Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre

N. Scaife
Norman.Scaife@imag.fr

C. Sofronis
Christos.Sofronis@imag.fr

P. Caspi
Paul.Caspi@imag.fr

S. Tripakis
Stavros.Tripakis@imag.fr

F. Maraninchi
Florence.Maraninchi@imag.fr

Laboratoire VERIMAG, Centre Equation
2, avenue de Vignate, 38610 GIERES, France

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.

Categories and Subject Descriptors: D.2.2 [Design Tools and Techniques]: Computer-aided software engineering (CASE)

General Terms: Design, Languages, Reliability, Verification

Keywords: Embedded software, Simulink, Lustre, Automatic translation

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 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 dataflow 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[2] [17]. Moreover, the industrial version of Lustre, SCADE, commercialized by Esterel-Technologies[3] 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

[1]Trademarks of the *MathWorks* company
[2]http://www.prover.com
[3]http://www.esterel-technologies.com

Stateflow is much harder than it was for Simulink[4]. 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,
- 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 2.3). This may be useful for designing less restrictive guidelines. Secondly, we show how to translate Stateflow into Lustre (Section 3) and also show how properties that may not be checked by the algorithms of Section 2.3 can be checked on the Lustre translation by means of model-checking (Section 4). This allows us to further enhance our notion of a "safe" subset. Finally, we discuss a prototype implementation and a simple case study (Section 5).

Related work

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 [18] 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 inter-level transitions. They also have a translator into the SAL system which allows various model-checking techniques to be applied to Stateflow.

[4]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.

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.[5] have a tool called REACTIS for automated test generation for Simulink/Stateflow models.

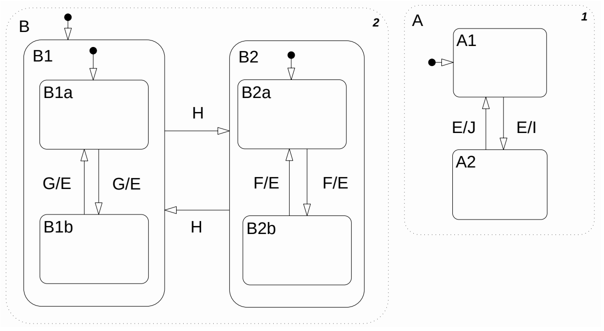
Our work differs from these in that we provide a set of simple static checks which are much "lighter" than model-checking. We are also able to use the generated Lustre program not only for model-checking but for C code generation while preserving the original semantics. Note that generation of C code from Lustre is now well-understood and we do not need to elaborate upon it here.

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 (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]. The following model has examples of both:



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

[5]`http://www.reactive-systems.com`

[6]The term "parallel" is misleading since such states are actually considered in sequential order.

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.

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 enabling 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.

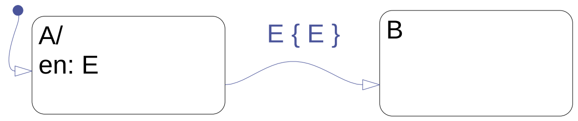
It should be emphasized that each of the executions *runs to completion* and this makes the behaviour of the overall algorithm very complex. In particular, *when any of the actions consists of broadcasting an event, the interpretation algorithm for that event is also run to completion before execution 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.

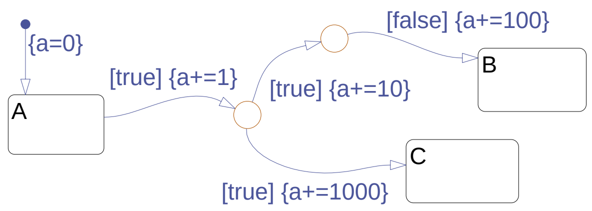
Another source of potential problems is the "run to completion" semantics of event broadcast. Every time an event is emitted the interpretation algorithm is called recursively, runs to completion, then execution resumes from the action statement immediately after the emission of the event. This can lead, semantically, to infinite recursion and in practice (i.e., during simulation) to stack overflow[9] as shown in the following model:



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 following model:



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".

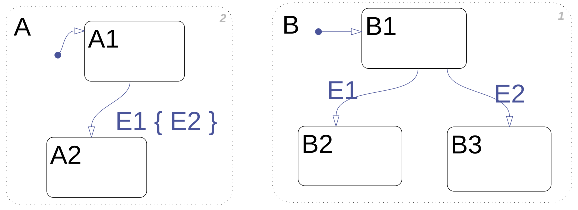
2.2.3 Dependence of semantics on graphical layout

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, in the following model, 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 in the top right-hand corner.)

[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: "Do not design your Stateflow diagram based on the expected execution order of transitions."

[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."*



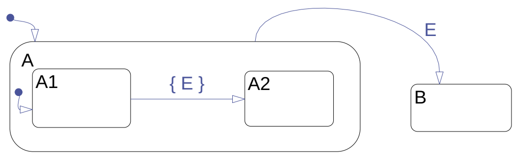
The order of exploration is important since it may lead to different results. In the case of "12 o'clock" rule, for example, if the top-most transition of the model in Section 2.2.2 emanated from the 11 o'clock position instead of the 1 o'clock position, then the final value of a would be 1001 instead 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 in this section. A and B are parallel states. When event $E1$ arrives, if A is explored first, then $E2$ will be emitted and the final global state will be $(A2, B3)$. But if B is explored first then the final global state will be $(A2, B2)$. 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 the model below. 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 $A1$ is not active anymore, since its parent is no longer active.



2.3 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 4.

2.3.1 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 and is hard to loosen, since termination is undecidable for programs with counters and loops.

2.3.2 Acyclicity of triggering and emitted events

An event E is said to be *triggering* a state s if the state has an "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 $v \to v'$ iff the following two conditions hold:

1. An event E is emitted in v which triggers 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.

2.3.3 No 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.

2.3.4 Outgoing junction conditions form a cover

In order to ensure absence of back-tracking 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 events are covered as well.

2.3.5 Outgoing junction conditions 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

[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.

expressing that a set of conditions are disjoint and form a cover. These logical properties can be submitted 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.

2.3.6 *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 variables shared between A and B are not assigned by either of these states. 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 $v \to v'$ such that v belongs to A and v' to B or vice-versa.

2.3.7 *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.

3. TRANSLATION INTO LUSTRE

The above checks on Stateflow models define a subset which is more likely to be correct according to the system designer's intentions than using the full Stateflow definition. However, it is restrictive since it disallows some of Stateflow's features which designers have become used to. We therefore extend our subset by employing analysis with sound theoretical underpinnings. One such framework is model-checking and we have access to Lesar [8], a well-established model-checker which takes Lustre as its input. A translation of Stateflow into Lustre thus opens up the 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.

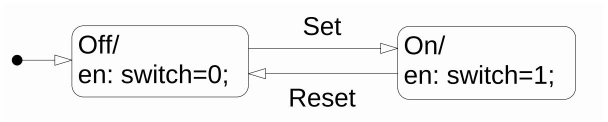
We have to be clear, however, about the difference between the subset of Stateflow which is "safe" in the sense of the previous discussion and that which is translatable into Lustre. We can copy the behaviour of Stateflow as precisely as required (given sufficient effort in building the translator) and 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 4 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 semantics-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 initialization 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*.

3.1 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 following simple chart:



in the Lustre code:

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

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.

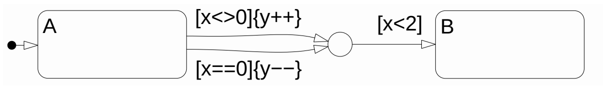
A more practical approach, therefore, is to split the above equations into their components and use explicit dependencies to force their order of evaluation. Inspecting the Lustre code above, the state update equation for each state consists of; an initialization value computed from default transitions (`true` 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:

```
node SetReset1(Set,Reset: bool)
returns (Off,On: bool; switch: int);
var Off1,On1: bool; switch1: int;
let
  Off1=true->if pre Off and Set then false else pre Off;
  On1=false->if pre On and Reset then false else pre On;
  Off=Off1->if pre On and Reset then true else Off1;
  On=On1->if pre Off and Set then true else On1;
  switch1=0->if Off and not pre Off then 0 else pre switch;
  switch=switch1->if On and not pre On then 1 else switch1;
tel.
```

Here, the state update code has been split into separate entry and exit sections with the order set by the use of the auxiliary variables `sOff1` and `sOn1`. The entry actions can then be inserted in the correct place. For this simple example they could be placed anywhere in the Lustre node but we would want transition actions to be evaluated between exiting the source state (exit actions) and entering the destination state (entry actions) for compatibility with Stateflow.

3.2 Compiling transition networks



The above is a Stateflow chart which illustrates 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. The actual code generated is as follows. Note that 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":

```
-- transition id=7 name=[x<>0]{y++}
node lv7_(x,y: int; lv8,lv7,exit: bool)
returns(lv8o,lv7o: bool; yo: int; exito: bool);
var cv7,cv8,end,end_1,exit_1: bool;
let
  cv7,end_1,exit_1=
    if not exit then cv7_(x) else (false,false,exit);
  yo=if cv7 then ca7(y) else y;
  cv8,end,exito=
    if cv7 and not end_1
    then cv8_(x) else (false,end_1,exit_1);
  lv7o,lv8o=if cv8 and end then (true,true) else (lv7,lv8);
tel

-- transition id=6 name=[x==0]{y--}
-- Note: Code identical to lv7_ with "6" in place of "7"

-- node id=3 name=A
node suAex(x,y: int; sA,term,init,exit: bool)
returns(sAo,lv6,lv8,lv7: bool; yo: int; exito: bool);
var exit_1,lv8_1: bool; y_1: int;
let
  lv6,lv8_1,y_1,exit_1=lv6_(x,y,false,false,exit);
  lv8,lv7,yo,exito=lv7_(x,y_1,lv8_1,false,exit_1);
  sAo=if exito and (lv7 or lv6) or term then false else sA;
tel.
```

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. 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.

A more serious problem is that junction networks can have loops which results in unbounded recursion and therefore a loss of synchronous semantics. To allow a synchronous semantics for Stateflow we have to outlaw such constructs in the general case. It is possible, however, to unroll such loops 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.

3.3 Hierarchy and parallel AND states

We initially make the assumption that inter-level transitions 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.

For example, The model in Section 2.1 illustrates a simple model with both parallel and exclusive sub-states. For both types of sub-state we insert the function calls to the sub-states 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 as follows:

```
-- State B (OR,[B1,B2])
node sf_7(F,G,H,E: event; sgB,sgB1in,sB1bin,sB1ain,sgB2in,
          sB2bin,sB2ain,term,init: bool)
returns(sgB1,sB1b,sB1a,sgB2,sB2b,sB2a: bool; Eo: event);
let
  sgB1t=sguB1en(sgB1_1,lv17,lv19,term,init,exit);
  sgB2t=sguB2en(sgB2_1,lv18,term,init,exit);
  sB1b,sB1a,E_1=
    sf_8(G,E,sgB1t,sB1bin,sB1ain,
         not sgB1t and sgB1in,sgB1t and not sgB1in);
  sB2b,sB2a,Eo=
    sf_9(F,E_1,sgB2t,sB2bin,sB2ain,
         not sgB2t and sgB2in,sgB2t and not sgB2in);
   sgB1=sgB1t; sgB2=sgB2t;
tel

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

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 role 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 the following table (`s` is the local state variable) and are embodied in auxiliary nodes (for example the state variable is computed by the node `sfs` in the above code):

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 init) and not s`
during	`false -> s and pre s`

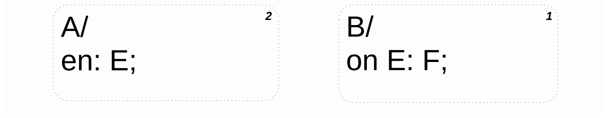
For exclusive sub-states the `init` and `term` flags are computed solely from the local state variable (`init = s and not pre s` and `term = not pre s and s`). The complication is that we need the value of the state variable at the end of the reaction without actually setting the variable itself because 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 3.4) the value of the state variable can be updated by actions. Note also that for the top-level call we set `init` to `true->false` and `term` to `false`.

3.4 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.

This is implemented by creating a `const`[11] 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.

The following is a trivial model with non-confluent parallel states which requires event broadcasting:



The two states `A` and `B` are evaluated in the order `B` then `A` but `A` emits event `E` whereas `B` receives it. The following code is generated, showing the event stack:

[11]A `const` value in Lustre 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);
let ...

-- 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.
```

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(...);
```

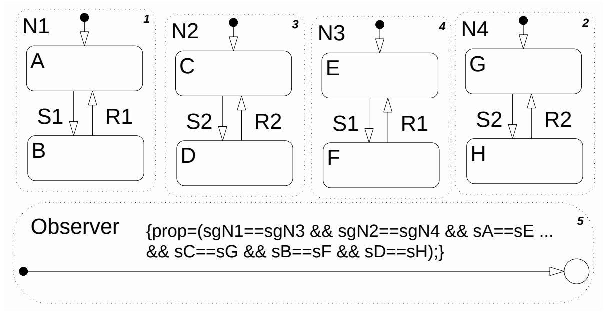
This static recursion technique allows us, in theory, to emulate the behaviour of Stateflow charts which exhibit bounded-stack behaviour. In practice, there is a heavy penalty to pay for static recursion since the recursion encompasses practically the entire program. This means that each event broadcast 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 accommodated in reasonable space and time.

3.5 The translatable subset of Stateflow

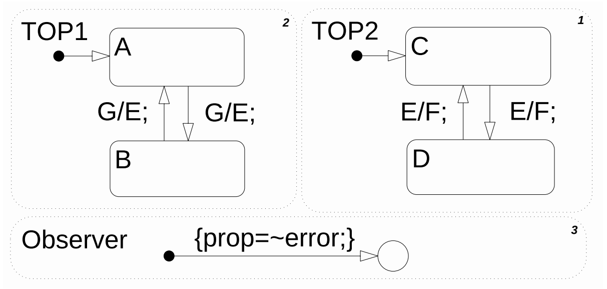
Currently, we can translate hierarchical and parallel AND states assuming *no* inter-level transitions. We can implement event broadcasting provided the broadcasting recursion is bounded by a reasonably small value. State entry, exit, during and on-actions as well as condition and transition actions 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, inter-level transitions and inner transitions, the details of these are outwith the scope of this paper. See the accompanying technical report [16].

4. ENLARGING THE “SAFE” SUBSET BY MODEL-CHECKING

The existence of a translation from Stateflow into Lustre allows us to immediately apply the existing model-checking tools for Lustre to Stateflow models. In this section we demonstrate two useful properties that can be model-checked in Stateflow models; confluence of parallel states (partially addressing the graphical layout problem in Section 2.2.3 and extending the static check for confluence in Section 2.3.6) and boundedness of event broadcasting (partially addressing the non-termination problems of Section 2.2.1 and extending the check for acyclic event emission in Section 2.3.2). Our translator is able to generate auxiliary Lustre nodes which are observers for properties supplied to the translator. Currently, these have to be generated manually but in future could be generated automatically for the specific properties addressed in Section 2.3.


N1
1
A
S1
R1
B
N2
3
C
S2
R2
D
N3
4
E
S1
R1
F
N4
2
G
S2
R2
H
Observer
5
{prop=(sgN1==sgN3 && sgN2==sgN4 && sA==sE ...
&& sC==sG && sB==sF && sD==sH);}

The above chart shows a set of parallel states. 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. 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.



The above is 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 shown in the following code:

```
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);
```

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.

Although these examples are trivial the analysis itself can be extended to models of greater 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.

5. TOOL AND CASE STUDY

5.1 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.

5.2 Case Study

Figure 1 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 seat belt status and generates the `belt_alarm` alarm if the seat belts are not on and the speed is greater than 10, and `locks` which monitors the door lock switch and controls the locks.

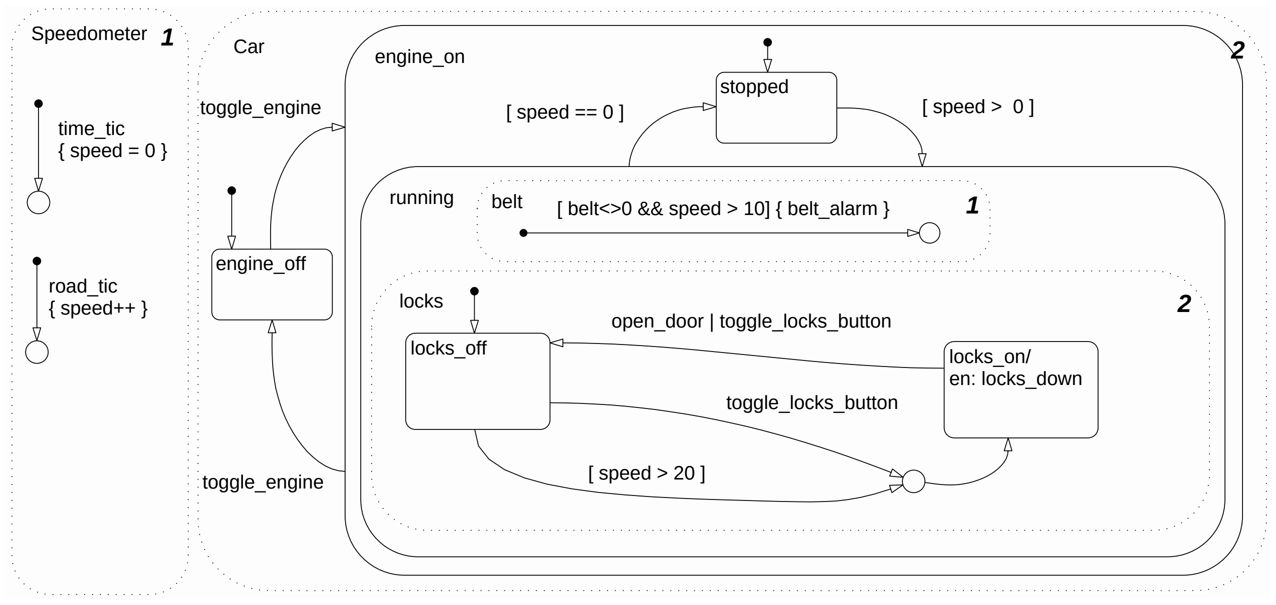


Figure 1: An alarm controller for a car

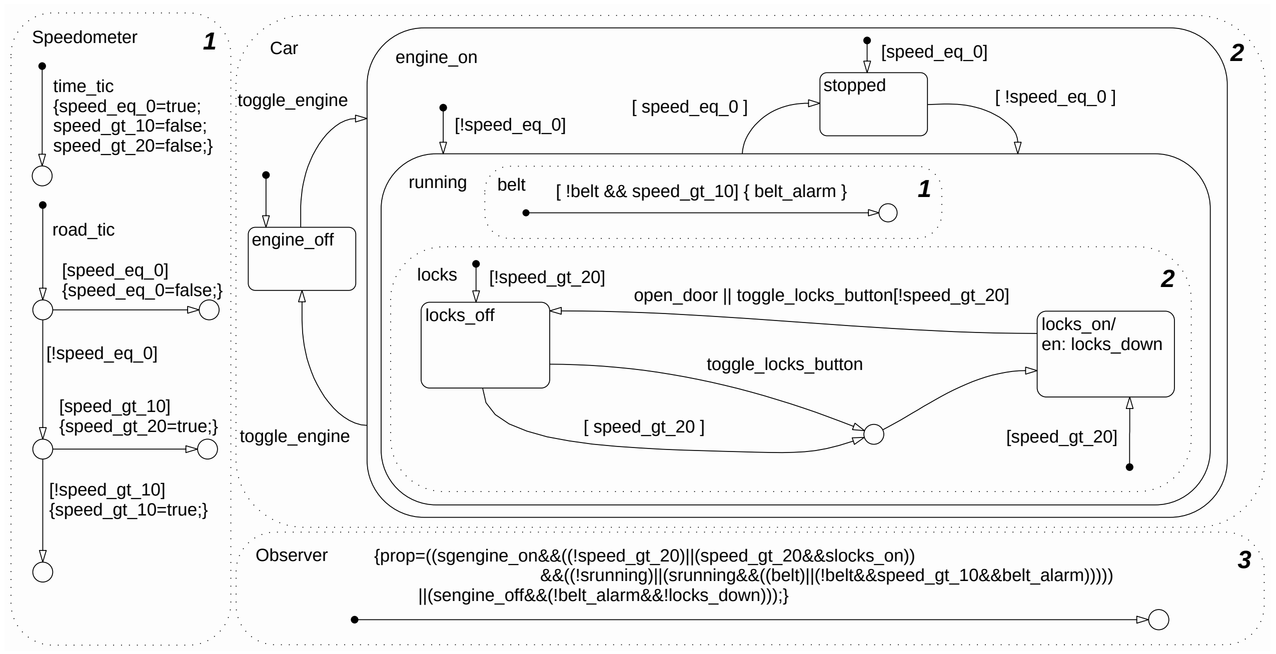


Figure 2: Abstracted and corrected version of the alarm controller

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 role 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 2. Here, the `Speedometer` state 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.

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
```

The model-checker has spotted 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 2. 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.

6. CONCLUSIONS

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 lightweight 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.

7. REFERENCES

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

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

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

[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*, LNCS. Springer Verlag, 2003.

[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.

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

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

[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.

[9] G. Hamon and J. Rushby. An operational semantics for stateflow. In *Proc. of Fundamental Approaches to Software Engineering*, Barcelona, Spain, March 2004.

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

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

[12] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A Compositional Approach to Statecharts Semantics. In D.Rosenblum, editor, *Proc. of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 120–129. ACM Press, 2000.

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

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

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

[16] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. Technical Report TR-2004-16, http://www-verimag.imag.fr.

[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.

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