The Stateflow to Lustre Translator User's Manual

Norman Scaife
Laboratoire VERIMAG, Centre Equation,
2, avenue de Vignate, 38610 GIERES, France
Norman.Scaife@imag.fr

January 18, 2005

SF2LUS Version 0.01b

`Document Revision: 1.5`

Contents

1 Introduction

The tool SF2LUS is an (as yet) *ad hoc* translation from a subset of STATEFLOW[1] [1] into the synchronous language LUSTRE [2]. The technical details of the translation can be found in the Verimag technical report [3] and in the paper published in EMSOFT'04 [4].

Here the SF2LUS program is described which is the implementation of the translator in Objective Caml. This is designed to work with the S2L translator [5] which translates a discrete-time subset of SIMULINK into LUSTRE. This package provides a script called SS2LUS which calls the SF2LUS translator to extract the STATEFLOW component as LUSTRE nodes and provides S2L with the information needed to incorporate the STATEFLOW LUSTRE code into the output.

The SF2LUS program can be used in isolation, however, to extract and test the STATEFLOW components independently of the SIMULINK. This requires a slightly different interface since constants and data imported from SIMULINK or MATLAB have to be defined locally and emulation is required for other features such as the time variable `t`.

Output from the tool is primarily LUSTRE V4. This version of LUSTRE includes bounded iteration using the `when` construct in conjunction with statically-evaluated constants and proved very useful in structuring STATEFLOW's unbounded iterations such as event broadcasts and junction loops. In general, however, it is intended that SF2LUS be used in conjunction with the planned STATEFLOW analysis tool to allow the elimination of unbounded recursion either by automated transformation or by manual editing of the STATEFLOW chart.

Other outputs include preliminary versions of SCADE [6] which allows STATEFLOW charts to be included as source code into the SCADE suite of tools by *Esterel Technologies, Inc.*, RELUC which is a commercial version of LUSTRE also by *Esterel* and some minor support required by the abstract interpretation tool NBAC (via LUS2NBAC provided in the LUSTRE distribution). These output formats are only partially supported, however. Note that the LUSTRE output itself can be used as input to a variety of tools including the model-checker LESAR [7].

[1] Trademark of the MathWorks inc.

2 The sf2lus translator

2.1 A simple worked example

Consider the very simple Stateflow chart shown in Figure 1[2].

Assume that the MATLAB model file is SetReset_r13.mdl. Running the translator in isolation on this model extracts the STATEFLOW component and generates a LUSTRE node for each top-level STATEFLOW chart in the model.

```
% sf2lus SetReset_r13.mdl -o SetReset_r13.lus
```

The LUSTRE nodes are called `sf_<CID>` where `<CID>` is the `id` field in the STATEFLOW `chart` entry. This node looks as follows, noting that the first STATEFLOW chart in the model almost always has number 2:

```
-- graph id=11 name=22,GCTOP
node sf_2(Set, Reset: event) returns(x: int);
var ini, lv5, lv5_1, lv6, lv6_1, lv7, lv7_1, ok22, ok22_1, ok22_2, ok22_3,
    s3, s3_1, s3_2, s3t, s4, s4_1, s4_2, s4t, trm: bool; x_1, x_2: int;
let
...
```

This is valid LUSTRE code and can be compiled with LUSTRE:

```
lustre SetReset_r13.lus sf_2 -o SetReset_r13.oc -0
poc -loop -o SetReset_r13.c SetReset_r13.oc
gcc -c SetReset_r13.c -o SetReset_r13.o
gcc -c SetReset_r13_loop.c -o SetReset_r13_loop.o
gcc SetReset_r13.o SetReset_r13_loop.o -o SetReset_r13
```

The resulting program can be run:

```
% SetReset_r13
##### STEP 1 ##################
Set (true=1/false=0) ? 0
Reset (true=1/false=0) ? 0
x = 0
##### STEP 2 ##################
Set (true=1/false=0) ? 1
Reset (true=1/false=0) ? 0
x = 1
##### STEP 3 ##################
Set (true=1/false=0) ? 0
Reset (true=1/false=0) ? 1
x = 0
```

Alternatively, we can simulate the code with LUCIOLE:

[2] This model can be found in the "`tests`" directory in the distribution

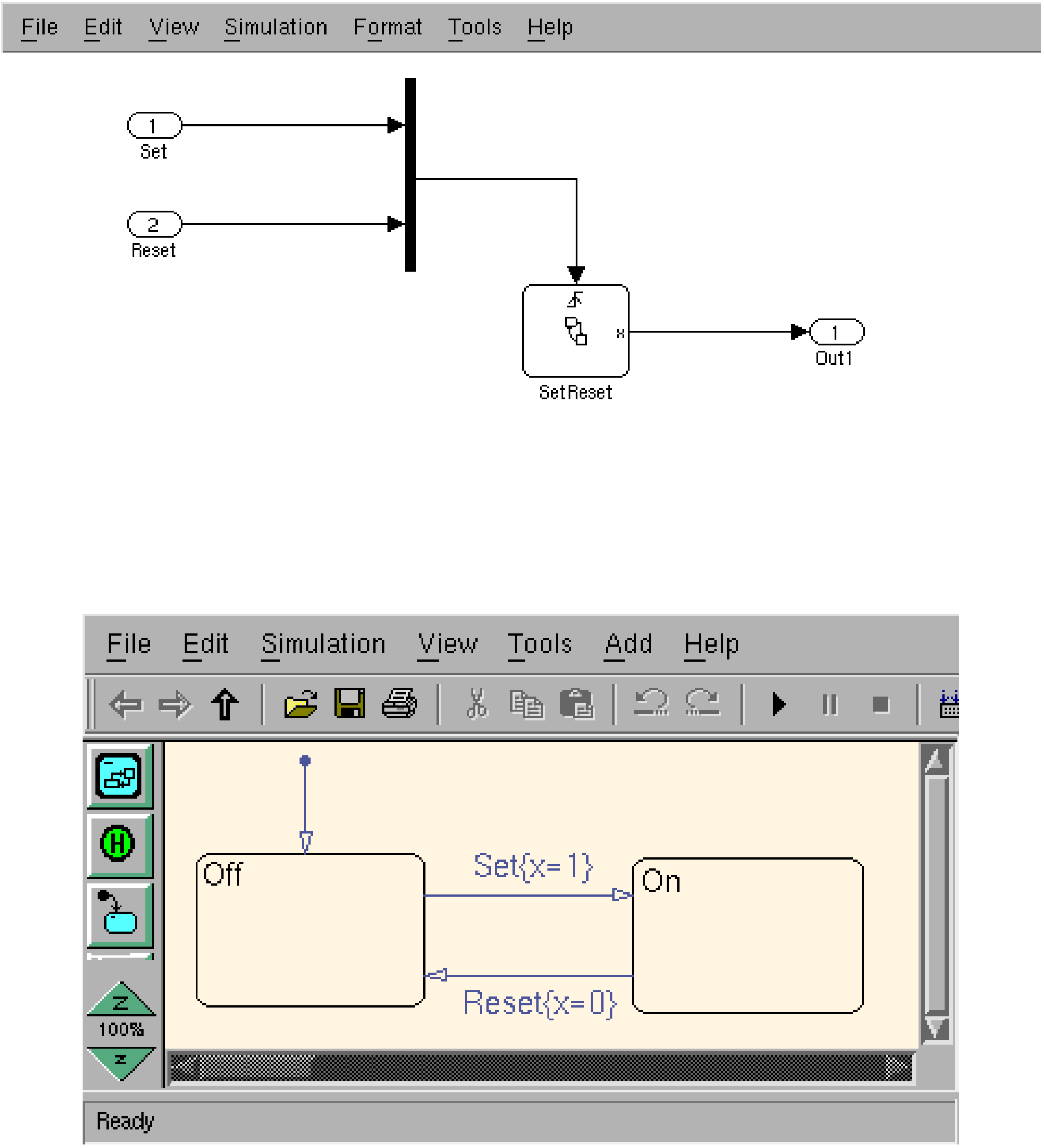
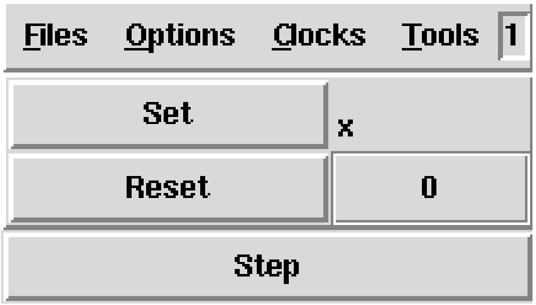


Figure 1: A simple Simulink/Stateflow chart

```
% luciole SetReset_r13.lus sf_2
```

resulting in something like the following:



2.2 Debugging Stateflow with sf2lus

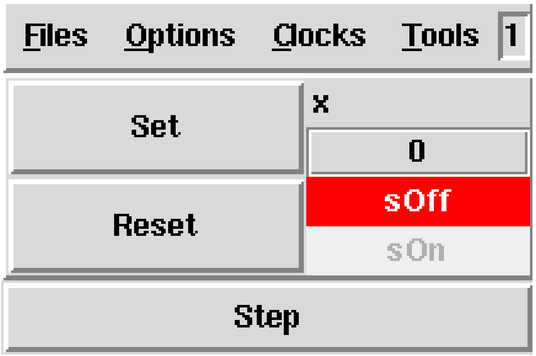
While this is the required behaviour from the Lustre code it does not say much about the internal state of the running Lustre program. Neither are the generated variables particularly meaningful since states are represented simply by their `id` numbers.

2.2.1 Viewing the internal state

If we wish to use sf2lus as a means of debugging the Stateflow component of a Simulink application then it might be useful both to give sensible names to the variables and to export some of the internal state of the chart, for example the state variables. We can do this as follows:

```
% sf2lus -names -states_visible SetReset_r13.mdl -o SetReset_r13.lus
```

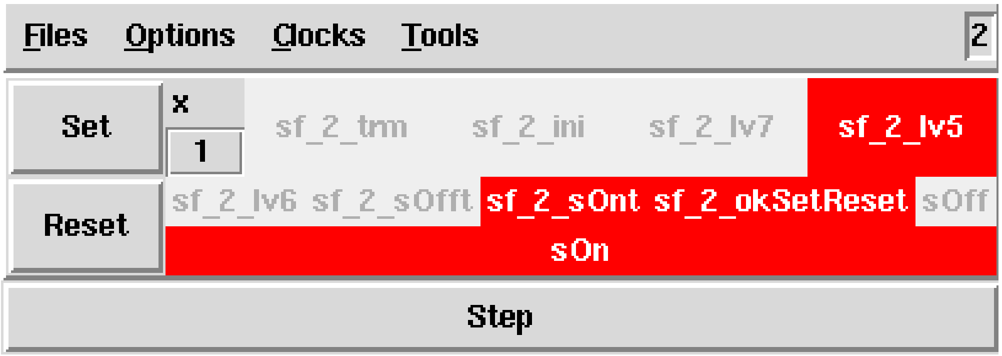
resulting in the following Luciole display:



Now we can see which states are active. State names are based on the node names in Stateflow preceded by `s` for a terminal state and `sg` for a state containing substates when there is hierarchy or parallel states. Note that these are abbreviated names so, for example, “state” is shortened to “s” and “subgraph” to “sg”. Use the `-long_names` option to generate the full names which results in voluminous output even for small charts. Note also that with `-long_names` the chart names become `stateflow_<CID>`.

2.2.2 Even more detailed output

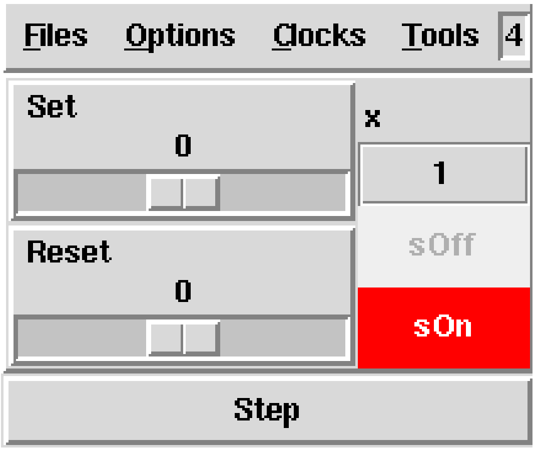
In fact, we can go further using the `-temps_visible` or `-locals_visible` options which attempt to dump the entire internal state of the working chart[3]. For example:



For this simple chart the output is small but the output can become prohibitively large for even modestly-sized charts. The most useful elements of this display are the transition validity flags (for example `sf_2_lv5`) which indicate which transition was traversed on the previous step.

2.2.3 Setting multiple boolean inputs with Luciole

One problem with Luciole is that it is not possible to set more than one boolean input simultaneously so sf2lus provides an option `-input_bools_ints` which transforms input booleans into integers. Note that this does not work for input events but Stateflow processes one event at a time so this restriction is not relevant unless, for example, one is model-checking the code. In any case, events can be turned into integers using the `-sends` option discussed in Section 3:



[3] At present this process is incomplete due to problems with event handling code. These options will work more fully when sf2lus has evolved further.

3 Events

The current version of SF2LUS has support for event broadcasts. This feature, however, is in a state of flux engendered by the implementation of inter-level transitions. Event broadcasting should not be used in conjunction with inter-level transitions (and are a bit "buggy" even with confluent events).

The intended uses of SF2LUS are such that using event broadcasts is not recommended and charts should be transformed into ones which do not require broadcasting. This results in safer code (no unbounded behaviour), if sometimes much less readable charts.

3.1 Event broadcasting

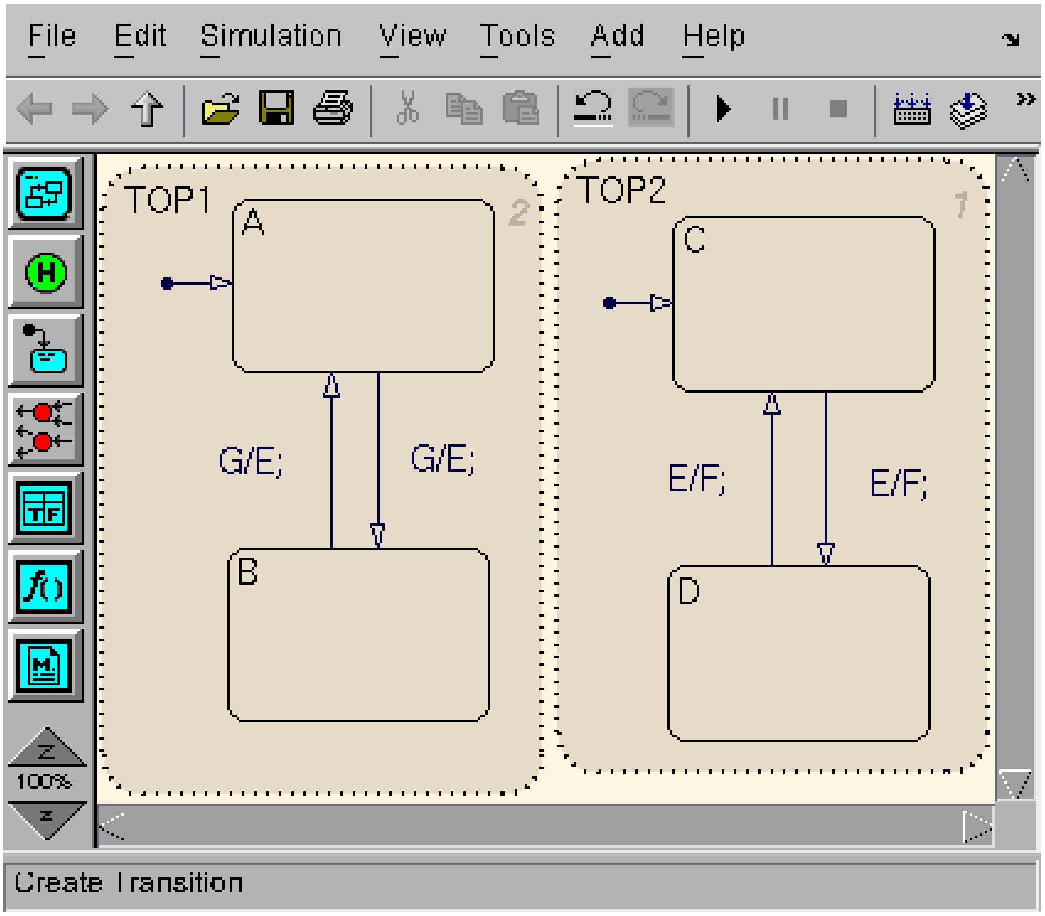
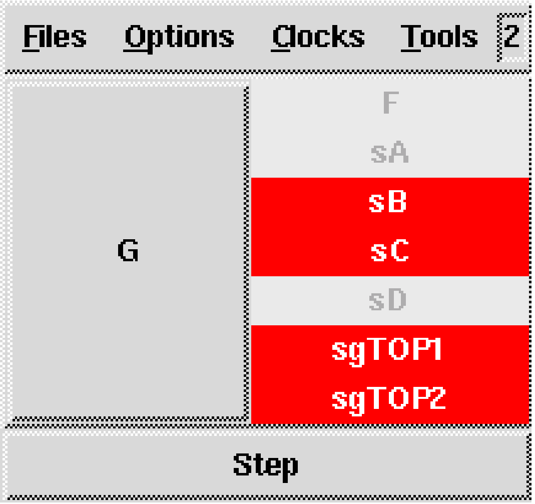


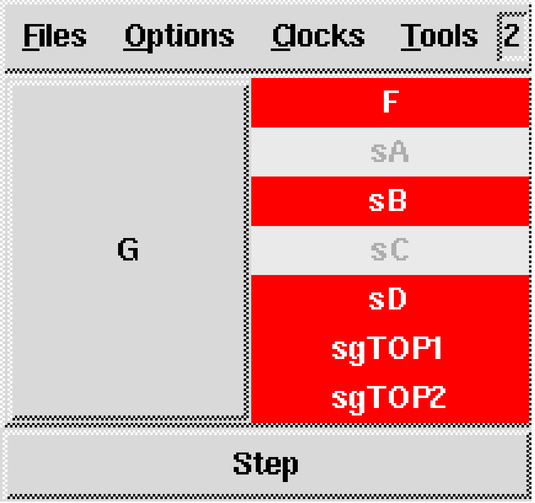
Figure 2: A Stateflow chart requiring event broadcasts

Figure 2 shows a STATEFLOW chart (`Events3.mdl`) with two parallel states `TOP1` and `TOP2`. Here, state `TOP2` receives events from `TOP1` but is executed *before* `TOP1` according to the priorities in the chart. If we naïvely translate this chart and send event `G` to it we get:



State A has exited and state B has entered in subgraph TOP1 but subgraph TOP2 has not received event E and stayed in state C.

To enable event broadcasts we use the -ess <n> option where <n> is the depth of the event stack we require. For this chart we can set the event stack size to 2 and we get the following:



This time event E has been sent to state TOP2 resulting in emission of the local event F. Bear in mind that this mechanism is statically implemented in Lustre so each event broadcast results in duplication of the entire chart at that point, *up to the event depth*. This results in huge code and Lustre soon runs out of resources to implement the expansion. In practice, the -ess parameter should not be set to more than about 4. Charts which require event stacks deeper than this should be redesigned.

3.2 Event sending

Stateflow's send facility for targetting an event at a specific state is difficult to implement in Lustre. In an imperative environment such as Stateflow this can be implemented simply by calling a function which implements the behaviour of the state when sending the event. This does not work in LUSTRE since it may result in dependency cycles. A partial implementation has been achieved which behaves in a similar manner to STATEFLOW by turning events into integers.

An event of 0 is inactive, an event of 1 is broadcast and any other number refers to an event targetted at the state with that `id`. All events are then broadcast but only relevant states action the event (either the event is 1 or it or one of the states's parents has the same `id` as the event). The only problem here is that passive states during the send are not completely switched off (this would require inordinate numbers of guards in the code) so during actions get executed event if the state is not targetted by the send.

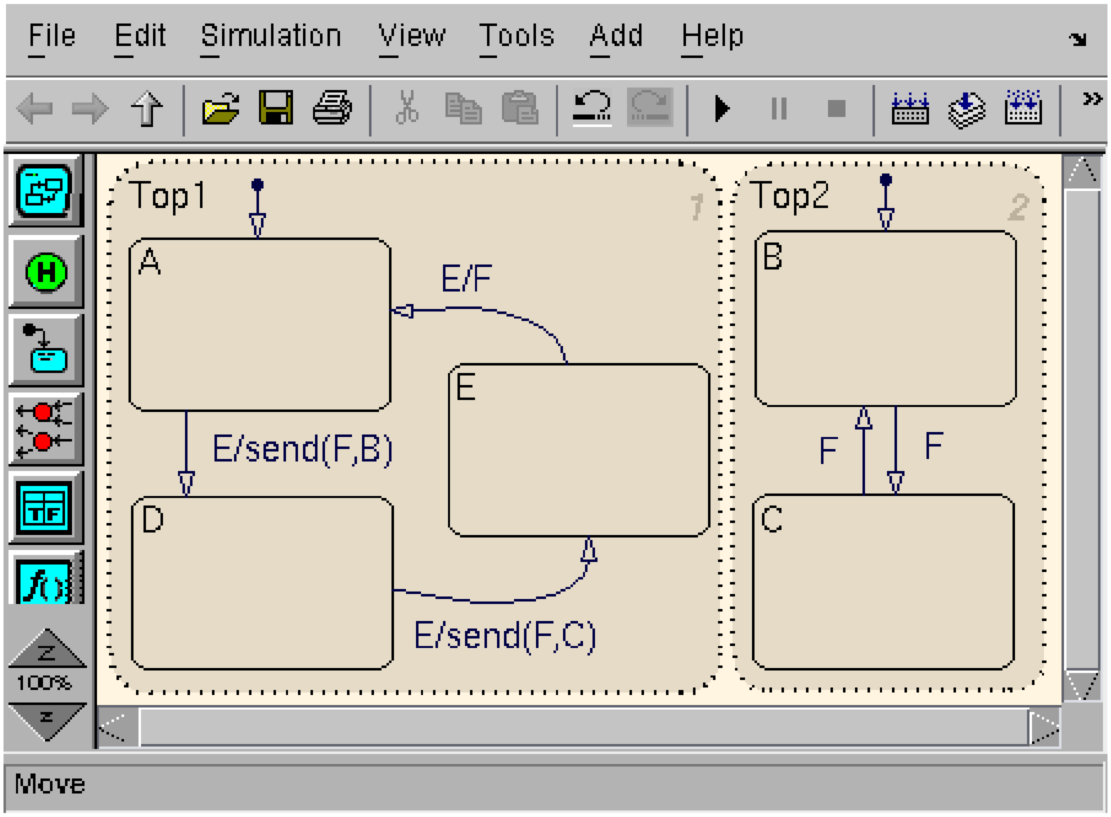
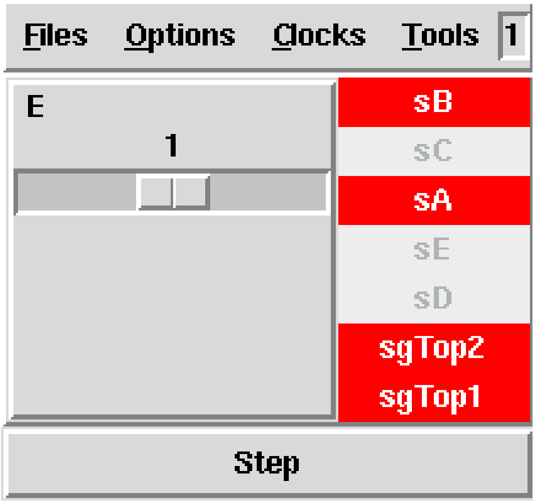


Figure 3: A chart requiring event sending

To trigger this mechanism use the `-sends` option. Figure 3 shows a chart which uses sends (`Events5_docs.mdl`), the resulting LUCIOLE display is as follows:



Note that we now have to set the event E to 1 to indicate a broadcast[4].

[4] The external interface to sent events will change in the near future to allow compatibility with SS2LUS.

4 Junction loops

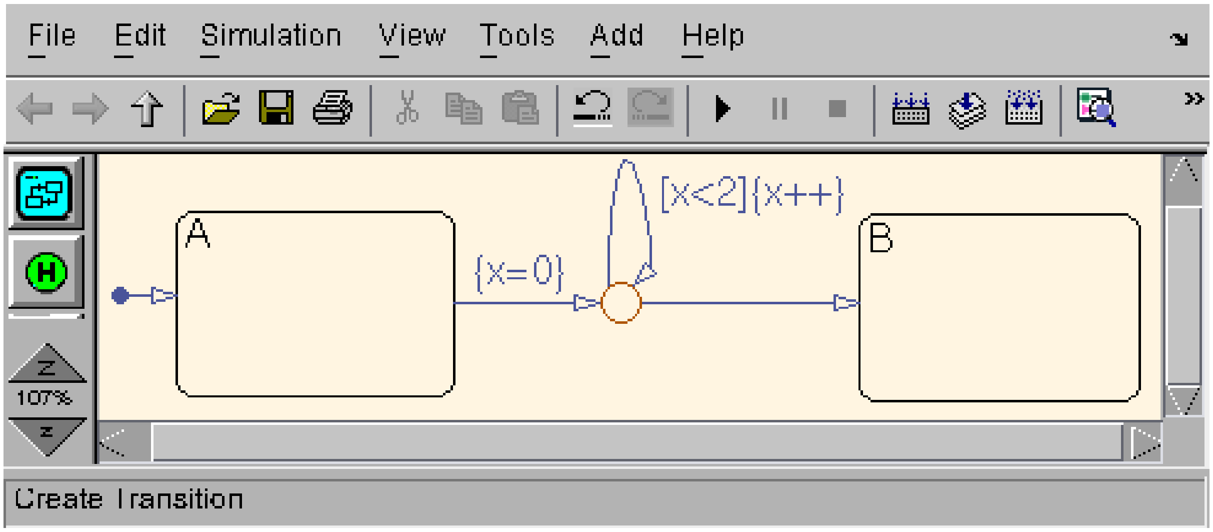


Figure 4: A Stateflow chart with a junction loop

The STATEFLOW chart shown in Figure 4 (Loops1_docs.mdl) with an observer has a junction loop, the transition labelled `[x<2]{x++}` initiates a loop in the network. Strictly speaking, we cannot arbitrarily translate this chart into LUSTRE as it stands since LUSTRE is compiled into a stackless machine and without a stack it is not possible to implement arbitrary recursion *within a reaction*. If we try to translate this chart we get:

```
% sf2lus Loops1.mdl -o Loops1.lus
Fatal error: exception Failure("Loops found in links")
```

Detecting loops is trivial. There are two possibilities:

1. We could give the junction itself a state so that the loop then occurs across synchronous reactions rather than within one.

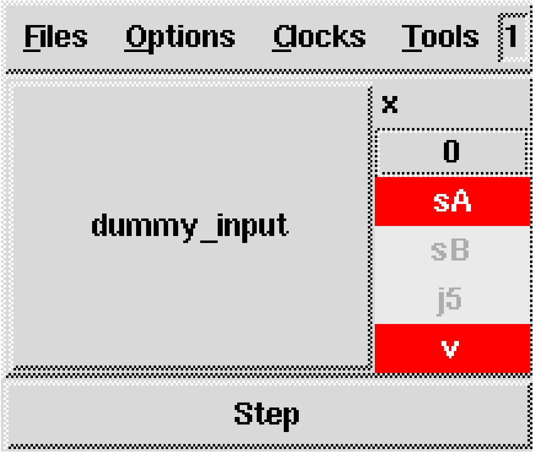
2. If we know the loop is bounded, as is the case here, we could *unfold* the loop into an equivalent one without a loop.

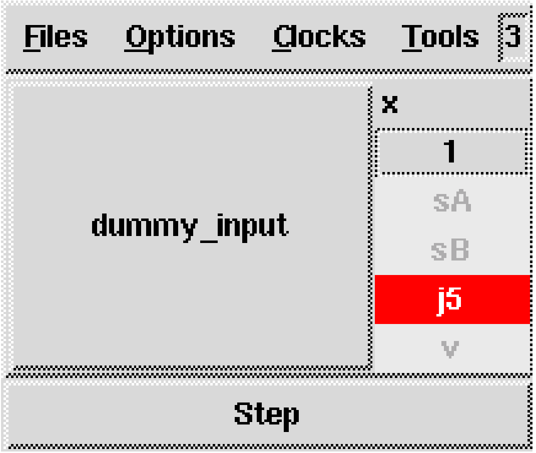
4.1 Junctions as states

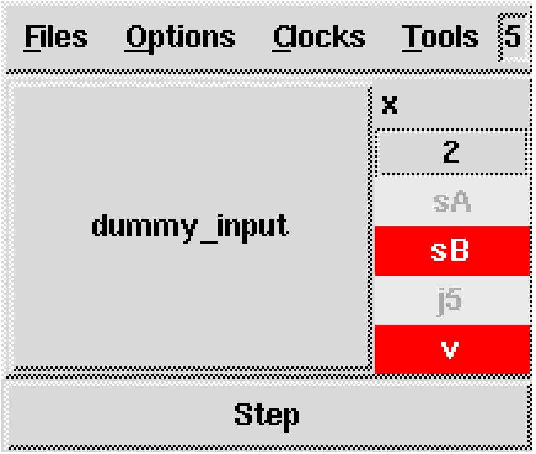
The first option is easily implemented in the translator using the `-junc_states` option:

```
% sf2lus -names -states_visible -junc_states Loops1.mdl -o Loops1.lus
```

This results in the following sequence when simulated with LUCIOLE:







An extra output `v` has been synthesized (or `valid` with `-full_names`) which indicates when the chart is in a state and not a junction. In theory arbitrary recursion could be implemented this way, passing the burden of termination to the client code but it is not a very satisfactory solution.

4.2 Loop unrolling

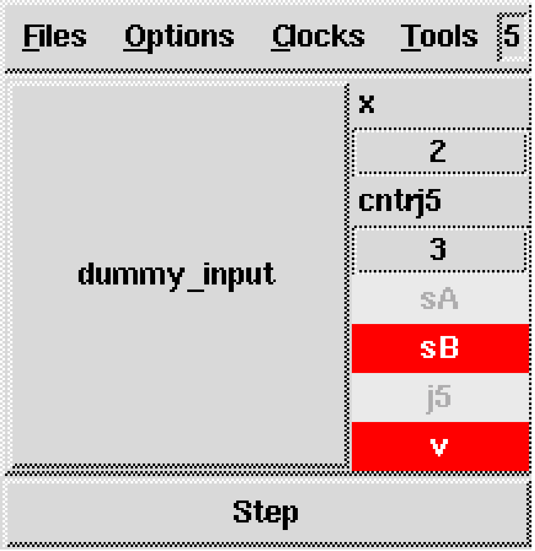
The SF2LUS translator thus supports:

- profiling of charts using the above mechanism,
- output of proof obligations to external programs and
- a (currently very primitive) form of loop unfolding to fixed bounds.

First, we add the `-counters` option to maintain counters on each junction:

```
% sf2lus -names -states_visible -junc_states -counters \
        Loops1.mdl -o Loops1.lus
```

After LUCIOLE simulation we get:



x
2
cntrj5
3
sA
sB
j5
v

So junction `j5` was entered 3 times during execution. We then annotate the junction in the STATEFLOW chart with the putative loop count maximum. Right-click on the junction, select "Properties" and enter:

```
cntrlim=3
```

in the "Description" field. With this annotation in place, the `-counters` option will also cause the translator to output the following observer for the loop counters. The observer node name is comprises `loop_counters_<CID>` where `<CID>` is the chart `id` number as for the toplevel STATEFLOW:

```
-- Observer for junction loop counters
node loop_counters_2(dummy_input: bool) returns(prop: bool);
var x, cntrj5: int; sA, sB, j5, v: bool;
let
  x, cntrj5, sA, sB, j5, v = sf_2(dummy_input);
  prop = cntrj5 <= 3;
tel
```

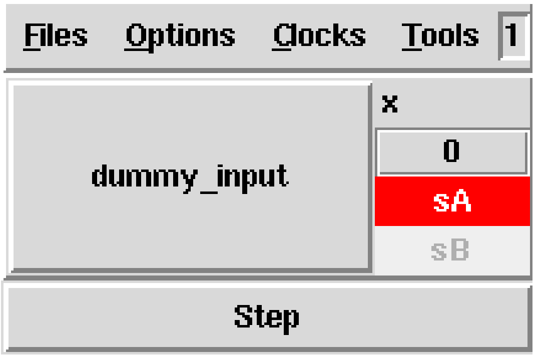
which can be passed to Nbac for validation:

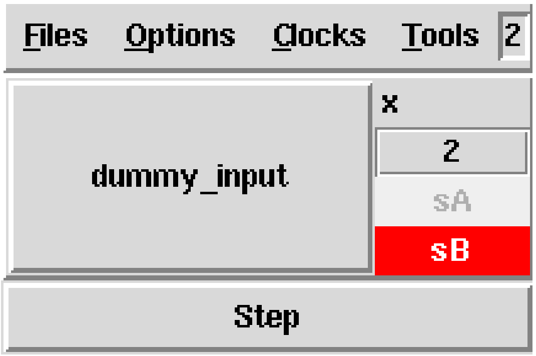
```
% lus2nbac Loops1.lus loop_counters_2
--Pollux Version 2.1
start normalisation ... done
Bool optimization : 171 -> 96 nodes
start minimal network generation ..... done (96 -> 77 nodes)
% nbacg -analysis 2 loop_counters_2.ba
...
SUCCESS: property proved
*** END ***
```

We now have confidence that junction `j5` will never be visited more than three time for *any possible set of inputs*. Thus we can unroll the loop three times:

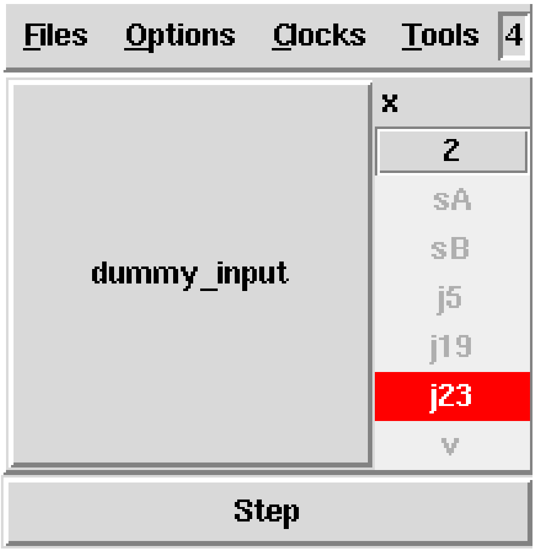
```
% sf2lus -names -states_visible -unroll Loops1.mdl -o Loops1.lus
```

This results in the following luciole trace which is how one would expect this chart to behave when translated into Lustre:





It is perhaps also instructive to add the `-junc_states` flag which show how the network has been duplicated. Here, the duplicated network appears as junctions `j19` and `j23`:



4.3 Discussion

The technique presented here is quite powerful. We used dynamic profiling (by treating junctions as states) to guess the bounds on the loops. We could then, in theory, use *static* analysis to unroll the loop into an equivalent one without loops but this is not valid until we have a proof of the bounds on the loops. The NBAC proof bridges this gap and validates the whole process.

However, there are very tight limits upon what can be done given finite resources for translation, proof and static analysis. It is possible that our guessed bounds may not be correct in which case NBAC will fail to provide a proof. It is also possible that NBAC may fail to provide the proof simply on resource limits or the nature of the chart (in general, proofs of this kind are undecidable). Finally, even if we have the proof it is possible that the static analysis can fail on resource limits, for example if the loop bounds were unreasonable.

What has been presented here is simply an example of the kind of analysis available for tackling imperative features of charts. It still depends upon the skill and experience of the chart designer to produce a design which both meets the requirements of the application and of abstract safety properties.

5 Observers and safety properties

Translating a Stateflow chart into Lustre allows the application of tools such as the model-checker Lesar or the abstract interpretation tool Nbac to be applied to the chart. The sf2lus translator supports this process with several features.

5.1 Observers in Lustre

Firstly, we can pass an expression to the translator and it will build an observer for that expression:

```
% sf2lus -names -states_visible SetReset_r13.mdl -o SetReset_r13.lus \
  -observe "((sOn and not sOff) or (sOff and not sOn))"
```

This will result in the following node being appended to the Lustre output:

```
-- observer for expr: ((sOn and not sOff) or (sOff and not sOn))
node verif_2_1(Set, Reset: event) returns(prop: bool);
var x: int;
let
  x = sf_2(Set, Reset);
  prop = ((sOn and not sOff) or (sOff and not sOn));
tel
```

The name of the observer node is `verif_<CID>_<index>` where `<CID>` is the chart `id` as before and `<index>` is a unique index for the observer (`verif` becomes `verify` under `-long_names`). This can then be passed to Lesar for checking:

```
% lesar SetReset_r13.lus verif_2_1 -v -diag
--Pollux Version 2.1
start normalisation ... done
start minimal network generation ..... done (90 -> 64 nodes)
building bdds ... 6 (on 6)

computing relevant statevars ... done (3 on 3)
DONE =>     3 states     5 transitions

=>total bdd memory : 59 nodes (~ 5.99 K)
TRUE PROPERTY
```

In fact the process can be automated somewhat since the translator automatically looks for an observer file when a model file is translated. This file has the same name as the model file but with `.mdl` replaced by `.obs`. The format of this file is simply one observer per line with blank lines and Lustre-style comments (starting with `--`) ignored. This behaviour can be switched off using the `-no_observers` option.

We can build complex safety properties using this mechanism but it is a bit inconvenient having to develop the observers in Lustre.

5.2 Observers in Stateflow

We can, however, build observers in STATEFLOW by providing parallel states at the toplevel which have one boolean free value.

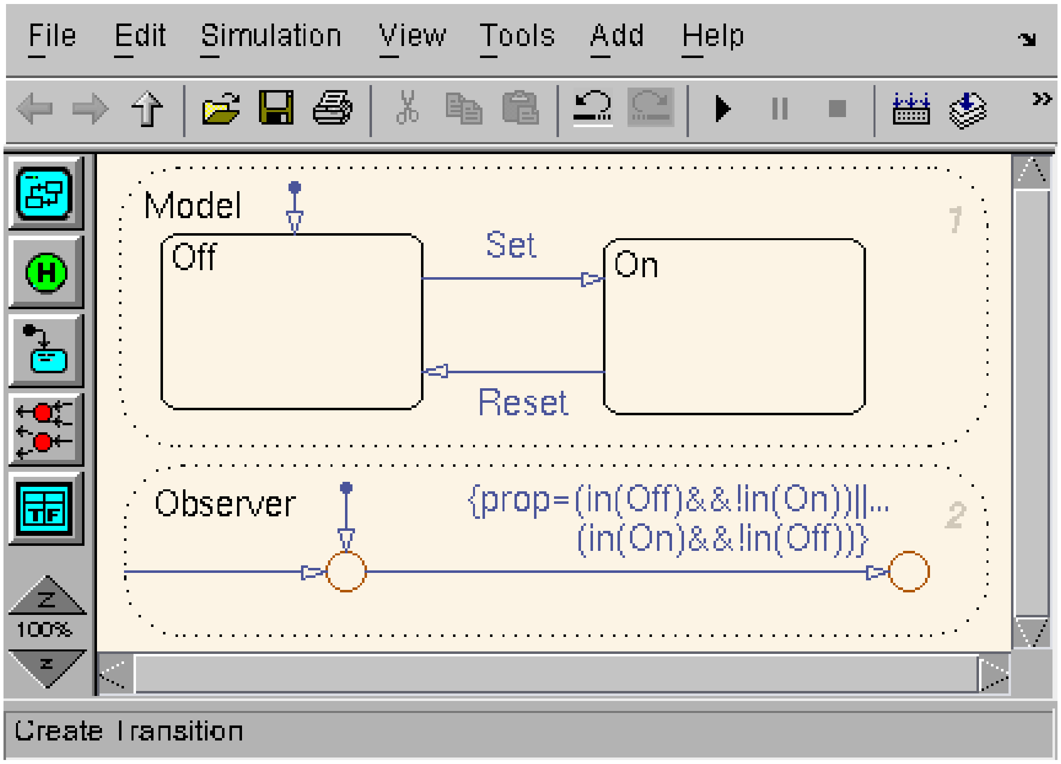


Figure 5: A Stateflow chart with an observer node

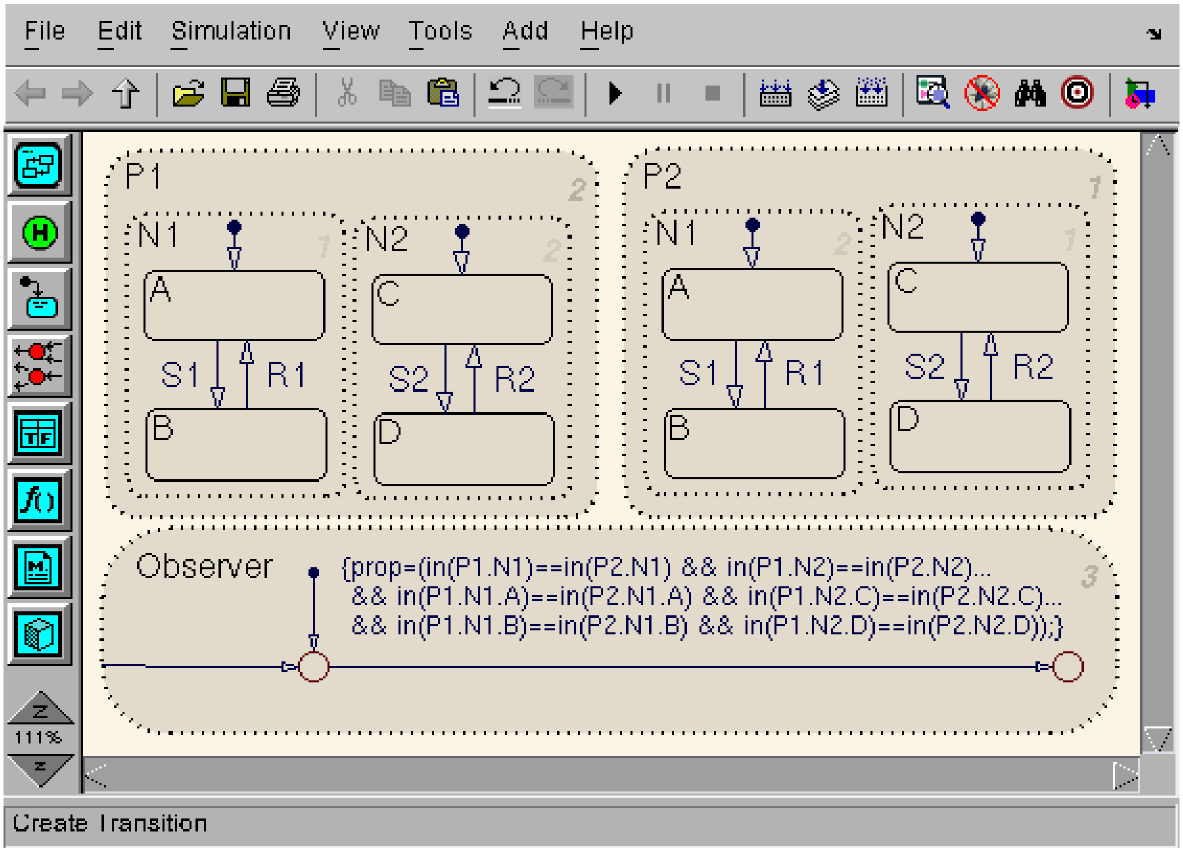
Figure 5 shows a STATEFLOW chart (`SetResetV.mdl`) with an observer implemented in STATEFLOW. If a toplevel node called "`Observer`" exists in the chart with a boolean output called "`prop`" (or "`property`" with `-long_names`) then a LUSTRE observer node for `prop` is automatically appended to the translated output. We can thus use all of STATEFLOW's action language to build observers which means that the designer does not have to learn LUSTRE and that the observed property is visible in the STATEFLOW chart.

5.3 Types of observers

We can write observers for application-specific safety properties in either LUSTRE or STATEFLOW. The EMSOFT'04 paper [4] describes a simple safety observer and how it was used to debug a STATEFLOW model (`taxi_verif_sf2.mdl`). However, we can also use these observers to help translate imperative features such as event broadcasting or parallel state confluence.

5.3.1 Parallel state confluence

Consider the chart in Figure 6 (`Parallel6V.mdl`). This contains two copies of a subchart with parallel states which are identical apart from the order in which the parallel states are visited.



```
{prop=(in(P1.N1)==in(P2.N1) && in(P1.N2)==in(P2.N2)...
 && in(P1.N1.A)==in(P2.N1.A) && in(P1.N2.C)==in(P2.N2.C)...
 && in(P1.N1.B)==in(P2.N1.B) && in(P1.N2.D)==in(P2.N2.D));}
```

Figure 6: An observer for parallel state confluence

The observer simply compares, state by state, the two parallel machines. If we translate the chart into Lustre and run Lesar on it:

```
% sf2lus Parallel6V.mdl -o Parallel6V.lus -names -states_visible
% lesar Parallel6V.lus verif_2_1
--Pollux Version 2.1

TRUE PROPERTY
```

We get a `TRUE` property showing that the two versions of the subchart are indeed equivalent and thus their parallel states are confluent.

The above chart implemented the observer in Stateflow but we may also wish to keep two copies of the chart and compare them externally. The problem here is that the namespaces of the two translated charts would be identical so sf2lus allows the visible node names to be prefixed, for example:

```
% sf2lus Parallel6_12.mdl -o Parallel6_12A.lus -prefix A -names -states_visible
% sf2lus Parallel6_21.mdl -o Parallel6_21B.lus -prefix B -names -states_visible
```

This allows a comparative observer to be built in Lustre, as follows:

```
include "Parallel6_12A.lus";
include "Parallel6_21B.lus";

node verif(S1, R1, S2, R2: bool) returns(prop: bool)
AsgN1, AsB, AsA, AsgN2, AsD, AsC,
BsgN1, BsB, BsA, BsgN2, BsD, BsC: bool;
let
  AsC, AsD, AsA, AsB, AsgN2, AsgN1 = Asf_2 (S1, R1, S2, R2);
  BsA, BsB, BsC, BsD, BsgN1, BsgN2 = Bsf_2 (S1, R1, S2, R2);
  prop = AsgN1 = BsgN1 and AsgN2 = BsgN2 and
         AsA = BsA and AsB = BsB and
         AsC = BsC and AsD = BsD;
tel
```

This simply includes both versions of the chart with alternate parallel state order and compares the outputs. The strange ordering of the output variables is due to the fact that inputs and outputs are indexed according to the `id` numbers of the data, events, states and junctions. Stateflow orders these according to its internal priorities.

In theory, these observers could be built automatically but for now they have to be built by hand.

5.3.2 Event stack depth

When the event stack mechanism is used there is an implicit assumption that the Stateflow model is bounded in its event stack. To support this the translator can add an additional output `err` (or `error` in `-long_names` format) which is set to `true` if there is an attempt to broadcast an event at the lowest level of the event stack. Consider again the chart in Figure 2. If we translate the chart with an event stack of 1 and run Lesar on the output:

```
% sf2lus Events3.mdl -o Events3.lus -errstate -ess 1
% lesar Events3.lus verif_2_1
--Pollux Version 2.1

FALSE PROPERTY
```

We get a FALSE property because we tried to broadcast the F output event while processing the E event at level 1 of the event stack. For this chart we actually need an event stack of 2:

```
% sf2lus Events3.mdl -o Events3.lus -errstate -ess 2
% lesar Events3.lus verif_2_1
--Pollux Version 2.1

TRUE PROPERTY
```

Beware, however, that the -errstate variable is not set if you do not define an event stack, a false positive will result. The SF2LUS translator assumes that you know the chart is confluent if you do not use an event stack.

5.3.3 Automatic observers

The observer in Section 5.2 is a simple internal consistency check upon the state variables. In fact, we can automate the generation of such an observer by scanning the chart and building an expression for each subgraph. For exclusive (OR) states there should be either no active state if the subgraph is inactive or *one and only one* active state if the subgraph is active. For parallel (AND) states there should be no active state if the subgraph is inactive or *all* the states should be active simultaneously. The -consistency option triggers the generation of just such an observer (called consistency_<CID>).

For example, the model in Figure 3 generates the following observer:

```
-- Observer for state consistency
node consistency_2(E: event) returns(prop: bool);
var y, x, z: real; sB, sC, sA, sE, sD, sgTop2, sgTop1: bool;
let
  y, x, z, sB, sC, sA, sE, sD, sgTop2, sgTop1 = sf_2(E);
  prop =
    ((sgTop1 and sgTop2) and
     (if sgTop1
      then ((sD and ((not sE) and (not sA))) or
            (((not sD) and (sE and (not sA))) or
            ((not sD) and ((not sE) and sA))))
      else (not (sD or (sE or sA))) and
      if sgTop2 then ((sC and (not sB)) or ((not sC) and sB))
      else (not (sC or sB))));
tel
```

Observers for junction entry counters are discussed in Section 4.2.

6 Using sf2lus with s2l and ss2lus

The ss2lus script is a simple program which allows Simulink models with Stateflow components to be translated into a monolithic Lustre program. The process is very simple, sf2lus is called to translate the Stateflow component into a temporary file, the name of which is then passed to s2l which translates the Simulink component generating calls to the sf2lus-generated nodes. The entire temporary Stateflow Lustre file is then appended to the translated Simulink. In practice, however, this interface is still in its infancy and only fairly simple and regular Simulink/Stateflow combinations are supported. For example, we do not handle callbacks from Stateflow to Simulink as yet, although this is possible and may be implemented in future.

6.1 ss2lus syntax

The sytax of the ss2lus command is very simple:

```
% ss2lus SetReset_r13.mdl --monoperiodic
```

This translates the given model file and combines the two components. Some of the options to sf2lus are mandatory and set by ss2lus. If you wish to pass options to s2l they can be passed on the ss2lus command line. Both s2l and sf2lus can receive additional arguments in the `S2LOPTS` and `SF2LUSOPTS` environment variables, respectively. Be careful with sf2lus options, however, since they may invalidate the interface between s2l and sf2lus. In particular, do not set `-names` or `-states_visible`, or any options which generate additional inputs or outputs. In fact, some of the features of sf2lus which allow management of Stateflow charts in isolation should also not be used. For instance, you need to ensure that all Stateflow components have at least one input and one output to ensure that the dummy arguments generated to allow legal Pollux output are not triggered.

6.2 Matlab workspace files

To assist in building the interface, a file describing sf2lus's view of the Matlab workspace is generated in a `.mws` file. This contains a list of Matlab workspace variables used by the sf2lus code and which have to be declared in the s2l output. It also contains information about any *pseudo-variables* generated by sf2lus (currently only the `t` time variable). Note that the `.mws` file is *not* deleted by either sf2lus or ss2lus since constants are read from this file. It is up to the programmer to maintain this file with respect to the current Matlab workspace. It may be possible, in future, to generate this file from Matlab automatically.

7 Miscellaneous features

Here we describe some features of SF2LUS which can be used to help debugging STATEFLOW charts. Most of these are more useful for debugging the translator itself, however.

7.1 Matlab revisions

-r13,-r14 Currently, only MATLAB revisions `r13` and `r14` are supported. These are controlled by the `-r13` and `-r14` options (`-r14` is the default).

7.2 Syntax control

STATEFLOW has a number of features which make it difficult to translate verbatim into LUSTRE without complex (and probably unreliable) compilation support. The following options have been implemented to help with compatibility:

-kw,-nkw Add/remove a keyword to/from the list of keywords allowed as identifiers. STATEFLOW allows keywords to be used as identifiers which is difficult to handle with a lex/yacc-style parser. These options allow the use of some selected keywords to be included or excluded from the syntax. Use these with care, strictly-speaking you should eliminate keywords from the chart altogether.

-paths Use full path names for states. This triggers the mechanism which allows full pathnames to be use in action code. This is not on by default, however, since it slows down the translator and pollutes the namespace since *all* variables have to be in long format, for example, the node reference "`A.B.C`" would probably become `sA_B_C` throughout the code. Note that local data is not supported yet.

This option is automatically set if more than one STATEFLOW chart is found in a model. This is to prevent name clashes between different STATEFLOW charts with identical state names. There is no check, as yet, for such states, in future this mechanism may only be triggered when such states exist.

-no_paths Do not automatically set `-paths`. If you know that there are no state name conflicts between the STATEFLOW charts in a model then this allows processing of multiple-STATEFLOW charts without full path names.

7.3 Output formatting

These are some options to control the format of the output LUSTRE file. You can actually find some of these documented in the OCAML manual under the Format library.

-margin Set the margin for formatted output.

-max_indent Set the maximum indent for formatted output.

-text_limit Limit output strings to this number of characters. This is not a `Format` library variable but is used to limit the amount of text copied into the LUSTRE file from STATEFLOW sources, for example when creating comments from state or transition labels.

7.4 Output language

Four output languages are currently supported, triggered by the following options:

-pollux Generate Lustre V4 output. This is the default mode and the most fully supported.

-nbac Generate Nbac output. Nbac is almost completely compatible with Lustre V4 via the lus2nbac utility supplied with the Lustre distribution. This option simply triggers emulation of the integer modulus function which seems to be missing from Nbac.

-reluc Use Reluc modifications. Again, since the sf2lus translator uses only a small subset of Lustre the output is almost compatible with Reluc. Currently, this option triggers some additional parenthesization which seems to be needed. Currently, arrays and the event stack are not supported since Reluc does not have Lustre's static recursion mechanism.

-scade Use SCADE modifications. There are some syntactic differences between SCADE and Lustre, some of which can be ironed out by a simple transformation on the output. These involve constructs such as:

```
(x,y) = if p then f(a,b) else (c,d);
```

which are not supported by SCADE. Again, there are no arrays or event stack.

7.5 Namespace management

Namespace management in the sf2lus translator is not fixed, for several reasons:

- There is the tension between providing human-readable Lustre output without making the code too verbose.
- Different users may have different preferences as to what is readable, depending upon their intended usage of the resulting code.
- It is difficult to translate namespaces accurately between two such widely differing languages as Stateflow and Lustre. It is simple to convert names from one syntax space to another but doing so while retaining the flavour of the original language is difficult.
- It is possible that different translations may have to coexist in the same context which will inevitably result in namespace collisions. This is exacerbated by the fact that Lustre V4 has no concept of modularity.

For these reasons, sf2lus supports several options controlling the way the output namespace is generated:

-names Use state names in variables. So, for example, a state called `A` generates a variable `sA` (or `state_A`).

-ids Use state ids in variables. A state with `id` 3 will be referenced by `s3` (`state_3`).

-names_ids Use both names and state ids in variables. For example `s3_A` (`state_3_A`).

-long_names Use unabrreviated names in the output. Currently, the complete list of abbreviations is:

_point		action	a	after	aft	at	at
before	bfr	call	ca	change	ch	condition	c
count	cnt	counter	cntr	counts	cnts	during	du
end	end	enter	ent	entry	en	error	err
event	ev	events	evs	every	evry	exit	ex
flag	flg	graph	g	history	h	in	in
increment	inc	init	ini	inners	ins	junction	j
link	l	okay	ok	out	o	pre	p
print	pr	property	prop	state	s	stateflow	sf
stub	st	subgraph	sg	term	trm	tmp	t
transition	tr	update	u	valid	v	verify	verif

-varprefix Prefix all variables (for namespace conflict avoidance). Do not use this, it is present for debug purposes only.

-prefix,-suffix Prefix/suffix all toplevel names. This is used, for instance, when one wishes to compare the output from two different translations. All the visible identifiers in the output code are prefixed by the given string, for example, "`-prefix A`" might give:

```
type Aevent = bool;
const Aset = true; Aclr = false;
node Asf_2(Set, Reset: Aevent) returns(sOff, sOn: bool);
```

7.6 General translator control

These options control the translation process at the most basic level. Some of these are discussed in the preceding sections. The effect of others is described in the associated papers.

-no_self_init Top level graph does not provide initialization. Normally, the `init` and `term` flags are automatically set to:

```
init = true -> false;
term = false;
```

at the top level of the code. This option disables this behaviour but is only present for debugging the translator.

-ess Event stack size. This sets the depth of the event stack, see Section 3 for usage information and *caveats*.

-sends Enable sends to specific states. This option triggers some additional processing which allows STATEFLOW's `send` function to be implemented. Note that events become integers which may affect subsequent analysis and that currently this feature is only partially implemented. See Section 3.2.

-junc_states Treat junctions as states. When this is set junctions are given a physical state (called `j<ID>` or `junction_<ID>`) and the chart can stay in a junction after a reaction. An additional output boolean (`v` or `valid`) is generated which is `true` if and only if the current state is not a junction. See Section 4.1.

-errstate Add error processing to event broadcasts. This generates an extra output boolean variable (`err` or `error`) which is set to `true` if an event is broadcast at the lowest level of the event stack. This logic is switched off if the event stack size is zero. See Section 5.3.2.

-unroll Unroll loops according to loop counters. Using the annotations in the Stateflow chart described in Section 4.2 transition networks involving loops are unfolded a fixed number of times resulting in a loop-free chart. The unrolling algorithm is currently very primitive and has complexity problems.

7.7 Data management

Handling Matlab's workspace is complicated by the fact that it is stored in a binary format which external tools cannot read. Hence, the translator has to make some assumptions about the workspace which it communicates via the `.mws` file. These options allow some additional control over workspace values:

-create_missing Add missing data to data dictionary. If a chart contains a reference to a variable not in Stateflow's data dictionary then it can be automatically created. All such variables have to have the same scope and type, however.

-missing_scope Scope for missing data (default: `INPUT_DATA`). Recognized values are:

r13/r14	`INPUT_EVENT`	`OUTPUT_EVENT`	`LOCAL_EVENT`
	`OUTPUT_DATA`	`INPUT_DATA`	`LOCAL_DATA`
	`TEMPORARY_DATA`	`CONSTANT_DATA`	
r14	`FUNCTION_INPUT_DATA`	`FUNCTION_OUTPUT_DATA`	`PARAMETER_DATA`

-missing_datatype Data type for missing data (default: `double`). Known values are (not all are supported):

`double`	`single`	`int8`	`int16`
`int32`	`uint8`	`uint16`	`uint32`
`boolean`	`fixpt`	`ml`	

-no_constants Omit workspace constants from output. This is used by ss2lus and prevents sf2lus from including constants defined in the Matlab workspace file from being included. The output is not legal Lustre since the constants are expected to be provided by s2l.

7.8 Time

The Stateflow implicit time variable `t` is slightly problematical since Lustre does not have any notion of absolute time. For stand-alone Stateflow-generated Lustre code the following options allow `t` to be generated automatically assuming a fixed time difference between reactions. If time is not emulated here then it is assumed to be an input to the chart. The Matlab workspace file contains an entry indicating whether `t` is an input or not.

-emulate_time Provide internal time value. References to Stateflow's time value are implemented internally in the Lustre code according to the following two options.

-start_time Start time for emulated time (default: 0.0). The value of `t` at reaction zero.

-time_increment Time increment for emulated time (default: 1.0). The `t` variable is incremented by this amount at the start of each reaction.

7.9 Observers

One of the main uses of the sf2lus translator is in the proof of safety properties. The following options support this activity:

-observe Add observer node for given expression. Generate a single Lustre node observing the expression given as a string of Lustre code. State variables can be observed provided the `-states_visible` option is set.

-no_observers Don't read observer file. By default, sf2lus looks for a file `<file>.obs` when given a model file `<file>.mdl`. If it exists it is assumed to be a file containing observable expressions, one per line. Lustre-style comments (`--`) are permitted.

-consistency Add a state consistency observer. This option causes an observer for the state variables to be generated. The observer is called `consistency_<CID>` and is mostly used to verify the translation process, see Section 5.3.3. The `-states_visible` option is set automatically.

-counters Add loop counters to junction networks. An additional integer output for each junction in the chart is generated. The counters are called `cntrj<ID>` where `<ID>` is the `id` number for the junction. Each counter is incremented when its junction is entered during transition path analysis. Currently, maximum values are not maintained so the values have to be checked after each reaction. In addition, an observer (called `loop_counters_<CID>`) is generated for all the junctions annotated as in Section 4.2. The `-junc_states` option is set automatically.

7.10 Debugging features

These are some miscellaneous options used to help in debugging the translator. Some may be of use in debugging Stateflow charts, however.

-trace Add trace output. Generates a print function for each node generated. These are supported by external C functions in a file `<file>_ext.c` which print out the arguments and results of each node as it is processed. Some effort is made to defeat demand-driven computation which can elide operations which are pure side-effects.

-trace_inputs Number of inputs to add to trace output. These seems to be a resource limit of the nummber of arguments which can be passed to C functions through Lustre's FFI. This sets the number of inputs to print for each function.

-trace_locals Number of locals to add to trace output. The number of locals to print for each node during a trace.

-states_visible Make state variables visible for toplevel graph. States are named as in Section 7.5.

-temps_visible Make temporary variables visible for toplevel graph. Currently, this is not complete since there are problems with the event stack due to inter-level transitions.

-stubs_visible Make stub nodes visible in output. When the event stack is implemented a skeleton node is generated at the start of translation as a *pro forma* event broadcast node. The output will not be legal LUSTRE if this option is set.

-locals_visible Make chart locals outputs. Again, only works for a subset of local values.

-no_typecheck Do not typecheck generated nodes. If the typechecker fails, this can either help debug the typechecker or generate partial output which can be hand-edited to give legal code.

-no_sequence Do not sequence generated nodes. Another debugging feature, sequencing refers to the generation of intermediate flows to force the evaluation order of imperative actions in LUSTRE. Output will not be legal LUSTRE.

-no_normalize Do not normalize generated nodes. Normalization refers to sequencing, typechecking and some other small transformations applied to the generated LUSTRE.

-input_bools_ints Transform input booleans into integers. Used to allow LUCIOLE to set more than one boolean input simultaneously, see Section 2.2.3.

-write_now Write output as generated. Normally, the LUSTRE code is built up in memory and then dumped at the end of translation. This is a debug feature which allows inspection of partial code when the translator crashes.

-g,-gp,-v Enable debug printouts. Enables debug printouts. Normally, these are disabled when the distribution is made.

-help,–help Display the list of options.

8 The display_graph utility

The SF2LUS distribution includes a utility program called `display_graph`. This parses a STATEFLOW model file exactly as SF2LUS itself and dumps out the internal data structure as a `dot` file. This is useful to help understanding the LUSTRE output from SF2LUS since it can annotate the states and links with `id` numbers.

The potentially unlimited complexity of STATEFLOW charts and the nature of GRAPHVIZ's plotting algorithms limits the size of chart which can be displayed. Also, GRAPHVIZ cannot implement inner transitions so they are represented by synthesized nodes standing for the parent node.

Figure 7 (`DisplayGraph1.mdl`) shows a sample output from `display_graph` for a simple test model. This shows most features of the output, dotted boxes are parallel states, solid boxes exclusive states, circles are junctions and points are the tails for default transitions. The transition labelled `dummy` has been synthesized by the parser and the diamond-shaped node labelled `C` is the proxy for the inner transition. Inter-level transitions are handled by GRAPHVIZ. The command line options are reasonably self-explanatory, see Appendix B.

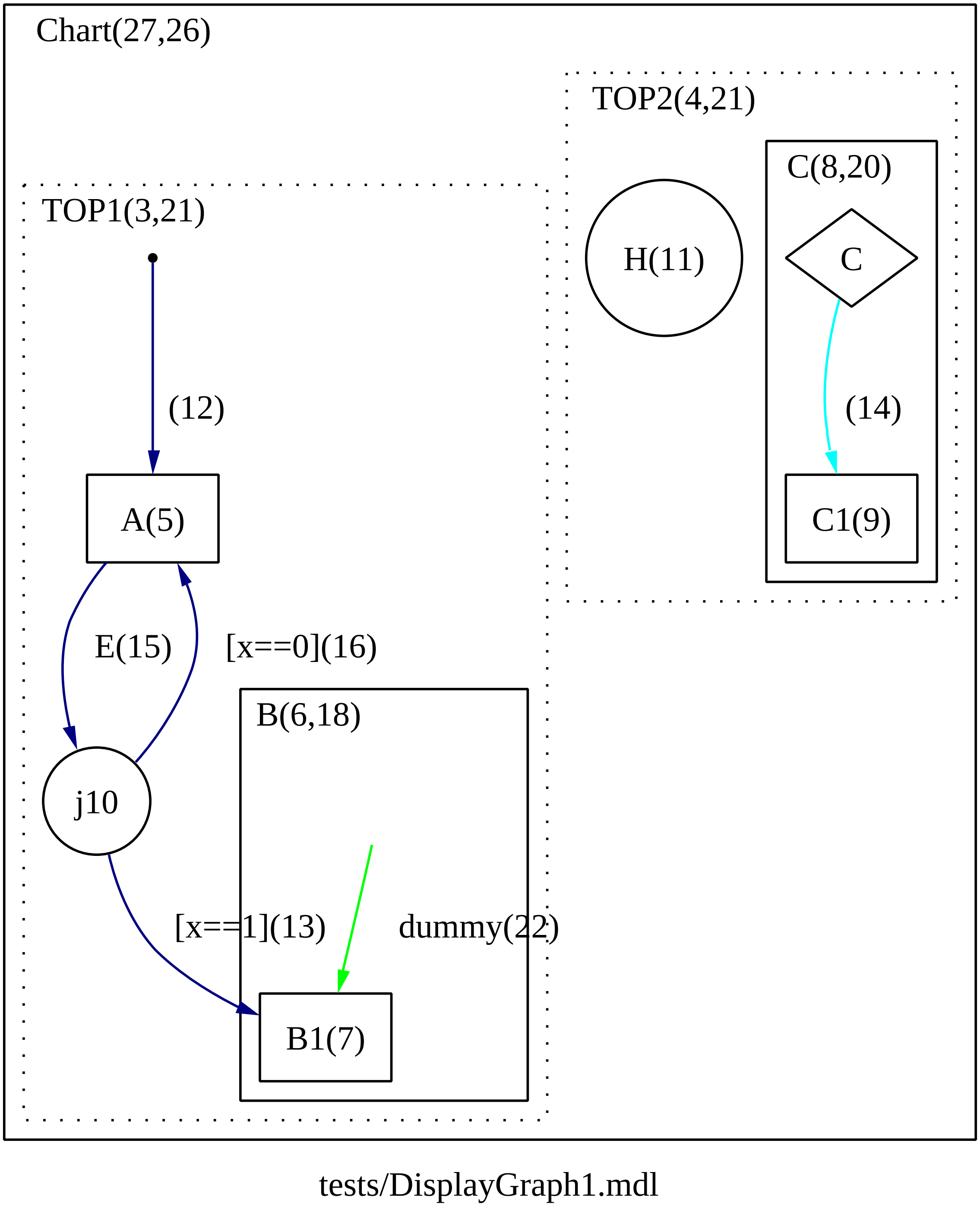


Figure 7: Sample output from display_graph

A sf2lus command line options

The following are the currently supported options which can be viewed using `sf2lus --help`:

```
Stateflow to Lustre (c) VERIMAG 2004

  Convert Stateflow into Lustre.

Syntax:

  sf2lus <options> file.mdl

Bug reports and enquiries to: "Paul Caspi" <Paul.Caspi@imag.fr>

Options:
  -r13                     Matlab version 13
  -r14                     Matlab version 14 (default)
  -kw <str>                Add a keyword to the keyword identifier list
  -nkw <str>               Remove a keyword from the keyword identifier list
  -paths                   Use full path names for states
  -no_paths                Do not set -paths automatically
  -I <dir>                 Append a directory to the search path
  -include <file>          Add a file to be included
  -o <file>                Name of output file, (default: stdout)
  -mws <file>              Name of Matlab workspace emulation file
  -margin <int>            Set the margin for formatted output
  -max_indent <int>        Set the maximum indent for formatted output
  -text_limit <int>        Limit output strings to this number of characters
  -pollux                  Use Pollux modifications (default)
  -nbac                    Use Nbac modifications
  -reluc                   Use Reluc modifications
  -scade                   Use Scade modifications
  -names                   Use state names in variables
  -ids                     Use state ids in variables
  -names_ids               Use both names and state ids in variables
  -long_names              Use unabrreviated names (eg. "s" -> "state")
  -no_self_init            Top level graph does not provide initialization
  -ess <int>               Event stack size
  -sends                   Enable sends to specific state (events become ints)
  -errstate                Add error output variable
  -junc_states             Treat junctions as states
  -create_missing          Add missing data to data dictionary
  -missing_scope <scope>   Scope for missing data (default: INPUT_DATA)
  -missing_datatype <type> Data type for missing data (default: double)
  -no_constants            Omit workspace constants from output
  -emulate_time            Provide internal time value
  -start_time <float>      Start time for emulated time
  -time_increment <float>  Time increment for emulated time
  -real_time               Provide real time value (in external C code)
  -varprefix <str>         Prefix all variables (for namespace conflict avoidance)
  -prefix <str>            Prefix all names (used for comparisons with lesar)
  -suffix <str>            Suffix all names (used for comparisons with lesar)
  -observe <expr>          Add observer node for given expression
```

```
-no_observers            Don't read observer file
-consistency             Add state consistency observer (sets -states_visible)
-counters                Add loop counters to junctions (sets -junc_states)
-unroll                  Unroll loops according to loop counters
-trace                   Add trace output
-trace_inputs <int>      Number of inputs to add to trace output
-trace_locals <int>      Number of locals to add to trace output
-export_cvs              Export condition variables (set if function call events)
-states_visible          Make state variables visible for toplevel graph
-temps_visible           Make temporary variables visible for toplevel graph
-stubs_visible           Make stub nodes visible in output (won't compile)
-locals_visible          Make chart locals outputs
-no_typecheck            Do not typecheck generated nodes
-no_sequence             Do not sequence generated nodes
-no_normalize            Do not normalize generated nodes
-input_bools_ints        Transform input booleans into ints (for luciole)
-write_now               Write output as generated (debug)
-g                       Enable debug printouts
-gp                      Enable parser debug printouts
-v                       Set debug level
-help                    Display this list of options
--help                   Display this list of options
```

B display_graph command line options

The following are the options supported by `display_graph` which can be viewed using `display_graph --help`:

```
Display Stateflow Graphs (c) VERIMAG 2004

  Display the internal structure used by sf2lus.

Syntax:

  display_graph <options> file.mdl

Bug reports and enquiries to: "Paul Caspi" <Paul.Caspi@imag.fr>

Options:
  -r13                     Matlab version 13
  -r14                     Matlab version 14 (default)
  -kw <str>                Add a keyword to the keyword identifier list
  -nkw <str>               Remove a keyword from the keyword identifier list
  -paths                   Use full path names for states
  -no_paths                Do not set -paths automatically
  -I <dir>                 Append a directory to the search path
  -include <file>          Add a file to be included
  -o <file>                Name of output file, (default: stdout)
  -mws <file>              Name of Matlab workspace emulation file
  -margin <int>            Set the margin for formatted output
  -max_indent <int>        Set the maximum indent for formatted output
  -text_limit <int>        Limit output strings to this number of characters
  -names                   Use state names in variables
  -ids                     Use state ids in variables
  -names_ids               Use both names and state ids in variables
  -long_names              Use unabrreviated names (eg. "s" -> "state")
  -string_escape_subgraph  string escape subgraphs for "string_of_graph"
  -colours                 Use colours in graph plotting
  -link_ids                Include link ids in dot output
  -node_ids                Include node ids in dot output
  -display_pointers        Display root, parent and this pointers
  -display_fromto          Display from, to and sublink pointers
  -display_local_data      Display local data attached to nodes
  -display_ids             Display ids only for "display_graph"
  -display_names           Display names only for "display_graph"
  -display_intermediates   Display intermediate graphs
  -display_wait            Wait for viewer after display
  -viewer <str>            Viewer for dot output files (default: ghostview)
  -dot_output_type <str>   Dot output type (dot -Ttype, default: ps)
  -dot_output_size <str>   Dot output size (dot -Gsize=<str>, default: 11,8)
  -dot_simulink            Include Simulink in dot output
  -dot_boxes               Include boxes in dot output
  -g                       Enable debug printouts
  -gp                      Enable parser debug printouts
  -v                       Set debug level
  -help                    Display this list of options
```

```
--help                   Display this list of options
```

References

[1] The MathWorks. Stateflow and stateflow coder, user's guide. Available at http://www.mathworks.com/products/stateflow/. 4

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 4

[3] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. Technical Report TR-2004-16, Laboratoire VERIMAG, Centre Equation, 2, avenue de Vignate, 38610 GIERES, France, 2004. http://www-verimag.imag.fr. 4

[4] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *Proc. EMSOFT 2004*, Pisa, Italy, Sep 2004. Springer. 4, 19

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time simulink to lustre. In R. Alur and I. Lee, editors, *EMSOFT'03*, Lecture Notes in Computer Science. Springer Verlag, 2003. 4

[6] Esterel Technologies, Inc. *SCADE Language - Reference Manual 2.1*. 4

[7] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992. 4