The `xes` simulator

panel may not contain all the parts). The *main source window* displays the Esterel source code of the program main module. It provides symbolic debugging at source code level, and can be unmapped if necessary.

It is possible to save a simulator for further use, thus avoiding typing many `.o` names on the `xes` command line and bypassing the time needed to link the simulated modules with the simulation master code. This is done using the `-o` option as follows:

```
xes -o xwatch watch.o watch_data.o
```

This builds a file called `xwatch.exe`, which can be fast-loaded by `xes` to run the simulation[1]:

```
xes xwatch.exe
```

6.2 Performing Reactions

To perform a reaction, one must build the input event and send it to the simulator, which replies by showing the associated output event.

6.2.1 Building The Input Event

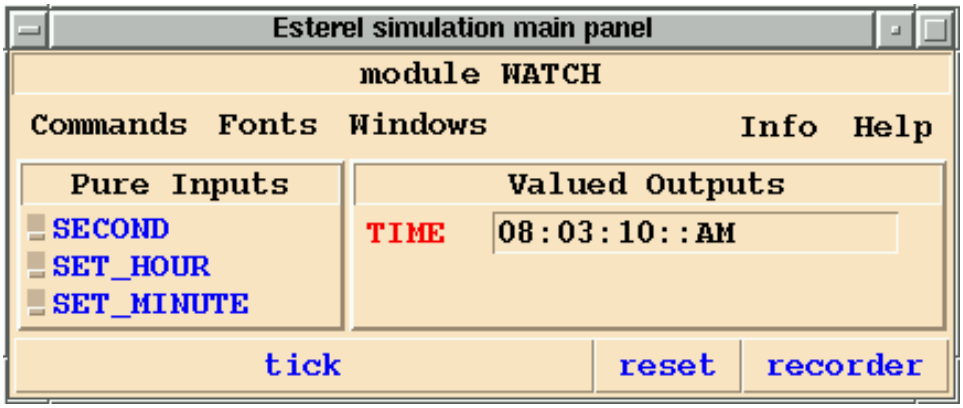


Figure 6.2: The main panel: building input events

[1]The `watch.exe` file is indeed executable, but it requires some libraries provided by the `xes` command; an alternative way is to call directly `xwatch.exe` with the `ESTEREL` source variable set to the Esterel distribution directory, see the `xes` man page.

The input signal part is made of buttons, one per input signal. Each button is labeled with the name of the signal. On the left of each button, there is a little checkbox whose role will be explained in Section 6.2.5. Figure 6.2 shows the main panel of `WATCH`, which has three pure input signals named `SECOND`, `SET_HOUR`, and `SET_MINUTE`.

When the mouse pointer is moved over an input signal name, the background of the button background gets darker. To set the input present, click with the left mouse button. The color of the button label then changes from blue to red. Clicking on a signal which was set present sets back the signal absent (blue). On Figure 6.2, `SET_HOUR` is set present in the current input event, while `SECOND` and `SET_MINUTE` are absent.

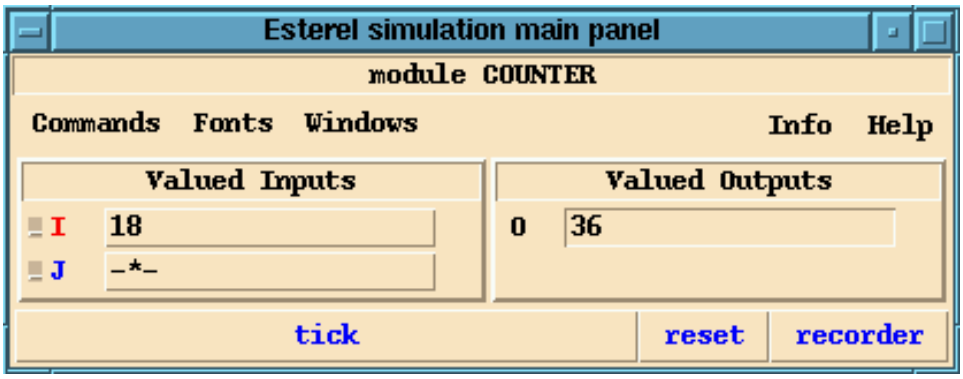


Figure 6.3: The main panel: valued input signals and sensors

Valued input signals are put together in a separate part of the main panel in order to associate each input signal button with a one-line text editor, as shown in Figure 6.3. The associated editor enable the user to input the signal value. Editor commands are Emacs-like. A summary of the main commands is given below.

CTRL-A *or* HOME Move to beginning of line.

CTRL-E *or* END Move to end of line.

CTRL-F *or* RIGHT Move forward one character.

CTRL-B *or* LEFT Move backwards one character.

DEL Delete the character before the cursor.

CTRL-D Delete the character after the cursor.

CTRL-K Kill to end of line.

Arrows and function keys can also be used if present on the keyboard.

> **WARNING:** *Changing the value of an input signal does not set the signal present. For this, one must click on the signal button.*

The syntax of input values is as usual for values of predefined types, except that `float` values do not need a trailing '`f`', unlike in Esterel, since their type is already known. For user types, the syntax is defined by the user-provided string conversion functions, see Section 4.3.4.

Sensors are gathered in a separate part that looks like the valued input signal part. Each sensor name is associated with a one-line text editor that enables the user to input the sensor's current value. Unlike for signals, sensor names are simply labels with no button semantics since sensors do not have a presence status.

6.2.2 Sending the tick

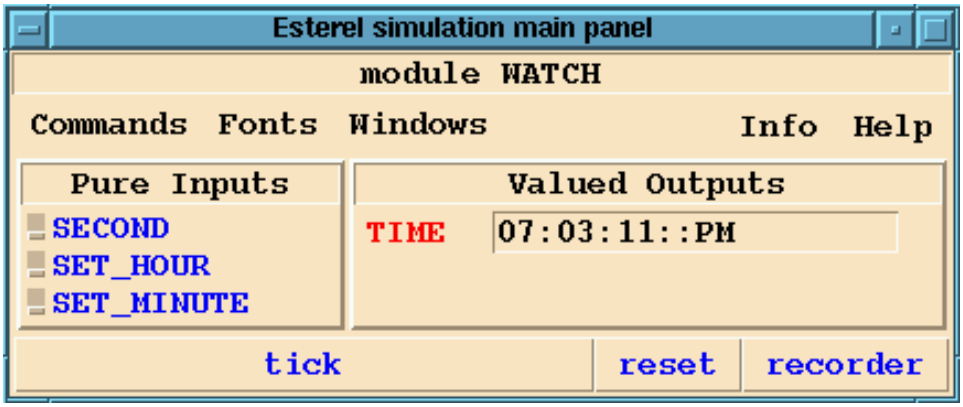


Figure 6.4: The main panel: performing a reaction

Once the desired input event is built, one triggers the reaction by clicking the big `tick` button. A double-click with the left button on an input event button also signals an implicit `tick` and triggers a reaction. Clicking on `tick` with all inputs blue provokes a blank event, i.e. an event where no signal is present.

The input event is first checked for consistency. In case of an error, such as an incorrect value or a violated input relation, a message box pops up to give information about the error, and the reaction function is not called. The simulation can be resumed from the current state by acknowledging the message. The input event remains as it was before clicking the `tick` button, and it must be corrected before retrying.

6.2.3 The Output Event

After the reaction function is called, the output signals emitted by the program's reaction are displayed in red, together with their values for valued signals. Figure 6.4 shows that the output signal `TIME` was emitted in the reaction.

In case of an access to an uninitialized variable during a reaction, no output is performed, an error occurs, and a message pops up. The simulation must be reset since it is impossible for the current version of `xes` to rebuild the previous state.

6.2.4 Building the Next Event

Once a reaction is performed, the input event is reset to all signals absent, unless the `Keep Inputs` option was selected from the `Commands` menu or `xes` was called with `-ki` option. When the `Keep Inputs` option is selected, the input event defaults to the previous event. It can be modified as before.

6.2.5 High/Low Inputs

To selectively keep some signals present by default (active low) instead of absent by default (active high), input switches are provided on the left of input buttons. The switches are activated only if the `High/Low Inputs` option is selected from the `Commands` menu or `xes` was called with `-hl` option. There is one high/low switch per input signal, as shown on Figure 6.4. The switches can take two different positions: the default low blue position and the high red position. When a switch is on its high red position, the corresponding input signal is set present by default. It can be set absent by clicking on its name.

The `High/Low` option is different from the `Keep Inputs` option. The switches allow the user to select input signals that will remain present by default, whatever their value in the previous reaction was. The `Keep Inputs` option tells the simulator to start from the previous input event for the next reaction.

6.2.6 Resetting the Program

The main panel provides a `reset` button to bring the Esterel program back to its initial state, i.e. to the state it had when the simulator was started. Variables and signals are reset to the '`-*-`' uninitialized value, or to their initial values.

6.2.7 Handling exec Statements

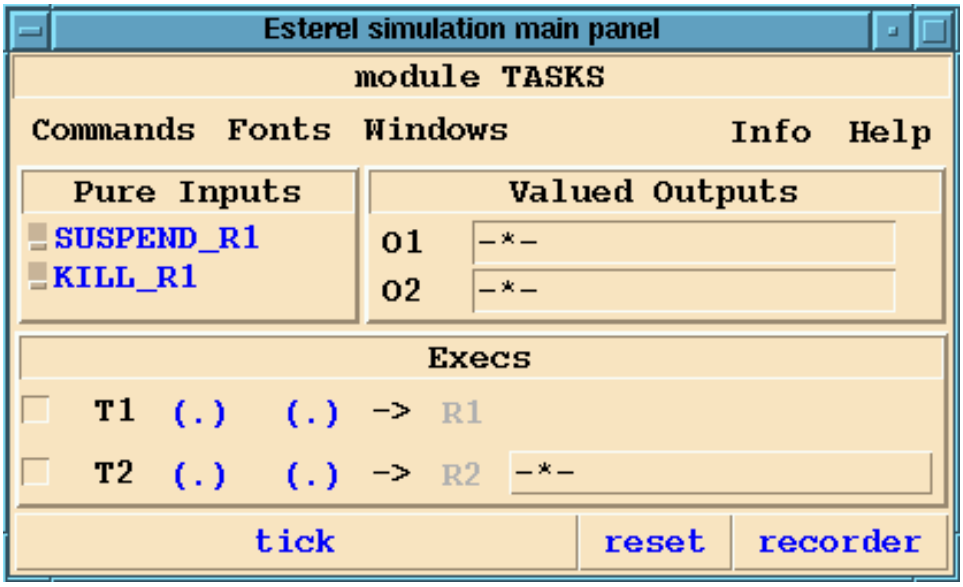


Figure 6.5: The main panel: handling `exec` statements

The handling of `exec` statements is put apart in the main panel, see Figure 6.5. The simulation of `exec` statement is entirely under the user's control and there is no proper task code to be simulated. The user is told when the task starts and with which parameter values. The user signals task return by explicitly inputting the return signal and setting the return values of reference parameters.

Each `exec` statement is displayed on a single line. To avoid lines becoming too long, reference and value argument lists are replaced by a phantom dot between parentheses. The actual argument values are popped when clicking on the phantom. The return signal is displayed on the right of the statement, after a right-arrow. There is a one-line value editor if the signal

is valued.

The name of the return signal is a button since a return signal is a particular input signal that notifies the completion of an `exec` statement. However, the button is active only if the `exec` statement it is associated with is active. In Figure 6.5, `R1` is a pure return signal while `R2` is a valued one.

The status of an `exec` statement is shown by the color of the task name:

black the statement is not active,

red the statement is active,

orange the statement is suspended.

In addition, a flag on the left of the task name indicates whether the task was started or killed in the reaction:

red the statement was started in the previous reaction,

black the statement was killed and not restarted in the previous reaction,

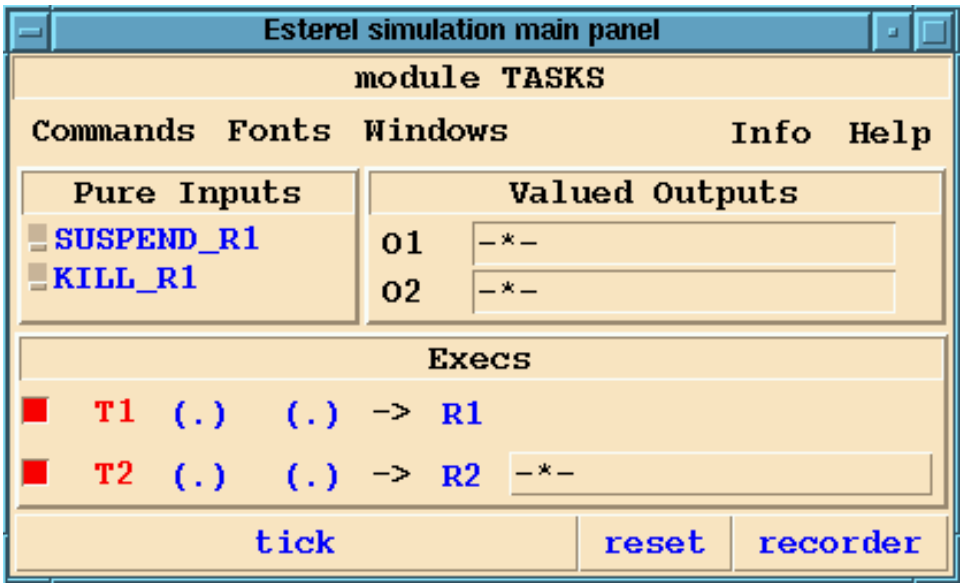
black and red the statement was killed and restarted right away.

This is illustrated in Figure 6.6, where `T1` and `T2` are active and were started in the reaction.

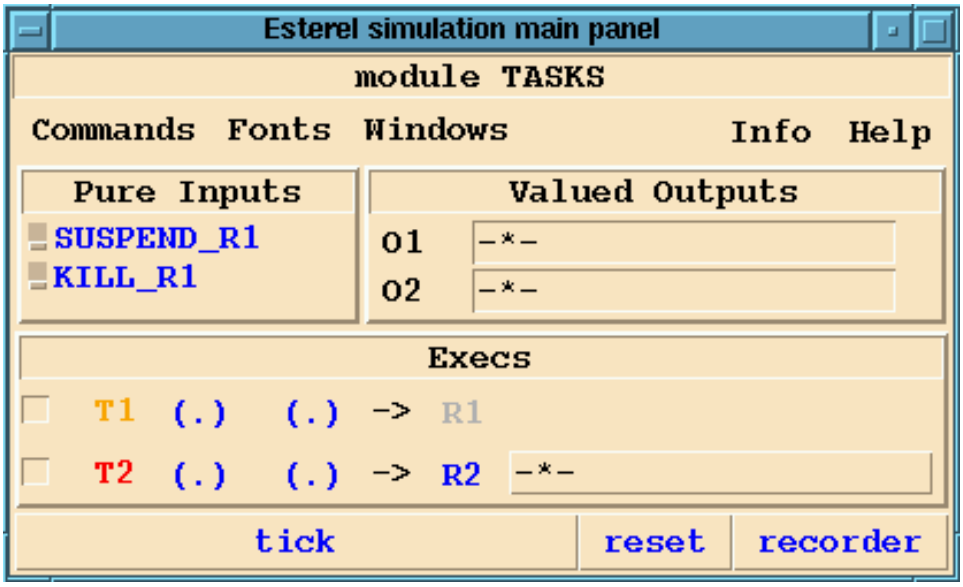
To simulate the completion of an active `exec` statement, i.e. task return, click on the return signal name. If the signal is valued, a value must be provided beforehand using the appropriate one-line text editor. If the task has reference arguments, a window pops up when the return signal is set present to specify the return values for these arguments. Double-clicking on a return signal is not usable if there are reference arguments to update. In Figure 6.7, the user has selected the `R2` return signal, and has to provide a value to update the `X` variable.

6.3 The Main Panel Menus

The main panel has three menus labeled `Commands`, `Fonts`, and `Windows`. A menu is opened by clicking the left mouse button. It can be torn off by clicking on its first dashed line item.



`T1` and `T2` are started and running



`T1` is suspended and `T2` is running

Figure 6.6: The main panel: status of `exec` statements

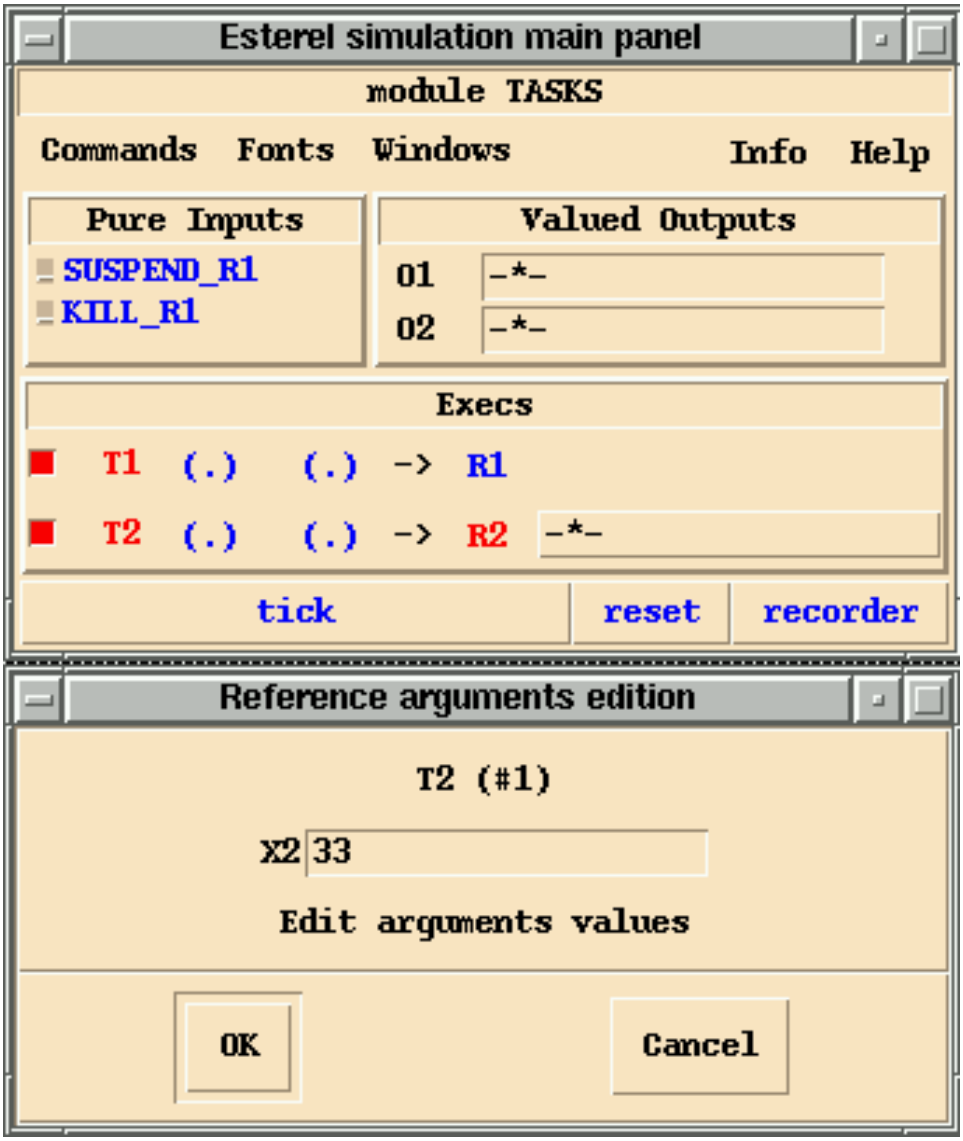


Figure 6.7: The main panel: completion of `exec` statements

6.3.1 The Commands Menu

The `Commands` menu has the following items:

Signal Browser When selected, this checkbutton enables the signal browsing facility, see Section 6.4.4. Notice that signal browsing is inactive by default.

High/Low Inputs If this toggle is false, which is the default, the active high/low switches on the left of the input buttons are deactivated. If true, the switches are activated, see Section 6.2.5. The toggle can be initialized to true by setting the `-hl` option of the `xes` command

Keep Inputs If this toggle is false, which is the default, selected inputs are unselected after a reaction. If true, selected inputs remain selected for the next reaction. It can be initialized to true by setting the `-ki` option of `xes`.

Clear Inputs Reset all preselected inputs to their default state, i.e. absent for the next event in default mode, or to the current high/low status if `High/Low Inputs` is activated.

Remove Breakpoints Removes all breakpoints in the program, see Section 6.4.

Recorder Pops up a tape player/recorder allowing Esterel session saving and playback, see Section 6.5.

Quit Ends the simulation.

6.3.2 The Fonts Menu

The `Fonts` menu controls global font sizes. Separate control is given on the font size of the panels (main panel, locals, traps, variables) and on the font size of the source and program tree windows. A local font size control is also available on each window.

6.3.3 The Windows Menu

The `Windows` menu controls the other simulation windows:

Program Tree Displays the whole tree of the main module and submodules of the Esterel program, see Section 6.4.3. The source code of a submodule can be popped up by clicking its name in the tree. Colors in the program tree are explained in Section 6.4.3.

Locals Pops up a window containing the local signals, which are shown in red when emitted in the last reaction. The same information is available on the source code, see Section 6.4.

Traps Pops up a window containing the trap names, which are shown in red when exited in the last reaction.

Variables Pops a window containing the variable names and values.

Source Windows The last items of the Windows menu are the names of current source file windows, see Section 6.4.

The `Locals`, `Traps`, or `Variables` windows are not available if the program does not use local signals, trap statements, or variables.

6.4 Symbolic Debugging

6.4.1 Finding the Source Code

At startup time, `xes` looks for the source code of the Esterel program in the current working directory or in the directory specified by the the `-D` source directory option of the `xes` command. If the source code is not found in that way, the user is prompted for the correct pathname of the source directory. If still no source code is found, source debugging cannot be performed but the rest of the simulation works normally. If the source files are found, the source Esterel code of the main module pops up in a source window whose name is added to the main panel `Windows` menu.

6.4.2 Source Windows

Figure 6.8 shows an example of a source window. Each submodule appears in a separate window. A submodule's window is popped either by clicking on the `run` (or `copymodule` in old syntax) statement in the father's source window, or by clicking on the module name in the `Program Tree` window. When a source window pops up for the first time, its name is added to the main panel `Windows` menu. If a single submodule is run several times, its instances are distinct in the tree and they appear in distinct windows.

The `Father` button in a source window menubar pops the father's module source code. The `Main Panel` button pops the main panel and the `Program Tree` button pops the program tree. This is useful when these windows ared buried under source windows.

Closing a source window using its `Close` button removes the window from the screen, but not from the source window list in the main panel's `Windows` menu. It can be recovered from there or from the program tree window.

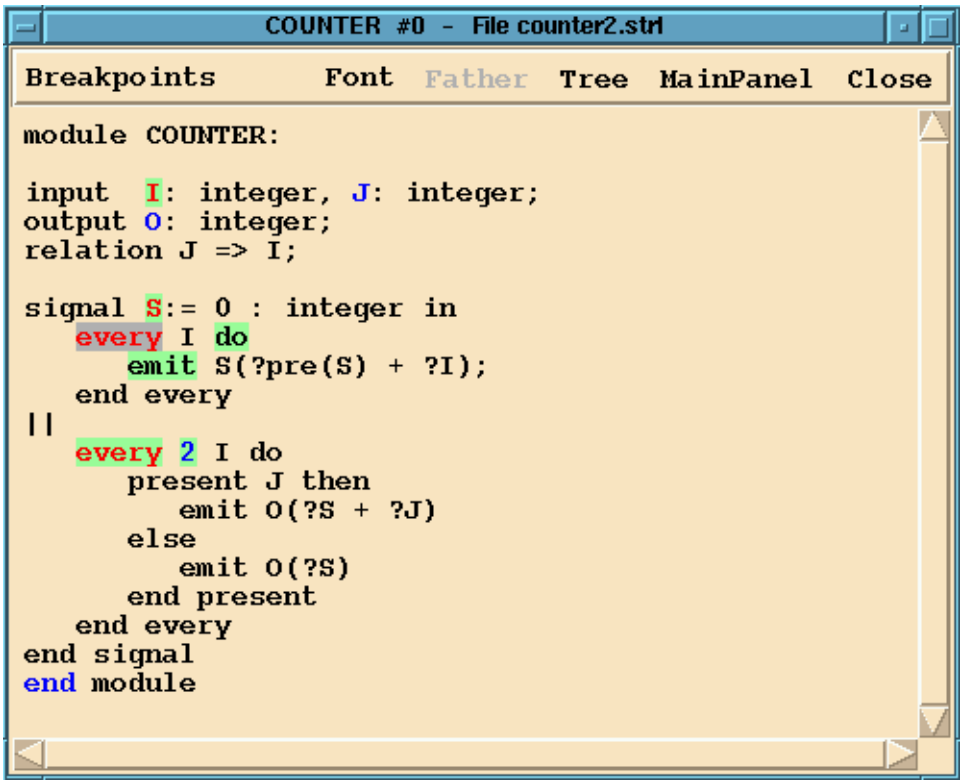


Figure 6.8: Symbolic debugging: a source window

6.4.3 The Program Tree

The program tree window is pictured in Figure 6.9. It is raised by clicking on the `Program Tree` entry of the `Windows` menu in the main panel. The program tree shows the module instantiation structure defined by the nested `run` Esterel statements. A given module appears as many times as it is instantiated in a program. Different occurrences are identified by different numbers appearing after the '`#`' character. The + or - symbol on the left of a module name controls the opening and closing of the subtree it controls.

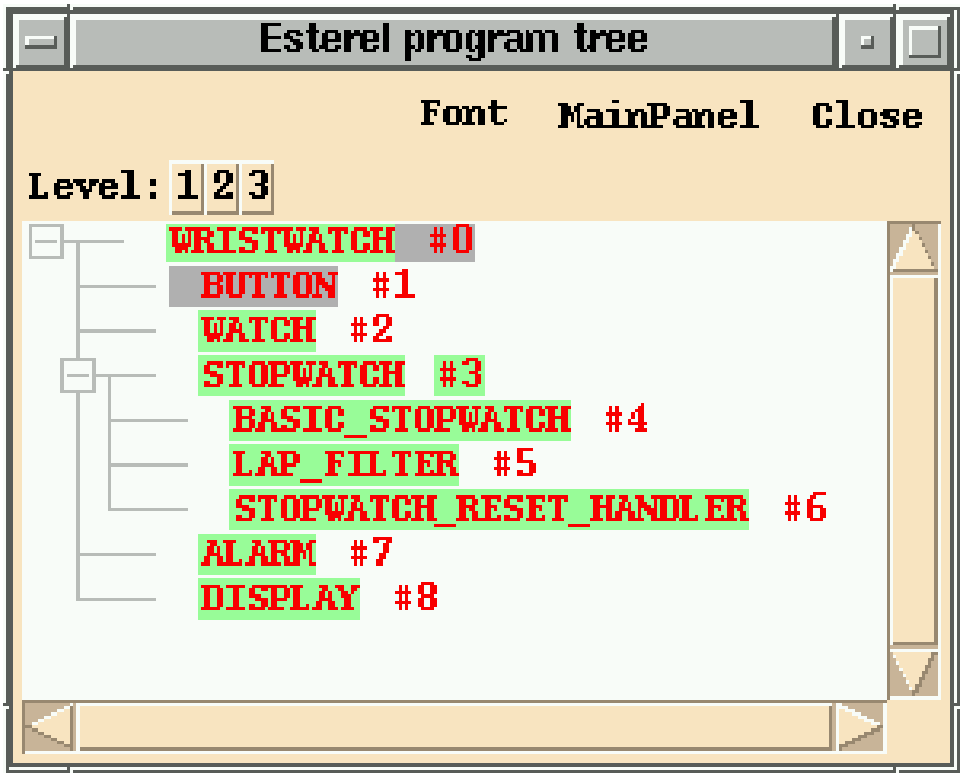


Figure 6.9: The Program Tree

Click on the numerical buttons on the top of the program tree window to open the tree up to a given level. Clicking on a module name opens a source code window for this module occurrence. The instance number appears in the window's title.

Each program tree line is composed of two parts: the *head*, which contains the module name and the blank character before it, and the *tail*, which contains the module number and the character before it. Various informations are displayed in the head and tail parts using the color codes explained below. Information displayed in the head is called *direct information*; it is about the module code proper. Information displayed in the tail is called *inherited* information; it is about the submodules of the considered module. An example of direct information is module activity: the foreground color of the head is read if the module is active. An example of indirect information is submodule activity: the tail foregroung color is red if there is an active submodule. The same principle holds for any color code.

Foreground and background of each module name displays an information which is inherited from the corresponding source window or the source windows of its sub-modules. Foreground colors may be red or blue. A background color may be green, pink, grey, orange, or the default color of the tree window itself.

Foreground is red (resp. blue) if the module is currently active (resp. inactive), i.e. some (resp. no) haltpoints in the module are currently active.

The program tree is fundamental for browsing. When interested by a particular aspect related to a color code, inspect the program tree to find where objects of this color appear in the head or tail and click on the module names to open the corresponding source windows. Folding and unfolding the tree may be necessary for big programs.

6.4.4 Signal browsing

In the Commands menu, the "Signal browser" checkbutton enables a signal browsing facility, which is available only for programs compiled with the `-I` option. To see where a particular signal is declared, emitted, accessed, or displayed, just click with mouse-button 3 on its name in a panel (main panel or locals or traps) or in its declaration in a source code, or on the blue or red keyword of instruction referencing this signal (emit, await,...). All occurences of the signal name in all the sub-windows of xes start blinking in an orange color background, together with the names of the modules where the signal appears (run instructions in source windows, lines in the program tree instances). Blinking is a bit heavy on the eye, but it is useful since the

current colors of objects still appear in the non-orange phase. To stop signal browsing, unselect the "Signal browser" checkbutton.

WARNING: the Esterel code must be compiled with the -I option.

6.4.5 Colors in Source Windows

Colored foregrounds and backgrounds are used to decorate the source code and the program tree, and some keywords also appear underlined. Colored items are selectable by the mouse, and the effect depends on the type of object.

The declaration point of a signal appears in blue if the signal was absent in the last reaction and in red if the signal was present. In either case, clicking on the signal name in the declaration pops the current signal value. Notice that interface signals of copied submodules remain black: these are not real objects since they are replaced by actual signals in the father module.

Clicking on a variable at its declaration point pops its value. Similarly, the value of a count expression appearing in a delay expression or in a repeat loop can be displayed by clicking on it. For example, click on the constant expression `2` in the statement "`every 2 I do`" to see the current value of the associated counter variable.

Keywords of statements that contain a haltpoint appear in blue or red foreground. These keywords are `pause`, `halt`, `await`, `every`, `each`, `exec`, and `immediate` in "`suspend...when immediate S`", which has a haltpoint to wait for the first non-occurrence of `S`. They appear in red if the previous reaction has precisely stopped at that point. For example, "`await S`" is red exactly when one is waiting for `S` there. An `every` or `each` statement is red if and only if its body is terminated; one is then simply waiting for the corresponding event. The `immediate` keyword in an immediate suspension is red while waiting for the first instant where `S` is absent. Since Esterel is a parallel language, several concurrent haltpoints can be red at the same time, see Figure 6.8.

In the program tree, the head (module name) is in red foreground if at least one haltpoint is red in the module own body, the tail is red if at least one submodule has an active haltpoint.

All statements that involve a preemption are underlined in either blue or red. These statements are `abort`, `every`, `each`, `suspend` and `upto`. The keyword is underlined in red if preemption is currently possible, i.e. if there is an active (red) control point in the statement body.

6.4.6 Breakpoints

Clicking on a haltpoint keyword sets or removes a breakpoint. Breakpoints set are shown using an grey background for the first character of the keyword. For example, a breakpoint is set on the first `every` statement in Figure 6.8. In the program tree, the head space before the module name is in grey background if there is at least one breakpoint in the module proper. The tail space after the module name is in grey background if there is at least one breakpoint in a submodule.

When the program reaches a breakpoint, an alert message window pops up and all keywords corresponding to reached breakpoints appear on grey background for all their characters instead of just the first one. In the program tree, the full module head is on grey background if there is at least one breakpoint reached in the module proper, the full tail is on grey background if there is at least one breakpoint reached in a submodule.

To remove a single breakpoint, click on it. To remove all breakpoints in a module instance, click on the `Remove` entry of the `Breakpoints` menu of the module source window. To remove all breakpoints in a module and in all its submodules, click on `Remove recursively` entry of the `Breakpoints` menu of the module source window. To remove all breakpoints in the whole program, click either on the `Remove recursively` entry of the main module source window on the `Remove Breakpoints` entry of the main panel `Commands` menu.

6.4.7 The Control Path

When a program is compiled with the `-I` constructive interpretation option, some source keywords are displayed with a green background as shown on Figure 6.10. They correspond to the statements executed in the previous reaction. This control path feature makes it possible to find out what the Esterel program actually did in the reaction. In the `WATCH` example of Figure 6.10, a `SET_MINUTE` signal was received in the previous reaction while the program was stopped on the `await-case` statement.

In the program tree, the head is on green background if there is at least one executed statement in the module proper, the tail is on green background if there is at least one executed statement in a submodule.

Red foreground and green background can coexist. For example, consider the `WATCH` example. In the reaction, the `await` keyword appears in red foreground on green background, since the implicit halt statement in the `await-case` statement has been reached (green background) and control has

stopped on it (red foreground). The previous `emit` statement and the `TIME` variable declaration are displayed on a green background too, since they were executed first in the reaction. The declaration of the output signal `TIME` is also on a green background to mean that the signal was emitted in the reaction.

In the second reaction, if neither of the three signals, `SECONDS`, `SET_HOUR`, and `SET_MINUTE` is present, the `await` keyword remains in red over green meaning that control stays in the `await` statement, and the `case` keywords change to green background meaning that the implicit present tests were performed.

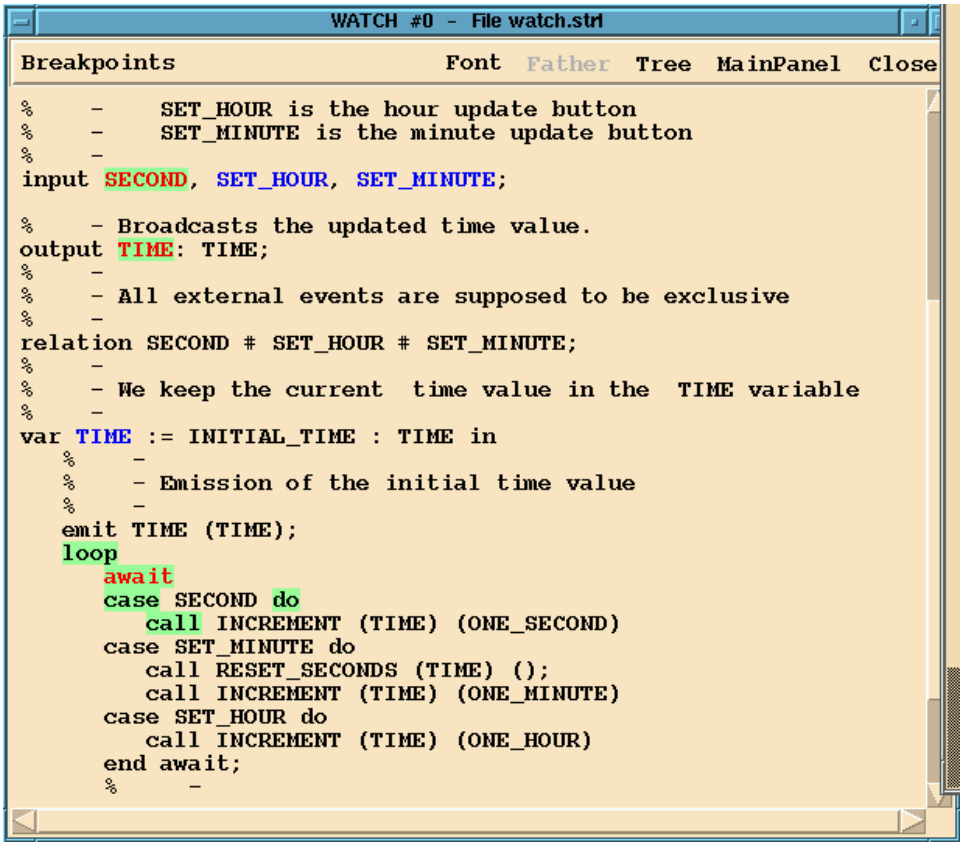


Figure 6.10: Symbolic debugging in interpretation mode

If, instead, the `SET_MINUTE` signal is present in the second reaction, only the first two `case` keywords change to green background. The third `case` is

not reached since the second one is successfully checked and the corresponding `do` statement is executed. The second `emit` statement is then executed and the `await` statement is started anew by the `loop` statement. All this is clearly shown by the green background on Figure 6.10.

Complex reincarnations are not shown in a fully satisfactory way, see Section 8.3.

6.4.8 Causality Errors

When a program is compiled with the `-I` option, causality problems may occur at run-time. When a causality error occurs, a dialog box pops up and the error is displayed on the source code and program tree using green and pink backgrounds. The interpretation of these color codes conforms to the constructive semantics used in the Esterel v5 compiler and described in [2].

A statement is shown on a green background if it *must* be executed in the transition. Statements that *cannot* be executed remain on standard background. Statements for which it is impossible to prove either that they must be executed or that they cannot be executed appear on a pink background. The same backgrounds are used for signals at their declaration point.

In the program tree, the head is on green (resp. pink) background if there is at least one green (resp. pink) statement in the module proper, the tail is on green (resp. pink) if the same holds for a submodule.

When faced with a causality problem, first identify the current state by finding keywords written in red foreground. Then, look for pink signals, since they are the ones with undetermined status. Finally, look for places where background changes from green to pink, since they are tests for undetermined signals. Browse modules using the program tree.

Let us consider the `P3` non-constructive program studied in [2]. Figure 6.11 shows the simulation windows after the user has clicked on the `tick` button to get the first reaction. The `present` statement must be executed, hence the `present` keyword is on a green background. But it is impossible to determine whether the `else` branch of the `present` statement should be executed or not using the constructive semantics. Therefore, the `else` keyword, the `emit` keyword, and the `O` interface signal declaration are displayed on a pink background.

Because of instantaneous loops in Esterel, the pink and green backgrounds may collide. For instance, it may happen that a statement within a loop must be executed, which results in a green background, and that it is undefined whether or not the statement must be instantaneously executed anew (i.e. reincarnated) by the loop, which results in a pink background.

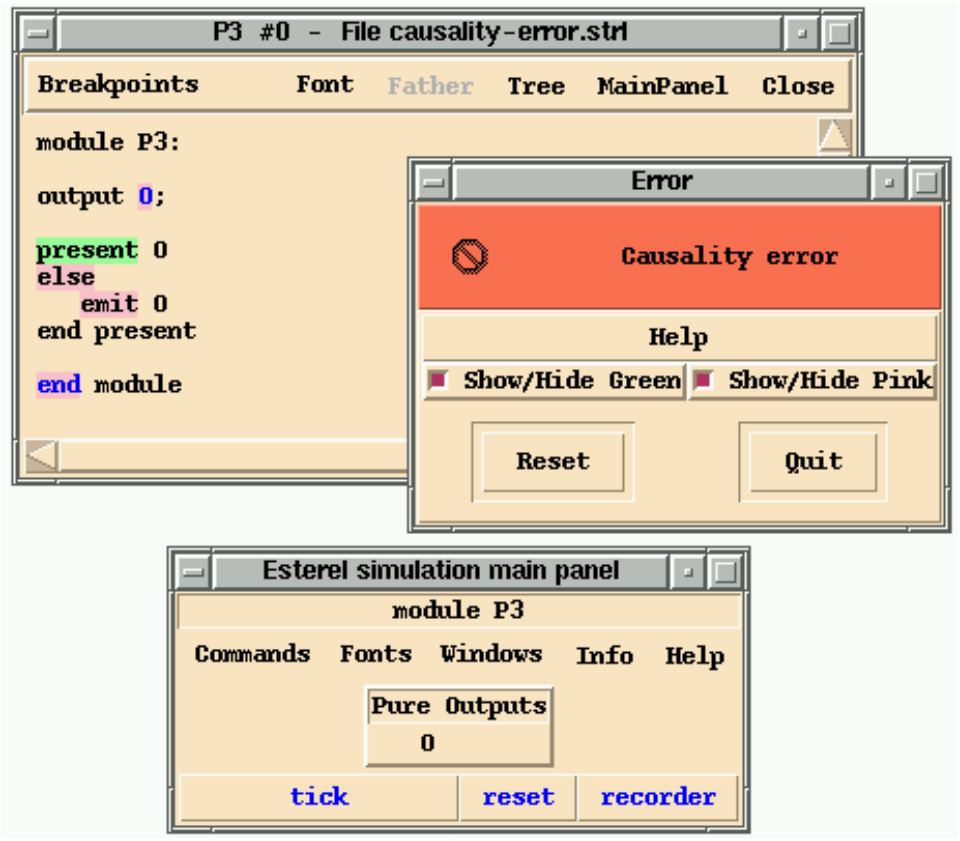


Figure 6.11: Causality errors

In such a case, the statement is initially displayed with a pink background. The causality error alert box contains two auxiliary buttons that allow to selectively hide or show green and pink backgrounds. When showing a background color afresh, the color takes precedence over the other color. This makes it possible to analyze causality errors in a finer way. A statement that should be both pink and green is first shown on a pink background and is then shown on a green background if the green background option is selected.

6.5 The Session Recorder

The session recorder / player is pictured in Figure 6.12. Il deals with event input files, which we call tapes. Its window splits in two parts: the lower part is for saving the current simulation session in tapes, the upper part is for playing already recorded tapes.

A tape is a file which contains recorded events in csimul format. The file name has special extension `.esi` for Esterel Simulation Input, but the old extension `.csimul` used prior to version v5_91 is still recognized by the tape browser.

6.5.1 Recording a Tape

The current session input sequence is automatically recorded in a recording buffer, which can be reset to the empty state by clicking on the `Erase` button. A `reset` command entered either from the input panel or from an input tape generates a `!reset` simulation command in the recording buffer. A counter counts the number of entries in the recording buffer.

At any time, the current contents of the buffer can be saved in a tape by clicking the "`Save as`" button in the bottom recording part of the recorder window. A browser makes it possible to define the directory and file names, which are the current directory and the name `untitled` by default. The file name appears in the filename display, and the full file path can be popped up by clicking in the name window. The `Save` button rewrites the current session in the currently selected file, overwriting its previous contents. The file name is blue if the contents of the file matches the contents of the recording buffer, and becomes red as soon as the recording buffer is modified.

WARNING: the save operation is performed only when you click on the `Save` or "`Save as`" buttons. If you quit without saving, you are asked for confirmation.

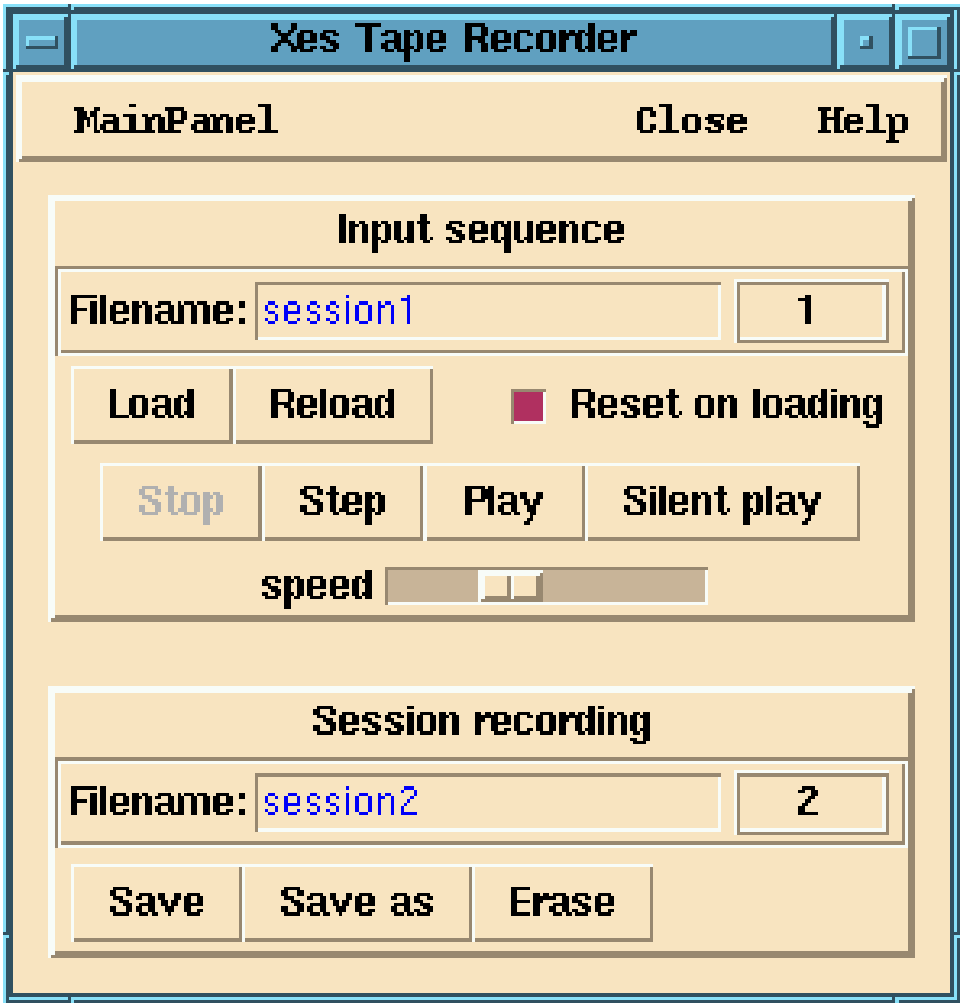


Figure 6.12: The Session Recorder

6.5.2 Playing a Tape

To play an existing tape, click on the `Load` button. You are asked for a file name by the browser. The name of a currently loaded tape appears in blue in the name display. The full path of the corresponding file may be popped up by clicking on the name display. When loaded, a tape can be played step by step with button `Step` or continuously with button `Play` or button "`Silent play`". An event counter is displayed. The tape automatically stops when reaching its end or if you click the `Stop` button. The playing speed can be tuned with the slider.

In `Step` or `Play` mode, a graphical update of the whole simulator is performed at each tick (emitted signals, nets, values, etc.), as if the event had been entered by hand. In "`Silent play`" mode, transitions of the automaton are performed without any graphical update except for the last one, making it possible to quickly reach the final state. A currently stopped tape (name in blue) or the last fully played tape (name in grey) can be repositionned at its beginning by clicking the `Reload` button. When loading or reloading a tape, automatic reset of the automaton is performed if the check button `Reset on loading`" is selected, which is the default. A `!reset` command is then added to the recording buffer, and the recording event counter is incremented.

6.5.3 Saving Conflicts

A read/write conflict occurs when trying to save the current recording in the tape that is currently loaded for playing. In this case, you are asked to choose between the following:

- eject the tape from the player and proceed with saving, overwriting the previous tape contents
- same, but immediately reload the tape in the player
- abort the save operation

6.5.4 The untitled Tape

The default `untitled` tape is handled in a special way. It obeys the same read/write rules as other named tapes, except that, if `untitled` is selected for recording, you are not asked for saving confirmation at the end of the session, even if your session is not totally saved. Since event recording is

always on, this avoids a systematic confirmation click if the recorder is never invoked.

The untitled tape is convenient for fast recording / playing. For example, if you want to take a snapshot of the current session and replay it right away, click on `Save` in the recording subwindow, and click immediately on `Reload` in the playing subwindow.

6.6 Options of the xes Command

The options of the `xes` command are listed when typing

```
xes -help
```

The main options are:

`-version`

Displays the version number.

`-info`

Displays more extended identification of `xes`.

`-access` Displays the access rights to `xes`.

`-o` *name*

Builds a fast loadable simulator called `name.exe`, without running it. Run this simulator by "`xes name.exe`".

`-script` *script*

Loads a tcl script file to change some of the `xes` resources. This may be useful for some X-window systems where `xes` looks ugly for reasons we cannot deal with! The default resource file is provided under the name `xes_res.tcl` in the Esterel distribution (subdirectory `lib/xes`). Change it according to your needs.

`-display` *display*

Specifies the X display.

`-geometry` *geometry*

Specifies the main panel position.

`-ki`

Runs the simulator in *keep inputs* mode by default.

`-np`

Suppresses source code parsing. Useful if the source language is not Esterel proper but generates Esterel (thanks to special features undocumented here).

`-ns`

Disables source code debugging.

`-np`

Disables source code parsing. This special feature is for people who write Esterel super-languages for which they want the simulator not to parse the source file (parsing is only used to underline abotrion statements).

Please use option `-info` to identify your version of `xes` when sending a bug report or a question.

Chapter 7

Simulation with csimul

By executing the object simulator generated using `libcsimul.a`, the user enters a `csimul` simulation session. The simulator repeatedly waits for an event input command or for a control command from the following list: `!show`, `!trace`, `!reset`, `!help`, `!module`, `!load`, and `?`. An input command defines an event to react to. The other commands control the simulator's behavior and various traces. The simulator can take its commands directly from the terminal or from files.

External tasks are entirely simulated by the user. The simulator tells when a task is started and when it is suspended or killed. The user provokes the task return and explicitly passes return arguments. Therefore, there is no code to write to simulate tasks.

7.1 Prompts, Help, Exit, and Interrupts

By default, the currently simulated module is the first module in the generated C file. Call it `PROG`. Before waiting for a command, the simulator prints a primary prompt `PROG>`. There is a secondary prompt '`>`' for commands that spread over several lines, as in:

```
$ prog
PROG> I
> J
K;
```

Comments start with a '`%`' character and end at end-of-line, as in Esterel. The help command '`?`' or `!help` prints a help message. The exit command '`.`' exits the simulator. All other commands must end by a semicolon '`;`'.

Any interrupt character typed when entering a command from the terminal cancels the command. When reading a command file, an interrupt character stops reading the file and processes the next possible input file.

Syntax errors in commands are recovered by typing a semicolon, the special prompt `recover>` indicating that the simulator tries to recover a syntax error.

7.2 The Input Command

The input command defines an input event for a reaction. It consists in a blank- or comma-separated list of input signals, exec returns, or sensors items. Values must be provided for valued signals and sensors. For a return signal of a currently active `exec` statement, the reference parameters return values must also be given as described below.

The input list is ended by a semicolon '`;`'. The given input signals are considered as simultaneous inputs for the reaction; they are all merged into the input event of the reaction.

When fed with an input line, the simulator checks that this line is a valid input. If so, it performs a reaction unless the input command entirely consists of sensor value definitions, in which case the sensor values are set without this provoking a reaction. The simulator prints the emitted output signals with their values, the list of tasks started, suspended, or killed, plus other information according to the current show or trace option described in Section 7.6. Here is a simple example:

```
PROG> ;  % tick
--- Output:
PROG> I J;
--- Output: O
```

7.2.1 Input Syntax

Input event components are entered as follows:

- A pure signal is simply denoted by its name.
- A valued signal or sensor is denoted by its name followed by its value, given either enclosed in parentheses as in `I(1)` or preceded by the '`=`' sign as in `I=1`.

- An exec return consists in the name of the return signal, followed by a value if the signal is valued, the `return` keyword, and the list of returned values for reference parameters. Here are examples:

```
R1 return ()
R2(3) return (5,6)
R3=5 return ("OK")
```

Values of signals, sensors, and task reference parameters are entered as strings to be passed to the appropriate conversion functions. Strings are enclosed in double quotes. Inner double quotes are doubled, as in Esterel. Integer, Boolean and floating-point values can also be entered without double quotes. For signal and sensors, the value is given after the signal or sensor name, either enclosed in parentheses or preceded by the '`=`' sign. For example, assume signals `I`, `J` and `K` have, respectively, type `integer`, `boolean` and user-defined `TIME`. Then the two following input lines are equivalent:

```
PROG> I(8), J(false), K(12:01:59);
PROG> I=8 J=false K="12:01:59";
PROG> I("8") J="false", K="12:01:59";
```

The simulation toplevel accepts floating point values in C-like format. For example `2.5` or `4.44e-3` are valid floating point values. Floating point values may be entered with or without double quotes as in

```
PROG> F(8.0), PI(3.14159), PLANCK("6.62.e-34");
```

7.2.2 Event generation

An input event is generated by an input command unless the command only sets the value of sensors. An empty input command such as

```
PROG> ;
--- Output: O
```

simply generates a tick, i.e. a reaction with no input. The input command

```
PROG> SENSOR(1);
PROG>
```

sets the value of the sensor `SENSOR` to `1` and does not call the reaction function. There is no output message and the simulator prompt is printed back.

Notice that no reaction is triggered by the ending semicolon '`;`' if there are only sensors in the input line.

In the next example, there is a sensor and an input signal. A reaction is performed:

```
PROG> INPUT_SIG(3), SENSOR(1);
--- Output: O
```

7.2.3 Input Checking

An input is invalid and rejected in the following cases:

- A signal appearing in the input list is neither an input, nor an inputoutput, nor a return signal, nor a sensor;
- A pure signal appears with a value;
- A valued signal appears without a value;
- A valued signal appears with a value that is not of the proper type;
- For a return signal, the associated list of values for update of reference variables is either missing or incorrect (wrong number of values, mismatched types, etc.);
- A list of values for reference variable updates appears with a signal which is not a return signal;
- A pure signal, a single signal, or a sensor appears several times in the input list;
- The input event violates some input relation.

If the input is invalid, the simulator prints a self-explanatory error message and sends back a new prompt without changing state. No execution of the reaction function is performed. The current input event is simply ignored.

7.2.4 Combined Signals in Input Events

A combined signal can appear several times in an input event. The actual input value is computed using the signal's combination function before the transition. Assume for example that `COMB` has addition for combination function. Then the input

```
PROG> COMB(1) COMB(2);
```

is equivalent to the input

```
PROG> COMB(3);
```

7.3 Output Printing

The output event corresponding to an input event is printed after the `Output:` keyword. Here are some examples of valid input lists and output events:

```
PROG> METER;
--- Output:
PROG> INTEGER_SENSOR(-4), INTEGER_SIGNAL(2001);
--- Output: TEMPERATURE(12)
PROG> FLOAT_SIGNAL(2.4E3), DOUBLE_SENSOR(2.4E3);
--- Output: AVERAGE(12.4E43) GOOD_SHOT
PROG> BOOLEAN_SIGNAL(false), STRING_SIGNAL("long-string");
--- Output: ECHO("long_string")
PROG> MS, SET_TIME("2:2:21:PM");
--- Output: TIME(2:02:21:PM) BEEP
PROG> COMBINED_INPUT(1), COMBINED_INPUT(2);
--- Output: OUTPUT(3)
```

For task execs, the changes in execution statuses are printed after the `Started`, `Suspended`, and `Killed` keywords. A keyword is followed by the number of `exec` task instances started, suspended, or killed, and by the list of those instances. Here are some examples:

```
EXEC> ;
--- Output:
--- Started: 2
T1 (-*-) () return R1
T2 (-*-) () return R2
--- Suspended: 0
--- Killed: 0

EXEC> R1 return (3);
--- Output: O1(3) THE_END
--- Started: 0
--- Suspended: 0
--- Killed: 0

EXEC> SUSPEND_R1
>     R2(3) return (4);
--- Output: O2(7)
--- Started: 0
--- Suspended: 1
T1 (12) () return R1
--- Killed: 0
--- Active: 0
```

```
EXEC> KILL_R1
>     R2(5) return (6);
--- Output: O2(11) THE_END
--- Started: 0
--- Suspended: 0
--- Killed: 1
T1 (6) () return R1
```

The '`-*-`' symbol means undefined value, which is ok for a reference task argument when the task is started.

More information about task statuses can be obtained by using the `!show` and `!trace` commands, see Section 7.6.

7.4 Input and Output Streams

By default, a simulator reads commands on the standard input stream. One can also run batch simulations by passing a list of simulation command file names to the simulator. Furthermore, if any of the file names is replaced by the symbol '`-`', then the standard input is read in place of a file, up to end-of-file (typically `^D`).

For example, the call

```
$ prog simcom1 simcom2 -
```

first executes the simulation commands contained in file `simcom1`, then the commands from file `simcom2` and finally prompts the user for new commands at the terminal. This is useful to bring the program to a given state before starting interactive simulation or to set trace options. The symbol '`-`' can appear more than once in the file list.

A file load command can also be executed from within an interactive or batch command sequence, up to the nesting permitted by the operating system. The command has the following syntax

```
!load "file";
```

The `!load` command echoes its input commands on the standard output stream. To reopen the standard input when reading a command file, use the "`!load -;`" command.

In interactive and batch modes, the simulator echoes its input on its standard output stream. Output also goes to the standard output. Error messages are written on the standard error stream. To redirect the output of the simulator, it is good practice to use the shell command

```
$ prog > LOG 2>&1
```

to get proper placing of error messages.

7.5 The reset Command

The `!reset` command resets restarts the simulation of the current module afresh. It has the syntax

```
!reset;
--- Automaton PROG reset
```

The simulator starts again and waits for input events.

7.6 The show and trace Commands

The `!show` command followed by an option list enables the user to see more than just the output event of the next reaction. It has no effect other than responding to the show request and giving back the simulator prompt. The `!show` command acts for only one reaction; to render its effect permanent, use the `!trace` command.

The `csimul` simulator recognizes fifteen basic `show` options: `state`, `halts`, `variables`, `sourcevariables`, `signalvariables`, `counters`, `signals`, `locals`, `traps`, `awaited`, `execs`, `started`, `suspended`, `killed`, `active`, and the special option `all` that comprises all other fifteen. The syntax of the `!show` command is:

```
!show option-list ;
```

where `option-list` stands for a non-empty list of valid `!show` options, for example:

```
PROG> !show state signals;
```

7.6.1 Showing the State

The `!show state` command prints the current state number:

```
PROG> !show state;
--- Current State: 1
```

This is meaningful only for automaton code generated by the `-A` option of the `esterel` command.

7.6.2 Showing Haltpoints

The `!show halts` command prints the halpoints that characterize the current control state:

```
PROG> !show halts;
--- Halts: 2
halt 1: line: 33, column: 7 of file: "prog.strl" (PROG#0)
halt 5: line: 67, column: 12 of file: "aux.strl" (AUX#2)
```

Let us explain what these lines mean. All temporal instructions like `await`, `upto`, `loop...each`, `every`, ... are expanded by the Esterel compiler as explained in [1]. Each expansion generates exactly one kernel `pause` statement. An automaton state is exactly a set of such kernel `pause` statements on which the control was halted in the current reaction. The above lines list exactly these statements together with the position of the source statement that generated them. Therefore, this list of positions is exactly a representation of the current state in the source program. The '`#`' character after the module name defines the instance number of the module in the program tree, see Section 6.4.3.

7.6.3 Showing Variables

The `!show variables` command prints the variables of the module. Each variable is identified by its C name and by its source name whenever possible, as in the debug format.

Variable values are converted to strings before being printed, using the output conversion functions described in Section 4.3.4. Variables that are not yet initialized are printed out as '`-*-`'. There are three sets of variables:

- Source variables, declared by the user in the Esterel program.
- Signal variables, associated with valued signals.
- Counter variables, associated with `repeat` statements and signal occurrence delays such as "`await 3 S`".

Here is an example of "`!show variables`" output:

```
PROG> !show variables;
--- Source Variables:
V3 = 0 (source variable VAR1)
V4 = -*- (source variable VAR2)
--- Signal Variables:
```

```
V1 = -*- (value of signal I1)
V2 = 10 (value of signal I2)
V5 = 20 (value of sensor S)
--- Counters:
V6 = 2 [line: 16, column: 13 of file: "prog.strl" (PROG#0)]
```

The `!show sourcevariables` command only prints the variables that are explicitly declared by the user in the Esterel program:

```
PROG> !show sourcevariables;
--- Source Variables:
V3 = 0 (source variable VAR1)
V4 = -*- (source variable VAR2)
```

The `!show signalvariables` command only prints the variables that hold the values of valued signals:

```
PROG> !show signalvariables;
--- Variables:
V1 = -*- (value of signal I1)
V2 = 10 (value of signal I2)
V5 = 20 (value of sensor S)
```

The `!show counters` command only prints the variables that hold the values of counters:

```
PROG> !show counters;
--- Counters:
V6 = 2 [line: 16, column: 13 of file: "prog.strl" (PROG#0)]
```

Counters are variables of type `integer`. They are associated with `repeat` statements and temporal statements that use count delays.

7.6.4 Showing Signals

The `!show signals` command shows three sets of signals: the set of local signals emitted in the previous reaction, the set of traps exited in the previous reaction, and a set of signals "*awaited*" by the program before the next reaction. The set of awaited signals is meaningful only for automaton code (`-A` option of `esterel`): the presence / absence status of a non-awaited signal cannot influence the next reaction (but its value can). Here is an example:

```
PROG> !show signals;
--- Local: A_LOCAL_SIGNAL
--- Trap: ALARM
--- Awaited: GO_BUTTON SECOND OTHER_BUTTON
```

The interpretation of the output of the "`!show signals`" command can lead to rather intricate discussions on the execution of the reaction. A local signal declaration can be reincarnated any given number of times in a reaction [2]. Moreover, each incarnation can have its own fate, i.e. during its (short) life, it may or may not be emitted, independently of other incarnations; if it is valued, an incarnation may be emitted with any value, independently of other incarnations. In such a case the "`!show signals`" command is clearly not sufficient to track the control flow through all the incarnations. An interpreter which could execute the reaction step-by-step and point back at each step to the corresponding source code instructions would be needed. See Section 8.3 for an example.

The output of the "`!show signals`" command collapses all the incarnations of a given signal in the following way:

- The signal is printed as "emitted" as soon as *one* of its incarnations has been emitted.
- The displayed value is the value of the most recent emitted incarnation, in constructive order, i.e. the one that corresponds to the last declaration entered in the reaction.

The `!show locals` command only prints the local signals:

```
PROG> !show locals;
--- Local: A_LOCAL_SIGNAL
```

The `!show traps` command only prints the exited traps:

```
PROG> !show traps;
--- Trap: ALARM
```

The `!show awaited` command only prints the awaited signals:

```
PROG> !show awaited;
--- Awaited: GO_BUTTON SECOND OTHER_BUTTON
```

7.6.5 Showing Tasks and Execs

Four options show the current statuses of `exec`uted tasks The `!show started` command lists the `exec` statements started in the previous reaction:

```
PROG> !show started;
--- Started: 2
T1 (6, 0) (5, "foo") return R1
T2 (-*-) (5) return R2 : integer
```

The `!show suspended` command lists the `exec` statements suspended in the previous reaction:

```
PROG> !show suspended;
--- Suspended: 1
T3 () () return R3
```

The `!show killed` command lists the `exec` statements killed in the previous reaction:

```
PROG> !show killed;
--- Killed: 1
T4 (2) () return R4
```

Finally, the `!show active` command lists the currently active `exec` statements:

```
PROG> !show active;
--- Active: 3
T1 (6, 0) (5, "foo") return R1
T2 (-*-) (5) return R2 : integer
T3 () () return R3
```

This option is particularly useful since active tasks are not printed by default. The "`!show execs`" command prints all the available information about all the `exec` statements of the module. For example:

```
PROG> !show execs;
--- Started: 2
T1 (6, 0) (5, "foo") return R1
T2 (-*-) (5) return R2 : integer
--- Suspended: 1
T3 () () return R3
--- Killed: 1
T4 (2) () return R4
--- Active: 3
T1 (6, 0) (5, "foo") return R1
T2 (-*-) (5) return R2 : integer
T3 () () return R3
```

7.6.6 The trace Command

The `!trace` command is just a "permanent `!show`" command. A `!show` is executed after each reaction until an `!untrace` command is issued. There are again sixteen basic forms:

```
!trace state
!trace halts
!trace variables
!trace sourcevariables
!trace signalvariables
!trace counters
!trace signals
!trace locals
!trace traps
!trace awaited
!trace started
!trace suspended
!trace killed
!trace active
!trace execs
!trace all
```

The arguments can be put in a single list, as in

```
PROG> !trace state variables;
PROG> !untrace variables signals;
```

In addition, one can type `!trace` alone to display the trace options currently in use:

```
PROG> !trace;
Enabled trace options: variables state
```

7.7 The module Command

When running the simulator, the current module is the first one in the C file. To change the current module, use the `!module` command with the new module name as argument; the prompt changes to indicate the new current module

```
PROG1> !module PROG2;
PROG2>
```

With no arguments, the `!module` command lists all modules that can be simulated:

```
PROG1> !module;
--- Modules: PROG1 PROG2
```

7.8 Simulation Errors

This section lists all the error messages that may appear during a simulation session. Error messages are printed on the standard error stream of the simulator.

7.8.1 Module Error

... `unknown module:` *symbol*

symbol is not the name of a module that can be simulated. Remember that only "root" modules in the module hierarchy can be simulated.

7.8.2 File Error

... `cannot open:` *file*

file cannot be opened; check file names and access rights.

7.8.3 Command Errors

... `unknown show option:` *symbol*

symbol is not a valid `!show` option. The simulator recognizes the following `!show` options: `state`, `halts`, `variables`, `sourcevariables`, `signalvariables`, `signals`, `locals`, `traps`, `awaited`, `started`, `suspended`, `killed`, `active`, and the special option `all`.

... `unknown trace option:` *symbol*

symbol is not a valid `!trace` option. The options are the same as for `!show`.

7.8.4 Input Errors

... `not an input:` *symbol*

symbol is not the name of either an input signal, an inputoutput signal, or a sensor.

... `single signal input more than once:` *signal*

A single signal cannot appear more than once in an input event.

... `signal` *signal* `should have a value of type` *type*

A valued signal is written with no value or with a value of the wrong type.

... `pure signal` *signal* `should have no value`

A pure signal appears with a value.

... `implication violated:` *ImplicationRelation*

The input event violates an implication relation.

... `exclusion violated:` *ExclusionRelation*

The input event violates an exclusion relation.

7.8.5 Variable Access Error

... `variable read before being written:` *variable (comment)*

A variable is read before being written. This points out a serious error in the Esterel program. The simulator will accept no further input unless the user performs a `!reset` command.

... `no input until reset (see previous errors)`

The simulator waits for a `!reset` command before accepting new inputs.

7.8.6 Task Errors

... `Reference argument #`*n* `of task` *task* `(terminated by` *signal*`) should have a value of type` *type*

The value for updating the *n*th reference argument of task *task* is not of the appropriate type *type*.

... `Task` *task* `(terminated by` *signal*`) has` *n* `reference arguments`

The number of values given for updating reference arguments of task *task* does not correspond to the actual number of reference arguments of task *task*.

Chapter 8

Simulation Examples

All the examples described in this chapter can be reproduced by the user on his/her own machine using `xes` and `csimul`, as a good introduction to the use of the Esterel C simulator. We give listings of `csimul` simulation sessions.

8.1 The Counter Example

We start with a small Esterel example whose simulation needs no user-written C code.

8.1.1 The Esterel source program

```
module COUNTER:

input  I: integer, J: integer;
output O: integer;
relation J => I;

signal S := 0 : integer in
   every I do
      emit S(pre(?S) + ?I)
   end every
||
   every 2 I do
      present J then
         emit O(?S + ?J)
      else
         emit O(?S)
```

```
      end present
   end every
end signal
end module
```

Here, `I` and `J` are two integer-valued input signals. The signal `J` can appear only together with `I`, as specified by the implication relation `J => I`.

The first branch of the parallel statement waits for each occurrence of `I` and broadcasts the sum of the values received so far to the other branch using the local signal `S`.

The second branch controls the emission of the output signal `O` every other occurrences of `I`. Its value is the sum of `I`'s values if `J` is not present. If `J` is present the value of `O` is incremented by the value of `J`.

8.1.2 Building the simulator

Assume that the Esterel source program is in file `counter.strl`. The esterel command is invoked with the `-simul` option to produce a simulation-instrumented `counter.c` file This C file is compiled and linked with the Esterel toplevel simulation library `libcsimul.a` to obtain a simulator named `counter`. The `-l` option of the UNIX `cc` command is used to search for and link with the Esterel simulation toplevel library, provided that it has been properly installed.

```
$ ls counter*
counter.strl
$ esterel -simul counter.strl
$ ls counter*
counter.strl counter.c
$ cc -o counter counter.c -lcsimul
$ ls counter*
counter.strl counter.c counter
```

8.1.3 Running the simulator

The simulation session is started by running `counter`. We begin by a blank input event to perform initializations.

```
$ ./counter
COUNTER> ;
--- Output:
```

Then, we input `I` with different values; the signal `O` is output only on even occurrences of `I`. Its value is the sum of all the previous values of `I`.

```
COUNTER> I(1);
--- Output:
COUNTER> I(2);
--- Output: O(3)
```

The `!show variables` option allows us to see the values of all the variables. Notice the "`-*-`" to show that signal `J` is not yet initialized. Variable `V7` contains the implicit occurrence counter for the "`every 2 I ...`" instruction.

```
COUNTER> !show variables;
--- Source Variables:
--- Signal Variables:
V0 = 2  (value of signal I)
V2 = -*-  (value of signal J)
V4 = 3  (value of signal O)
V5 = 3  (value of signal S)
--- Counters:
V7 = 2 [line: 13, column: 13 of file: "counter.strl" (COUNTER#0)]
```

Let us try some other `!trace` options. The `signals` option shows the awaited signals (i.e. those whcih can influence the current reaction), and the local signal emissions.

```
COUNTER> !trace signals;
--- Awaited: I J
```

The signal `I` is mandatory to obtain a non-empty reaction, then `J` can be significant. Let us try a couple of values of `I` with `J` present; `J` is only significant in even occurrences of `I`.

```
COUNTER> I(1), J(2);
--- Output:
--- Local: S(4)
--- Trap:
--- Awaited: I J
COUNTER> I(1), J(2);
--- Output: O(7)
--- Local: S(5)
--- Trap:
--- Awaited: I J
```

Let us try some erroneous inputs (remember that J is integer-valued and can only be present if I is).

```
COUNTER> J("error");
*** Error: signal J should have a value of type integer
COUNTER> J(1);
*** Error: implication violated: J => I
COUNTER> I(1), K(2);
*** Error: not an input: K
COUNTER> =-=-=-=-=
*** Syntax error: last token read: = -- recovery on ‘‘;’’
or EOF
recover> sdfgjhjkfgdshkjdfgs
recover> ;
*** Syntax error recovered on ‘‘;’’
```

Finally we terminate the session and return to the shell

```
COUNTER> .
$
```

8.2 The Watch Example

This more elaborate example involves data-handling code. It is a simplified version of the full wristwatch program to be found in the Esterel distribution.

8.2.1 The Esterel source program

The WATCH module describes a basic timekeeper that maintains a time variable. It reacts to a periodic (quartz) signal SECOND and to two update commands SET_HOUR and SET_MINUTE. The internal time value is broadcast at each change via the output signal TIME. Here, we use a time-holding variable instead of the pre(?S) operator, to give an alternative programming example. The code of the WATCH module is given hereafter:

```
module WATCH:

type TIME;
constant INITIAL_TIME : TIME;
constant ONE_SECOND, ONE_MINUTE, ONE_HOUR : TIME;

% The INCREMENT procedure is used for time arithmetics
```

```
% The RESET_SECONDS procedure sets seconds to zero.

procedure INCREMENT (TIME) (TIME),
          RESET_SECONDS (TIME) ();

% SECOND is the watch internal quartz
% SET_HOUR is the hour update button
% SET_MINUTE is the minute update button

input SECOND, SET_HOUR, SET_MINUTE;

% Broadcasts the updated time value.

output TIME: TIME;

% All external events are supposed to be exclusive

relation SECOND # SET_HOUR # SET_MINUTE;


% We keep the current  time value in the  TIME variable

var TIME := INITIAL_TIME : TIME in
   % Emission of the initial time value
   emit TIME (TIME);
   loop
      await
      case SECOND do
         call INCREMENT (TIME) (ONE_SECOND)
      case SET_MINUTE do
         call RESET_SECONDS (TIME) ();
         call INCREMENT (TIME) (ONE_MINUTE)
      case SET_HOUR do
         call INCREMENT (TIME) (ONE_HOUR)
      end await;
      % Emission of the updated value of  TIME
      emit TIME (TIME)
   end loop
end var
end module
```

8.2.2 The Data-Handling Code

Since the module uses one user-type, the simulation requires the definition of the type and of the required data-handling functions. The `TIME` type is defined in file `watch.h` as a structure:

```
typedef struct
{
    int hours;
    int minutes;
    int seconds;
    int am_pm_flag;
} TIME;
```

Assignment, output conversion, constants and procedures are written in `watch_data.c`. There is no need for input string conversion and checking since we do not input times. Nevertheless, the corresponding functions are defined in the Esterel distribution file `watch_data.c` to serve as examples for the user.

```
#include "watch.h"
/* Assignment */
_TIME(tp, t)
    TIME *tp;
    TIME t;
{
   tp->seconds    = t.seconds;
   tp->minutes    = t.minutes;
   tp->hours      = t.hours;
   tp->am_pm_flag = t.am_pm_flag;
}
/* Output Conversion : pretty-printing TIME objects */
char *_TIME_to_text (time)
    TIME time;
{
   static char outbuf[13]="";
   sprintf(outbuf, "%02d:%02d:%02d::%s",
           time.hours,
           time.minutes,
           time.seconds,
           (time.am_pm_flag ? "AM" : "PM"));
   return(outbuf);
}
```

```
/* Constants and Procedures */
TIME INITIAL_TIME = {1,2,3,0};
TIME ONE_SECOND   = {0,0,1,0};
TIME ONE_MINUTE   = {0,1,0,0};
TIME ONE_HOUR     = {1,0,0,0};
INCREMENT(tp, t)
    TIME *tp;
    TIME t;
{
   tp->seconds += t.seconds;
   if (tp->seconds >= 60) {
      tp->seconds -= 60;
      tp->minutes++;
   }
   tp->minutes += t.minutes;
   if (tp->minutes >= 60) {
      tp->minutes -= 60;
      tp->hours++;
   }
   tp->hours += t.hours;
   if (tp->hours > 12) {
      tp->hours -= 12;
      tp->am_pm_flag = !tp->am_pm_flag;
   }
}
RESET_SECONDS(tp)
    TIME *tp;
{
   tp->seconds = 0;
}
```

One could also inline the definitions of assignment and constants in `watch.h` as explained in Section 4.3.

8.2.3 Building the simulator

The simulator is built from the files `watch.strl`, `watch.h` and `watch_data.c`. The `-l` option of the UNIX `cc` command is used to search for and link with the Esterel simulation toplevel library, provided that it has been properly installed.

```
$ esterel -simul watch.strl
$ cc -o watch watch.c watch_data.c -lcsimul
```

8.2.4 Running the simulator

The user executes the `watch` simulator and enters a simulation session.

```
$ ./watch
WATCH> ;
--- Output: TIME(01:02:03::PM)

WATCH> !trace all;
--- State: 2 of 3
--- Halts: 1
halt 1: line: 35, column: 7 of file: "watch.strl" (WATCH#0)
--- Awaited: SECOND SET_HOUR SET_MINUTE

WATCH> SECOND;
--- Output: TIME(01:02:04::PM)
--- Local:
--- Trap:
--- Source Variables:
V4 = 01:02:04::PM  (source variable WATCH.TIME)
--- Signal Variables:
V3 = 01:02:04::PM  (value of signal TIME)
--- Counters:
--- State: 2 of 3
--- Halts: 1
halt 1: line: 35, column: 7 of file: "watch.strl" (WATCH#0)
--- Awaited: SECOND SET_HOUR SET_MINUTE

WATCH> SET_HOUR;
--- Output: TIME(02:02:04::PM)
--- Local:
--- Trap:
--- Source Variables:
V4 = 02:02:04::PM  (source variable WATCH.TIME)
--- Signal Variables:
V3 = 02:02:04::PM  (value of signal TIME)
--- Counters:
--- State: 2 of 3
--- Halts: 1
halt 1: line: 35, column: 7 of file: "watch.strl" (WATCH#0)
--- Awaited: SECOND SET_HOUR SET_MINUTE
```

```
WATCH> SET_MINUTE;
--- Output: TIME(02:03:00::PM)
--- Local:
--- Trap:
--- Source Variables:
V4 = 02:03:00::PM  (source variable WATCH.TIME)
--- Signal Variables:
V3 = 02:03:00::PM  (value of signal TIME)
--- Counters:
--- State: 2 of 3
--- Halts: 1
halt 1: line: 35, column: 7 of file: "watch.strl" (WATCH#0)
--- Awaited: SECOND SET_HOUR SET_MINUTE

WATCH> !untrace state;


WATCH> SECOND SET_MINUTE;
"stdin", line 13: *** Error: exclusion violated: SECOND
# SET_HOUR # SET_MINUTE

WATCH> !untrace variables;

WATCH> SECOND;
--- Output: TIME(02:03:01::PM)
--- Local:
--- Trap:
--- Halts: 1
halt 1: line: 35, column: 7 of file: "watch.strl" (WATCH#0)
--- Awaited: SECOND SET_HOUR SET_MINUTE

WATCH> SET_HOUR;
--- Output: TIME(03:03:01::PM)
--- Local:
--- Trap:
--- Halts: 1
halt 1: line: 35, column: 7 of file: "watch.strl" (WATCH#0)
--- Awaited: SECOND SET_HOUR SET_MINUTE

WATCH> .
$
```

8.3 Local Signal Reincarnation

We now present the rather intricate example of a program in which a local signal is reincarnated several times in the same reaction. This section may well be skipped at first reading. Its purpose is to highlight the problems in the tracing of local Esterel objects.

```
module M:

input  S;
output O: combine integer with +;

loop
   var X := false : boolean in
      trap T in
         await S do
            exit T
         end await
      ||
         loop
            emit O(1);
            signal L : boolean in
               emit L(X)
            end signal ;
            X := true;
            await S
         end loop
      end trap
   end var
end loop
end module
```

In the first instant, the program initializes variable `X` to `false`, emits `O(1)` and `L(false)`), and sets variable `X` is to `true`. The output event is `O(1)`. The program is then stopped on the two "`await S`" statements:

```
$ ./local
M> ;
--- Output: O(1)

M> !show locals sourcevariables;
--- Local: L(false)
--- Source Variables:
V3 = true  (source variable M.X)
```

Here is a microstep-by-microstep description of how the reaction proceeds upon the next event with `S` present:

- The first branch of the parallel exits `T`.
- In the second branch, `S` causes the termination of the inner loop body, which is immediately restarted. Then `O(1)` and `L(true)` are emitted. Notice that variable `X` is still equal to `true` at this stage. The second branch stops on the new instantiation of the "`await S`" statements.
- The parallel exits the "`trap T`" block. The outer loop is restarted and it resets `X` to `false`.
- The first branch of the new parallel incarnation stops on the "`await S`" statement.
- The second branch emits `O(1)` and `L(false)`.

Therefore, the output event is `O(2)` because of `O`'s combination function. In this reaction, two incarnations of `L` were brought alive, logically simultaneously but causally in succession. One was emitted with value `true`, the other with value `false`.

The output of the simulator is the following one. In the first instant, the program emits `O(1)` and, internally, `L(false)`. Then, as explained above, when `S` is present, two `O(1)` emissions are executed. After combination, this yields `O(2)`. The `!show signals` command only prints `L` as emitted with value `false`, i.e. the last incarnation. The incarnation with value `true` is lost.

```
M> !reset;
--- Automaton M reset

M> ;
--- Output: O(1)

M> !show signals;
--- Output: O(1)
--- Local: L(false)
--- Trap:

M> S;
--- Output: O(2)
```

```
M> !show signals;
--- Output: O(2)
--- Local: L(false)
--- Trap: T
```

Source code debugging in `xes` would also hide the `true` incarnation of `L`.

Chapter 9

Constructive Cyclic Programs

We now present the way in which Esterel v5_91 handles cyclic programs. Here are the three basic facts to remember:

- For Pure Esterel programs, there is no limitation and the class of accepted programs is exactly the class of constructive programs described in [1, 2].
- For plain Esterel programs with values, all constructive programs are correctly handled when using the `-I` interpretation option. If a program is non-constructive for a state and an input, this fact is found at run-time and reported either by a -1 error return code in embedded code or by an error message in `xes` or `csimul` simulation. The `-Icheck` option performs a complete compile-time constructiveness analysis, which guarantees that the program will never encounter a constructiveness error at run-time.
- For plain Esterel programs, the `-causal` option performs the same constructiveness analysis as `-Icheck` and generates sorted circuit code in `ssc` format for a (useful) subclass of programs, composed of programs where the computing actions can be statically ordered without requiring duplication. The other programs cannot be handled yet and must be compiled by the `-I` option, which generates less efficient code.

In the sequel, we detail the different options and the way unsorted circuit code is generated. The basic examples of non-constructive programs presented in [2] can be found in the Esterel v5_91 distribution tape.

9.1 A Non-Trivial Non-Constructive Program

Consider the following Pure Esterel program, also to be found in the distribution tape:

```
module NonConstructive :
input I1, I2;
output O1, O2;
pause;
[
   present [O1 and I1] then
      emit O2
   end present
||
   present [O2 and I2] then
      emit O1
   end present
]
end module
```

The `NonConstructive` program is constructive at first instant, since it only executes "`pause`", but it is non-constructive at second instant if the input signals `I1` and `I2` are both present. In that case, one cannot determine whether the signals `O1` and `O2` *must* or *cannot* be emitted since the program's body reduces to

```
   present O1 then
      emit O2
   end present
||
   present O2 then
      emit O1
   end present
```

which is a typical example of a non-constructive program, see [2].

9.1.1 Checking for Constructiveness

Type the following command:

```
esterel -Icheck non-constructive.strl
```

This prints a message on `stderr` and pops an error box. The error message contains an input sequence that leads to a non-constructive state. Here, the sequence is

```
;
I1 I2;
```

It is formed by the empty event followed by the event where both `I1` and `I2` are present. The input sequence is printed in a format that can be read by the `xes` and `csimul` simulators.

To view the source of the error, click on `Show`. This pops a window which displays the source of the program using a color code that emphasizes the problem just as for option `-I`:

- The keywords written in red foreground identify the (reachable) state in which the error occurs.
- The statements that *must* be executed and the signals that *must* be present are shown on a green background. In particular, the present input signals appear on green background at their declaration point.
- The statements that *cannot* be executed and the signals that *cannot* be present remain on standard background. In particular, the absent input signals remain on standard background.
- The statements and signals for which we cannot prove either *must* or *cannot* are shown on a pink background.

To identify the problem, look at the current state and at the boundary between green and red backgrounds.

For `NonConstructive`, the current state is identified by the red foreground color of "`pause`". The input signals I1 and I2 are shown on green background since they are present, and the first `present` statement is also on green background since it must be executed. The rest is shown on pink background. Click on the `Help` button for a more detailed explanation.

9.1.2 Adding Relations for Constructiveness

In `NonConstructive`, one can forbid simultaneous presence of `I1` and `I2` by asserting the relation `I1#I2`. The program becomes:

```
module Constructive:
input I1, I2;
relation I1 # I2;
output O1, O2;
await tick;
[
   present [O1 and I1] then
      emit O2
   end present
||
   present [O2 and I2] then
      emit O1
   end present
]
end module
```

The constructiveness analysis takes care of relations and the `Constructive` program is found constructive by the `-Icheck` option.

Notice that both `Constructive` an `NonConstructive` are statically cyclic and are rejected by the default sorted-circuit generation option of the `esterel` command. When using the `-I` option, `NonConstructive` will return an error code -1 if fed with a blank event and then with the event "`I1, I2`". In `Constructive`, the relation states that the user guarantees that such a second event cannot be generated. The satisfaction of relations is taken for granted by the generated code. It is not checked at run-time, unless in simulation mode (`-simul` option). *If the run-time input event violates the relation, unspecified behavior can occur.*

9.1.3 Generating Sorted Circuit Code

The `-Icheck` option told us that the code generated with option `-I` is run-time safe and will behave correctly provided that the relations are satisfied. However, this code is not very efficient, and unsorted circuit code should be preferred whenever possible to embed the program. This code is generated as follows:

```
esterel -causal constructive.strl
```

The generated file `constructive.c` can be used just as any other C file generated by the Esterel compiler (for acyclic programs, the `-causal` option only performs a topological sorting just as a standard option). The

`-causal` option is compatible with the other Esterel options. However, option `-causal` is submitted to some restrictions for programs that handle data. These restrictions are described below.

Notice that only the constructivity check is performed by default by the `-causal` option. The check that single signals are emitted only once is additionnally performed if the `-single` option is given to `esterel` command:

```
esterel -causal -single constructive.strl
```

9.2 Data Handling in Cyclic Programs

Plain Esterel involves data handled by valued signals, variables, etc. To explain how valued signals are handled, we first show the translation of a simple Esterel instruction into low-level code. The instruction is

```
   emit X(1)
||
   emit Y(?X+1)
```

The semantics is that `X` is emitted with value 1 and `Y` is emitted with value 2. Clearly, the emission of `X` must precede that of `Y` since `Y` reads the value of `X`. In the low-level (`ic`) code, we add `Updated` and `Access` auxiliary statements to enforce the constructive order. Call `VS` and `VY` the low-level variables that hold the values of `X` and `Y`. The translation is:

```
   VX := 1;
   Updated VX;
   Emit X
||
   Access X;
   VY = VX+1;
   Updated VY;
   Emit Y
```

The law is that an `Access` statement (reader) can be executed only if all `Updated` statement (writers) for the same variable that can be executed in the same reaction have already been executed. This is necessary to ensure uniqueness of the signal values. For example, the following sequence is correct since it respects the constraint:

```
VX := 1;
Emit X;
VY = VX+1;
Emit Y
```

Valued signals can lead to data-dependency cycles that make it impossible to find an execution order. The simplest example is

```
emit S(?S+1)
```

The low-level code is

```
Access VS;
VS := VS+1;
Update VS;
emit S(VS)
```

There is no way to respect the execution ordering constraint. Here, probably, a `pre(?S)` operator is missing, and the correct code should be emit `S(pre(?S)+1)`.

9.2.1 Static vs. Dynamic Ordering

When compiled with the `-I` option, the order in which action execution occurs is found dynamically and may differ for each input and each state. If there is an order that respects control propagation and correct access to signal values, the interpreter will find one. Therefore, all constructive programs can be executed with option `-I`.

However, when compiled with the `-causal` option that generates sorted circuit code, we currently look for a input-independent and state-independent *static ordering* of the data actions and tests in the program. This ordering does not only depend on the program's control, it also depends on data dependencies in actions.

Let us say that two data actions or data tests `A` and `B` are in *potential conflict* if the following conditions are met:

- There is a static instantaneous cycle through control and signals that passes by `A` and `B`.
- Either one action writes a variable that the other one reads or one action is a test.

Clearly, two actions that are not in potential conflict commute and can be arbitrarily ordered. The actions are in *real conflict* when not in one of the following cases:

- *The actions belong to the same computation thread*, which implies that there is no way to execute them in parallel. There are two subcases:
  - One of them can follow the other one instantaneously in the program's control flow, in which case the control flow order must be preserved. Notice that there can be no reciprocal dependency, otherwise there would be an instantaneous loop and the program would have been rejected beforehand.
  - They respectively appear in the `then` and `else` branches of a test, or they are separated by some delay statement, in which case the order can be arbitrary since at most one of them can be executed in an instant.
- *The actions are in different threads but cannot be executed in the same instant*, in which case the order can be arbitrary. If needed, this is checked using a full state reachability analysis, which can be expensive.
- *One of the actions is the emission of a signal and the other one reads the signal's value.* In this case, as seen above, the emitter must appear before the reader.

If these rules are met, then option `-causal` generates correct code. Otherwise, an error message is generated.

9.2.2 A Cyclic Program accepted by sccausal

Here is a fairly tricky program for which `sccausal` can generate code (since this version v5_91). The example is extracted from a real program written at Dassault Aviation.

```
module BAR :
output O : integer;
signal S1 := 1 : integer, S2 := 1 : integer in
   loop
      pause;
      emit O(?S1)
   end loop
||
```

```
      loop
         pause;
         emit S1(?S2);
         pause;
         emit S2(?O)
      end loop
   end signal
   end module
```

There is a cycle through "`emit O(?S1)`", which reads `S1` and writes `O`, "`emit S1(?S2)`", which reads `S2` and writes `S1`, and "`emit S2(?O))`", which reads `O` and writes `S2`. The data actions are respectively `A0 : V0:=VS1`, `A1 : VS1:=VS2`, and `A2 : VS2:=V0`. The actions `A1` and `A2` are in potential conflict since `A1` reads `VS2` and `A2` writes `VS2`. However, `A1` and `A2` cannot be performed in the same reaction since they are separated by `pause` statements. Therefore, their order can be arbitrary in the sorted circuit code. Because of the data dependencies, `A1` must precede `A0` and `A0` must precede `A2`. The sorted circuit code can be generated with the order `A1`, `A0`, `A2`.

9.2.3 Malik's Counter Example

An example of a constructive program that cannot be compiled using `-causal` is an Esterel rephrasing of Malik's example in [4]:

```
module MalikExample :
input I;
input X : integer;
output O : integer;
signal F : integer, G : integer in
   var XF, XG : integer in
      present I then XF := ?X else XF := ?G end;
      emit F(XF/2)
   ||
      present I then XG := ?F else XG := ?X end;
      emit G(XG+1)
   ||
      present I then emit O(?G) else emit O(?F) end
   end var
end signal
end module
```

In the first instant, assume the input is `X=6` and `I`. The low-level computation action order found by option `-I` is:

```
XF := 6;    (VX)
VF := 3;    (XF/2)
XG := 3;    (VF)
VG := 4;    (XG+1)
```

and the result is $4 = (6/2)+1$. Consider now `X=6` with `I` absent. The action order is:

```
XG := 6;    (VX)
VG := 7;    (XG+1)
XF := 7;    (VG)
VF := 3;    (XF/2)
```

and the result is $3 = (6+1)/2$. Since the addition and division can be executed in any order, they cannot be statically ordered as required by sorted circuit code and only the `-I` option can handle the program. Generation of sequential code would require some action duplication, which is not yet implemented.

When called with the `-causal` option on `MalikExample`, the Esterel compiler prints the following error message:

```
*** sccausal: Cyclic dependency between read/write actions:
        - malik.strl, line 11, col 22, MalikExample#0;
        - malik.strl, line 12, col 7, MalikExample#0;
        - malik.strl, line 8, col 36, MalikExample#0;
        - malik.strl, line 9, col 7, MalikExample#0;
        - malik.strl, line 11, col 22, MalikExample#0;
*** sccausal: Error #25.4.5:
        Can not find a static order for these read/write
actions.
```

9.2.4 An Example With Tests

The following example involve data tests:

```
module CyclicActions :
input I;
signal S1, S2 in
   present I then
      emit S1
   else
      emit S2
   end
||
   present S1 then
      if true then emit S2 end
   end
||
   present S2 then
      if true then emit S1 end
   end
end signal
end module
```

This example is somewhat artificial because of the `true` tests, but it also illustrates the fact that the contents of data action is not taken care of in the analysis. A meaningless `true` test is handled symbolically, exactly as a meaningful '`X>0`' test, since no partial evaluation is performed on data.

Here, there is a cyclic dependency between the control signals `S1, S2`, but the program is constructive because the presence test for `I` always cuts the cycle in one way or another. This is easily checked using option `-I`.

However, the data actions that compute the test values cannot be statically ordered. Each of them depends on the other one by an instantaneous path. For example, the second test depends on the first one through the `then` part of the first test, the emission of `S2`, and the test for `S2`. Therefore, the second test should be executed after the first one. Conversely, the first test should be executed after the second one because of `S1`, and option `-causal` cannot generate code. Here also, more clever action handling would be necessary. The error message printed by the Esterel compiler is:

```
*** sccausal: Error: Cyclic dependency between data test
actions:
        - CyclicActions.strl, line 13, col 7, CyclicActions#0;
        - CyclicActions.strl, line 17, col 7, CyclicActions#0;
        - CyclicActions.strl, line 13, col 7, CyclicActions#0;
*** sccausal: Error #25.4.5:
        Can not find a static order for these data test
actions.
```

Bibliography

[1] G. Berry. *The Esterel Language Primer, version v5_90.* Available from http://www.inria.fr/meije/esterel, 2000.

[2] G. Berry. *The Constructive Semantics of Esterel.* Available from http://www.inria.fr/meije/personnel/Gerard.Berry.html, Version 3, June 1999.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[4] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer–Aided Design*, 13(7):950–956, 1994.

[5] E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. ICCAD'96*, 1996.

[6] E. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proc. DAC'97, Anaheim*, 1997.

[7] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. International Design and Test Conference ITDC 96, Paris, France*, 1996.

Index