Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Defining and translating a "safe" subset of Simulink/Stateflow into Lustre

N. Scaife, C. Sofronis, P. Caspi, S. Tripakis and F. Maraninchi

Report n^o TR-2004-16

July 15, 2004

Defining and translating a "safe" subset of Simulink/Stateflow into Lustre

N. Scaife, C. Sofronis, P. Caspi, S. Tripakis and F. Maraninchi

July 15, 2004

Abstract

The SIMULINK/STATEFLOW toolset is an integrated suite enabling model-based design and has become popular in the automotive and aeronautics industries. We have previously developed a translator called S2L from SIMULINK to the synchronous language LUSTRE and we build upon that work by encompassing STATEFLOW as well. STATEFLOW is problematical for synchronous languages because of its unbounded behaviour so we propose analysis techniques to define a subset of STATEFLOW for which we can define a synchronous semantics. We go further and define a "safe" subset of STATEFLOW which elides features which are potential sources of errors in STATEFLOW designs. We give an informal presentation of the STATEFLOW to LUSTRE translation process and show how our model-checking tool LESAR can be used to verify some of the semantical checks we have proposed. Finally, we present a small case-study.

Keywords: Model-based Design, Simulink, Stateflow, Lustre, Formal Methods, Safety-critical, Model-checking.

Reviewers: Reviewed by EMSOFT04 referees

Notes: This work has been partially supported by the European Community through IST projects Next-TTA and Rise. This report presents an extended version of a paper accepted by the Fourth International Conference on Embedded Software EMSOFT04

How to cite this report:

@techreport { scaife04defining,
title = { Defining and translating a "safe" subset
of Simulink/Stateflow into Lustre},
authors = { N. Scaife, C. Sofronis, P. Caspi, S. Tripakis and F. Maraninchi},
institution = { Verimag Technical Report },
number = {TR-2004-16},
year = { 2004},
note = { This is the full version of the paper accepted by EMSOFT'04.}
}

Contents

1 Introduction

Embedded and real-time systems are often safety-critical and require high-quality design and guaranteed properties. Model-based design has been advocated as the method of choice for dealing with systems such as these. The design process consists of building models on which the required system properties are carefully checked and assessed and then deriving implementations such that these properties are preserved. This allows high quality to be achieved at a lower cost.

SIMULINK/STATEFLOW[1] is a very popular tool-chain in this setting and is considered a *de facto* standard in many domains such as control systems and the automotive and aircraft industries. SIMULINK is a block-diagram based formalism while STATEFLOW provides hierarchical and parallel state machine notations borrowed from STATECHARTS [10]. In many cases, designers need to use both models and a strength of SIMULINK/STATEFLOW is the integration of these complementary formalisms within the same tool-chain. However, the tool-chain was originally designed for simulation purposes and, as such, it lacks many desirable features when it comes to model-based design, such as static checks, formal semantics [2] and associated formal methods such as formal analysis and synthesis techniques (for example, verification, testing and code generation).

In previous work [4], we have shown how to translate a subset of SIMULINK into LUSTRE [7], a synchronous data-flow language which, as opposed to SIMULINK, is formally based and endowed with several formal tools such as the LESAR model-checker [8] and the Prover Plug-in from Prover Technology[3] [17]. Moreover, the industrial version of LUSTRE, SCADE, commercialised by Esterel-Technologies[4] is equipped with a DO178-B Level A qualified code generator, which makes it well-adapted to be used in safety-critical projects. Thus, the intended use of our translator is quite clear: after a system is designed using SIMULINK, the LUSTRE translation can be used to guarantee the formal status of the model, formally check properties of this model and, finally, generate code which preserves the semantics of the original model.

This work aims at extending the previous work by including support for STATEFLOW. This is compulsory because, as said above, SIMULINK/STATEFLOW is an integrated tool-chain and many applications use both complementary tools.

However, STATEFLOW raises many more semantic problems than SIMULINK and the task of identifying a "clean" subset of STATEFLOW is much harder than it was for SIMULINK[5]. This is why many STATEFLOW users have guidelines restricting the use of unsafe constructs [6]. The problem with these guidelines is that:

- there is no common agreement between the various guidelines in use,

[1] Trademarks of the *MathWorks* company

[2] The semantics of SIMULINK/STATEFLOW is precisely but informally described in a several-hundred-pages long document [18]. The *MathWorks* implementation is the reference for this behaviour.

[3] http://www.prover.com

[4] http://www.esterel-technologies.com

[5] The reason for this state of affairs may come from the fact that the field of hierarchical and parallel state machines is much younger than that of block-diagrams. Furthermore, the problem of communicating parallel state machines is an intricate one and, despite several interesting approaches [10, 3, 14] does not seem to have yet reached a satisfactory solution.

- in the absence of automated checking tools they may appear too restrictive to designers and
- many legacy STATEFLOW models predate the existence of guidelines and bringing them into conformance with these guidelines would require considerable effort.

The contributions of this paper are two-fold. Firstly, we list the semantic problems associated with STATEFLOW (Section 2) and propose light-weight static checking algorithms which ensure that a model is free of such problems and can therefore be considered "safe" (Section 3). This may be useful for designing less restrictive guidelines. Secondly, we show how to translate STATEFLOW into LUSTRE (Section 5) and also show how properties that may not be checked by the algorithms of Section 3 can be checked on the LUSTRE translation by means of model-checking (Section 6). This allows us to further enhance our notion of a "safe" subset. Finally, we discuss a prototype implementation and a simple case study (Section 7).

Related work

STATEFLOW evolved as the finite state machine component of SIMULINK and as such must be viewed principally as a simulator itself. As such its designers have concentrated upon a user-friendly interface and supporting as many useful features as possible while not impeding the design process by allowing too many unsafe features. For this purpose STATEFLOW itself is equipped with many *run-time* error detection features such as stack overflow or consistent state checking. A formal semantics for such a system is probably not necessary and may even place too many restrictions upon the tool for use by non-specialists. However, STATEFLOW has become increasingly used as a design verification tool for which effort can be economised by using the SIMULINK/STATEFLOW itself for code generation. In such an arena a formal semantics is essential and there have been some recent attempts to define a semantics for STATEFLOW.

STATECHARTS [10] are sometimes compared with STATEFLOW since both are visual representations of state machines. There has been much work into formalization of STATECHARTS either by translating into a known system such as hierarchical automata [15] or by deriving a semantics for a suitable subset [12]. The two systems have a very different semantics, however, for example STATEFLOW has no notion of true concurrency so work in this area would be difficult to adapt for STATEFLOW directly.

One attempt includes Tiwari [19] who describes analyses for SIMULINK/STATEFLOW models by translating into communicating pushdown automata. These automata are represented in SAL [2] which allows formal methods such as model-checking and theorem proving techniques to be applied to these models. Essentially, the system is treated as a special hybrid automata and algebraic loops involving STATEFLOW charts are not considered.

Hamon and Rushby have developed a structural operational semantics for STATEFLOW [9] for which they have an interpreter to allow comparison with STATEFLOW. Their subset of STATEFLOW seems to have been inspired by the Ford guidelines [6], for instance loops are forbidden in event broadcasting and local events can only be sent to parallel states. They have other restrictions as well, such as forbidding transitions out of parallel states but in general support most of

the STATEFLOW definition including supertransitions. They also have a translator into the SAL system which allows various model-checking techniques to be applied to STATEFLOW.

Banphawatthanarak *et al.* describe a translator from STATEFLOW into the SMV model checker [1]. As for our translator they do not work from a formal semantics for STATEFLOW and the main issue seems to be the ordering of actions.

Finally, Reactive Systems Inc. [16] have a tool called REACTIS for automated test generation for SIMULINK/STATEFLOW models.

Our work differs from these in that the generated LUSTRE program can be used not only for model-checking but also and primarily for C code generation while preserving the original semantics. Also, we provide a set of simple static checks which are much "lighter" than model-checking.

2 A safe subset of Stateflow

Before we can attempt to define which features of STATEFLOW are suitable for translation into Lustre, we have to illustrate some of the semantical issues with STATEFLOW, which are also likely to cause problems with our translator. These issues range from "serious" ones, such as non-termination of a simulation step or stack overflow, to more "minor" ones, such as dependence of the semantics upon the positions of objects in the STATEFLOW diagram. First, we briefly describe the STATEFLOW language and informally explain its semantics (for a formal semantics, see [9]).

2.1 A short description of Stateflow

STATEFLOW is a graphical language resembling Statecharts [10]. The semantics of STATEFLOW are embodied in the interpretation algorithm of the STATEFLOW simulator, documented in a 900-page long User's Guide [18] (terminology is borrowed from that guide). A STATEFLOW *chart* has a hierarchical structure, where states can be refined into either *exclusive (OR)* states connected with transitions or *parallel (AND)* states, which are not connected.[6] Figure 14 shows an example: A and B are parallel states (with *parent* the root state), while all their *child* states are exclusive. A transition can be a complex (possibly cyclic) flow graph made of *segments* joining connective *junctions*. Each segment can bear a complex label with the following syntax (all fields are optional):

$$E[C]\{A_c\}/A_t$$

where E is an *event*, C is the *condition* (i.e., guard), A_c is the *condition action* and A_t is the *transition action*. A_c and A_t are written in the *action language* of STATEFLOW, which contains assignments, emissions of events, and so on. Actions written in the action language can also annotate states. A state can have an *entry action*, a *during action*, an *exit action* and *on event* E *actions*, where E is an event.

[6] Notice that parallel states are not executed concurrently, but sequentially.

The interpretation algorithm is triggered every time an event arrives from SIMULINK or from within the STATEFLOW model itself.[7] The algorithm then executes the following steps:

Search for active states: this search is performed hierarchically, from top to bottom. At each level of hierarchy, when there are parallel states, the search order is a *graphical two dimensional one: states are searched from top to bottom and from left to right*, in order to impose determinism upon the STATEFLOW semantics.

Search for valid transitions: once an active state is found, its transitions are searched based on several enabledness criteria: the event of the transition must be present and its condition must be true. The goal is to find a transition which is *valid* all the way from the source state to the destination state. In particular, when the transition is multi-segment, the condition actions of each segment are executed while searching and traversing the transition graph. The search order is again deterministic: transitions are searched according to the *12 o'clock rule*.[8]

Execute a valid transition: once a valid transition is found, STATEFLOW follows these steps: execute the exit action of the source state, set the source state to inactive, execute the transition actions of the transition path, set the destination state to active and finally execute the entry action of the destination state.

Idling: when an active state has no valid output transitions an active state performs its during action and the state remains active.

Termination: occurs when there are no active states.

It should be emphasized that each of the executions *runs to completion* and this makes the behaviour of the overall algorithm very complex. In particular, *when any of the actions consists of broadcasting an event, the interpretation algorithm for that event is also run to completion before execution proceeds.* This means that the interpretation algorithm is recursive and uses a *stack*. However, as we will see, the stack does not store the full state, which leads to problems of side effect (Section 2.2.2). Also, without care, the stack may overflow (Section 2.2.1).

2.2 Semantical issues with Stateflow

2.2.1 Non-termination and stack overflow

As already mentioned, a transition in STATEFLOW can be multi-segment and the segment graph can have cycles. Such a cycle can lead to non-termination of the interpretation algorithm during the search for valid transition step.

Another source of potential problems is the run-to-completion semantics of event broadcast. Every time an event is emitted the interpretation algorithm is called recursively, runs to completion, then execution resumes from the action statement immediately after the emission of the

[7] The SIMULINK event is often a SIMULINK *trigger*, although it can also be the simulation step of the global SIMULINK-STATEFLOW model.

[8] Notice that this is considered harmful even in the STATEFLOW documentation, where it is stated: "Do not design your Stateflow diagram based on the expected execution order of transitions."

event. This can lead, semantically, to infinite recursion and in practice (i.e., during simulation) to stack overflow.[9]

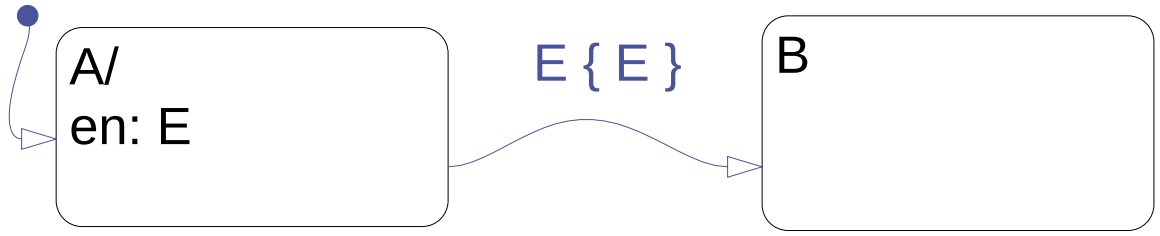

Figure 1: Stack overflow

A model resulting in stack overflow is shown in Figure 1. When the default state A is entered event E is emitted in the entry action of A. E results in a recursive call of the interpretation algorithm and since A is active its outgoing transition is tested. Since the current event E matches the transition event (and because of the absence of condition) the condition action is executed, emitting E again. This results in a new call of the interpretation algorithm which repeats the same sequence of steps until stack overflow.

2.2.2 Backtracking without "undo"

While searching for a valid transition, STATEFLOW explores the segment/junction graph, until a destination state is reached. If, during this search, a junction is reached without any enabled outgoing segments, the search backtracks to the previous junction (or state) and looks for another segment. This backtrack, however, does not restore the values of variables which might have been modified by a condition action. Thus, the search for valid transitions can have side effects on the values of variables.

An example of such a behavior is generated by the model shown in Figure 2. The final value of variable a when state C is entered will be 1011 and not 1001 as might be expected. This is because when the segment with condition "false" is reached, the algorithm backtracks without "undoing" the action "a+=10".

2.2.3 Dependence of semantics on graphical layout

In order to enforce determinism in the search order for active states and valid transitions (thus ensuring that the interpretation algorithm is deterministic) STATEFLOW uses two rules: the "top-to-bottom, left-to-right" rule for states and the "12 o'clock" rule for transitions. These rules imply that the semantics of a model depend on its graphical layout. For example, as the model

[9] This is recognized in the official documentation: *"Broadcasting an event in the action language is most useful as a means of synchronization among AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. Cyclic behavior can be detected only during simulation."*

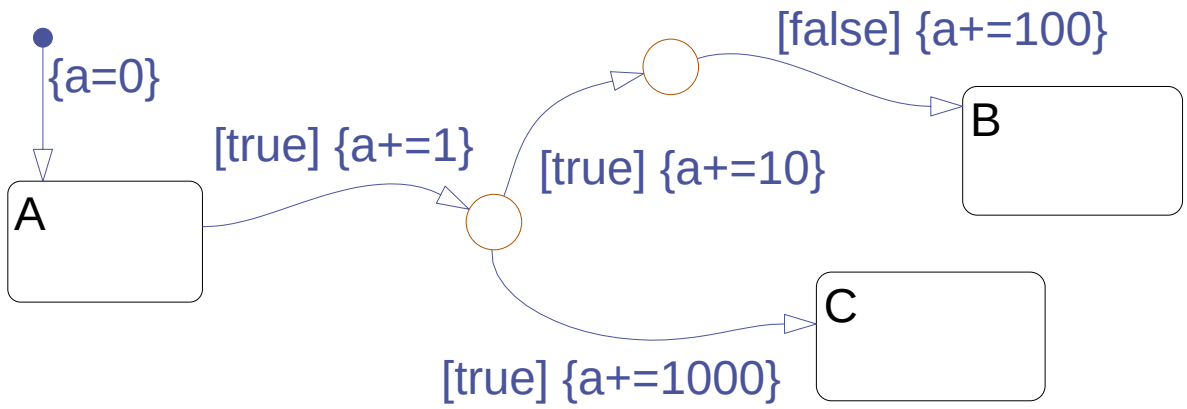

Figure 2: Example of backtracking

is drawn in Figure 3, parallel state A will be explored before B because it is to its left. But if B was drawn slightly higher, then it would be explored first. (Notice that STATEFLOW annotates parallel states with numbers indicating their execution order, e.g., as shown in Figure 3.)

The order of exploration is important since it may lead to different results. In the case of "12 o'clock" rule, for example, if the top-most transition of the model of Figure 2 emanated from the 11 o'clock position instead of the 1 o'clock position, then the final value of a would be 1001 instead of 1011.

Exploration order also influences the semantics in the case of parallel states, even in the absence of variables and assignments. An example is given by the model of Figure 3. A and B are parallel states. When event E_1 arrives, if A is explored first, then E_2 will be emitted and the final global state will be (A_2, B_3). But if B is explored first then the final global state will be (A_2, B_2). Thus, parallel states in STATEFLOW do not enjoy the property of *confluence*.

2.2.4 Other problems

Due to lack of space, we cannot cover all semantical issues with STATEFLOW. We end this part by briefly mentioning two more potential problems. The first is the possibility of having so-called *super-transitions* crossing different levels of the state hierarchy. This is a feature of Statecharts as well, but is generally considered harmful in the Statecharts community [10]. Many proposals disallow such transitions for the sake of simpler semantics [12].

The second problem is termed *early return logic* in the STATEFLOW manual. This problem is illustrated in Figure 4. When event E is emitted, the interpretation algorithm is called recursively. Parent state A is active, thus, its outgoing transition is explored and, since event E is present, the transition is taken. This makes A inactive, and B active. When the stack is popped and execution of the previous instance of the interpretation algorithm resumes, state A_1 is not active anymore, since its parent is no longer active.

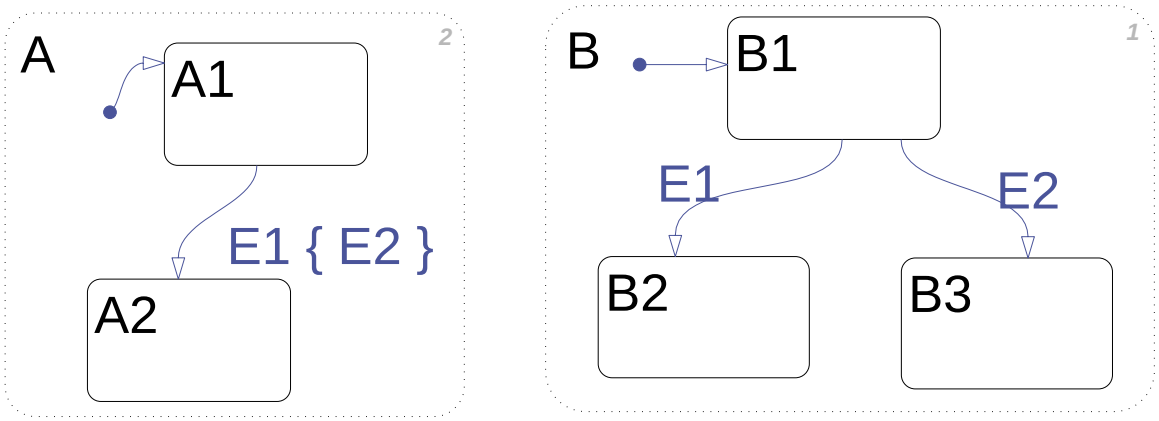

Figure 3: Example of non-confluence

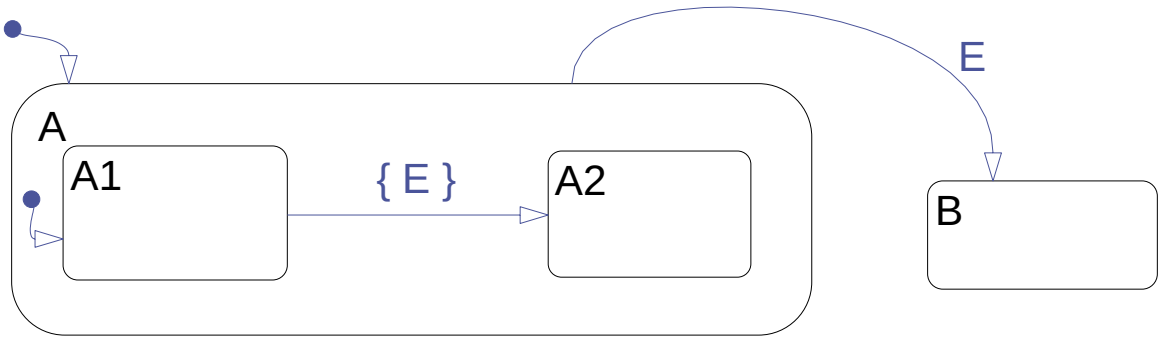

Figure 4: "Early return logic" problem

3 Simple conditions identifying a "safe" subset of Stateflow

In this section we present a sufficient number of simple conditions for avoiding error-prone models such as those discussed previously. The conditions can be statically checked using mostly light-weight techniques. The conditions identify a preliminary, albeit strict, "safe" subset of STATEFLOW. A larger subset can be identified through "heavier" checks such as model-checking, as discussed in Section 6.

Absence of multi-segment loops: If no graph of junctions and transition segments contains a loop (a condition which can easily be checked statically) then the model will not suffer from non-termination problems referred to in Section 2.2.1. This condition is quite strict, however, it is hard to loosen, since termination is undecidable for programs with counters and loops.

Acyclicity in the graph of triggering and emitted events: An event E is said to be *triggering* a state s if the state has a "on event E: A" action or an outgoing transition which can be triggered by E (i.e., E appears in the event field of the transition label or the event field is empty). E is said to be *emitted* in s if it appears in the entry, during, exit or on-event action of s, or in the condition or transition action[10] of one of the outgoing transitions of s. Given a STATEFLOW model, we construct the following graph. Nodes of the graph are all states in the model. For each pair of nodes v and v', we add an edge $v \rightarrow v'$ iff the following two conditions hold:

1. There is an event E which is emitted in v and triggering v'.
2. Either $v = v'$ or the first common parent state of v and v' is a parallel state.

The idea is that v can emit event E which can then trigger v', but only if v and v' can be active at the same time. If the graph above has no directed cycle then the model will not suffer from stack overflow problems.

Absence of assignments in intermediate segments: In order to avoid side effects due to lack of "undo", we can simply check that all variable assignments in a multi-segment transition appear either in transition actions (which are executed only once a destination state has been reached) or in the condition action of the last segment (whose destination is a state and not a junction). This ensures that even in case the algorithm backtracks, no variable has been modified. An alternative is to avoid backtracking altogether, as is done with the following check.

Conditions of outgoing junction segments form a cover: In order to ensure absence of backtracking when multi-segment transitions are explored, we can check that for each junction, the disjunction of all conditions in outgoing segments is the condition *true*. If segments also carry triggering events, we must ensure that all possible emitted events are covered as well.

Conditions of outgoing junction segments are disjoint: In order to ensure that the STATEFLOW model does not depend on the 12 o'clock rule, we must check that for each state or junction, the conditions of its outgoing transitions are pair-wise disjoint. This implies at most one transition is enabled at any given time. In the presence of triggering events, we can relax this by performing the check for each group of transitions associated with a single event E (or having no triggering event).

It should be noted that checking whether STATEFLOW conditions are disjoint or form a cover is an undecidable problem, because of the generality of these conditions. From a STATEFLOW design, we can extract very easily the logical properties expressing that a set of conditions are disjoint and form a cover. These logical properties can be transmitted as a proof obligation to some external tool such as a theorem prover. However, for most practical cases, recognizing common sub-expressions is sufficient for establishing that some conditions are disjoint and form a cover.

[10] In fact, transition action events can probably be omitted from the set of emitted events of s, resulting in a less strict check. We are currently investigating the correctness of this modification.

Checks for confluence: In order to ensure that the semantics of a given STATEFLOW model does not depend on the order of exploring two parallel states A and B, we must check two things. First, that A and B do not access the same variable x (both write x or one reads and the other writes x). But this is not sufficient, as shown in Section 2.2.3, because event broadcasting alone can cause problems. A simple solution is to check that in the aforementioned graph of triggering and emitted events, there is no edge $v \rightarrow v'$ such that v belongs to A and v' to B or vice-versa.

Checks for "early return logic": To ensure that our model is free of "early return logic" problems, we can check that for every state s and each of its outgoing transitions having a triggering event, this event is not emitted somewhere in s. Note that if a transition has no triggering event then this transition is enabled for any event, thus, we must check that no event is emitted in s.

4 A Description Language for Stateflow

Figure 5 defines a simple language in which we express our input STATEFLOW and describe the analysis and translation. We use the convention that capital letters represent lists of syntactic objects, for instance `N` stands for $\emptyset_N$ | `n.N`. This language essentially defines the data structure in the STATEFLOW model files but with some additional elements synthesized to make the analysis simpler. For example, Figures 6 and 7 show a simple STATEFLOW chart and its translation into our language. Our language defines a graph structure with subgraphs, STATEFLOW states and junctions are nodes in the graph, transitions are the vertices of the graph. Note that we actually translate the enclosing SIMULINK into the graph so the top-level syntactic object `sf` is actually a subgraph. Subgraphs have no physical representation in the STATEFLOW model file so we synthesize new subgraphs as the model file is parsed. Similarly, the source for default transitions has no associated object in STATEFLOW so we generate a default point node `(14,_point)` in its place.

This is only a partial representation of the full STATEFLOW model. For example, we do not represent the action language here, of which only a subset can sensibly be translated into LUSTRE. However, we can represent the full STATEFLOW definition in our language, including such features as `inner`, `outer` and `inter-level` transitions.

5 Translation into Lustre

The checks on a STATEFLOW model described in Section 2 define a subset which is much more likely to be correct according to the system designer's intentions than using the full STATEFLOW definition. It is restrictive, however, since it disallows some of STATEFLOW's programming features which designers have become used to. We would therefore like to extend our subset by employing analysis with sound theoretical underpinnings. One such framework is model-checking and we have access to the well-established model-checker called LESAR [8] which takes LUSTRE as its input. A translation of STATEFLOW into LUSTRE therefore opens up the

Graphs			
Stateflow	sf	=	(id, N, L, dd)
Graph	g	=	(id, N, L)
Nodes			
Node	n	=	(id, nt)
Node type	nt	=	p | j | s | sg
Point	p		
Junction	j		
State	s	=	SA
Subgraph	sg	=	(SA, ao, g)
And/Or	ao	=	`AND` | `OR`
State action	sa	=	(sn, a) | (`on`, et, a)
State action name	sn	=	`entry` | `during` | `exit`
Transitions			
Transition	l	=	(ui, sn, dn, et, c, ca, ta)
Source node	sn	=	n
Destination node	dn	=	n
Condition action	ca	=	a
Transition action	ta	=	a
Event or Temp	et	=	$\emptyset_{et}$ | e | t | et_1 `or` et_2
Temp	t	=	(tn, `int`, e)
Temporal name	tn	=	`after` | `before` | `at` | `every`
Actions			
Condition	c	=	$\emptyset_c$ | <*condition code*>
Action	a	=	$\emptyset_a$ | <*action code*>
Data			
Data dictionary	dd	=	(E, D)
Event	e	=	(ui, nm, sc)
Data	d	=	(ui, nm, sc, ty, init)
Scope	sc	=	`INPUT` | `OUTPUT` | `LOCAL` | `TEMP` | `CONST`
Identifiers			
Identifier	id	=	(ui, os)
Unique integer	ui	=	*unique* `int`
Optional string	os	=	"" | `string`
Name	nm	=	*non-empty* `string`

Figure 5: A language defining a subset of STATEFLOW

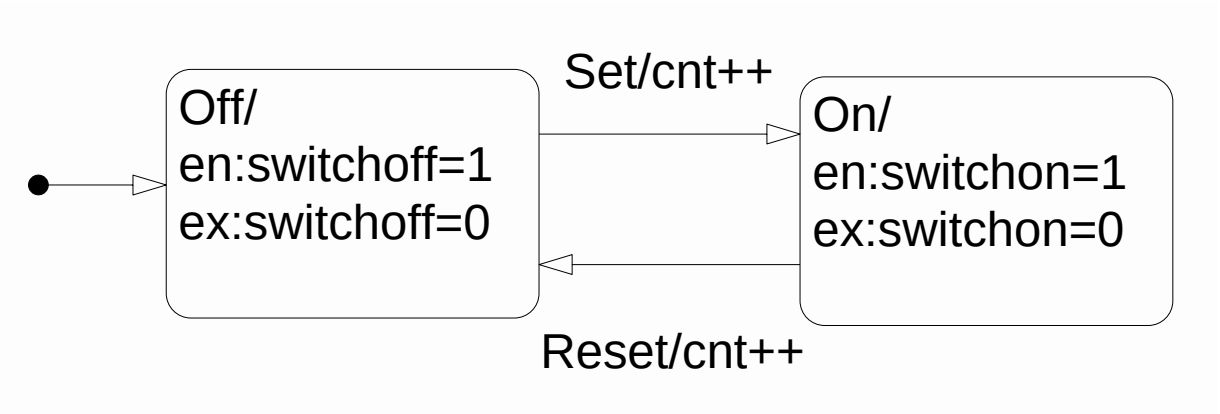

Figure 6: A simple STATEFLOW chart

```
sf=(id=(13,Switch2),
    N=[(id=(3,Off),
        nt=s(SA=[(exit,<switchoff=0>), (entry,<switchoff=1>)])),
       (id=(4,On),
        nt=s(SA=[(exit,<switchon=0>), (entry,<switchon=1>)])),
       (id=(14,_point), nt=p)],
    L=[(ui=5, sn=3, dn=4, et=Set, c=∅c, ca=∅a, ta=<cnt++>),
       (ui=6, sn=4, dn=3, et=Reset, c=∅c, ca=∅a, ta=<cnt++>),
       (ui=7, sn=14, dn=3, et=∅et, c=∅c, ca=∅a, ta=∅a)],
    dd=(E=[(ui=8, nm=Set, sc=INPUT), (ui=9, nm=Reset, sc=INPUT)],
        D=[(ui=11, nm=switchoff, sc=OUTPUT, ty=int, init=∅exp),
           (ui=10, nm=switchon, sc=OUTPUT, ty=int, init=∅exp),
           (ui=12, nm=cnt, sc=OUTPUT, ty=int, init=∅exp)]))
```

Figure 7: The simple chart in the language

possibility of allowing some of the "unsafe" features of STATEFLOW to be used with confidence provided we can verify the intended properties of the model using LESAR.

We have to be clear, however, about the difference between the subset of STATEFLOW which is "safe" in the sense of the previous discussion and that which is translatable into LUSTRE. We can copy the behaviour of STATEFLOW as precisely as required (given sufficient effort in building the translator) and can even implement loops and recursion *provided we can prove that the behaviour is bounded*. The generated program, however, does not have any guaranteed safety properties since all the previous discussion about the semantical problems with STATEFLOW are carried over into the LUSTRE translation. This is where model-checking and other formal methods can be applied. In this section we describe the translation process informally and in Section 6 we show how some of the previously mentioned properties can be verified and our subset extended using the LESAR model-checker.

Needless to say, the goal of the translation is not simply to provide a way to model-check Stateflow models. It is also to allow for semantic-preserving code generation and implementation on uni-processor or multi-processor architectures [5].

LUSTRE is a synchronous language where variables are *flows*, i.e. a notionally infinite stream of values. Each value of a flow is its instantaneous value in a particular *reaction* and for each time instant, *outputs* can only depend on current or previous *inputs*. The previous value of a flow is accessed by the `pre` operator and initialisation is performed by the "followed by" operator `->`. In the translator these are the only temporal operators used. In particular, the `when` and `current` LUSTRE operators are not used, because STATEFLOW models are *single-clock*.

5.1 Encoding of states

The most obvious method of encoding states into LUSTRE is to represent each state as a boolean variable and a section of code to update that variable according to the validity of the input and output transitions. For example, one can envisage a very simple and elegant encoding of the boolean component (i.e. *without* the entry actions) of the example in Figure 6 in the LUSTRE code depicted in Figure 8. Here a state becomes true if any of its predecessor states are true and there is a valid transition chain from that state. It becomes false if it is currently true and there is a valid transition chain to any of its successor states. Otherwise it remains in the same state. The initial values of the states are defined by the validity of the default transitions.

This code is semantically correct for a system consisting only of states but it is difficult to incorporate the imperative actions attached to both states and transitions in STATEFLOW. For example, if the above code had included the entry actions in the states then all the values referenced by the action code would have to be updated in each branch of the if-then tree. This causes two problems. Firstly, for even quite small charts the number of values being updated can become large and this has to be multiplied by the complexity introduced by the network of transitions each state participates in. Secondly, the action language is an imperative language for which it would be difficult to compile a single expression for each sequence of actions. Note also that if more than one state updates the same value then causality loops and multiple definitions could arise.

A more practical approach, therefore, is to split the above equations into their components

```
node SetReset0(Set, Reset: bool)
returns (sOff, sOn: bool);
let
  sOff = true ->
    if pre sOff and Set then false
    else if (pre sOn and Reset) then true
    else pre sOff;
  sOn = false ->
    if pre sOn and Reset then false
    else if (pre sOff and Set) then true
    else pre sOn;
tel.
```

Figure 8: Simple LUSTRE encoding of the example

and use explicit dependencies to force their order of evaluation. Inspecting the code in Figure 8 the state update equation for each state consists of:

- An initialization value computed from default transitions (`true` for `sOff`),
- a value for each outgoing transition (`Set` for `sOff`),
- an exit clause (`(pre sOff and Set)` for `sOff`),
- an entry clause (`(pre sOn and Reset)` for `sOff`) and
- a no-change value (`pre sOff`).

Explicitly separating these components allows us to insert the action code at the correct point in the computation of a reaction. This results in the rather dense encoding shown in Figure 9. Here, the code has been split into several sections.

- **Initial values.** These are the intial values for all variables, `false` for states and the initial value from the data dictionary for STATEFLOW variables.
- **Transition validity.** In this section the values for the transitions are computed. For convenience in the translator these are actually calls to predefined nodes generated in advance from the transitions' events and actions. Note that the test for the activity of the source state is included in the transition's validity test.
- **State exits.** Any states which are `true` and have a valid outgoing transition are set to `false`.
- **Exit actions.** The code for any exiting state's exit actions is computed. This section also includes during actions for states which remain active and `on` actions also for active states.
- **Transition actions.** The code for the transition actions is executed. Note that the exiting state's value is `false` while this occurs.

- **State entries.** Any states which are `false` and have a valid incoming transition are set to `true`.

- **Entry actions.** Entering states action code is executed with the state's variable now `true`.

This sequence corresponds to the sequence of events in STATEFLOW's interpretation algorithm. Note that by "transition valid" we do not mean that the transition is valid with respect to the current context but that this is a transition which will be traversed in the current reaction. Thus the arbitration between competing outgoing transitions has to be resolved by the transition valid computation.

There are some additional complications in the code shown in Figure 9, for instance the use of the `init` and `term` flags which are used to control initialization and termination of subgraphs but these are discussed in the later sections.

```
node SetReset1(Set, Reset, init, term: bool)
returns (sOff, sOn: bool; switchon, switchoff, cnt: int);
var sOff_1, sOff_2, sOn_1, sOn_2, lv5, lv6, lv7: bool;
    switchon_1, switchon_2, switchoff_1, switchoff_2,
    cnt_1, cnt_2: int;
let
  -- initial values
  sOff_1      = false -> pre sOff;
  sOn_1       = false -> pre sOn;
  switchon_1  = 0     -> pre switchon;
  switchoff_1 = 0     -> pre switchoff;
  cnt_1       = 0     -> pre cnt;
  -- link validity
  lv5 = if sOff_1 then Set   else false;
  lv6 = if sOn_1  then Reset else false;
  lv7 = if init and not (sOff_1 or sOn_1) then true else false;
  -- state exits
  sOff_2 = if sOff_1 and (lv5 or term) then false else sOff_1;
  sOn_2  = if sOn_1  and (lv6 or term) then false else sOn_1;
  -- exit actions
  switchoff_2 = if not sOff and sOff_1 then 0 else switchoff_1;
  switchon_2  = if not sOn  and sOn_1  then 0 else switchon_1;
  -- transition actions
  cnt_2 = if lv5 then cnt_1+1 else cnt_1;
  cnt   = if lv6 then cnt_2+1 else cnt_2;
  -- state entries
  sOff = if not sOff_2 and (lv7 or lv6) then true else sOff_2;
  sOn  = if not sOn_2  and lv5 then true else sOn_2;
  -- entry actions
  switchoff = if sOff and not sOff_1 then 1 else switchoff_2;
  switchon  = if sOn  and not sOn_1  then 1 else switchon_2;
tel.
```

Figure 9: Alternative LUSTRE encoding of the example

5.2 Compiling transition networks

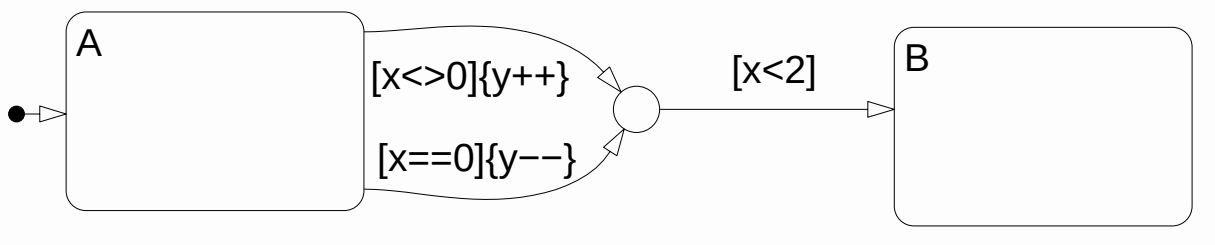

Figure 10: A STATEFLOW chart with a junction

Figure 10 shows a STATEFLOW chart with a junction. Junctions in STATEFLOW do not have a physical state and can be thought of as nodes in an `if-then` tree. This is thus the most sensible encoding of junctions. One problem, however, is that junction networks can be sourced from more than one state and a single state can have more than one output to the same junction. These can be handled quite easily if one allows a certain amount of code duplication, the common subnetwork for two joining outgoing transitions being compiled twice.

We could devise a very natural scheme for LUSTRE to handle this but again it becomes difficult to insert the condition and transition actions into the `if-then` tree in LUSTRE. Figure 11 shows the actual code generated[11]. The functions `cv{678}_` not shown compute the condition code for their respective transitions. Note how the `cv8_` call is duplicated between `lv6_` and `lv7_`. Essentially, the junction tree is turned into a flattened representation with two flags, "`end`" which signifies the termination of the tree (either a destination state or a terminal junction) and "`exit`" which is true if the terminal was a state. One slight inefficiency is the use of these flags to defeat further computation after the termination point is reached (the `not end` clauses). These two flags correspond to the `End`, `No` and `Fire` transition values in [9], the semantics of our junction processing is identical to the semantics described therein.

There is also a slight problem with the "transition valid" section in the code shown in Figure 9. For the example shown there can only ever be one transition valid flag `true` at each instant but when a state has (potentially competing) outgoing transitions there has to be some kind of arbitration between them, hopefully using the same arbitration as STATEFLOW itself. In fact the statements are chained together with a common flag which indicates when a valid transition has been found. This is called the `ok` variable and a revised transition validity computation section is shown in Figure 12. In fact we need a separate `ok` flag *for each subgraph*, this is explained later when inter-level transitions are discussed.

A more serious problem is that junction networks can have loops which results in unbounded recursion and therefore a loss of synchronous semantics. Figure 13 shows a simple `for`-loop. There are a number of possibilities for handling this.

[11] Our code examples have been condensed for brevity and use abbreviated variable names. `cv` means "condition valid", `lv` "transition valid", `ca` "condition action" and `su` "state update"

```
-- link id=7 name=[x<>0]{y++}
node lv7_(x, y: int; ok, lv7, lv8: bool)
returns(yo: int; oko, lv7o, lv8o: bool);
var cv7, cv8, end, end_1, end_2, ok_1: bool;
let
  end_1 = false;
  ok_1, cv7, end_2 =
    if (not (end_1 or ok)) then cv7_(x) else (ok, false, end_1);
  yo = if cv7 then ca7(y) else (y);
  oko, cv8, end =
    if ((not end_2) and cv7) then cv8_(x) else (ok_1, false, end_2);
  lv7o, lv8o = if (cv8 and end) then (true, true) else (lv7, lv8);
tel

-- link id=6 name=[x==0]{y--}
node lv6_(x, y: int; ok, lv6, lv8: bool)
returns(yo: int; oko, lv6o, lv8o: bool);
var cv6, cv8, end, end_1, end_2, ok_1: bool;
let
  end_1 = false;
  ok_1, cv6, end_2 =
    if (not (end_1 or ok)) then cv6_(x) else (ok, false, end_1);
  yo = if cv6 then ca6(y) else (y);
  oko, cv8, end =
    if ((not end_2) and cv6) then cv8_(x) else (ok_1, false, end_2);
  lv6o, lv8o = if (cv8 and end) then (true, true) else (lv6, lv8);
tel

-- node id=3 name=A
node suAlv(x, y: int; ok, sA, trm, ini: bool)
returns(yo: int; oko, lv6, lv7, lv8: bool);
var lv8_1, ok_1: bool; y_1: int;
let
  y_1, ok_1, lv6, lv8_1 = lv6_(x, y, ok, false, false);
  yo, oko, lv7, lv8 = lv7_(x, y_1, ok_1, false, lv8_1);
tel
```

Figure 11: Code generated for the junctions example

```
ok_1 = false;
lv5, ok_2 = if not ok_1 and sOff_1 then (Set, Set)     else (false, ok_1);
lv6, ok_3 = if not ok_2 and sOn_1  then (Reset, Reset) else (false, ok_2);
lv7, ok   = if not ok_3 and init and not (sOff_1 or sOn_1)
            then (init, init) else (false, ok_3);
```

Figure 12: Chaining together transition valid computations

Junctions as states. An easy solution would be to give junctions a physical state in the executing LUSTRE program. This effectively moves the non-termination problem outward into the code calling the STATEFLOW model but also moves the burden of the proof of non-termination to the client code. This has been implemented in our translator where we also provide an additional status flag called "`valid`" as an output which is `true` only if the current state is not a junction. In theory, the client code could loop over the STATEFLOW code until this flag becomes `true` at which point the other outputs are also valid.

Loop unrolling with external proof obligations. This is unsatisfactory from the point of view of using the translator as a development tool. We would prefer to simply impose a synchronous semantics upon STATEFLOW and outlaw such constructs if they cannot be proven to be bounded. Given a synchronous semantics for STATEFLOW we have to outlaw such constructs in the general case. It is possible, however, to unroll such loops (Figure 13 also shows the expansion of the simple loop) without loss of generality, provided bounds can be proven on the number of iterations. This means we can generate proof obligations for external tools such as Nbac [11]. If a bound exists and is feasible we can unroll loops individually as required. This requires further investigation. Currently, we detect all junction loops and reject models which have them.

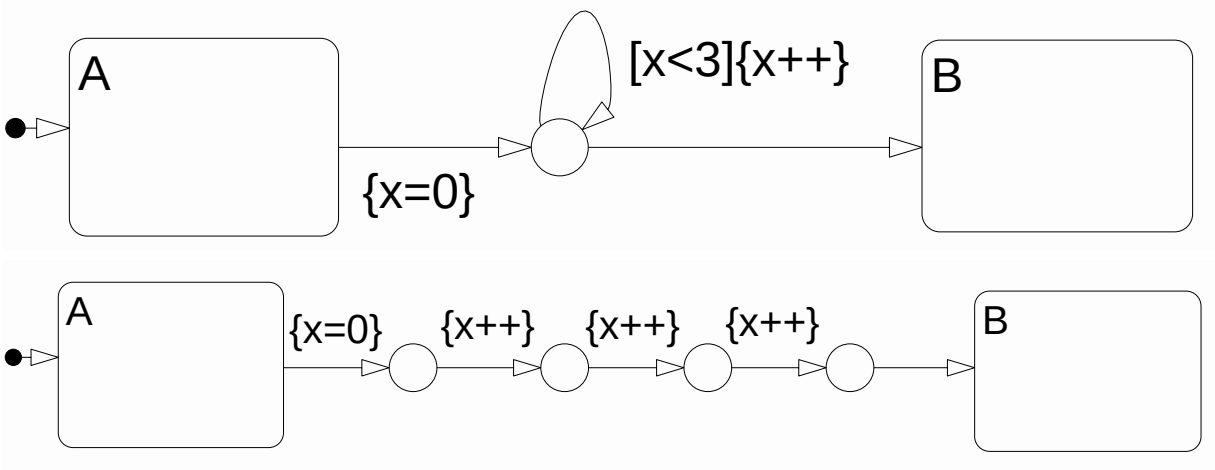

Figure 13: A `for`-loop implemented in STATEFLOW junctions and its expansion

5.3 Hierarchy and parallel AND states

We initially make the assumption that supertransitions are not allowed. This restriction could be removed in future since there is no reason why they could not be implemented but the analysis of hierarchical and parallel states is greatly simplified by this assumption. In fact the entire hierarchy boils down to simple function calls of nested states, the only complication being the initialization and termination of the nested states.

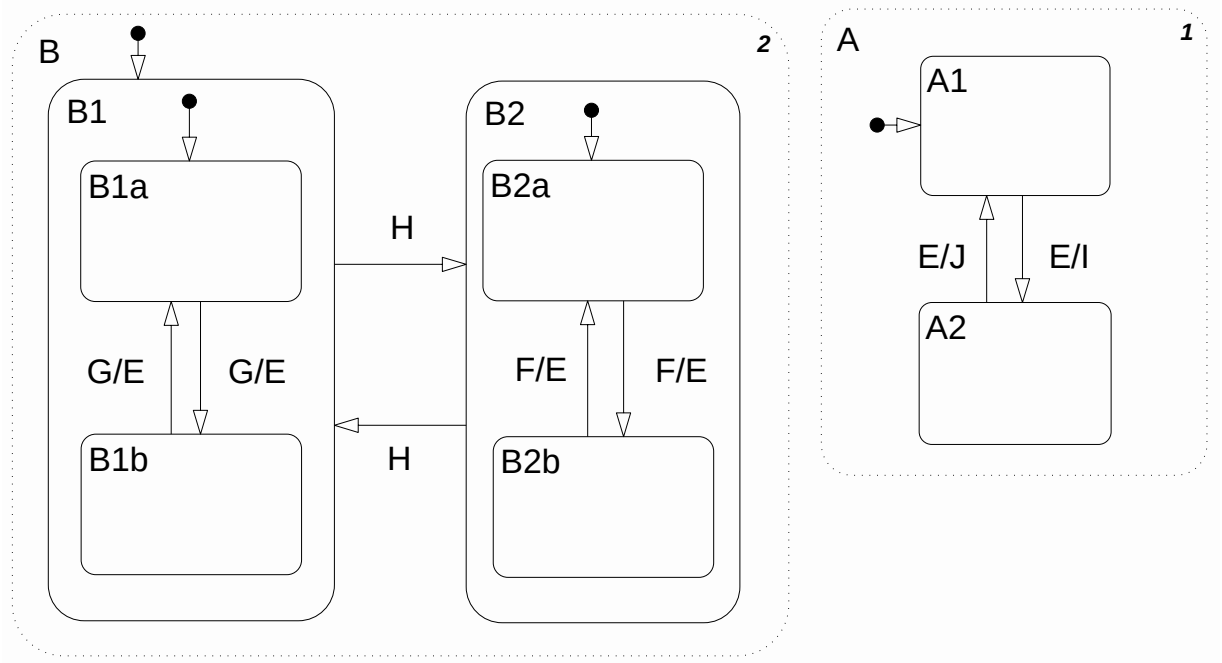

Figure 14: A model with parallel (AND) and exclusive (OR) decompositions

For example, Figure 14 illustrates a simple model with both parallel and exclusive substates. For both types of substate we insert the function calls to the substates after computation of the local state variables, the LUSTRE nodes generated for the top-level state (parallel) and state `B` (exclusive) for this model are depicted in Figure 15.

Initialization and termination are controlled by two variables, "`init`" and "`term`" which are passed down the hierarchy. This is a standard method for implementing state machines in synchronous languages [13]. One way of viewing the `init` value is as a *pseudo-state* which the model is in prior to execution and in fact this plays the rôle of the state variable for default transitions. For parallel states the local state variable depends only on the `init` and `term` variables, as do the flags for entry, exit and during actions. These are computed as in Figure 16 (`s` is the local state variable) and are embodied in auxiliary nodes (for example the state variable is computed by the node `sfs` in Figure 15).

For exclusive substates the `init` and `term` flags are computed solely from the local state variable (`init = s and not pre s` and `term = not pre s and s`). The complication is that we need the value of the state variable at the end of the reaction without actually setting the variable itself because the nested states have to be executed using the input value. This is why we call the state entry computation beforehand (`sgu8_B1en` for example) but save the value in a temporary variable (`sg8_B1t`) and then update the actual value at the end of the computation. The temporary value then stands for the new value and the input value (`sg_8B1in`) for the previous one. Actually, for the code presented here this is unnecessary but when event broadcasting is enabled (Section 5.6) the value of the state variable can be updated by actions.

```
-- State B (OR,[B1,B2])
node sf_7(F,G,H,E: event; okB2,okB1,okSubgraph36,sB2ain,sB2bin,sB1ain,
          sB1bin,sgB2in,sgB1in,sgB,trm, ini: bool)
returns(okB2o,okB1o,okSubgraph36o,sB2a,sB2b,sB1a,sB1b,sgB2,sgB1: bool;
        Eo: event);
let
  ...
  sgB1t = sguB1en(okSubgraph36o,lv16,lv18,sgB1_1,trm,ini);
  sgB2t = sguB2en(okSubgraph36o,lv17,sgB2_1,trm,ini);
  okB1o,sB1a,sB1b,E_1 =
    sf_8(G,E,okB1,okSubgraph36o,lv17,sB1ain,sB1bin,sgB1t,
         ((not sgB1t) and sgB1in),(sgB1t and (not sgB1in)));
  okB2o,sB2a,sB2b,Eo =
    sf_11(F,E_1,okB2,okSubgraph36o,lv18,sB2ain,sB2bin,sgB2t,
          ((not sgB2t) and sgB2in),(sgB2t and (not sgB2in)));
  ...
tel

-- Toplevel graph (AND,[A,B])
node sf_2(F,G,H: event) returns(I,J: event);
let
  ...
  sgA = sfs(ini,trm);
  J,I,okA,sA1,sA2 = sf_4(E_1,I_1,J_1,okA_1,sA1_1,sA2_1,sgA,trm,ini);
  sgB = sfs(ini,trm);
  okB2,okB1,okSubgraph36,sB2a,sB2b,sB1a,sB1b,sgB2,sgB1,E =
    sf_7(F,G,H,E_1,okB2_1,okB1_1,okSubgraph36_1,sB2a_1,sB2b_1,sB1a_1,
         sB1b_1,sgB2_1,sgB1_1,sgB,trm,ini);
  ...
tel
```

Figure 15: LUSTRE code fragments for parallel and hierarchical states

state	(init and not term) -> (init or pre s) and (not term)
entry	init -> s and not pre s
exit	(init and term) ->
((pre s or ((not pre s) and init)) and (not s))
during	false -> s and pre s

Figure 16: Computation of parallel state variables

Note also that for the top-level call we set `init` to `true->false` and `term` to `false`.

5.4 Inter-level and inner transitions

The methods described so far work in a natural way for STATEFLOW charts which are structured as trees, which allows the LUSTRE code also to be structured as a tree. One consequence of this is that we can map states onto LUSTRE nodes and still retain the same action sequences as STATEFLOW. STATEFLOW, however, allows inter-level transitions, ie. between states not at the same level of the node hierarchy which means that the model becomes a more general graph structure rather than a tree. This in itself does not break any of the characteristics of a synchronous implementation but it does greatly complicate the translation. As such, early versions of the translator simply outlawed transitions of this type in favour of a much simpler analysis. A large amount of legacy STATEFLOW code uses inter-level transitions, however, so a preliminary version of our translator which can handle inter-level transitions has been developed.

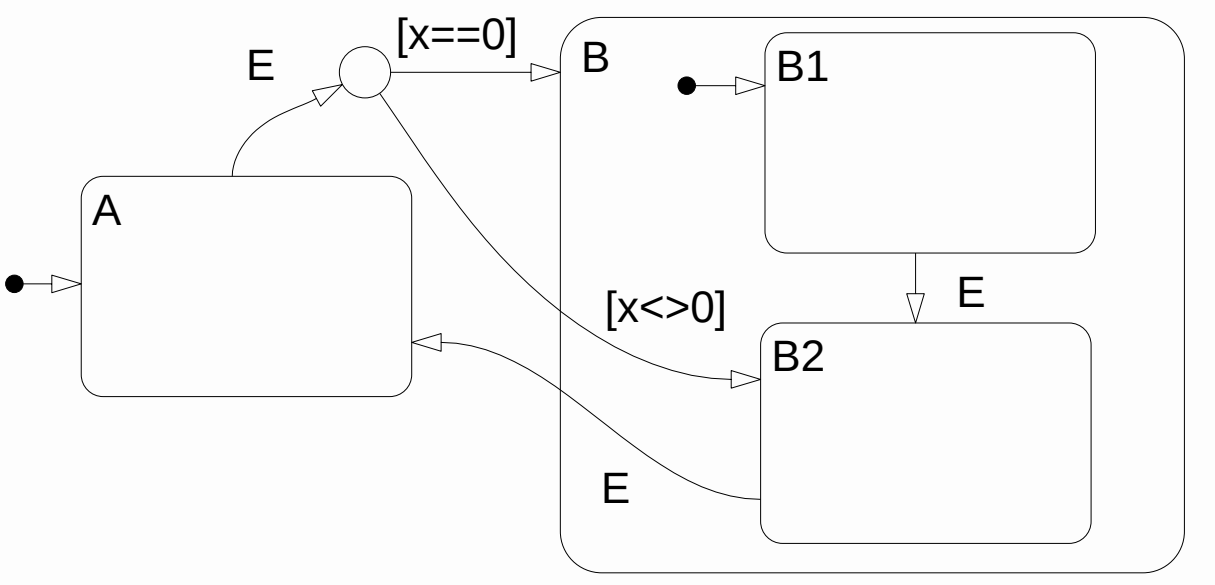

Figure 17: A model with inter-level transitions

Figure 17 illustrates a simple STATEFLOW chart with an inter-level transition network, from `A` to `B` and `B2`. Figure 18 shows the different kinds of inner transitions that can be used. The top transition `[x==0]` is an inner transition which terminates in the parent state `A`, transitions `[x==1]` and `[x==2]` show inner transitions to and from a substate of `A` and transition `[x==3]` terminates in a junction (this style of inner transition is known as a *flowchart* in STATEFLOW terminology).

These charts show a number of problems with inter-level transitions:

- The inter-level transition from the junction to `B2` in Figure 17 acts *in lieu* of a default

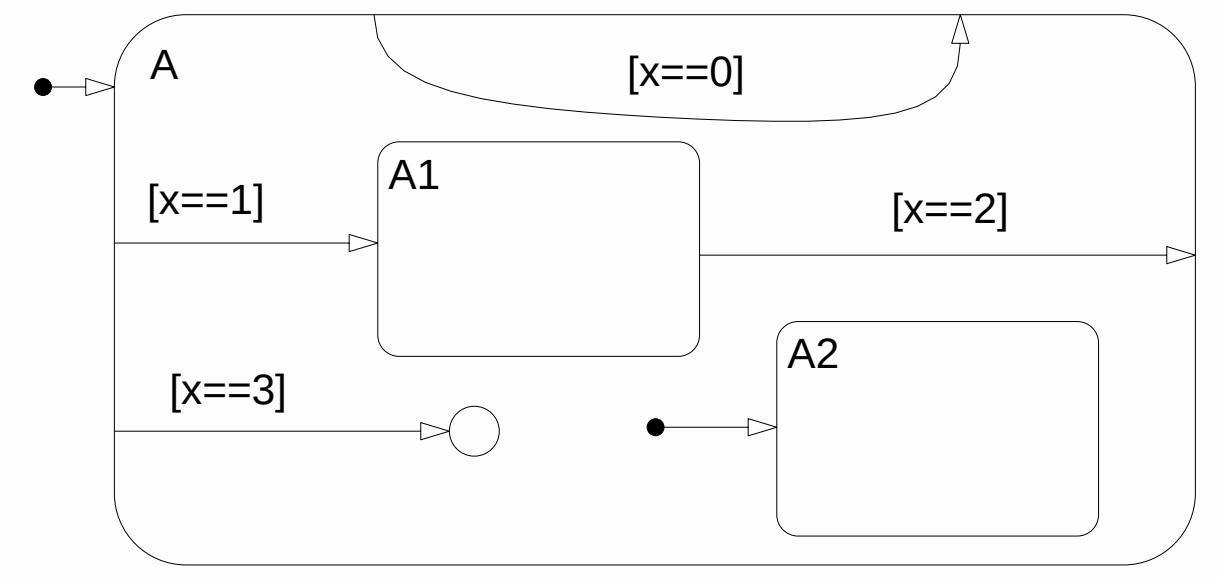

Figure 18: A model with inner transitions

transition when it is taken so any default transition in the states traversed by the path have to be ignored.

- Any transition which traverses a state inwards results in activation of that state and likewise any transition which traverses outwards results in deactivation of the state.

and inner transitions:

- The semantics of inner transitions mean that, for example, transition `[x==1]` in Figure 18 acts as a default transition for state `A` *when this transition results in termination of a currently active substate but not when state `A` is entered from outside*. The other three transitions do not have this property since none of them terminate in an internal state.
- State `A` neither exits nor enters when an inner transition is taken and its `during` actions are executed *before* the inner transitions are taken. Thus, if either of the transitions `[x==0]` or `[x==2]` are taken state `A2` is reached. Note, however, that if `A1` is the active substate and transition `[x==3]` is taken then state `A1` remains active.
- Note that transitions `[x==0]`, `[x==1]` and `[x==3]` are considered to emanate from the same source and thus require arbitration and are subject to STATEFLOW's check for multiple valid transitions. They also take precedence over default transitions when an inner transition is taken.
- Only one inner transition can be taken at a time so that if state `A1` exits on transition `[x==2]` it cannot return on transition `[x==1]`.

- Flowchart transitions are taken, if valid, each time the state is active and a higher priority transition is not valid. Inner transitions are pritoritized according to the 12 o'clock rule so that they are checked in the order `[x==0]`, `[x==3]` then `[x==1]`. Inner transitions from the parent state take precedence over those emanating from substates. They do not result in a change of state and are evaluated purely for their side-effects.

In addition, a transition network can be *mixed* ie. has paths through it which can be inter-level, inner, flowchart or normal paths, or an arbitrary combination of all of them. Note also that both inner and inter-level transitions can lead to inconsistent states if not implemented properly. This results in a highly complex semantics for STATEFLOW transitions which would be extremely difficult to emulate precisely. The semantics in [9] follows STATEFLOW's interpretation algorithm very closely but is essentially an imperative method which would be difficult to adapt to LUSTRE's synchronous semantics.

We have, instead, implemented a compromise solution which behaves in a very similar manner to STATEFLOW with some distortions on the state, condition and transition actions. This solution is based on splitting transition networks into separate *paths* and associating them with the *outermost* point traversed by any transition in the path. Evaluation then proceeds top-down as before but computing transition validity when the transitions come into scope. The results of this computation can then be passed down the hierarchy. For instance, the transitions labeled `E` and `F` in Figure 19 are computed at the top level of the hierarchy and then flags corresponding to their validity are passed as arguments to the nodes generated for states `A` and `B`. States `A1` and `B1` then include these additional parameters in their entry and exit clauses.

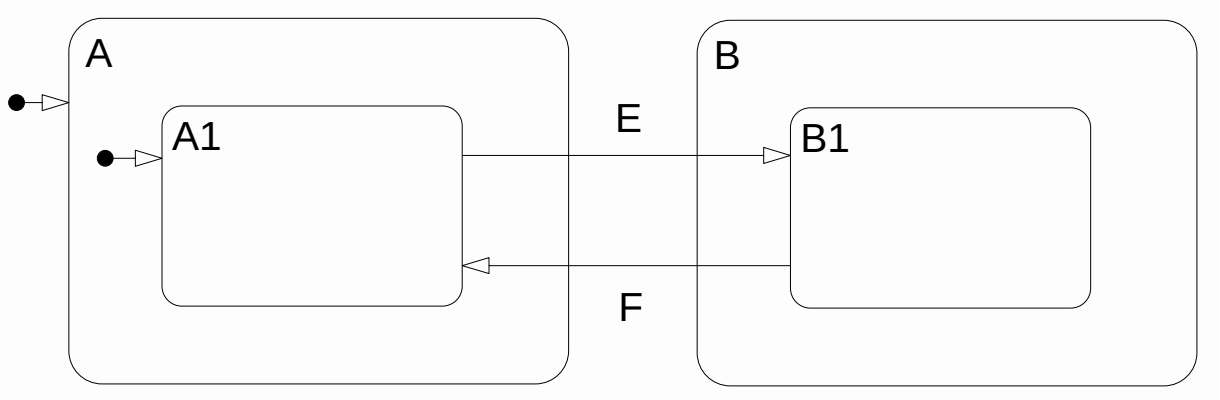

Figure 19: Inter-level transitions with action order distortion

The problem then arises as to how to ensure that the sequence of exit action followed by transition action followed by entry action is in the correct order. If substates are checked in a fixed order then at least one of transitions `E` or `F` must be evaluated in reverse, ie. the entry and exit actions will be executed in the wrong order. Several solutions are possible:

- We could dynamically order the calls to the nodes for `A` and `B` according to which transitions have been computed as valid.

- We could move the entry, transition and exit actions to either the source for the transition path, the outermost scope of the transition path or the destination of the transition path.
- We could lift all the actions to the top-level node in the hierarchy and impose an order on the actions based upon some abtraction of STATEFLOW's interpretation algorithm.

All of these options would result in other more subtle distortions in the actions as compared to STATEFLOW. They also have the additional complexity of computing all the entry and exit actions for states along the paths traversed.

Currently, none of these options are implemented so we can guarantee the correct order of action execution only for *for non-inter-level transitions*. We can, however, guarantee that all actions which would have been executed within a single LUSTRE reaction will get executed in some order.

5.4.1 Inter-level transitions

The basic scheme, however, is relatively easy to implement for inter-level transitions provided we are careful to compute the correct arguments (transitions) to the substate nodes.

The only major complication is the computation of the `ok` value for inter-level transitions. Because of the presence of default transitions we need an `ok` flag for each substate because the computation of transition validity is disjoint for each default transition taken within the hierarchy. We also need a separate `ok` flag for each parallel state because transition computations are also disjoint between parallel states. Luckily, STATEFLOW outlaws inter-level transitions between parallel states but we still need a flag for each subgraph because of default transitions. This means that we need to associate an `ok` value with each transition (in fact we associate it with the flag for its *source* graph) so that the transition is only valid if both its validity flag and associated `ok` flag are true.

Figure 20 shows the code produced for Figure 17. This is a direct implementation of the scheme described above. Points to note about this code include:

- transitions 8 (`default for A`), 9 (`A to junction`), 10 (`B2 to A`), 12 (`junction to B`) and 13 (`junction to B2`) are all computed at the top-level, of which 9, 10 and 13 are passed to the node for subgraph `B`,
- the node for state `B` augments these with the transitions 11 (`B1 to B2`) and 14 (`default for B1`),
- the complex predicate for the default transition to state `B1` has to take into account whether state `B` is being entered by inter-level transition 13 or normal transition 12,
- the distortions in the actions (state `A` enters before `B2` exits if transition 10 is taken) and
- the computation of the `ok` flags, for example, transition 12 uses `okTop` whereas transition 10 uses `okB`.

```
-- link id=8  name=        point -> A
-- link id=9  name=E       A -> junction
-- link id=10 name=E       B2 -> A
-- link id=11 name=E       B1 -> B2
-- link id=12 name=[x==0]  junction -> B
-- link id=13 name=[x<>0]  junction -> B2
-- link id=14 name=        point -> B1

-- graph id=17 name=B,NONTOP
node sf_4(E:event; okB,okTop,lv9,lv10,lv13,sB1in,sB2in,sgB,trm,ini:bool)
returns(okBo,sB1,sB2:bool);
var lv11,lv11_1,lv14,lv14_1,okB_1,sB1_1,sB2_1:bool;
let
  lv11_1,lv14_1=(false,false);
  okB_1,lv11=if sB1in then suB1lv(E,okB,sB1in,trm,ini) else (okB,lv11_1);
  okBo,lv14=if not ((okTop and lv13) and (okTop and lv9))) and
              (ini and (not (sB2in or sB1in))
            then iniu19__pointlv(okB_1,trm,ini) else (okB_1,lv14_1);
  sB2_1=if sB2in then suB2ex(okBo,lv10,sB2in,trm,ini) else (sB2in);
  sB1_1=if sB1in then suB1ex(okBo,lv11,sB1in,trm,ini) else (sB1in);
  sB2=suB2en(okBo,okTop,lv11,lv9,lv13,sB2_1,trm,ini);
  sB1=suB1en(okBo,lv14,sB1_1,trm,ini);
tel

-- graph id=18 name=Top,GCTOP
node sf_2(E:event; x:int) returns(sB1,sB2,sgB,sA:bool);
var ini,lv10,lv10_1,lv12,lv12_1,lv13,lv13_1,lv8,lv8_1,lv9,lv9_1,okB,okB_1,
    okB_2,okTop,okTop_1,okTop_2,okTop_3,sA_1,sA_2,sAt,sB1_1,sB2_1,
    sgB_1,sgB_2,sgBt,trm:bool;
let
  sA_1=false -> pre sA; sgB_1=false -> pre sgB;
  sB2_1=false -> pre sB2; sB1_1=false -> pre sB1;
  okTop_1,okB_1=(false,false);
  lv10_1,lv9_1,lv12_1,lv13_1,lv8_1=(false,false,false,false,false);
  okB_2,okTop_2,lv10=
    if sB2_1 then suB2lv(E,okB_1,okTop_1,sB2_1,trm,ini)
    else (okB_1,okTop_1,lv10_1);
  okTop_3,lv9,lv12,lv13=
    if sA_1 then suAlv(E,x,okTop_2,sA_1,trm,ini)
    else (okTop_2,lv9_1,lv12_1,lv13_1);
  okTop,lv8=
    if ini and not (sA_1 or sgB_1)
    then iniu20__pointlv(okTop_3,trm,ini) else (okTop_3,lv8_1);
  sA_2=if sA_1 then suAex(okTop,lv9,lv12,lv13,sA_1,trm,ini) else (sA_1);
  sgB_2=if sgB_1 then sguBex(okB_2,lv10,sgB_1,trm,ini) else (sgB_1);
  sA=suAen(okB_2,okTop,lv8,lv10,sA_2,trm,ini);
  sgB=sguBen(okTop,lv9,lv12,lv13,sgB_2,trm,ini);
  okB,sB1,sB2=sf_4(E,okB_2,okTop,lv9,lv10,lv13,sB1_1,sB2_1,sgB,
                   sgB and trm -> (not sgB) and (pre sgB),
                   sgB -> sgB and not (pre sgB));
tel.
```

Figure 20: LUSTRE code fragments for inter-level transitions

5.4.2 Inner transitions

Although we have treated inter-level and inner transitions separately here, they are intimately interlinked due to the possibility of a single path through a transition network having transitions of both types. It is even possible for a single transition to be of both types. In some ways, inner transitions are simpler than inter-level transitions since they are nearly local (only involving the immediate parent state) but are more complicated in the way they interact with other transitions at the same level.

```
-- link id=7  name=[x==0] A -> A
-- link id=8  name=[x==1] A -> A1
-- link id=9  name=[x==2] A1 -> A
-- link id=10 name=[x==3] A -> j6
-- link id=11 name=       p17 -> A
-- link id=12 name=       p16 -> A2

-- node id=3 name=A
node sguAlv(x:int; okA,sgA,trm,ini:bool) returns(okAo,lv7,lv8,lv10:bool);
var okA_1,okA_2:bool;
let
  okA_1,lv7=lv7_(x,okA,false);
  okA_2,lv8=lv8_(x,okA_1,false);
  okAo,lv10=lv10_(x,okA_2,false);
tel

-- graph id=14 name=A,NONTOP
node sf_3(x:int; okA,sA1in,sA2in,sgA,trm,ini:bool)
returns(okAo,sA1,sA2:bool);
var inner,lv10,lv10_1,lv12,lv12_1,lv7,lv7_1,lv8,
    lv8_1,lv9,lv9_1,okA_1,okA_2,sA1_1,sA2_1:bool;
let
  lv9_1,lv7_1,lv8_1,lv10_1,lv12_1=(false,false,false,false,false);
  okA_1,lv9=if sA1in then suA1lv(x,okA,sA1in,trm,ini) else (okA,lv9_1);
  okA_2,lv7,lv8,lv10=if sgA and (not ini)
                     then sguAlv(x,okA_1,sgA,trm,ini)
                     else (okA_1,lv7_1,lv8_1,lv10_1);
  inner=((okA_2 and lv9) or (okA_2 and lv7)) and
        (not ((okA_2 and lv10) or (okA_2 and lv8)));
  okAo,lv12 =if inner or (ini and (not (sA2in or sA1in)))
             then iniu16__pointlv(okA_2,inner,trm,ini)
             else (okA_2,lv12_1);
  sA2_1=if sA2in
        then suA2ex(okAo,lv7,lv8,lv10,sA2in,trm,ini) else sA2in;
  sA1_1=if sA1in
        then suA1ex(okAo,lv7,lv8,lv9,lv10,sA1in,trm,ini) else sA1in;
  sA2=suA2en(okAo,lv12,sA2_1,trm,ini);
  sA1=suA1en(okAo,lv8,sA1_1,trm,ini);
tel.
```

Figure 21: LUSTRE code fragments for inner transitions

Figure 21 shows the code produced for Figure 18. Only the node for state A is shown since

all inner transition activity is computed at this level. The main problem is to arrange the priorities of the transitions in the correct order. This is achieved with the aid of an additional flag, called `inner`, which is `true` when an inner transition is about to be taken which terminates in the parent state. This is needed to control the default transition (transition 12 to state `A2`). If an inner transition terminates in the parent state (transitions 7, `[x==0]` and 9, `[x==2]`) then the default transition is traversed. Inner transitions which terminate in either substates (transition 8, `[x==1]`) or junctions (transition 10, `[x==3]`) result in the default transition being overridden. This results in the complex boolean condition for `inner` in Figure 21.

The remainder of the computation of transition validity is relatively simple given this flag:

- Transition 9 is computed as an ordinary transition between substates since its source is the substate `A1`.
- The inner transitions 7, 8 and 10 should be prioritized according to the STATEFLOW model. Unfortunately, we have so far been unable to deduce this order from the STATEFLOW model file so they are current computed in arbitrary order.
- Once all the inner transitions are computed the `inner` flag can be generated.
- Once the `inner` flag is produced the default transitions can be computed if either the normal default transition conditions exist or the `inner` flag is `true`. Note that the `inner` flag is passed to the function which computes the default transition's validity.
- The rest of the code is same as if there were no inner transitions.

This behaviour is fairly accurate with respect to executions of STATEFLOW charts with some very subtle departures, for example we do not handle inner transitions to history junctions correctly.

5.5 Action language translation

There are two basic options for translating the simple imperative language implemented by STATEFLOW into the synchronous language LUSTRE. One possibility would be to generate C code from the action code and use the external function call facility of LUSTRE to call the action code. This has appeal since this translation would be essentially a one-to-one correspondence between semantic objects. However, the model-checking and other verification tools are unable to work with embedded C code and we lose expressive power for our system. The alternative, and harder, approach is to translate the action code into LUSTRE. The problem here is that we need to impose a sequential order on the generated LUSTRE statements which matches the execution order in the STATEFLOW. We also have efficiency problems since any values in the context not referenced by the action code have to be copied across but we are not concerned with the efficiency of the generated LUSTRE code at this point.

5.5.1 Pseudo-lustre

To ease this translation we have defined a simple sequential subset of LUSTRE characterised by the following properties:

- LUSTRE statements are considered to be evaluated from top to bottom.
- Any inputs which are updated have an output value created for them[12].
- Any outputs referenced before their first definition have inputs created for them.
- Values referenced within `pre` statements are not considered as instances.
- Values on the left hand side of the equations are made unique.
- References to sequenced values on the right hand side are transformed to refer to the most recent instance.

```
-- a) Untransformed
node test(x: int)
returns(y: int);
let
  x = x + y;
  x = x + 1;
  y = y + 1;
tel.
```

```
-- b) Transformed
node test(x, y_in: int)
returns(y, x_out: int);
var x_1: int;
let
  x_1 = x_in + y_in;
  x_out = x_1 + 1;
  y_out = y_in + 1;
tel.
```

Figure 22: Transformation of pseudo-LUSTRE

For example, Figure 22 shows the transformation of a simple test node. This transformation allows us to virtually transliterate the action code directly into LUSTRE with minimal alteration. In fact, this style of LUSTRE is also useful for code generated elsewhere and is used ubiquitously in the translator.

5.5.2 Stateflow arrays to Lustre arrays

The only significant complication is arrays for which we synthesize access code which allows them to behave as variables. For example, the action code `x[0]++`, where `x` has type `int^3` is translated into:

```
xo = aset1_i(3, 0, aget1_i(3, 0, x) + 1, x);
```

We synthesize a function "`a(get|set|fill)<n>_<ts>`" for each `<n>`-dimensional array value of type `<ts>`. Note that each time an array value is accessed or updated the entire array is searched or copied resulting in very inefficient code.

[12] Created output variables are suffixed with the string "`_out`" (or simply "`o`" in abbreviated form) and created inputs are suffixed with "`_in`". Unique variables are suffixed by an integer.

5.5.3 Temporal logic operators

```
-- Counter function for temporal events
node sfcnts(s, E: bool) returns(a: bool; cnt: int);
var inc: int;
let
   a = s -> s and not pre s;
   inc = if a and E then 1 else 0;
   cnt = inc ->
     if a then inc
     else if s and E
          then (pre cnt) + 1
          else pre cnt;
tel

-- after function for temporal events
node sfaft(n: int; s, E: bool) returns(flg: bool);
var a: bool; cnt: int;
let
   a, cnt = sfcnts(s, E);
   flg = n >= 0 and s and (cnt >= n);
tel
```

Figure 23: Counter function for temporal logic

A similar complication arises for temporal logic code, for which we synthesize auxiliary LUSTRE routines. Figure 23 shows the synthesized code for the `after` temporal operator. The main issue is that the counter is only incremented when the state is active so we have to pass the associated state variable to the counter function. The code shown here simply tests if the current count (`cnt`) is greater than the required number of counts (`n`), the code for `before`, `at` and `every` being similar.

5.6 Event broadcasting

One of the most difficult aspects of STATEFLOW to translate is the generation of events *within the* STATEFLOW *model*, these are called *local* events in STATEFLOW terminology. The problem is that STATEFLOW implements these by running the interpretation algorithm to completion on each transmitted local event which implies the possibility of unbounded behaviour (since transmission of one event can trigger the transmission of another). On the other hand, LUSTRE provides a bounded (and known at compile time) recursion mechanism. Therefore, if we can prove (or assume) that the implicit recursion is bounded by a constant k, then we can translate the STATEFLOW model into a LUSTRE program with recursion bounded by k.

Up to now, nothing we have described implies any kind of recursive behaviour in the translator, we could simply generate the code by preserving the hierarchy in the original STATEFLOW model. Now, however, we have to know the arguments to the top-level call when we implement a broadcast event. We could either make the translator a fix-point computation where the arguments to previously generated graphs are updated when event broadcasts happen, we could use

a two-pass method where the first pass computes the nodes and arguments and the second generates the code or, since the *only* recursion point is the top-level call we could simply predict the arguments to this node and use that for the broadcasts. Currently, we use the last (and simplest) option but if, for example, we were to implement the `send` function as a function call to the relevant node we would require a more general analysis.

Another slight complication is that LUSTRE will not accept a constant value for the top-level node. Bounded recursion requires the presence of a recursion variables which have to be statically evaluated. We thus generate a proxy node for the top-level call and seed this with the value of the recursion variable. We implement bounded recursion by creating a `const`[13] recursion variable for event broadcasts which we call the "event stack size". We can then call the top-level node at the point where an event is broadcast, reducing this constant by one. This allows emulation of the recursive nature of STATEFLOW's interpretation algorithm *up to a finite limit set by the event stack size*. If we have a proof of the bound on event broadcast recursion then our behaviour will be the same as STATEFLOW's.

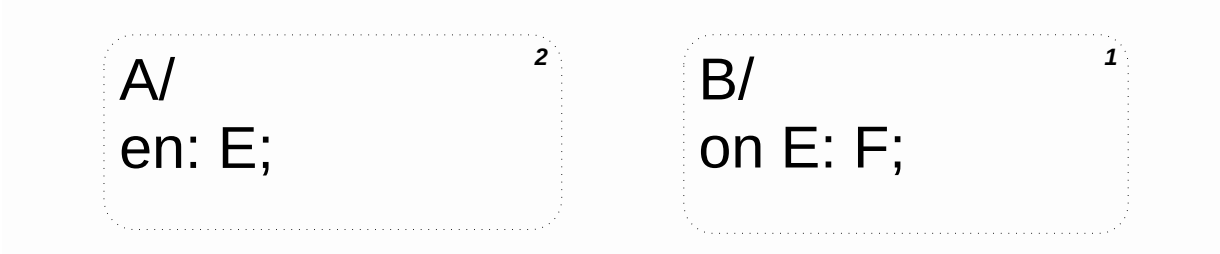

Figure 24: A model with non-confluent parallel states requiring event broadcasting

In Figure 24 the two states `A` and `B` are evaluated in the order `B` then `A` but `A` emits event `E` whereas `B` receives it. Figure 25 shows the relevant parts of the generated code. The event broadcast routines simply call the recursion point (`sf_2ca`). At the point of call, all events are cleared (`clr`) and the event being broadcast is set. The recursion point is the `sf_2ca` node and the top-level function (`sf_2`) is simply a wrapper for `sf_2ca` replacing the recursion variable (`const n`) with the event stack size. This is needed because LUSTRE will not accept a `const` value as an input to the top-level node.

Within this scheme it is possible to implement STATEFLOW's "early return logic" which is intended to reduce the possibility of inconsistent states arising from the misuse of event broadcasts. It results, however, in messy and inefficient code since virtually all activity after the potential processing of an event has to be guarded with a check of the parent or source state. This has been partially implemented in our translator, for example, in the above code, if state `A` was within another state, say `A1`, then the call to the entry action for state `A` would actually be something like:

```
if (sgA1 and enA) then enaA1(...);
```

[13] A `const` value in LUSTRE is not actually a constant. It refers to a value which can be statically evaluated at compilation time.

```
-- entry action for node id=3 name=A
node enaA(F,E: event; sB,sA,term,init: bool;
          const n: int)
returns(Fo: event; sBo,sAo: bool; Eo: event);
let
  Fo,sBo,sAo,Eo=
    with n=0 then (F,sB,sA,E)
    else sf_2ca(clr,set,sB,sA,term,init,n-1);
tel

-- graph id=7 name=Parallel5,call
node sf_2ca(F,E: event; sB,sA,term,init: bool;
            const n: int)
returns(Fo: event; sBo,sAo: bool; Eo: event);
...

-- graph id=7 name=Parallel5,top
node sf_2(dummy_input: bool) returns(F: event);
let
  ...
  F,sB,sA,E=sf_2ca(F_1,E_1,sB_1,sA_1,term,init,1);
tel
```

Figure 25: Code showing event stack

This static recursion technique allows us, in theory, to emulate the behaviour of STATEFLOW charts which exhibit bounded-stack behaviour. In practice, there is a heavy penalty to pay for static recursion since the recursion encompasses practically the entire program. This means that each event broadcast point results in expansion of the whole program at that point, down to the level of the event stack. Practical experience with the translator shows that an event stack size of 4 is about the greatest that can be accomodated in reasonable space and time.

Finally, we can easily accomodate STATEFLOW's `send` facility which allows sending of an event to a named state. One possibility is to view this as a function call of the target state [9], however, this would require generalization of the recursion mechanism to allow calls to intermediate nodes. A simpler solution is to simply treat events as integers and use the convention that 0 is an inactive event, 1 is a broadcast event and events with other integer values are targetted at the state with that identity number. For this purpose we abstract the `event` type and provide constants for event testing:

```
type event = int;
const set = 1; clr = 0;
```

The `on` action for state `B` (id number 4) would thus become guarded by:

```
if ((E = set) or (E = 4)) then ...
```

5.7 History junctions

History junctions are a STATEFLOW feature which allow states to "remember" their previous configuration in between activations. This is easily handled by our translator by keeping local variables within each node corresponding to a state with a history junction. The only complication is how to trigger storage and restoring of the history values. Luckily, the `init` and `term` flags correspond almost exactly to the semantics of history junctions, we only need to store them when `term` is `true` and restore them when `init` is `true`. Figure 26 shows the relevant code.

```
node sf_3(sAin,sBin,sgTOP,term,init: bool)
returns(sA,sB: bool);
var sAh,sBh,...: bool;
let
  sB_1,sA_1=(false,false) ->
    if init then (pre sBh,pre sAh) else (sBin,sAin);
  ...
  sBh,sAh=(false,false) ->
    if term then (sBin,sAin) else (pre sBh,pre sAh);
tel
```

Figure 26: Saving and restoring history values

There is also a slight complication with entry actions since the normal entry action flag computation does not take into account the fact that a state may become active by being "remembered" rather than entered via a transition. A simple fix for this is to augment the flag with a test for external activation of the state. For example, in Figure 26, if state `A` had an entry action we would add the following line *after* the state entry predicate:

```
enA = enA or ((not sgAin) and sgAt);
```

Where `sgAin` is the initial value of the state and `sgAt` is the final value of the state (which does take into account the possibility of activation by memory). The exit actions have no such problems.

5.8 Implicit event keywords

This is a feature of STATEFLOW which allows an event to be generated upon some specific conditions; a state entering, a state exiting, data changing or upon every time instant (`tick`) ie. every time the chart wakes up. The most obvious method of implementing these is to actually generate a physical event for these occurrences. The problem here is that their existence needs to be known in advance. Thus for each implicit event encountered during parsing of the model we build a table of these events. This can then be used to generate an event when the relevant code is encountered, for example when a state enters or exits or when data changes. We have defined a naming convention for such events, for example the state entry event is: `<state name>_enter_event`, and these events are generated in the state's entry action (one is created if the state has no entry

action). This has also been implemented for statements in condition, transition and state actions for changing data. The `tick` keyword is easily handled by defining the `tick_event` event at the toplevel and setting this event each time the chart is entered.

5.9 Translation fidelity

It is not possible to formally verify the equivalence of STATEFLOW's and our translator's behaviours, principally because of a lack of a formal definition for STATEFLOW. Our translator was developed, however, directly from the STATEFLOW documentation and its description of the intepretation algorithm which, as far as possible, we have encoded into LUSTRE. We have also manually verified the equivalence of the two systems on a substantial set of example STATEFLOW models based around the subset of STATEFLOW which we currently support. From our point of view, however, the primary reference for the behaviour of the translated code is the LUSTRE translation. In a real-world example we would perform tests and validation upon the LUSTRE code and not upon the STATEFLOW model directly.

This also applies to our link with SAFE STATE MACHINES (SSM) by *Esterel Technologies, Inc.*. SSM, like STATEFLOW, is also a graphical interface to a finite state machine system but, unlike STATEFLOW, is based on a sound formal semantics and there exists a formal translation path into languages such as LUSTRE. The question exists, however, as to how to translate legacy STATEFLOW code into SSM and the issues embodied in our translator also apply to translation from STATEFLOW into SSM. Our translator can, however, be used as a reference semantics for this translation since its output should, in theory, have the same semantics as the output from STATEFLOW → SSM → LUSTRE.

5.10 The translatable subset of Stateflow

Currently, we can translate hierarchical and parallel AND states assuming *no* inter-level transitions. We can implement event broadcasting provided the broadcasting recursion is bounded by a reasonably small value. State entry, exit, during and on-actions as well as condition and transition actions for transitions are all supported. Only part of the action language is translatable but we can implement array processing and so-called *temporal logic operators*. This gives basic functionality. In addition, however, we can implement sending of events to specific states, history junctions and inner transitions.

6 Enlarging the "safe" subset by model-checking

For future work we intend to provide a separate semantical analysis of STATEFLOW models which should either transform the model into a semantically equivalent model which conforms to a specified subset which we can implement or reject the model with reasons as to why it does not conform. For now, however, the existence of a translation from STATEFLOW into LUSTRE allows us to immediately apply the existing model-checking tools for LUSTRE to STATEFLOW models.

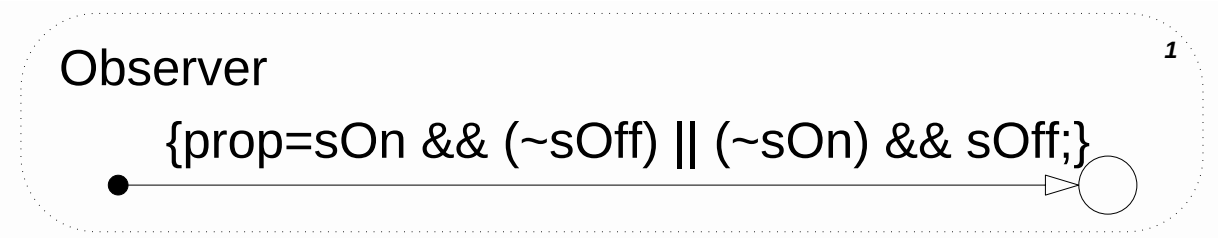

Figure 27: Simple observer in Stateflow

For example, Figure 27 shows a simple observe implemented in STATEFLOW for the model in Figure 6. Here the property is a trivial mutual exclusion of states and LESAR verifies this property without consuming any significant time or memory.

In this section we demonstrate two useful properties that can be model-checked in STATEFLOW models, i.e. confluence of parallel states and boundedness of event broadcasting. Our translator is able to generate auxiliary LUSTRE nodes which are observers for properties supplied to the translator. Currently, these are LUSTRE expressions but it should be possible to allow these expressions to be supplied by the STATEFLOW model in the form of graphical functions or some other form of annotation. This would obviate the necessity of the user learning LUSTRE's syntax and semantics. In this section we simply demonstrate two useful properties that can be model-checked in STATEFLOW models, i.e. confluence of parallel states and boundedness of event broadcasting.

Figure 28 shows a set of parallel states[14]. States `N1` and `N2` (executed in the order `N1` then `N2`) and states `N3` and `N4` (executed `N4` then `N3`) form two versions of the same simple machine except for the order of parallel execution. The figure also shows an observer which directly compares equivalent state variables between the two machines. Running LESAR on the generated LUSTRE code results in a `TRUE` value so we can deduce that the order of execution of parallel states in the machine `N1`/`N2` (or `N3`/`N4`) is irrelevant.

Figure 29 shows a STATEFLOW chart which requires either parallel state confluence or the use of an event stack. State `TOP1` generates a local event `E` upon receiving input event `G`. Event `E` is received by state `TOP2` which then emits output event `F`. To allow detection of event stack overflow the translator generates an additional local value "`error`" which is set if there is an attempt to broadcast an event when the event stack counter is zero. The broadcast statement for event `F` is show in Figure 30.

If `TOP2` is executed before `TOP1` we need event broadcasts to allow `E` to be received by `TOP2`. Furthermore, if output event `F` is to be broadcast we need a minimum event stack of 2 which is verified by LESAR. Model-checking using the `error` property gives a `FALSE` property for an event stack depth of 1 but a `TRUE` property if the event stack is set to 2. Finally, if we reverse the order of execution of states `TOP1` and `TOP2` we can get a `TRUE` property with an event stack size of zero.

[14]The state variable names are accessible in our translator so `sOn` refers to the variable for the `On` state. These *pseudo-variables* have to be included in STATEFLOW's data dictionary.

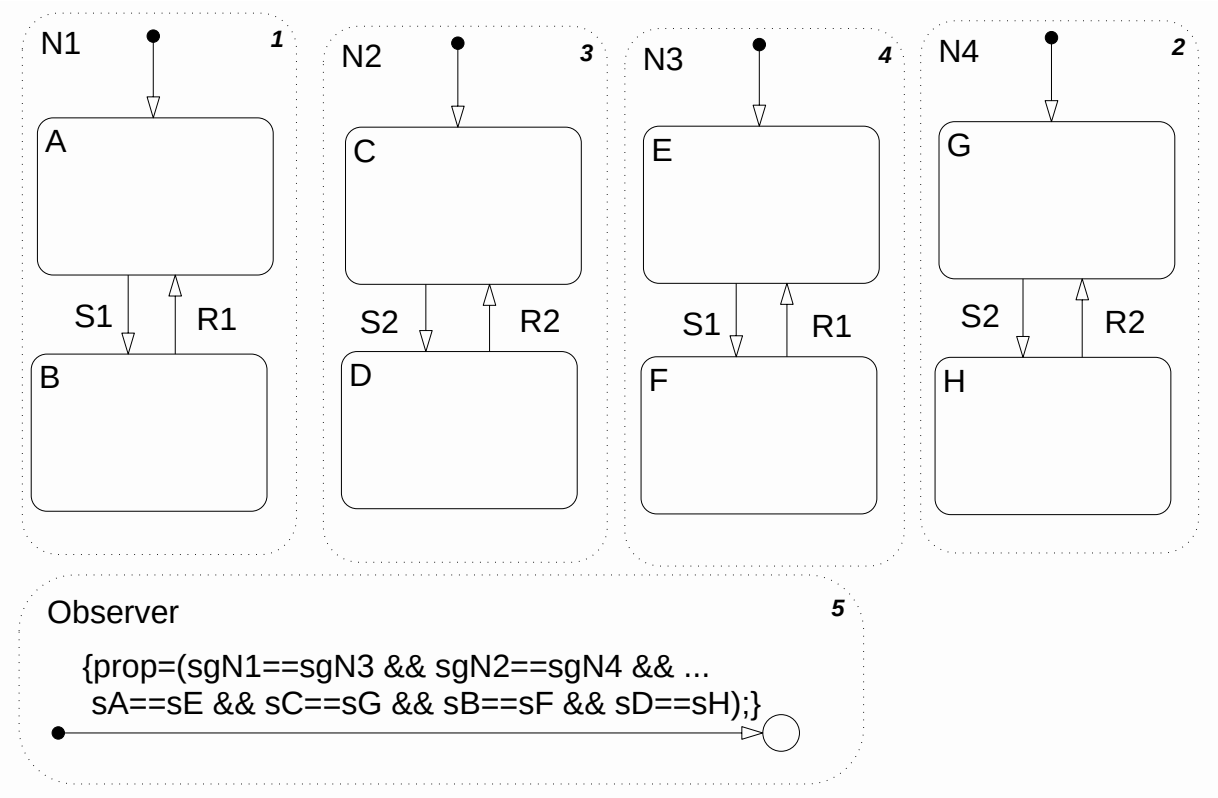

Figure 28: An observer for parallel state confluence

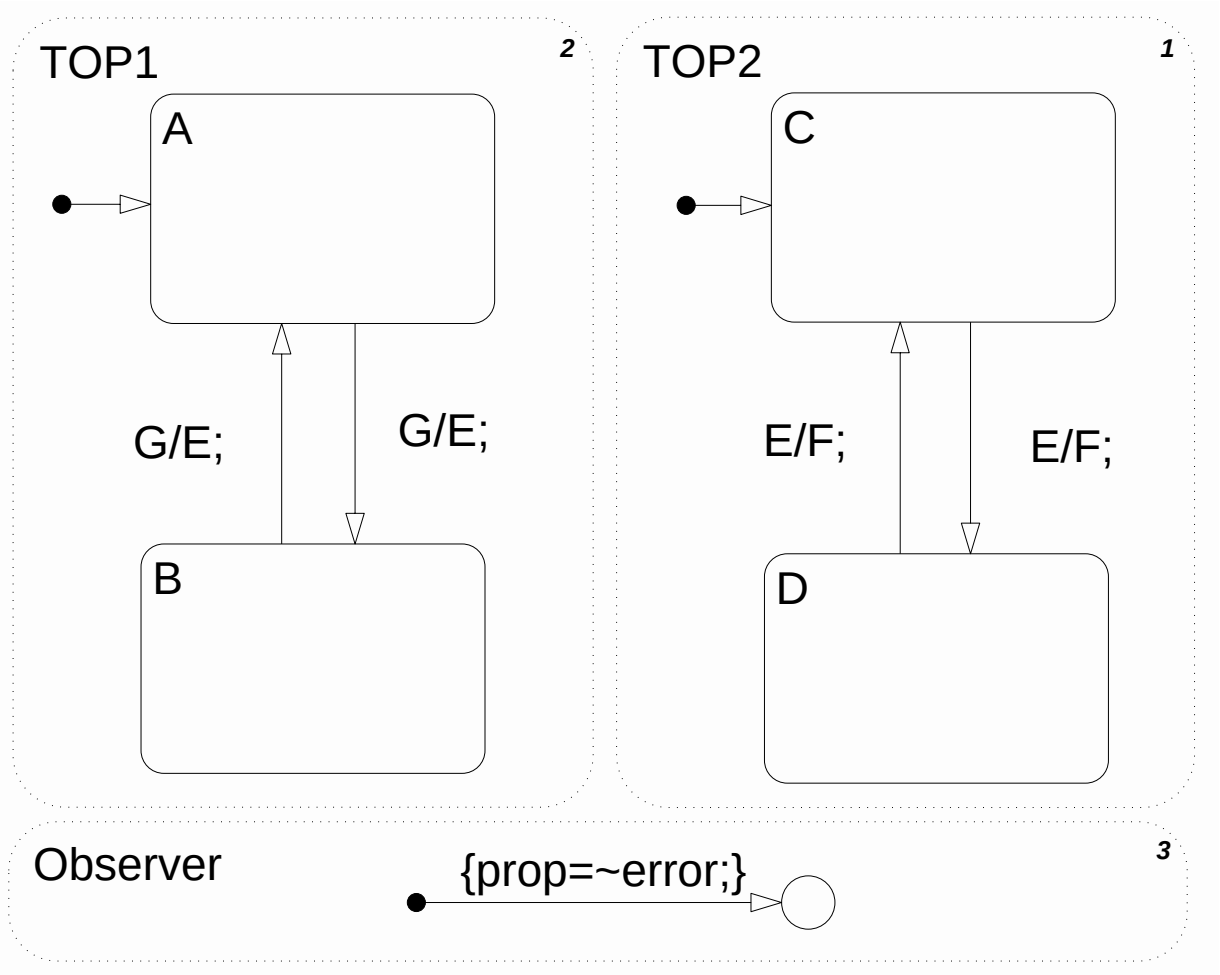

Figure 29: An observer for event stack overflow

```
propo,Fo,sAo,sBo,sCo,sDo,sgObservero,sgTOP1o,sgTOP2o,
erroro,Eo =
  with n = 0
  then (prop,F,sA,sB,sC,sD,sgObserver,
        sgTOP1,sgTOP2,true,E)
  else sf_2ca(clr,clr,set,prop,sA,sB,sC,sD,
              sgObserver,sgTOP1,sgTOP2,error,
              term,init,n-1);
```

Figure 30: Code for event broadcast with error detection

Although these examples are trivial the analysis itself can be extended to models of complexity. We envisage using the model-checking not just for verification of safety properties but also as a means of enhancing the subset of STATEFLOW which we are able to implement. A designer can use model-checking to spot where his design does not conform and where to fix the model to bring it into conformance.

7 Tool and case study

7.1 Prototype implementation

We have developed a prototype translator of SIMULINK/STATEFLOW to LUSTRE, called SS2LUS. The tool integrates and extends the existing SIMULINK-to-LUSTRE translator S2L [5] with a new module, called SF2LUS. All examples shown in the paper have been translated automatically with the tool.

S2L and SF2LUS interface in a "clean" manner: whenever S2L finds a STATEFLOW block, it submits it to SF2LUS which translates it into a LUSTRE node and returns this node (body plus type signature) back to S2L. Type and clock inference, which have been major issues in S2L, are much easier with STATEFLOW. Types of variables are explicitly declared in STATEFLOW, so they need not be inferred. In fact, type checking is required by the translator but this is solely for constant and operator resolution and it suffices to typecheck the generated LUSTRE code. The STATEFLOW block is triggered by a SIMULINK signal and uses a single clock, thus, no clock inference is needed either.

What we have, therefore, is a development tool for SIMULINK/STATEFLOW which allows, firstly, verification of subset inclusion for our various subsets of STATEFLOW, secondly, verification of application-specific model properties using model-checking and finally, an alternative means of code-generation for SIMULINK/STATEFLOW models via the various LUSTRE compilers and interpreters. To demonstrate this tool's applicability, we present a simple case study.

7.2 Case Study

Figure 31 shows a hypothetical alarm monitoring system for a car. This contains two parallel states, `Speedometer` which adjusts the `speed` variable according to input events and `Car` which is hierarchical, the outer layer `engine_on` monitoring the engine status and the next inner layer monitoring the car's speed. The innermost level has two parallel states, `belt` which monitors the seatbelt status and generates the `belt_alarm` alarm if the seatbelts are not on and the speed is greater than 10, and `locks` which monitors the door lock switch and controls the locks.

LESAR only has limited support for numerical values, and does not handle the `speed` variable very well. Since we now have a LUSTRE program, we could use the tool Nbac [11], which is based on abstract interpretation techniques, to handle the `speed` variable. However, the only rôle of this variable in the model is in boolean tests so we can abstract this variable and use an equivalent set of boolean flags. This chart is shown in Figure 32. Here, the `Speedometer` state

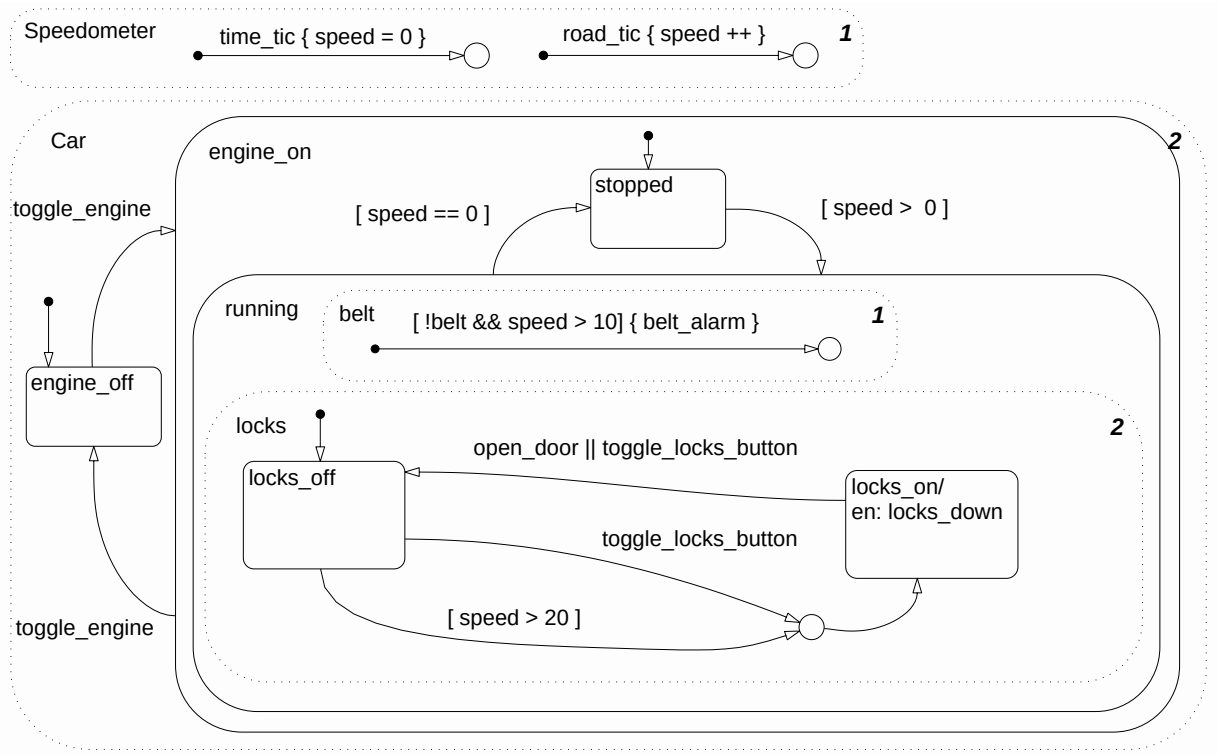

Figure 31: An alarm controller for a car

outputs flags according to whether the speed is zero, non-zero or greater than 10 or 20. The rest of the model has been suitably transformed. The observer for this model states that there should be no alarms when the engine is off and that the door locks should always be on when the speed is greater than 20. Furthermore, the belt alarm should be on if the speed is greater than 10 and the `belt` status is off.

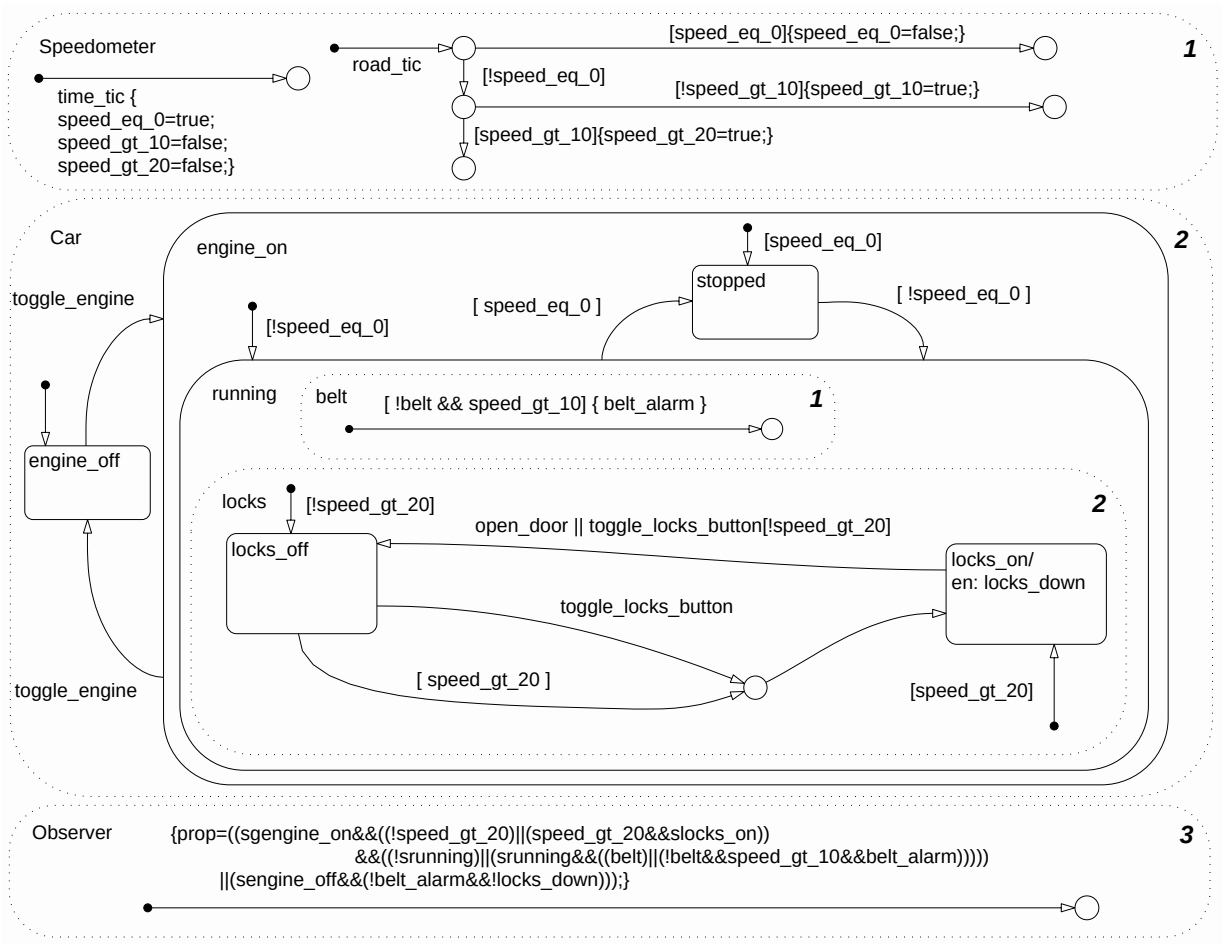

Figure 32: Abstracted and corrected version of the alarm controller

Running LESAR on the original model results in a FALSE property with the following counterexample:

```
--- TRANSITION 1 ---
road_tic
--- TRANSITION 2 ---
toggle_engine and not time_tic and road_tic
--- TRANSITION 3 ---
not toggle_engine and not time_tic and road_tic
```

What the model-checker has spotted is that if the engine is switched on while the car is moving (not an impossibility by any means) then it is possible to reach a state where the speed is greater than 20 and not be in the `locks_on` state. The solution is simple, split up the default transitions in the `engine_on` and `locks` states (for example, `[speed_eq_0]` and `[!speed_eq_0]`) so that the correct state is reached depending upon the initial conditions when these states are entered. These additional default transitions are shown in Figure 32. The new model gives a `TRUE` LESAR property with the observer shown.

This model is perhaps not a realistic application but even with such a simple model the properties verified by LESAR are not intuitively obvious. It is also not very well-written STATEFLOW since the use of conditions on default transitions is warned against in the STATEFLOW documentation. The point, however, is that given suitable observers and verification by model-checking, even badly written STATEFLOW can be used with confidence.

8 Conclusions and further work

The success of SIMULINK/STATEFLOW lies partly in the integration of heterogeneous modeling styles, namely, dataflow and automata based. In this paper, we have extended our previous work on translating discrete-time SIMULINK to LUSTRE by incorporating a large part of STATEFLOW. Our method and tool, although still incomplete (we cannot handle arbitrary for-loops, for instance), translates most of STATEFLOW, including features which may be considered "unsafe" (e.g., backtracking and dependence on graphical layout). This is important for reasons of legacy. Still, realizing the importance of identifying a "safe" subset of STATEFLOW and perhaps developing standard guidelines which restrict engineers to this subset, we have also provided a number of light-weight static checks which guarantee absence of most semantic problems of STATEFLOW. In the case where a model fails these checks, the generated LUSTRE program can be model-checked instead. Finally, the LUSTRE program can be used for C code generation, which is guaranteed to preserve the semantics.

We are currently working on extending the capabilities of the translator and experimenting with more case studies. We are also studying ways to make the static checks less strict. Finally, we are examining the limits of preserving the structure of the STATEFLOW model in the generated LUSTRE program. This is useful, among other reasons, for debugging purposes, in particular, when mapping the model-checker diagnostics back to the original STATEFLOW model.

References

[1] C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Proceedings of the Tenth IEEE International Symposium on Computer Aided Control System Design*, pages 581–586, Hawaii, Aug 1999. 1

[2] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Mu noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center. 1

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. 5

[4] 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. 1

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *ACM-SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003. 5, 7.1

[6] Ford. Structured Analysis Using Matlab/Simulink/Stateflow - Modeling Style Guidelines. Technical report, Ford Motor Company, 1999. 1, 1

[7] 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. 1

[8] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Algebraic Methodology and Software Technology*, pages 83–96, 1993. 1, 5

[9] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Proceedings of Fundamental Approaches to Software Engineering (FASE)*, Barcelona, Spain, March 2004. 1, 2, 5.2, 5.4, 5.6

[10] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 1, 5, 1, 2.1, 2.2.4

[11] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium*, pages 39–50, 1999. 5.2, 7.2

[12] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A Compositional Approach to Statecharts Semantics. In D.Rosenblum, editor, *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 120–129. ACM Press, 2000. 1, 2.2.4

[13] F. Maraninchi and N. Halbwachs. Compiling ARGOS into boolean equations. In *Proc. 4th Int. School and Symposium "Formal Techniques in Real Time and Fault Tolerant Systems" FTRTFT*, pages 72–89, Uppsala, Sweden, 1996. 5.3

[14] F. Maraninchi and Y. Rémond. Argos: an Automaton-Based Synchronous Language. *Computer Languages*, (27):61–92, 2001. 5

[15] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical Automata as a Model for Statecharts. In *Asian Computing Science Conference (ASIAN'97)*, number 1345 in Lecture Notes in Computer Science. Springer, December 1997. 1

[16] Reactive Systems Inc. http://www.reactive-systems.com. 1

[17] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. Formal Methods in Computer Aided Design (FMCAD 2000)*, LNCS. Springer, Nov 2000. 1

[18] The MathWorks. Stateflow and stateflow coder, user's guide, version 5. Available at http://www.mathworks.com/products/stateflow/. 2, 2.1

[19] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002. `http://www.csl.sri.com/~tiwari/stateflow.html`. 1