EPiC Series in Computing

Volume 46, 2017, Pages 144–161

LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning


# Automated analysis of Stateflow models*

Hamza Bourbouh[1,3,5], Pierre-Loic Garoche[2], Christophe Garion[1]
Arie Gurfinkel[4], Temesghen Kahsai[5,6], and Xavier Thirioux[3]

[1] ISAE-SUPAERO/DISC, France
[2] ONERA/DTIM, France
[3] ENSEEIHT/IRIT, France
[4] University of Waterloo, Canada
[5] NASA Ames, USA
[6] CMU, USA

**Abstract**

Stateflow is a widely used modeling framework for embedded and cyberphysical systems where control software interacts with physical processes. In this work, we present a framework and a fully automated safety verification technique for Stateflow models. Our approach is two-folded: (i) we faithfully compile Stateflow models into hierarchical state machines, and (ii) we use automated logic-based verification engine to decide the validity of safety properties. The starting point of our approach is a denotational semantics of Stateflow. We propose a compilation process using continuation-passing style (CPS) denotational semantics. Our compilation technique preserves the structural and modal behavior of the system. The overall approach is implemented as an open source toolbox that can be integrated into the existing Mathworks Simulink/Stateflow modeling framework. We present preliminary experimental evaluations that illustrate the effectiveness of our approach in code generation and safety verification of industrial scale Stateflow models.

## 1 Introduction

The widespread deployment of cyberphysical systems in safety critical scenarios like automotive, avionics, and medical devices, has made formal and automated analysis of such systems necessary. This is witnessed by the sheer number of extensive approaches proposed in the verification community. Stateflow [22] is a widely used modeling framework for embedded and cyberphysical systems where control software interacts with physical processes. Specifically, Stateflow is a toolbox developed by MathWorks Inc. that extends Simulink [20] with an environment for modeling and simulating reactive systems. A Stateflow diagram can be included in a Simulink model as one of the blocks interacting with other Simulink components using input and output signals. Stateflow is a highly complex language with no formal semantics[1]: its semantics is only

*This work was partially supported by the ANR-INSE-2012 CAFEIN project, CNES project No. R-S16/BS-0004-045 and NASA Contract No. NNX14AI09G.

[1] At least not provided as a reference by the tool provider.

described through examples on the MathWorks website [22] without any formal definition. A Stateflow diagram has a hierarchical structure, which can be either arranged in *parallel* in which all states become active whenever the diagram is activated; or *sequentially*, in which states are connected with transitions and only one of them can be active.

Over the years several approaches have been proposed for the analysis of Stateflow diagrams. Such approaches often lack one or many of the following desired features: (i) a convincing formal semantics; (ii) a faithful compilation that preserves the hierarchical structure of the Stateflow model; (iii) fully automated analysis engine; and last but not least (iv) an open source tool that is easy to use and able to handle realistic models. The aim of this paper is to provide a framework in which all those points are properly addressed. We based our work upon a series of papers by Hamon [15, 14, 16] providing operational and denotational semantics for Stateflow, designing interpreters for Stateflow.

*A CPS semantics for Stateflow.* The starting point of our approach is the expression of the denotational semantics of Stateflow [14] as a pure *continuation-passing style* (CPS) denotational semantics. CPS has been proposed in the 70s by Plotkins [24] for $\lambda$-calculus call-by-value semantics and later developed for efficient compilation means, for example in the long line of Danvy's works, e.g., [18] and Appel's book [3]. As recalled by Danvy, CPS terms can be simply expressed yet enjoy a number of useful properties, e.g., "offering a good format for compilation and optimization". The following equations define the Plotkin's call-by-value CPS rules:

$$\begin{array}{rcl} [\![x]\!]\ \kappa & = & \kappa\ x \\ [\![\lambda x.e]\!]\ \kappa & = & \kappa\ (\lambda x \cdot \lambda k \cdot [\![e]\!]\ k) \\ [\![e_0 e_1]\!]\ \kappa & = & [\![e_0]\!]\ (\lambda v_0.[\![e_1]\!]\ (\lambda v_1 \cdot v_0\ v_1\ \kappa)) \end{array}$$

The key idea is to associate to each function an additional argument, the explicit continuation $\kappa : t \to t$. This continuation is an endomorphic map over $t$ values on which control is explicitly modeled: function calls, intermediate values, evaluation order, etc. In compilation, CPS $\beta$-reduction amounts to characterize a global continuation, which, when evaluated, produces the generated code.

**Contributions.** This paper makes the following contributions:

- We adapted Hamon's denotational semantics [14] to a pure CPS semantics, solving some of its flaws (see §2.2).
- We instantiate such CPS semantics for different uses, such as a *model interpreter* and a *code generator* for Stateflow models. Such framework has several advantages, including a formal semantics of Stateflow that preserves the hierarchical structure of the model.
- We implemented the general CPS semantics and the proposed instantiations in OCaml: an interpreter and a code generator both for imperative code and Lustre automaton.
- The Lustre automaton code generator from Stateflow has been implemented and integrated in CoCoSim [17] – an automated analysis framework for Simulink models. CoCoSim among other features provides an intuitive user interface that facilitates the modeling of safety properties, code generation, verification, and graphical debugging of failed properties.
- We used CoCoSim fitted with this new capability to address Stateflow model compilation and verification on a set of industrial scale benchmarks and have experimentally evaluated our approach using two evaluation scales: (i) does the tool generate faithful code (wrt.

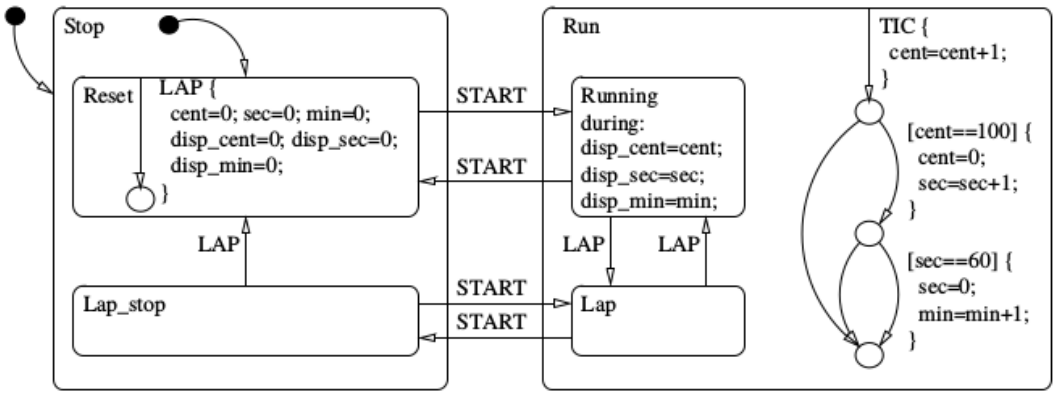



Figure 1: Stopwatch Stateflow model (from [14]).

the intended Stateflow semantics)? and (ii) is the tool able to efficiently verify safety properties? In Section 5 we provide evidence that answers positively both questions.

**Synopsis.** The paper is structured as follows. In the next two paragraphs, we first informally illustrate the Stateflow semantics and then give an overview of related works. In Section 2, we describe a CPS denotational semantics for Stateflow. In Section 3, we illustrate instantiations of the CPS semantics that allow us to generate both an interpreter and a code generator. Based on the code generator from Section 3, we describe in section 4 a compiler from Stateflow to Lustre automaton. Finally, in Section 5, we describe an open source implementation of our proposed technique and present the experimental evaluations that illustrate the effectiveness of our approach.

**Stateflow semantics.** Let us look at a concrete example that illustrates different Stateflow constructs. Figure 1 shows the `Stopwatch` model capturing a basic behavior of a stopwatch. This system interacts with its environment via 3 signals: *START*, *LAP* and *TIC*. *TIC* models the time increment of the system. *START* and *LAP* events model the user interaction with the device. There are 2 top level states *Run* and *Stop*, and 4 inner states *Reset*, *Lap_stop*, *Running* and *Lap*. The device is initialized in *Stop* mode and switches back and forth between *Stop* and *Run*. When in *Running* state, it can be paused using the *LAP* signal. Each *TIC* signal received when in *Run* state increments the timer, performing side effects on the internal timer variables (i.e., *cent, sec, min*). This specific behavior is modeled by a set of transitions using only junctions and starting with an inner transition. This model is interesting since it relies on multiple constructs of Stateflow: *inner* and *outer* transitions, *junctions* and hierarchical states.

**Related work.** Despite its lack of formal semantics, Stateflow is widely used in the industry both for modeling purposes and code generation [23]. It has been studied for formal modeling and verification by numerous approaches, e.g., targeting automata [4, 19, 33], hybrid automata [2], process calculi such as CPS [6, 34], transition systems [23], tabular expressions [28] or the synchronous language Lustre [27]. As a general remark, each of these approaches makes some restriction on the considered language, e.g., on events, inner transitions or junctions, and synthesizes an encoding in the target language, supported by verification tools or test case generation, independently of the code generated from the same model.

An approach related to ours is the work described in [27]. This approach translates a Stateflow model into Lustre dataflow language [5]. Since Lustre-like languages such as Scade [29] are used in the industry to generate embedded code, the approach is compatible both with compilation and verification. The approach is however very limited in respecting the full Stateflow semantics. A tool, `sf2lus`, is provided and performs the translation for the considered subset. This approach differs from ours in several ways: (i) Our translation keeps state machines structure and by consequence makes it easy to read and traces state information; (ii) sf2lus handles events erroneously at the same time, which makes the behavior unsound wrt. Stateflow events semantics. Events occur (and wake the chart) in an ascending order based on their port numbers once at a time. Such a chart can then be executed multiple times in the same time step; (iii) `sf2lus` does not support newer versions of Stateflow; and last (iv) our tool is developed in Matlab and is directly integrated into the Simulink/Stateflow environment, which allows among other features the use of Matlab simulation tool to further investigate failed properties.

Another line of related works are developed in a series of papers by Hamon [14, 15] on which we based our work. In these papers, a formal semantics for Stateflow is provided, either in operational or denotational flavors. Notice that [32] starts from Hamon's denotational semantics to define a structural operational semantics for three Statecharts languages, including Stateflow, but does not provide a compilation schema or analysis techniques. [30] uses a translation of Stateflow models to pushdown systems to generate and check invariants, but does not go further in properties to analyze. Until now, these are the only formal reference semantics available for Stateflow.

Last, Simulink Design Verifier (SLDV) is a toolbox provided by Mathworks to perform automated formal analysis of Simulink/Stateflow models. However, SLDV is a commercial distributed tool, therefore details on implementation and functionality are not available to the public.

## 2 Denotational Semantics for Stateflow

In this section, we describe our first contribution: a continuation-passing style denotational semantics for Stateflow. We start from the syntax and denotational semantics[2] described in [14]. However, we will revisit such semantics using a continuation-passing style (CPS). This new formulation allows to directly and easily capture the intended semantics of Stateflow. We use a syntactic representation of Stateflow models described by the grammar presented in Table 1.

**Syntax.** A program $(s, src_{i\in[1\ldots n]})$ is composed of state definitions $s : sd$ and junctions $j : T$, with a main node $s$. A single state is defined by the *entry*, *during* and *exit* actions $(a_e, a_d, a_x)$, *outer* and *inner* transitions $T_o$ and $T_i$, as well as component content $C$. A component content $C$ is either an $Or(T, sl)$ state with initializing transitions $T$ and substates $sl$, or an $And(sl)$ state where all $sl$ substates are run in parallel. A transition list is an ordered sequence of transitions. Each

$$
\begin{array}{rcl}
P & ::= & (s, [src_0, \ldots, src_n]) \\
src_i & ::= & s : sd \mid j : T \\
sd & ::= & ((a_e, a_d, a_x), T_o, T_i, C) \\
C & ::= & Or\ (T, [s_0, \ldots, s_n]) \\
 & & \mid And\ ([s_0, \ldots, s_n]) \\
t & ::= & (e, c, (a_c, a_t), d) \\
T & ::= & \emptyset \mid t.T \\
d & ::= & p \mid j \\
p & ::= & \emptyset \mid s.p
\end{array}
$$

Table 1: Syntactic representation of Stateflow models.

[2] This syntax and semantics does not handle additional constructs such as loops, history junctions or call to external C functions. The restriction of covered constructs is further detailed in Sect. 5

$$
\begin{array}{rl}
main.run.running: & ((\emptyset_a, \texttt{disp} = (\texttt{cent}, \texttt{sec}, \texttt{min}), \emptyset_a), \\
& [(START, true, \emptyset_a, \emptyset_a, P\ main.stop.reset); \\
& (LAP, true, \emptyset_a, \emptyset_a, P\ main.run.lap)], [], Or\ ([], )) \\
main.run.lap: & ((\emptyset_a, \emptyset_a, \emptyset_a), \\
& [(START, true, \emptyset_a, \emptyset_a, P\ main.stop.lap\_stop); \\
& (LAP, true, \emptyset_a, \emptyset_a, P\ main.run.running)], [], Or\ ([], )) \\
main.run: & ((\emptyset_a, \emptyset_a, \emptyset_a), [], \\
& [(TIC, true, \texttt{cent} = \texttt{cent} + \texttt{1}, \emptyset_a, J\ j1)], Or\ ([], \{running; lap\})) \\
j1: & [(noevent, \texttt{cent} == \texttt{100}, \texttt{cont} = \texttt{0}; \texttt{sec} = \texttt{sec} + \texttt{1}, \emptyset_a, J\ j2); \\
& (noevent, \texttt{cent!} = \texttt{100}, \emptyset_a, \emptyset_a, J\ j3)] \\
j2: & [(noevent, \texttt{sec} == \texttt{60}, \texttt{min} = \texttt{min} + \texttt{1}, \emptyset_a, P\ main.run); \\
& (noevent, \texttt{sec!} = \texttt{60}, \emptyset_a, \emptyset_a, J\ j3)] \\
j3: & []
\end{array}
$$

Figure 2: Encoding of the Stopwatch Stateflow model using the syntax from Table 1.

transition is associated with an event, a condition, side effect condition actions $a_c$, transition actions $a_t$ and a destination $d$. Destinations could be either states or junctions leading to further transitions.

**Modeling Stopwatch example.** Figure 2 describes the Stopwatch model using the syntax from Table 1. Transitions from a state $s$ to another $s'$ are defined as outer $T_o$ and inner transitions $T_i$ of node $s$, depending on the target node. The transitions associated to a node content of type $Or$ describe the initialization transitions when entering into the node. Junctions enable the definition of transitions combining multiple conditions. The semantics of transitions is far from trivial: a transition path is evaluated one segment (atomic transition) after the other, performing side effects on the state via *condition actions* $a_c$ on the go, while *transition actions* $a_t$ are only performed when all conditions over the path are satisfied and the path reaches a state not a junction. If a given path is eventually fireable, the exit actions of the original node are performed, then the transition actions, to conclude with entry actions of the target node.

## 2.1 CPS Denotational Semantics for Stateflow

**Motivating Continuation-passing style.** The denotational semantics presented in [14] relies on continuations to model the actions of path computation. Indeed, the actions associated to an atomic transition (a segment) are performed either immediately for condition actions and eventually for transition actions. Success and fail continuations allow to capture this complex behavior in functions, representing side effects as denotations. However, values manipulated by this denotational semantics were explicitly first order and represented by environments $\rho$ of type $Env$: $\rho ::= \{x_0 : v_0, \ldots, x_n : v_n, s_0 : b_0, \ldots, s_k : b_k\}$. These environments represented both the values of variables $x_i$ and the active status of states $s_i$.

The encoding of [14] could be significantly improved by rearranging arguments so as to push environments in rightmost position and defining a pure continuation-passing style (CPS) denotational semantics, point-free wrt. environments. Indeed, in some situations, the author would drop continuations and evaluate explicit intermediate environments.

The following definitions characterize our CPS denotational semantics for Stateflow, following precisely the semantics of [14] while solving its flaws. The semantics is higher order and environments are never made explicit: the evaluation of a model component acts as a transformer.

**Conditions.** Transitions in Stateflow are computed based on the current environment and an active event, evaluating conditions. Without loss of generality, we assume that the event is part of the environment and is not made explicit in the rules. An active event `e` could be checked as a regular condition using the predicate `event(e)`. We recall that the environment contains both variables mapped to values and active status of Stateflow states. To clarify the expression, we made explicit the check whether a state characterized by $p$ is active using the predicate `active(p)`.

**Actions.** The key ingredient of transition computation in Stateflow is the sequence of actions applied on the current environment, updating values of variables and changing active and inactive states. Actions act as transformers and are the values manipulated by our CPS denotational semantics. We denote by $Den$ this transformer type.

Basic action constructors are left free here but typically express some imperative assignment of an expression to an environment variable. In addition, we introduce the actions `open p` and `close p` which switch the Boolean status of state $p$ to `true` or `false` respectively. In order to generalize the approach, we express disjunctions as actions using the constructor $\mathcal{I}te(condition, Den, Den)$.

A primitive action (assignment or open/close action) semantics, i.e., its interpretation as a transformer, is defined using the function $\mathcal{A}[\![\cdot]\!] : action \rightarrow Den$. (Actions) transformers can be combined using the operator $\gg$: $Den \rightarrow Den \rightarrow Den$, which is associative: $a_1 \gg a_2 \gg a_3$ means that action $a_1$ is performed before action $a_2$ followed by $a_3$. Last, the default action, identity, is denoted $\mathcal{I}d$.

**Denotational semantics as a functions map.** Semantics functions are associated to state names and a global continuation environment $\theta$ of type $KEnv$ is defined as follows: $\theta ::= \{\ p_i : (\mathcal{S}[\![p_i : sd_i]\!]^e_m\ \theta, \mathcal{S}[\![p_i : sd_i]\!]^d\ \theta, \mathcal{S}[\![p_i : sd_i]\!]^x_m\ \theta), j_j : \mathcal{T}[\![T_j]\!]\theta, \ldots, j_j : \mathcal{T}[\![T_j]\!]\theta\}$. Functions $\mathcal{S}[\![p_0 : sd_0]\!]^e_m$, $\mathcal{S}[\![p_0 : sd_0]\!]^d$ and $\mathcal{S}[\![p_0 : sd_0]\!]^x_m$ denote respectively the semantics of a state when entering it, executing it, or exiting it. Note that entry and exit actions are parametrized by a mode $m \in Mode = L \mid S$. This mode, either loose ($L$) or strict ($S$), captures the difference between inner and outer transitions for entry and exit actions. Junctions are associated to transition list semantics function $\mathcal{T}$. The $\theta$ map captures the semantics of all components of the Stateflow model and is typically provided as argument of denotations.

**Transitions semantics.** Stateflow semantics is rather complex. A Stateflow transition amounts to evaluate a sequence of atomic transitions. Depending on some dynamic conditions, each atomic transition may be eventually fired or not. In all cases, it will impact the environment through side effects (condition actions). We introduce three continuations: `success` of type $k^+ ::= Den$ modeling a fired transition, a `fail` continuations of type $k^- ::= Den$ modeling an unfired one and a third case $\texttt{fail}^{glob}$ of type $k^-$ capturing complex executions in which a series of junctions ends in a terminal junction. In case of a transition leading to another state, some entry or exit actions may be performed[3]. They are captured by the `wrapper` continuation of type $w ::= p \rightarrow Den \rightarrow Den$.

The evaluation of a destination path which is a state amounts to apply the wrapper on the success continuation. Otherwise, when the destination is a junction, the transition list semantics is evaluated with the same continuations.

$\mathcal{D}[\![p]\!]\ (\theta : KEnv)\ (wrap : w)\ (success : k^+)\ (fail\ \ fail^{glob} : k^-) : Den = wrap\ \ p\ \ success$

[3]Note that those actions are not performed for a transition ending in a terminal junction.

$\mathcal{D}[\![j]\!]\ \theta\ wrap\ success\ fail\ fail^{glob} = \theta^{\mathbf{j}}(j)\ wrap\ success\ fail\ fail^{glob}$

Atomic transition semantics introduces an $\mathcal{I}te$ action: in case of an unfeasible condition, the (regular) fail continuation is used, otherwise a new success continuation is built, evaluating the transition actions of the atomic transition. The action associated to the then-branch combines the condition actions followed by the new destination evaluation and relies on the newly defined success continuation.

$\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ (\theta : KEnv)\ (wrapper : w)\ (success : k^+)\ (fail\ fail^{glob} : k^-) : Den =$
$\quad \mathcal{I}te(event(e_t) \wedge c,$
$\quad\quad (\texttt{let}\ success' = success \gg (\mathcal{A}[\![a_t]\!])\ \texttt{in}$
$\quad\quad\ (\mathcal{A}[\![a_c]\!]) \gg (\mathcal{D}[\![d]\!]\ \theta\ wrapper\ success'\ fail\ fail^{glob})),$
$\quad fail)$

Evaluation of a list of transitions performs a left-to-right traversal of the list: the unfeasibility of the head transition leads to the evaluation of the next, involving definition of fail continuations. As a special case, when provided with an empty list of transitions it always evaluates to the global fail continuation.

$\mathcal{T}[\![\emptyset]\!]\ (\theta : KEnv)\ (wrapper : w)\ (success : k^+)\ (fail\ fail^{glob} : k^-) : Den = fail^{glob}$
$\mathcal{T}[\![t.T]\!]\ \theta\ wrapper\ success\ fail\ fail^{glob} =$
$\quad \texttt{let}\ fail' = \mathcal{T}[\![T]\!]\ \theta\ wrapper\ success\ fail\ fail^{glob}\ \texttt{in}$
$\quad \tau[\![t]\!]\ \theta\ wrapper\ success\ fail'\ fail^{glob}$

**State semantics.** State semantics involves the opening and closing actions of states. We introduce wrapper functions dedicated to inner, outer transitions, as well as entering of states.

Wrapper considers a source and destination paths, $p_s$ and $p_d$, and identifies the common prefix $p$ of both paths. Depending on the context (inner or outer transition), it will compose respectively the exit actions of remaining $p_s$, the transition actions continuation, and the entering actions of the remaining $p_d$. Outer transitions will involve loose state semantics while inner transitions involve strict one. A specific wrapper $\texttt{open\_path}^{\mathbf{e}}$ is introduced to enter substates of a path.

$\texttt{open\_path}^{\mathbf{v}}\ (\theta : KEnv)\ (p : Path)\ (p_s : Path)\ (p_d : Path) : w =$
$\texttt{if hd}(p_s) = \texttt{hd}(p_d) \wedge \texttt{hd}(p_s) \neq \emptyset\ \texttt{then}$
$\quad \texttt{open\_path}^{v}\ \theta\ p.\texttt{hd}(p_s)\ \texttt{tl}(p_s)\ \texttt{tl}(p_d)$
$\texttt{else match v with}$
$\quad \mathbf{o}\ \texttt{->}\ \lambda den.\theta^{\mathbf{x}}_L(p.\texttt{hd}(p_s)) \gg den \gg \theta^{\mathbf{e}}_L(p.\texttt{hd}(p_d))\ \texttt{tl}(p_d)$
$\quad \mathbf{i}\ \texttt{->}\ \lambda den.\theta^{\mathbf{x}}_S(p.\texttt{hd}(p_s)) \gg den \gg \theta^{\mathbf{e}}_S(p.\texttt{hd}(p_d))\ \texttt{tl}(p_d)$
$\quad \mathbf{e}\ \texttt{->}\ \lambda den.den \gg \theta^{\mathbf{e}}_L(p.\texttt{hd}(p_d))\ \texttt{tl}(p_d)$

This definition assumes that $hd$ and $tl$ functions, returning head and tail of a list, are extended to handle empty lists, i.e., $\texttt{hd}\ \emptyset = \emptyset$ and $\texttt{tl}\ \emptyset = \emptyset$.

We now define the core semantics functions of states, $\mathcal{S}[\![\cdot]\!]^{\{\mathbf{e},\mathbf{d},\mathbf{x}\}}_m$. Note that mode parameter $m$ is provided as an index. Entering or exiting a path executes the entry and exit actions of all states in the path, respectively. Depending on the outer or inner status of a transition, the entry or exit actions of the root node shall or shall not be evaluated. The following definitions capture Stateflow semantics, handling specificities such as transitions from a node to a child or parent.

$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_S\ (\theta : KEnv)\ (\emptyset : Path) : Den = (\mathcal{C}[\![C]\!]^{\mathbf{e}}\ p\ \theta)$
$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_S\ \theta\ s.p_d = (\theta^{\mathbf{e}}_L(p.s)\ p_d)$
$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{x}}_S\ (\theta : KEnv) : Den = (\mathcal{C}[\![C]\!]^{\mathbf{x}}\ p\ \theta)$

$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_L\ \theta\ \emptyset\ = (\mathcal{A}[\![a_e]\!]\ \theta) \gg (\mathcal{A}[\![\texttt{open}\ p]\!]) \gg (\mathcal{C}[\![C]\!]^{\mathbf{e}}\ p\ \theta)$
$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{e}}_L\ \theta\ s.p_d = (\mathcal{A}[\![a_e]\!]\ \theta) \gg (\mathcal{A}[\![\texttt{open}\ p]\!]) \gg (\theta^{\mathbf{e}}_L(p.s)\ p_d)$
$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_0, T_i, C)]\!]^{\mathbf{x}}_L\ \theta = (\mathcal{C}[\![C]\!]^{\mathbf{x}}\ p\ \theta) \gg (\mathcal{A}[\![a_x]\!]\ \theta) \gg (\mathcal{A}[\![\texttt{close}\ p]\!])$

The definition for during actions is the following. First outer transitions are evaluated. If none succeeds, the during action of the node is computed. Then, inner transitions apply. If no transition can be fired at all, the components of the node are computed.

$$
\begin{array}{l}
\mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^{\mathbf{d}}\ (\theta : KEnv) : Den = \\
\quad \texttt{let}\ wrapper_i = \texttt{open\_path}^{\mathbf{i}}\ \emptyset\ p\ \texttt{in} \\
\quad \texttt{let}\ wrapper_o = \texttt{open\_path}^{\mathbf{o}}\ \emptyset\ p\ \texttt{in} \\
\quad \texttt{let}\ fail_o = \\
\qquad \texttt{let}\ fail_i = \mathcal{C}[\![C]\!]^{\mathbf{d}}\ p\ \theta\ \texttt{in} \\
\qquad\quad (\mathcal{A}[\![a_d]\!]\ \theta) \gg (\mathcal{T}[\![T_i]\!]\ \theta\ wrapper_i\ \mathcal{I}d\ fail_i\ fail_i)\ \texttt{in} \\
\quad \mathcal{T}[\![T_o]\!]\ \theta\ wrapper_o\ \mathcal{I}d\ fail_o\ fail_o
\end{array}
$$

Component semantics definitions follow. Entry transitions associated to an *Or* node initiate the component and shall not fail (the $\bot$ value is unreachable). Execution or exiting of an *Or* component applies on the active element. Parallel states rely on `fold_right` to ensure proper function compositions.

$$
\begin{array}{l}
\mathcal{C}[\![Or(T, \emptyset)]\!]^{\mathbf{e}}\ p\ \theta = \mathcal{I}d \\
\mathcal{C}[\![Or(T, S)]\!]^{\mathbf{e}}\ p\ \theta = \mathcal{T}[\![T]\!]\ \theta\ (\texttt{open\_path}^{\mathbf{e}}\ \emptyset\ p)\ \mathcal{I}d\ \bot\ \bot \\
\mathcal{C}[\![Or(T, \emptyset)]\!]^{\mathbf{d}}\ p\ \theta = \mathcal{I}d \\
\mathcal{C}[\![Or(T, x.S)]\!]^{\mathbf{d}}\ p\ \theta = \mathcal{I}te(\texttt{active}(p.x), \theta^{\mathbf{d}}(p.x), \mathcal{C}[\![Or(T, S)]\!]^{\mathbf{d}}\ p\ \theta) \\
\mathcal{C}[\![Or(T, \emptyset)]\!]^{\mathbf{x}}\ p\ \theta = \mathcal{I}d \\
\mathcal{C}[\![Or(T, x.S)]\!]^{\mathbf{x}}\ p\ \theta = \mathcal{I}te(\texttt{active}(p.x), \theta^{\mathbf{x}}_L(p.x), \mathcal{C}[\![Or(T, S)]\!]^{\mathbf{x}}\ p\ \theta) \\
\mathcal{C}[\![And(S)]\!]^{\mathbf{e}}\ p\ \theta = \texttt{fold\_right}\ (\lambda x.\lambda res.\theta^{\mathbf{e}}_L(p.x)\ \emptyset \gg res)\ S\ \mathcal{I}d \\
\mathcal{C}[\![And(S)]\!]^{\mathbf{d}}\ p\ \theta = \texttt{fold\_right}\ (\lambda x.\lambda res.\theta^{\mathbf{d}}(p.x) \gg res)\ S\ \mathcal{I}d \\
\mathcal{C}[\![And(S)]\!]^{\mathbf{x}}\ p\ \theta = \texttt{fold\_right}\ (\lambda x.\lambda res.\theta^{\mathbf{x}}_L(p.x) \gg res)\ S\ \mathcal{I}d
\end{array}
$$

**Program semantics.** The evaluation of the main program produces a transformer:

$$\mathcal{P}[\![(s, Srcs]\!] : Den = \mathcal{I}te(\texttt{active}(s), \theta^{\mathbf{d}}(s), \theta^{\mathbf{e}}_L(s)\ \emptyset)$$

where $\theta$ is built using *Srcs* and initial environment assumes all states are inactive, including the main one, $s$.

## 2.2 Comparison with Hamon's denotational semantics

The previous definitions are directly extracted from [14] but were modified to solve minor soundness flaws and to be compatible with the pure CPS semantics we designed. Without developing much about the soundness flaws[4], let us highlight the main differences in the semantics definitions:

- we adapted the rule to match our understanding of non trivial Stateflow constructs, as exhibited by current Stateflow simulation engine. For example sequences of actions performed when leaving a state and entering another one follow a specific order: in [14] the use of success and fail continuations was improperly combined. Our encoding introduced a new argument `wrapper` used in $\mathcal{D}$, $\tau$ and $\mathcal{T}$. Open and close actions bind dedicated wrappers that reorder actions. Our version follows current Stateflow behavior.
- regarding CPS, as explained at the beginning of the section, the $\rho$ argument is abstracted away and gives rise to a point-free semantics. Every dynamic access to environment as

[4] In fairness to this work, it is somehow difficult at the present time to figure out what the undocumented semantics of Stateflow circa 2005 may look like and to what extent it was well specified. The appendix sections compares the original semantics of [14] and the one we provide, both compared to the current behavior of Stateflow simulator. Notice also that [32] corrects some flaws in Hamon's denotational semantics.

$$
\begin{array}{rl}
\mathcal{A}[\![\texttt{open } p]\!](\rho) = & \rho\ [p \mapsto \texttt{true}] \\
\mathcal{A}[\![\texttt{close } p]\!](\rho) = & \rho\ [p \mapsto \texttt{false}] \\
\mathcal{A}[\![v = expr]\!](\rho) = & \rho\ [v \mapsto [\![expr]\!]_\rho] \\
\mathcal{I}te(cond, T, E)(\rho) = & \texttt{if } [\![cond]\!]_\rho \texttt{ then } T(\rho) \\
& \texttt{else } E(\rho) \\
(D_1 \gg D_2)(\rho) = & D_2 \circ D_1(\rho) \\
\mathcal{I}d(\rho) = & \rho \\
\bot = & \texttt{assert false}
\end{array}
$$

(a) Interpreter

$$
\begin{array}{rl}
\mathcal{A}[\![\texttt{open } p]\!] = & p = \texttt{true} \\
\mathcal{A}[\![\texttt{close } p]\!] = & p = \texttt{false} \\
\mathcal{A}[\![v = expr]\!] = & v = expr \\
\mathcal{I}te(cond, T, E) = & \texttt{if } cond \texttt{ then } T \\
& \texttt{else } E \\
(D_1 \gg D_2) = & D_1 \ ; \ D_2 \\
\mathcal{I}d = & \texttt{nop} \\
\bot = & \texttt{assert false}
\end{array}
$$

(b) Code Generator

Figure 3: Instantiations

found in Hamon's work is removed. For instance, dynamic if-then-else statements are lifted to a dedicated constructor to postpone their execution; similarly action evaluation is always introduced within computed continuations and never evaluated directly (see e.g., $\tau[\![\cdot]\!]$ and $S[\![\cdot]\!]^{\mathbf{d}}$). This brings far more flexibility in the purpose and design of semantics functions and allows for instance to define interpreters, code generators, source to source transformations, etc.

# 3 Modular code generation for Stateflow

The formal semantics presented in the previous section can be instantiated with appropriate definitions for the primitive elements of the denotational semantics: $\mathcal{A}[\![\cdot]\!]$, $\mathcal{I}te(\cdot,\cdot,\cdot)$, $\gg$, $\bot$ and $\mathcal{I}d$.

We present here different settings for the instantiation either as an interpreter or as code generator. The next section will address our main goal: generate Lustre automata from Stateflow model while preserving the hierarchical structure model.

**Interpreter instantiation.** The denotational semantics of [14] can be obtained where transformers modify environments: $Den = Env \to Env$. Figure 3a details the associated definitions. An environment $\rho$ defines a map from variables to values, including active status of states. $\rho[v \to c]$ represents the substitution of a variable $v$ to value $c$ in environment $\rho$. We assume that $[\![expr]\!]_\rho$ represents the evaluation of expression $expr$ in $\rho$, with a Boolean interpretation when evaluating a condition expression. The bottom construct throws an exception, but should not happen for well constructed models. We recall that events are part of the environment and are accessible through predicate $active(e)$ using in conditional expressions. Such an instantiation provides a simulator for the model: when provided with an initial environment, it computes the successor environment.

**Code generator instantiation.** One can also synthesize imperative code by synthesizing an abstract syntax tree while evaluating transformers. $Den$ denotes here an abstract syntax tree (AST):

$$
\begin{array}{rcl}
Den & ::= & Den;Den \\
& | & \texttt{if } cond \texttt{ then } Den \texttt{ else } Den \\
& | & v = expr \ \mid \texttt{nop} \mid \texttt{assert false}.
\end{array}
$$

Figure 3b provides such simple instantiation. Applying the code generator on the Stopwatch example from Fig. 1 generates a rather large program: about 800 LOC, for an overall number of 220 actions and 135 conditions, nested up to depth 13.

**Preserving hierarchical structure.** Stateflow semantics is global since environment is shared among all states. However, the transitions are attached locally to states. We can preserve this hierarchical structure by associating a procedure to each state execution denotation: each call to the denotation $\theta^{\mathbf{e}}(p)$, $\theta^{\mathbf{d}}(p)$ or $\theta^{\mathbf{x}}(p)$ could be respectively substituted by a call to a procedure `thetae_p`, `thetad_p` or `thetax_p` instead of executing $\mathcal{S}[\![p : sd]\!]^{\mathbf{e},\mathbf{d},\mathbf{x}}\ \theta$. This is possible since all arguments of these semantics functions are static (paths, modes, etc).

For a program $(s, srcs)$, the total code generation is then performed state by state, generating procedures `thetad_p` for each state $p$ declared in program sources $srcs$. The main procedure being the one associated to state $s$. For the Stopwatch example in Fig. 1, it generates 7 procedures, for an overall number of about 270 LOC, 100 actions and 55 conditions, nested up to depth 7. We have implemented in OCaml an interpreter and a code generator instantiation of the CPS denotational semantics. The code can be found in [1].

In our approach, modularity is itself modular as we can choose either to turn every semantics function into a procedure or on the contrary to inline its results. In this respect, turning junction-related semantics function $\theta^{\mathbf{j}}(j)$, which amounts to computing $\mathcal{T}[\![j : T]\!]\ \theta$ into a procedure helps in factorizing out common prefixes of transition sequences, provided one can defunctionalize [9] its arguments $wrapper$, $success$ and $fail$, expressing as first-order values.

For Stateflow models with complex transition sequences between junctions, this would greatly help factorizing out common junctions occurring in many paths, avoiding combinatorial blow-ups. This is left for future work.

# 4 Stateflow models as Lustre automata

In this section, we describe the compilation of Stateflow models into Lustre automata. Lustre [5] is a dataflow language, bearing similarities with Simulink/Stateflow, but is endowed with a synchronous semantics, which yields a more disciplined and predictable language, suited to verification activities. Lustre programs are expressed in terms of *nodes*, which directly model subsystems in a modular fashion, with an externally visible set of inputs and outputs. A *node* can be seen as a mapping of a finite set of input streams to a finite set of output streams. Operationally, a node has a cyclic behavior: at each clock tick $t$, it takes as input the value of each input stream at instant $t$ and *synchronously* returns the value of each output stream at same instant $t$. A Lustre node is built from a set of dataflow equations of the form $x = e$ where $x$ is a variable denoting an output or a locally defined stream and $e$ is an expression, in a certain stream algebra, whose variables are input, output, or local streams. Each variable is assigned exactly once at each cycle. Most of Lustre operators are point-wise lifting to streams of the usual operators over stream values. Still, a Lustre node may access to variable values at previous cycles, up to to a bounded past.

Automata are supported since Lustre V6 [8, 7]. The overall behavior of an automaton is pictured in Fig. 4. An automaton consists of states, each with its own set of equations and possibly local variables. At each instant, two pairs of variables are computed: a putative $state\_in$ and an actual state $state\_act$ and also, for both states, two booleans $restart\_in$ and $restart\_act$, that tell whether their respective state equations should be reset before execution. The actual state is obtained via an immediate (`unless`) transition from the putative state, whereas the next putative state is obtained via a normal (`until`) transition from the actual state. Only the actual state equations are executed at each instant. Finally, a state reset function is driven by the `restart`/`resume` keyword switches. A more complete presentation of these constructs is given in [11].

We are able to generate a Lustre AST mimicking the execution of the previous imperative

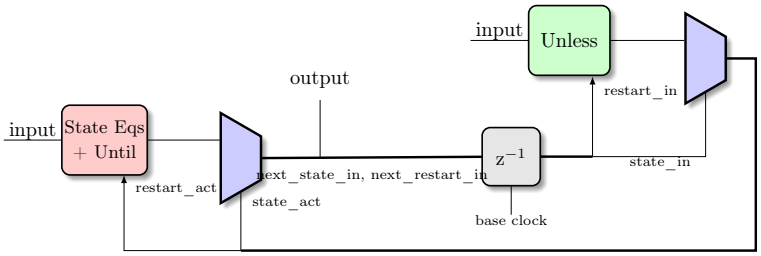



Figure 4: Automaton as a pure dataflow.

$$
\begin{array}{rl}
\mathcal{A}[\![\texttt{open}\ p]\!]\ in\ out := & \widetilde{out} = \widetilde{in}[in\_p \mapsto \texttt{true}] \\
\mathcal{A}[\![\texttt{close}\ p]\!]\ in\ out := & \widetilde{out} = \widetilde{in}[in\_p \mapsto \texttt{false}] \\
\mathcal{A}[\![v = expr]\!]\ in\ out = & \widetilde{out} = \widetilde{in}[in\_v \mapsto [\![expr]\!]_{in}] \\
\mathcal{A}[\![\texttt{call}\ p]\!]\ in\ out := & \widetilde{out} = \texttt{thetad\_p}(\widetilde{in}) \\
(L_1 \gg L_2)\ in\ out := & (L_1\ in\ name_{uid})\ ; \\
 & (L_2\ name_{uid}\ out) \\
\mathcal{I}d\ in\ out := & \widetilde{out} = \widetilde{in} \\
\bot\ in\ out := & \texttt{assert false}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}te(cond, T, E)\ in\ out := \\
\ \texttt{automaton}\ name_{uid} \\
\ \ \texttt{state Cond}: \\
\ \ \ \texttt{unless}\ [\![\neg cond]\!]_{in}\ \texttt{restart}\ NotCond \\
\ \ \ \texttt{let}\ (T\ in\ out);\ \texttt{tel} \\
\ \ \texttt{state NotCond}: \\
\ \ \ \texttt{unless}\ [\![cond]\!]_{in}\ \texttt{restart}\ Cond \\
\ \ \ \texttt{let}\ (E\ in\ out);\ \texttt{tel}
\end{array}
$$

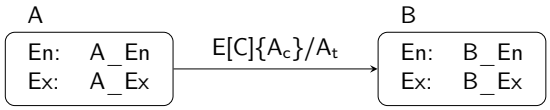
$$
\begin{array}{l}
\texttt{node thetad\_p}\ (\widetilde{in}\ :\ \widetilde{T_{in}})\ \texttt{returns}\ (\widetilde{out}\ :\ \widetilde{T_{out}}) \\
\ \texttt{let}\ (\mathcal{S}^{\mathbf{d}}[\![p]\!]\ in\ out);\ \texttt{tel}
\end{array}
$$

Figure 5: Lustre instantiation

code. The translation is also modular: a Lustre node is built for each component and each condition within a component is turned into an automaton. The use of intermediate local variables in the Lustre source allows to make the control-flow explicit. The denotation is defined as $Den = (Name \to Name \to LustreAST)$, taking two names *in* and *out* standing for input and output variables and producing a piece of code, assigning output from input. Lustre automata encode the very semantics of imperative conditional statements, whereas standard Lustre conditional is a strict operator which doesn't suit our needs. This instantiation is presented in Figure 5, assuming a supplementary node call action `call` $p$. We denote by $\widetilde{in}$ and $\widetilde{out}$ the sequence of variables present in the Stateflow environment, prefixed by names *in* and *out* respectively. Similarly, $\widetilde{T_{in}}$ and $\widetilde{T_{out}}$ denote their respective types.

Note that the action transformer function generates a set of Lustre flow definitions. Its evaluation combines all atomic transitions of Stateflow of an end-to-end transition into a single Lustre automaton state. No intermediate step is introduced. The language of actions is, for the moment, limited to basic Lustre flow definitions, but is left free in the general semantics of [16].

**Main compilation schema.** We have implemented this compilation as a component of the CoCoSim tool framework. In order to illustrate how the compiler works, let us consider a simple example. Figure 6 presents a simple transition between two states A and B. Each state is compiled into a corresponding Lustre automaton state (suffixed with _IDL). As Lustre only allows computations in states, the transition between A and B is compiled into a Lustre state that executes the transition actions. A_EXIT_B_ENTRY thus represents the transition in




```
automaton ab
  state CENTER_POINT:
    unless id=1 and E and C restart
         A_EXIT_B_ENTRY;
    unless id=1 restart A_IDL;
    unless id=2 restart B_IDL;
    let
      outputs = old_outputs;
    tel

  state A_IDL:
    let
      outputs = A_during_action(old_outputs,
          inputs);
    tel
    until true restart CENTER_POINT;
```

```
state A_EXIT_B_ENTRY:
  let
    -- execute the actions: condition
         action,
    -- A exit action, transition action, B
         entry action
  tel
  until true restart CENTER_POINT;

state B_IDL:
  let
    outputs = B_during_action(old_outputs,
        inputs);
  tel
  until true restart CENTER_POINT;
```

Figure 6: A simple transition encoding.

our example and contains all the actions to be executed: condition actions, exiting state A, transition actions and entering state B. id indicates the active state at the beginning of each clock. "Set B to active" means "update id to 2". The Lustre code associated to the previous transition A→ B is presented in the same Figure. In order to make the presentation simpler, nested Ite constructs as synthesized by our CPS semantics have been merged into a single automaton with more states.

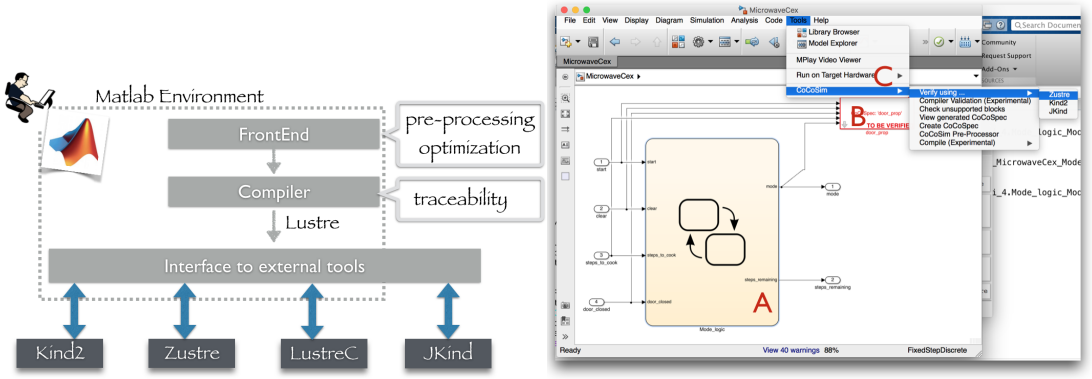
# 5 Experimental evaluation

In this section, we describe the implementation of our proposed approach as a component in the CoCoSim [17] tool framework. Moreover, we report the results from our experimental evaluation.

## 5.1 CoCoSim

CoCoSim [17] is an open source automated logic-based analysis framework for Simulink models. CoCoSim consists of two main components: (i) a compiler that compiles a subset of Simulink models to Lustre [5]; (ii) an interface to multiple model checkers based on Lustre. In this paper, we have extended CoCoSim to support compilation and verification of Stateflow models. The current version of CoCoSim provides capability to verify user supplied safety requirements. The CoCoSim workflow is pretty straightforward: a user specifies a safety requirement (property) in the Simulink/Stateflow model. This is done using a synchronous observer method. Subsequently, CoCoSim can be called directly on the Matlab environment[5] and the result of the analysis is reported directly on the Simulink model. If the property is violated, CoCoSim reports the set of input values that leads to the error state. Such information is then used to simulate the model, which allows the user to debug why a property is failing.

[5]The implementation in Matlab provides easy access to priorities for transition, typing information and eases the construction of traceability maps enabling the expression at model level of invariants or counter-examples.




(a) (b)

Figure 7: (a) The overall architecture of CoCoSim. (b) A screenshot of a verification scenario of a Stateflow model using CoCoSim: part (A) shows the Stateflow model; part (B) is the synchronous observer block, where the property to be verified is specified; part (C) is the CoCoSim dropdown menu provided as a Matlab toolbox.

The analyzer as well as the traceability information of the CoCoSim toolchain produces both a complete environment to simulate the model in Simulink as well as the sequence of active Lustre states and Lustre transitions computed, leading to a violation of the property. The computation of the associated list of Stateflow states and transitions could be rebuilt from that information. This is left as a future work.

The verification technique via observer-based method was first introduced by Halbwachs et al. [13], where a safety property of an I/O machine $M$ is defined in terms of another machine called a synchronous observer. The observer watches the inputs and outputs of $M$, and if they ever violate the safety property it emits an alarm signal. Various work has been devoted in applying the observer-based method in different areas, e.g., [31]. In [25], Rushby explains how observer-based methods are quite versatile. For instance, they can be used to increase expressiveness, specify assumptions and axioms, but also to generate test cases. Synchronous observers are specified using the same language as the system under test. This is means that the user of CoCoSim can specify the desired requirements using Simulink blocks. Its main novelties can be summarized as: (i) It decouples the input language syntax from the underlying verification technique; (ii) It provides a convenient way to express safety requirements; (iii) It allows the combination of automated logic-based verification results with traditional Matlab based tools; and finally, (iv) it offers a convenient way to debug failed properties. An important aspect of CoCoSim is its ability to simplify the development and integration of new analysis techniques targeting Simulink/Stateflow models. This is accomplished via a modular architecture (see Fig. 7a).

**Supported Stateflow blocks** Our implementation in CoCoSim version v0.1 supports most known Stateflow constructs such as: *states* (Exclusive (OR) and Parallel (AND) states), most used *state actions* (entry, during and exit actions), *transitions* with the following transition labels: event only, event and condition, condition only, action only or event and condition and action combined. In addition to these constructs, CoCoSim supports default transitions, inner and outer transitions, self-loop transitions, inter-level transitions, connective junctions

and history junctions. CoCoSim also supports basic data types (int, real and booleans) and also arrays with static indexes (e.g., [2]). More specific charts [6] are supported such as: flow charts, Stateflow functions (graph functions), Hierarchical states, *enter*, *exit*, *send* and *after* operators.

The current version of CoCoSim does not support events emission in actions (state action or transition actions); it supports instead the *send* operator that can replace events emission. We also assume that the model does not have an unbound behavior (such as loop in junctions). In addition, transitions with more than one event are not yet supported[7].

**Datatypes.** Our work focuses on providing a global semantics to Stateflow, which we believe is hard enough. We do not address specific details such as how to represent Simulink datatypes in the target language Lustre. Several simple solutions nevertheless come to mind. One could provide each of these types in the form of an external Lustre library, totally hiding their implementations, at the cost of making explicit all the implicit coercions at work in the Simulink semantics. Another solution would be for instance to map every specifically-sized integer type to the unspecified Lustre integer type (that is what we currently do) and then to annotate the Lustre program in order to take care of sizes when generating C code or Horn clauses.

## 5.2 Experimental evaluation

We have performed a set of experimental evaluations to demonstrate the effectiveness of CoCoSim. Our experiments are carried out in a machine with the following specifications: Intel Core i5, 8Go RAM, 750Go HD, with Ubuntu 16.04. We used CoCoSim v0.1 which runs on Matlab 2014b and up, however, in this experiments we used Matlab 2016a with Stateflow version 8.7.

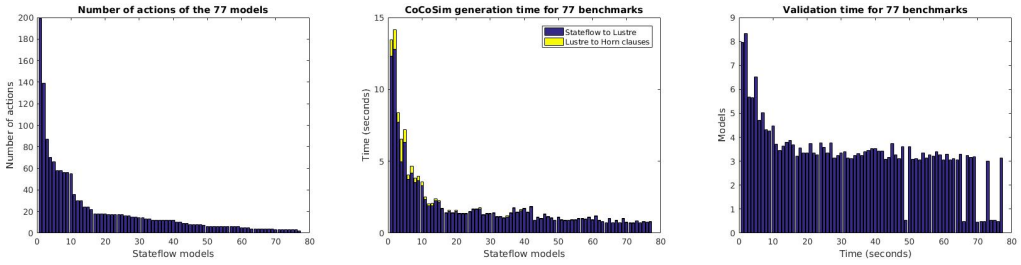
The first experiment was performed in order to illustrate the soundness of our compilation scheme. In particular, we would like to answer the question "*how faithful is the code generated via* CoCoSim?". In order to answer this question, we have compared the C code generated via CoCoSim with the one generated by Mathwork's *Simulink® Coder™* [21]. The latter is a popular product routinely used in different industrial application for code generation. Specifically, we have used a set of randomly generated test vectors with 100 iterations to validate the code generated by CoCoSim against the one generated by *Simulink® Coder™*. Such validation process is given as an option in CoCoSim. This allows a user to validate that the compiled code via CoCoSim conforms at least with the code generated by *Simulink® Coder™*. It also allowed us to demonstrate that other translation tools such as sf2lus [27] do not respect the full Stateflow semantic comparing to ours.

In our experiments, we have used a set of 77 Stateflow models [8] to evaluate the soundness and runtime of the compilation. Fig. 8a shows the size of the different models: number of actions is the sum of all state actions (entry, exit and during actions), condition and transition actions in the model. Fig. 8b illustrates the amount of time the compiler took to generate first Lustre code and then Horn clauses (used for verification). The models are given in decreasing number of actions, we can easily observe that generation time increases with the number of actions of the models. On average, CoCoSim takes about 1.85s to generate Lustre code. Fig. 8c shows the times for the validation script. On average, the validation process takes about 3.35s.

[6] https://github.com/coco-team/regression-test

[7] We are working on CoCoSim v0.2 to support: external C functions call and Matlab code, Arrays, On_event action and others.

[8] https://github.com/coco-team/regression-test




(a) (b) (c)

Figure 8: Runtime and validation experiments of CoCoSim.

| models | # props | # safe | # unsafe | # timeout | safe (time) | unsafe (time) |
|---|---|---|---|---|---|---|
| Microwave | 15 | 15 | 0 | 0 | 65.51 | 0 |
| NasaDockingApproach | 4 | 3 | 0 | 1 | 360 | 0 |
| GPCA_System_Monitor | 1 | 1 | 0 | 0 | 0.64 | 0 |
| GPCA_Logging | 1 | 1 | 0 | 0 | 4.88 | 0 |
| GPCA_Top_Level_Mode | 3 | 3 | 0 | 0 | 36 | 0 |
| GPCA_CONFIG | 1 | 0 | 1 | 0 | 0 | 19.34 |
| GPCA_INFUSION_MGR | 7 | 5 | 0 | 2 | 596.51 | 0 |
| GPCA_Alarm | 8 | 0 | 6 | 2 | 0 | 281.12 |

Figure 9: Experimental results of safety verification on a set of use cases.

## 5.3 Safety verification

The second experiment was performed in order to illustrate the effectiveness of our approach for the verification of safety properties. We have used 3 set of use cases. The first one is a Stateflow model of a microwave that captures the modal behavior of a typical microwave control software [9]. CoCoSim were able to verify 15 properties of this model using the backend solver Zustre [10] as the backend solving engine. The second use case is a Stateflow model that captures the complex behavior of the Space Shuttle when docking with the International Space Station (ISS) [26]. As the shuttle approaches the ISS, it goes through several operational modes related to how the shuttle is to orient itself for capture, dock with the ISS, and capture the ISS docking latch, among several other operational modes. The model describing this behavior is quite intricate and consists of a hierarchical and parallel state machines with three levels of hierarchy and multiple parallel state machines, including a total of 64 states. Using CoCoSim we were able to verify 2 out of 4 safety properties.

The third use case is the Generic Patient Controlled Analgesic infusion pump system. It consists of four main components: *Alarm*, *Infusion*, *Mode* and *Logging*. A more detailed description of the model can be found in [12]. Figure 9 summarizes the safety verification experimental evaluation results using CoCoSim.

[9]This model is developed by Rockwell Collins

# 6 Conclusion

In this paper, we have described an automated technique for the formal analysis of Stateflow models. Our approach faithfully compiles Stateflow models into hierarchical state machines, and uses automated logic-based verification engine to decide the validity of safety properties. We capture the compilation process using continuation-passing style (CPS) denotational semantics. Our compilation technique preserves the structural and modal behavior of the system, making the safety analysis of such models more tractable. We have presented a preliminary experimental evaluation that illustrates the effectiveness of our approach in the safety verification of industrial scale Stateflow models.

# References

[1] Continuation-passing style denotational semantics for stateflow. https://github.com/ploc/stateflow_CPS_semantics.

[2] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electr. Notes Theor. Comput. Sci.*, 109:43–56, 2004.

[3] A. W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.

[4] P. Boström and L. Morel. Mode-automata in simulink/stateflow. In *TUCS Technical Report No 772, September 2006*, 2006.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

[6] C. Chen, J. Sun, Y. Liu, J. S. Dong, and M. Zheng. Formal modeling and validation of stateflow diagrams. *International Journal on Software Tools for Technology Transfer*, 14(6):653–671, 2012.

[7] J. Colaço, G. Hamon, and M. Pouzet. Mixing signals and modes in synchronous data-flow systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, pages 73–82, 2006.

[8] J. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, pages 173–182, 2005.

[9] O. Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 131–142, New York, NY, USA, 2008. ACM.

[10] P. Garoche, A. Gurfinkel, and T. Kahsai. Synthesizing modular invariants for synchronous code. In N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors, *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014.*, volume 169 of *EPTCS*, pages 19–30, 2014.

[11] P. Garoche, T. Kahsai, and X. Thirioux. Hierarchical state machines as modular horn clauses. In *Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016.*, pages 15–28, 2016.

[12] C. Group. Generic patient controlled analgesia infusion pump project. https://crysis.cs.umn.edu/gpca.shtml.

[13] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993*, Workshops in Computing, pages 83–96. Springer, 1993.

[14] G. Hamon. A denotational semantics for stateflow. In *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, pages 164–172, 2005.

[15] G. Hamon and J. M. Rushby. An operational semantics for stateflow. In *FASE 2004*, pages 229–243, 2004.

[16] G. Hamon and J. M. Rushby. An operational semantics for stateflow. *STTT*, 9(5-6):447–456, 2007.

[17] T. Kahsai. CoCoSim – automated analysis framework for simulink. https://github.com/coco-team/cocoSim.

[18] J. L. Lawall and O. Danvy. Separating stages in the continuation-passing style transformation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 124–136, 1993.

[19] M. Li and R. Kumar. Recursive modeling of stateflow as input/output-extended automaton. *IEEE Trans. Automation Science and Engineering*, 11(4):1229–1239, 2014.

[20] Mathworks. Simulink. http://www.mathorks.com/products/simulink/.

[21] MathWorks. Simulink coder. https://www.mathworks.com/help/dsp/ug/generate-code-from-simulink.html.

[22] MathWorks. Stateflow. http://www.mathworks.com/products/stateflow/.

[23] P. J. Pingree, E. Mikk, G. J. Holzmann, M. H. Smith, and D. Dams. Validation of mission critical software design and implementation using model checking [spacecraft]. In *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, volume 1, pages 6A4–1–6A4–12 vol.1, Oct 2002.

[24] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[25] J. Rushby. The versatile synchronous observer. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software, A Festschrift Symposium in Honor of Kokichi Futatsugi*, volume 8373 of *Lecture Notes in Computer Science*, pages 110–128, Kanazawa, Japan, Apr. 2014. Springer-Verlag.

[26] M. Sampson and V. Derevenko. Interface definition document (idd) for international space station (iss) visiting vehicles (vvs). *NASA Technical Report*, 2000.

[27] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *EMSOFT 2004*, pages 259–268, 2004.

[28] N. K. Singh, M. Lawford, T. S. E. Maibaum, and A. Wassyng. Stateflow to tabular expressions. In H. Q. Thang, L. A. Phuong, L. D. Raedt, Y. Deville, M. Bui, T. T. D. Linh, N. T. Oanh, D. V. Sang, and N. B. Ngoc, editors, *Proceedings of the Sixth International Symposium on Information and Communication Technology, Hue City, Vietnam, December 3-4, 2015*, pages 312–319. ACM, 2015.

[29] E. Technologies. Scade.

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

[31] M. Westhead, S. Nadjm-tehrani, and F. H. E. U. K. Verification of embedded systems using synchronous observers. In *In Proceedings of the 4th International Conference on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS 1135*, pages 405–419. Springer Verlag, 1996.

[32] M. Whalen. A parametric structural operational semantics for stateflow, uml statecharts, and rhapsody. Technical report, UMSEC, 2010. https://www.umsec.umn.edu/sites/www.umsec.umn.edu/files/Parametric%20SOS%202_1.pdf.

[33] Y. Yang, Y. Jiang, M. Gu, and J. Sun. Verifying simulink/stateflow model: timed automata approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 852–857, 2016.

[34] L. Zou, N. Zhan, S. Wang, and M. Fränzle. Formal verification of simulink/stateflow diagrams. In

B. Finkbeiner, G. Pu, and L. Zhang, editors, *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015*, volume 9364 of *Lecture Notes in Computer Science*, pages 464–481. Springer, 2015.

# A Semantics of stopwatch example

We illustrate here the output of the stopwatch example as defined in Figures 1 and 2 when used with our Ocaml interpreter.

**Interpreter.** When provided with a simple sequence of events $[NO\_EVENT; START; TIC]$, it produces the following changes of active states. Note that a first step without provided event setup the model in its initial active state. We call `./sf_sem -model stopwatch -eval 2`

###1
main -> false
main.run -> false
main.run.lap -> false
main.run.running -> false
main.stop -> false
main.stop.lap_stop -> false
main.stop.reset -> false

###2
– Event none –
– no action performed –
main -> true
main.stop -> true
main.stop.reset -> true

###3
– Event START –
– no action performed –
main.run -> true
main.run.lap -> false
main.run.running -> true
main.stop -> false
main.stop.reset -> false

###4
– Event TIC –
– action performed –
cent+=1
cent==100
cont=0;sec+=1
sec==60
sec=0; min+=1
disp=(cent,sec,min)

« no state changed »

**Code generator.** We can also produce imperative-like code using the instianciation of Figure 3b. In our prototype, the modularity is tuned at three different levels: (i) no modularity at all, generating about 600 lines of code; (ii) modular with respect to $\mathcal{S}^d$, ie. for each state of the automaton, generating about 270 lines of code including seven functions focused on each state behavior; (iii) last, modular for each element: entry, during, exit actions, generating about 328 including 23 functions for each set of actions: entry, during or exit action of each state. We present here an excerpt of the code obtained with the second option: `./sf_sem -model stopwatch -modular 1 -gen_imp`

```
principal =
if Active(main) then <CallD(main)>
else <Open(main)>;
  if true then <Open(main.stop)>;
    if true then <Open(main.stop.reset)
        >
           else bot
    endif
         else bot
  endif
endif
...
component CallD(main.stop.reset) =
begin
  if Event(START) then
    if Active(main.stop.reset)
    then <Close(main.stop.reset)>
    else if Active(main.stop.lap_stop)
         then <Close(main.stop.lap_stop
             )>
         else <Nil>  endif
    endif;
    <Close(main.stop)>;
    <Open(main.run)>;
    <Open(main.run.running)>
  else if Event(LAP) then <reset
      counter>
                    else <Nil>  endif
  endif
end
```

The generation of Lustre is even more verbose: 2713 loc for inline calls, 1154 loc for state modular Lustre generation and 1132 with the fully modular model. We do not present here such generated code but refer the reader for the tool website to evaluate it. The generated is compatible with our tool LustreC and therefore can be used either to generate embeddable C code or to perform model-checking analyzes with our tool Zustre.