

POC

Pascal Raymond
Verimag
Centre Equation, 2 rue de Vignate
38610 - Gieres, France
e-mail: Pascal.Raymond@imag.fr
home-page: <http://www.imag.fr/VERIMAG/PEOPLE/Pascal.Raymond>

December 2, 2025

POC is a pretty printer for “oc” code (Lustre-Esterel object code). Moreover, the output format is (mostly) standard ansi c, so `poc` can be used as an alternative tool for the “occ” code generator¹.

The code generated by `poc` consists essentially in a procedure implementing a step of the reactive program described in the `oc` file. In order to run the reactive program, the user must write his own main loop around this “step” procedure.

The following sections precise what is generated by `poc`, and what the user must write to use it. Let us call `toto.oc` the source file, `TOTO` the `oc` module defined in this file, `toto.c` and `toto.h` the files generated by `poc`, and `loop.c` the main program written by the user.

Contents

1	Compilation into Ansi C	2
1.1	Execution context	2
1.2	Step procedure	2
1.3	Inputs and outputs	2
1.4	A simple example	3
1.5	External objects	3
1.6	Standard main loop	4
2	Pragmas	4
2.1	Pragmas in the Ansi C header file	4

¹this tool can be found on the Esterel web site <http://zenon.inria.fr/meije/esterel/>

1 Compilation into Ansi C

1.1 Execution context

Unlike `occ`, the code generated by `poc` allows multiple allocations of a reactive module. The memory needed for an instance of the reactive module is defined in `toto.c`, and declared in `toto.h`:

```
struct TOTO_ctx;
```

The user does not have to know what this structure is made of, he is only allowed to manipulate pointers. The user can get a new context using a procedure declared in `toto.h`:

```
struct TOTO_ctx * TOTO_new_ctx(void* client_data);
```

The user can associate “whatever he wants” to a new execution context using the `client_data` argument. This information is necessary if the user wants to run concurrently several instances of a same reactive module, as it is explained in 1.3.

1.2 Step procedure

The procedure implementing a step of the reactive module is declared in `toto.h`:

```
void TOTO_step(struct TOTO_ctx * ctx);
```

This procedure is called with an execution context previously created by a call of `TOTO_new_ctx`. This step procedure has no input/output parameters, since communication between the main loop and the reactive module are made via input/output procedures. More precisely, the user must call input procedures to set the input values before he calls the step procedure. The step procedure calls output procedures to send its outputs to the environment.

1.3 Inputs and outputs

Communications between `toto.c` and `loop.c` are made via input and output procedures. The input procedures are defined in `toto.c` and used in the main loop, the output procedures are defined by the user (in `loop.c` for instance), and used in `toto.c`.

Inputs For each input `IN`, of type `TYP`, `toto.c` contains the definition of the procedure:

```
void TOTO_I_IN(struct TOTO_ctx* context, TYP value);
```

Note that if `IN` is a pure signal, the `value` parameter is omitted:

```
void TOTO_I_IN(struct TOTO_ctx* context);
```

Outputs For each output `OUT`, of type `TYP`, the user must define a procedure:

```
void TOTO_O_OUT(void* client_data, TYP value);
```

Note that if `OUT` is a pure signal, the `value` parameter is omitted:

```
void TOTO_O_OUT(void* client_data);
```

Output procedures are called within the `TOTO_step(TOTO_ctx* ctx)`, using the client data which has been associated to `ctx` when it was created.

1.4 A simple example

Here is a (quite stupid) example of a main loop using an `oc` (whose name is `sum`) module with two real inputs (`x` and `y`) and a single real output `s`; note that the “client data” is not necessary, since the loop uses only one instance of the reactive module:

```
#include <stdlib.h>
#include "sum.h"

void sum_O_s(void* cdata, _float _V){
    printf("result: %f\n", _V);
}

main(){
    _float x;
    _float y;
    struct sum_ctx* prg = sum_new_ctx(NULL);

    while(1){
        printf("(float) x ?\n");
        scanf("%f", &x);
        sum_I_x(prg, x);
        printf("(float) y ?\n");
        scanf("%f", &y);
        sum_I_y(prg, y);
        sum_step(prg);
    }
}
```

Let `sum.oc` be the name of the `oc` file, and `loop.c` be the name of the main program, just type:

```
poc sum.oc
gcc sum.c loop.c
```

1.5 External objects

All the external objects declared in the `oc` program are supposed to be implemented in `c` by the users. Some informations are necessary for the compilation of the `c` code generated by `poc`, while others are only necessary for linking.

Compiling The `toto.c` program generated by `poc` cannot be compiled unless the external types are defined. `poc` supposes that those definitions are in a file called “`toto_ext.h`”.

Linking External constants, functions and procedures are declared in `toto.c` as imported objects, so `toto.c` can be compiled separately. Indeed the user must define those objects somewhere, and link the corresponding code with the `poc` code if he wants to build a executable program!

1.6 Standard main loop

When it is called with the `-loop` option, `poc` produces an extra c-file `TOTO_loop.c`. This code contains a main procedure implementing a loop which reads inputs on `stdin` and write outputs to `stdout`. If the `oc` program does not need external object, it is a simple way to obtain executable code. For instance:

```
poc sum.oc -loop
gcc sum.c sum_loop.c
```

produces a interactive `a.out` program, which allows the user to test his code.

If the `oc` program needs external objects, the user must write all the necessary code (as it is explained in 1.5), plus two procedures for each external type TYP:

- `TYP _get_TYP(char* name)` reads a value of type TYP on `stdin`, and returns it. The argument `"name"` is the name of the input, it can be used to make the procedure more "interactive".
- `void _put_TYP(TYP val)` prints the value `val` on `stdout`.

2 Pragmas

In order to allow automatic manipulation of the code generated, `poc` generates special comments (pragmas) in the generated files.

2.1 Pragmas in the Ansi C header file

Pragmas are single line Ansi C comments, beginning with the string `"poc:"`.

Node name Let `foo` be the name of the reactive module, the pragma giving this information is:

```
//poc:MODULE foo
```

Inputs For each input `nme` of type `typ`, `poc` generates the pragma:

```
//poc:IN typ nme
```

Outputs For each output `nme` of type `typ`, `poc` generates the pragma:

```
//poc:OUT typ nme
```

Indeed, it is much more simple to parse this kind of pragmas than real "C" code, so one can more easely build automatic tools using the code generated by `poc`.