Constructive Analysis of Esterel Programs
The -Icheck Option of the Esterel v5 Compiler

The Esterel Team
esterel-request@sophia.inria.fr
http://www.esterel.org

May 17, 2000

Contents

1 Introduction

The `-Icheck` option of the Esterel v5 compiler makes it possible to check Esterel programs for two essential properties:

1. The constructiveness of the reaction for any reachable state and any valid input.
2. The respect of single signal emission constraints for any reachable state and any valid input.

To perform the checks, just type

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

or any other valid `esterel` command line containing the `-Icheck` option.

The option first performs an exact constructive causality analysis of an Esterel program with respect to the constructive Esterel semantics defined in [1]. This semantics is used in the Esterel v5 compiler when programs are compiled with the `-I` option. If a program passes the check, then the C code generated by option `-I` will never detect a causality problem at run-time, and the C reaction function will never return the error code -1. If the program is found to be non-constructive, then this fact is reported, a counter-example input sequence is given, and the problem is displayed graphically on the source code as for option `-I`.

The option also checks for single emission of single signals. This check was available for Esterel v3 but it had disappeared in Esterel v4. It is now available again in Esterel v5 (`-single` option). If a single signal can be emitted more than once at a given instant, then the error is reported, a counter-example input sequence is given, and a graphical error message is displayed.

The `-Icheck` option only performs a check on the program. No code is generated. Beware: the `-Icheck` option uses a state reachability analysis technique that is expensive in computation time and space. The algorithm is described in [2].

The `sccheck` processor called by the option is based on the TiGeR Binary Decision Diagram library, which is a property of Digital Equipment Corp. and is distributed by the XORIX company. Please read the attached copyright license.

2 Examples

All the basic examples of non-constructive programs are given in [1]. They can be found in the distribution directory. Try them yourself.

2.1 A Non-Trivial Non-Constructive Program

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

```
module NonConstructive :
input 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 `NonConstructive` program is constructive at first instant, since it only executes "`await tick`", 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 boils down 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 [1].

Type the following command:

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

(`esterel` is assumed to refer to `esterelv5_90` or later). This prints an input 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 Esterel simulators.

To view the error proper, 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 instruction "await tick". 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.

2.2 Relations and 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 provoke a run-time error 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).

2.3 Constructiveness and Data-Paths

In Esterel, data tests and actions are controlled by signals. In a constructive cyclic behavior, the set of data tests and actions have to be ordered in a deterministic way. For a program that is checked constructive for its control part, it is checked whether the set of actions appearing in each cycle can be ordered or not.

We consider a simple example for which the control part is constructive but a cyclic dependency exists between actions so that they can not be ordered:

```
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
```

In this example s cyclic dependency between the control signals `S1, S2`. In fact, there is no reachable states for which there is an input event that lead to an actual cyclic dependency. Indeed, the dependency is broken by the presence status of the input signal `I`: if `I` is present, then `S1` is emitted; so `S1` does not depend on `S2`, while `S2` depends on `S1`. When `I` is absent, the reverse situation happens. The control part is thus constructive. However, there is a cyclic dependency between the data test actions performed in the bodies of the two lasts `present` statements. Actually, there is no way to order the two data test actions. Hence, this program is rejected by the compiler. The error message that is printed out on `stderr` lists the list of data action tests for which such a cyclic dependency exists.

3 Checking Single 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 by the `-Icheck` option, only for programs previously found to be constructive.

The following program does not respect the single 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 will be displayed, one for each signal that violates the single property. As for constructiveness checking, each counter-example is composed of an input sequence and of a graphical presentation of the execution path that leads to the error.

As before, one can make this program correct by adding the relation

```
relation I1 # 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.

4 Exploiting the Symbolic Input Sequences

With each error is associated an input sequence printed on `stderr`. For pure signal programs without counters, the sequence is directly usable as an input to simulators generated using the `libcsimul.a` and `libxsimul.a` libraries. If the program handles values, the input sequence is a *symbolic* one: it includes no values for valued input signals, and it also contains strange names that correspond to results of tests executed in the reaction. Tests are performed by `if` statements or `repeat` statements.

In a future release of the simulation libraries, you will be able to replay symbolic sequences in the simulators. Until this is available, proceed as follows if you really need to know which test a name characterizes:

- Build the `lc` code, using option `-lc` or option `-Klc` of the `esterel` command.
- Take the first number in the name, for example 30 in `Ift_30_5_`.

- Find the statement numbered “`30:`” in the `lc` code. It should be a `Test:` statement. At the end of the line, you will find a `lc:` source pragma of the form `%lc:` *l c i*`%`. Then *l* is a line number, *c* is a column number, and *i* is a module instance number.
- Find entry *i* in the module instance table, which is the first table in the `lc` module. Here you find the source file name *f* and possibly a directory name *d* that tell you where to find the source file.
- The source test is to be found in directory *d*, file *f*, line *l*, column *c*.

A test that prints a name in the input sequence takes its `then` branch. A test that prints no name takes its `else` branch if it is executed at all in the reaction.

5 Additional Options

The options common to all Esterel processors are available:

- **-version**
  Print the version name on the standard error output stream and terminate, ignoring all other arguments.
- **-info**
  Display various information about the current `sccheck` processor on the standard error output stream and terminate, ignoring all other arguments.
- **-v**
  Verbose mode. Explain what is happening (for experts).
- **-access**
  Print access rights to the processor on the standard error output stream and terminate, ignoring all other arguments.

If `sccheck` exhausts the available memory (that you should set to the maximum available, for example under Unix using the `ulimit` command), TiGeR outputs a failure message. In that case, try the following option:

-sift Enables dynamic reordering of the BDD variables when performing BDD operations. This option reduces the size of the memory used and the number of BDD node labels involved, but it increases the time of the computation.

To pass these options to `sccheck`, use the standard `esterel` command syntax:

```
esterel -Icheck:"-sift -v" foo.strl
```

6 The `sccausal` Code Generator

The `sccheck` processor is in fact a shell script that calls the `sccausal` processor that can also perform sorted circuit (`ssc`) code generation for constructive programs. Try for example

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

This command performs the same causality analysis as `sccheck` and produces `ssc` and `C` code exactly as for acyclic programs. The resulting C program can be used just as any other C program generated by the Esterel v5 compiler. The `-causal` option can be combined with other options of the `esterel` command as usual.

Technically, the `-causal` option tells the `esterel` compiler to replace the standard `scssc` circuit sorting processor by `sccausal`. The `sccausal` processor first tries a topological sorting, just as `scssc`. It the sort succeeds, the code is directly generated. Otherwise, if the program is found constructive, `sccausal` replaces the original cyclic circuit by an equivalent acyclic circuit obtained by direct implementation of the output and register BDDs using gates. As far as circuit's size is concerned, this is not the best way of making the circuit acyclic. We are currently working on more elaborate cycle removal techniques.

In its current state, the `sccausal` processor may fail generating code if a strongly connected control component contains data actions that can not be ordered (share variables in read/write mode, or dependencies like in the example of section 2.3). An error message is then printed. We give no more explanation here, contact us if you are confronted in such a situation and/or if you want the details.

Warning: by default, the `-causal` option does not check for single emission of single signals. To perform the check, either use the `-Icheck` option first, or pass the `-single` option to `sccausal` by typing

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

References

[1] G. Berry. *The constructive semantics of Esterel.* http://www.esterel.org/, 1996.

[2] T. Shiple and G. Berry. Constructive analysis of cyclic circuits. In *Proc. ITDC'96, Paris*, 1996.