Automated Analysis of Stateflow Models

LPAR 2017, Maun, Botswana

Hamza Bourbouh, Christophe Garion, Pierre-Loïc Garoche, Arie Gurfinkel, Temesghen Kahsai & Xavier Thirioux

May, 9th 2017
Cocosim

Matlab Environment

FrontEnd

Compiler

Lustre

Interface to external tools

Kind2

Zustre

LustreC

JKind

pre-processing optimization

traceability
Cocosim & Stateflow
The Stopwatch Stateflow model

from Hamon, “A denotational semantics for stateflow”.


Extreme semantics

Hierarchical state machines, but:

- emission of signals restarts the global transitions evaluation
- non termination – stack overflow
  - loops in sequences of atomic transitions
  - unbounded number of atomic transitions steps for each step
- backtracking with side effects
- transition order depends on graphical layout
Motivation – Theoretical roots

1. Do we want to analyze this?
Motivation – Theoretical roots

1. Do we want to analyze this? Yes.
   ⇒ People are using it and asking for verification means
Motivation – Theoretical roots

1. Do we want to analyze this? Yes.
   ⇒ People are using it and asking for verification means
2. Any sound semantics bases?
Motivation – Theoretical roots

1. Do we want to analyze this? Yes.
   ⇒ People are using it and asking for verification means

2. Any sound semantics bases? Yes!

A Denotational Semantics for Stateflow *

Grégoire Hamon
Chalmers Institute of Technology
Göteborg, Sweden
hamon@cs.chalmers.se

ABSTRACT
We present a denotational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite. This semantics makes use of constructors for even the most complex constructions (as inter-level transitions, junctions, or immediate application of this semantics is a big obstacle to static analysis, verification, or automatic test-cases generation of Stateflow designs. These are allowed a parallel description of the system, has a purely abstract scheme for the language.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Semantics
D.2.6 [Software Engineering]: Programming Environments—Graphical User Interfaces

General Terms
Design, Languages

Keywords
Stateflow, denotational semantics, continu

An Operational Semantics for Stateflow*

Grégoire Hamon and John Rushby

Abstract. We present a formal operational semantics for Stateflow, in which transitions, in particular, have been taken, which has great advantages for understanding the behavior of a program, does not formalize a compilation process.

1. INTRODUCTION
As embedded systems grow in complexity and criticality, designers increasingly face problems of scalability and quality. Alongside these developments has been an increase in the number of embedded systems, and their increasing complexity. This has led to the development of many new tools and languages for designing and analyzing such systems. One of the most widespread model-based development environments is Stateflow, which is used to specify the discrete controller behavior of a system. Stateflow is a component of the Simulink graphical simulation environment in Matlab. Stateflow inherits all its simulation and code generation capabilities from Simulink. As part of the Matlab tool suite, Stateflow is probably the most versatile design tool of its kind. This integration of the suite makes it easier to check properties of a design and they can also apply formal methods. Formal methods can provide tools allowing a parallel description of the system, has a purely abstract scheme for the language.

We present a denotational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite. This semantics makes use of constructors for even the most complex constructions (as inter-level transitions, junctions, or immediate application of this semantics is a big obstacle to static analysis, verification, or automatic test-cases generation of Stateflow designs. These are allowed a parallel description of the system, has a purely abstract scheme for the language.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Semantics
D.2.6 [Software Engineering]: Programming Environments—Graphical User Interfaces

General Terms
Design, Languages

Keywords
Stateflow, denotational semantics, continu

Abstract. We present a formal operational semantics for Stateflow, in which transitions, in particular, have been taken, which has great advantages for understanding the behavior of a program, does not formalize a compilation process.

1. INTRODUCTION
As embedded systems grow in complexity and criticality, designers increasingly face problems of scalability and quality. Alongside these developments has been an increase in the number of embedded systems, and their increasing complexity. This has led to the development of many new tools and languages for designing and analyzing such systems. One of the most widespread model-based development environments is Stateflow, which is used to specify the discrete controller behavior of a system. Stateflow is a component of the Simulink graphical simulation environment in Matlab. Stateflow inherits all its simulation and code generation capabilities from Simulink. As part of the Matlab tool suite, Stateflow is probably the most versatile design tool of its kind. This integration of the suite makes it easier to check properties of a design and they can also apply formal methods. Formal methods can provide tools allowing a parallel description of the system, has a purely abstract scheme for the language.

We present a denotational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite. This semantics makes use of constructors for even the most complex constructions (as inter-level transitions, junctions, or immediate application of this semantics is a big obstacle to static analysis, verification, or automatic test-cases generation of Stateflow designs. These are allowed a parallel description of the system, has a purely abstract scheme for the language.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Semantics
D.2.6 [Software Engineering]: Programming Environments—Graphical User Interfaces

General Terms
Design, Languages

Keywords
Stateflow, denotational semantics, continu

Abstract. We present a formal operational semantics for Stateflow, in which transitions, in particular, have been taken, which has great advantages for understanding the behavior of a program, does not formalize a compilation process.

1. INTRODUCTION
As embedded systems grow in complexity and criticality, designers increasingly face problems of scalability and quality. Alongside these developments has been an increase in the number of embedded systems, and their increasing complexity. This has led to the development of many new tools and languages for designing and analyzing such systems. One of the most widespread model-based development environments is Stateflow, which is used to specify the discrete controller behavior of a system. Stateflow is a component of the Simulink graphical simulation environment in Matlab. Stateflow inherits all its simulation and code generation capabilities from Simulink. As part of the Matlab tool suite, Stateflow is probably the most versatile design tool of its kind. This integration of the suite makes it easier to check properties of a design and they can also apply formal methods. Formal methods can provide tools allowing a parallel description of the system, has a purely abstract scheme for the language.

We present a denotational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite. This semantics makes use of constructors for even the most complex constructions (as inter-level transitions, junctions, or immediate application of this semantics is a big obstacle to static analysis, verification, or automatic test-cases generation of Stateflow designs. These are allowed a parallel description of the system, has a purely abstract scheme for the language.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Semantics
D.2.6 [Software Engineering]: Programming Environments—Graphical User Interfaces

General Terms
Design, Languages

Keywords
Stateflow, denotational semantics, continu

Abstract. We present a formal operational semantics for Stateflow, in which transitions, in particular, have been taken, which has great advantages for understanding the behavior of a program, does not formalize a compilation process.

1. INTRODUCTION
As embedded systems grow in complexity and criticality, designers increasingly face problems of scalability and quality. Alongside these developments has been an increase in the number of embedded systems, and their increasing complexity. This has led to the development of many new tools and languages for designing and analyzing such systems. One of the most widespread model-based development environments is Stateflow, which is used to specify the discrete controller behavior of a system. Stateflow is a component of the Simulink graphical simulation environment in Matlab. Stateflow inherits all its simulation and code generation capabilities from Simulink. As part of the Matlab tool suite, Stateflow is probably the most versatile design tool of its kind. This integration of the suite makes it easier to check properties of a design and they can also apply formal methods. Formal methods can provide tools allowing a parallel description of the system, has a purely abstract scheme for the language.

We present a denotational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite. This semantics makes use of constructors for even the most complex constructions (as inter-level transitions, junctions, or immediate application of this semantics is a big obstacle to static analysis, verification, or automatic test-cases generation of Stateflow designs. These are allowed a parallel description of the system, has a purely abstract scheme for the language.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Semantics
D.2.6 [Software Engineering]: Programming Environments—Graphical User Interfaces

General Terms
Design, Languages

Keywords
Stateflow, denotational semantics, continu

Abstract. We present a formal operational semantics for Stateflow, in which transitions, in particular, have been taken, which has great advantages for understanding the behavior of a program, does not formalize a compilation process.

1. INTRODUCTION
As embedded systems grow in complexity and criticality, designers increasingly face problems of scalability and quality. Alongside these developments has been an increase in the number of embedded systems, and their increasing complexity. This has led to the development of many new tools and languages for designing and analyzing such systems. One of the most widespread model-based development environments is Stateflow, which is used to specify the discrete controller behavior of a system. Stateflow is a component of the Simulink graphical simulation environment in Matlab. Stateflow inherits all its simulation and code generation capabilities from Simulink. As part of the Matlab tool suite, Stateflow is probably the most versatile design tool of its kind. This integration of the suite makes it easier to check properties of a design and they can also apply formal methods. Formal methods can provide tools allowing a parallel description of the system, has a purely abstract scheme for the language.
The Stateflow Language

Program $P ::= (s, [src_0, \ldots, src_n])$
SrcComp $src ::= s : sd | j : T$
StateDef $sd ::= ((a_e, a_d, a_x), T_o, T_i, C)$
Comp $C ::= Or (T, [s_0, \ldots, s_n]) | And ([s_0, \ldots, s_n])$
Trans $t ::= (e, c, (a_c, a_t), d)$
Dest $T ::= \emptyset | t.T$
TransList $d ::= p | j$
Path $p ::= \emptyset | s.p$

No dynamic execution of signals
The Stopwatch Encoding

\[
\text{main.run.running} : (((\emptyset_a, \text{disp} = (\text{cent}, \text{sec}, \text{min}), \emptyset_a),
\text{[} (\text{START}, \text{true}, \emptyset_a, \emptyset_a, P \text{ main.stop.reset});
(LAP, \text{true}, \emptyset_a, \emptyset_a, P \text{ main.run.lap})], [], \text{Or} ([], ))
\]

\[
\text{main.run.lap} : (((\emptyset_a, \emptyset_a, \emptyset_a),
\text{[} (\text{START}, \text{true}, \emptyset_a, \emptyset_a, P \text{ main.stop.lap_stop});
(LAP, \text{true}, \emptyset_a, \emptyset_a, P \text{ main.run.running})], [], \text{Or} ([], ))
\]

\[
\text{main.run} : (((\emptyset_a, \emptyset_a, \emptyset_a), []),
\text{[} (\text{TIC}, \text{true}, \text{cent}+ = 1, \emptyset_a, J j1)], \text{Or} ([], \{\text{running}; \text{lap}\}))
\]

\[
j1 : [(\text{noevent}, \text{cent} == 100, \text{cont} = 0; \text{sec}+ = 1, \emptyset_a, J j2); (\text{noevent}, \text{cent}! = 100, \emptyset_a, \emptyset_a, J j3)]
\]

\[
j2 : [(\text{noevent}, \text{sec} == 60, \text{min}+ = 1, \emptyset_a, P \text{ main.run}); (\text{noevent}, \text{sec}! = 60, \emptyset_a, \emptyset_a, J j3)]
\]

\[
j3 : []
\]
An execution of the Stopwatch model

from Hamon, “A denotational semantics for stateflow”.

```plaintext
### 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
```

--- Event none ---

--- no action performed ---
An execution of the Stopwatch model

from Hamon, “A denotational semantics for stateflow”.

```plaintext
### 2
main -> true
main.run -> false
main.run.lap -> false               -- Event START --
main.run.running -> false           -- no action performed --
main.stop -> true
main.stop.lap_stop -> false
main.stop.reset -> true
```
An execution of the Stopwatch model

from Hamon, “A denotational semantics for stateflow”.

### 3
main -> true
main.run -> true
main.run.lap -> false
main.run.running -> true
main.stop -> false
main.stop.lap_stop -> false
main.stop.reset -> false

-- Event TIC --
-- action performed --
cent += 1
cent == 100
cont = 0; sec += 1
sec == 60
sec = 0; min += 1
disp = (cent, sec, min)
An execution of the Stopwatch model

from Hamon, “A denotational semantics for stateflow”.

### 4
main -> true
main.run -> true
main.run.lap -> false
main.run.running -> true
main.stop -> false
main.stop.lap_stop -> false
main.stop.reset -> false
-- Event START --
-- no action performed --
An execution of the Stopwatch model

from Hamon, “A denotational semantics for stateflow”.

```plaintext
### 5
main -> true
main.run -> false
main.run.lap -> false
main.run.running -> false
main.stop -> true
main.stop.lap_stop -> false
main.stop.reset -> true
```

---

Event TIC ---
--- no action performed ---
An execution of the Stopwatch model

from Hamon, “A denotational semantics for stateflow”.

### 6
main -> true
main.run -> false
main.run.lap -> false
main.run.running -> false
main.stop -> true
main.stop.lap_stop -> false
main.stop.reset -> true
Hamon’s Interpreter: Environments

- **Static environment of semantic functions:**

\[
\theta : \text{KEnv} ::= \{ p_0 : (S[p_0 : sd_0]^e \theta, S[p_0 : sd_0]^d \theta, S[p_0 : sd_0]^x \theta), \ldots, p_n : (S[p_n : sd_n]^e \theta, S[p_n : sd_n]^d \theta, S[p_n : sd_n]^x \theta) \}
\]

- **Dynamic environment of states/variables:**

\[
\rho : \text{Env} ::= \{ x_0 : v_0, \ldots, x_n : v_n, s_0 : b_0, \ldots, s_k : b_k \}
\]
Hamon’s Interpreter: Basics

- Continuations (as arguments) denote success/failure:
  
  \[ k_+ : Env \rightarrow path \rightarrow Env \]
  
  \[ k_- : Env \rightarrow Env \]

- Primitive operators:
  
  \[ \mathcal{A}^{[\cdot]} : action \rightarrow KEnv \rightarrow Env \rightarrow Env \]
  
  \[ \mathcal{B}^{[\cdot]} : condition \rightarrow KEnv \rightarrow Bool \]

- Predefined actions:
  
  \[ open\ p , \ close\ p \]
Hamon’s Interpreter: Transitions

- Transitions: if feasible transition, update the success continuation and continue path evaluation. If not, fail continuation

\[
\tau[(e_t, c, (a_c, a_t), d)] \theta \rho \text{ success fail } e = \\
\text{if } (e_t = e) \land (B[c] \rho) \text{ then} \\
\quad \text{let } \text{success}' = \\
\quad \quad \lambda \rho_s. \lambda p. \text{if } p = [] \text{ then } \text{success } \rho_s \ p \\
\quad \quad \quad \text{else } \text{success } (A[a_t] \theta \rho_s) \ p \text{ in} \\
\quad \quad \quad \quad \text{in } D[d] \theta (A[a_c] \theta \rho) \text{ success'} \ fail \ e \\
\quad \text{else} \\
\quad \quad \text{fail } \rho
\]
Hamon’s Interpreter: Transitions

- Transitions: if feasible transition, update the success continuation and continue path evaluation. If not, fail continuation

\[
\tau[\langle e_t, c, (a_c, a_t), d \rangle] \; \varrho \; \text{success} \; \text{fail} \; e = \\
\quad \text{if } (e_t = e) \land (B[c] \; \varrho) \text{ then} \\
\quad \quad \text{let } \text{success}' = \\
\quad \quad \quad \lambda \varrho_s. \lambda p. \text{if } p = [] \text{ then success } \varrho_s \; p \\
\quad \quad \quad \quad \quad \text{else success } (A[a_c] \; \varrho \; \varrho_s) \; p \text{ in} \\
\quad \quad \quad \quad \quad \quad D[d] \; \varrho \; (A[a_c] \; \varrho) \; \text{success}' \; \text{fail} \; e \\
\quad \quad \text{else} \\
\quad \quad \quad \text{fail } \varrho
\]

- Lists of Transitions: evaluate in order, building fail continuations

\[
T[t.0] \; \varrho \; \text{success} \; \text{fail} \; e = T[t] \; \varrho \; \text{success} \; \text{fail} \; e \\
T[t.t'.T] \; \varrho \; \text{success} \; \text{fail} \; e = \\
\quad \text{let } \text{fail}' = \lambda \varrho_f. T[t'.T] \; \varrho_f \; \text{success} \; \text{fail} \; e \text{ in} \\
\quad T[t] \; \varrho \; \text{success} \; \text{fail}' \; e
\]

Disclaimer: talk focuses on transitions, state opening/closing is also handled in the paper.
Hamon’s Interpreter: Transitions

- Transitions: if feasible transition, update the success continuation and continue path evaluation. If not, fail continuation

\[
\tau[(e_t, c, (a_c, a_t), d)] \theta \rho \text{ success fail } e = \\
\text{ if } (e_t = e) \land (B[c] \rho) \text{ then } \\
\text{ let } \text{success}' = \\
\lambda \rho_s . \lambda p . \text{if } p = [] \text{ then success } \rho_s p \text{ else success } (A[a_t] \theta \rho_s) p \text{ in } \\
D[d] \theta (A[a_c] \theta \rho) \text{ success}' \text{ fail } e \\
\text{ else } \\
\text{ fail } \rho
\]

- Lists of Transitions: evaluate in order, building fail continuations

\[
T[t.0] \theta \rho \text{ success fail } e = T[t] \theta \rho \text{ success fail } e \\
T[t.t'.T] \theta \rho \text{ success fail } e = \\
\text{ let } \text{fail}' = \lambda \rho_f . T[t'.T] \theta \rho_f \text{ success fail } e \text{ in } \\
T[t] \theta \rho \text{ success fail}' e
\]

- Destinations: final states \( p \) or intermediate junction \( j \)

\[
D[p] \theta \rho \text{ success fail } e = \text{success } \rho \rho p \\
D[j] \theta \rho \text{ success fail } e = \theta^i(j) \rho \text{ success fail } e
\]
Hamon’s Interpreter: Transitions

- Transitions: if feasible transition, update the success continuation and continue path evaluation. If not, fail continuation

\[
\tau[(e_t, c, (a_c, a_t), d)] \; \theta \; \rho \; \text{success} \; \text{fail} \; e = \\
\text{if } (e_t = e) \land (B[c] \; \rho) \text{ then} \\
\quad \text{let } \text{success}' = \\
\quad \quad \lambda \rho_s. \lambda p. \text{if } p = [] \text{ then } \text{success } \rho_s \; p \\
\quad \quad \text{else } \text{success } (A[a_t] \; \theta \; \rho_s) \; p \text{ in} \\
\quad \quad \quad D[d] \; \theta \; (A[a_c] \; \theta \; \rho) \; \text{success}' \; \text{fail} \; e \\
\quad \text{else} \\
\quad \quad \text{fail } \rho
\]

- Lists of Transitions: evaluate in order, building fail continuations

\[
T[t.0] \; \theta \; \rho \; \text{success} \; \text{fail} \; e = \tau[t] \; \theta \; \rho \; \text{success} \; \text{fail} \; e \\
T[t.t'.T] \; \theta \; \rho \; \text{success} \; \text{fail} \; e = \\
\quad \text{let } \text{fail}' = \lambda \rho_f. T[t'.T] \; \theta \; \rho_f \; \text{success} \; \text{fail} \; e \text{ in} \\
\quad \tau[t] \; \theta \; \rho \; \text{success} \; \text{fail}' \; e
\]

- Destinations: final states \(p\) or intermediate junction \(j\)

\[
D[p] \; \theta \; \rho \; \text{success} \; \text{fail} \; e = \text{success } \rho \; p \\
D[j] \; \theta \; \rho \; \text{success} \; \text{fail} \; e = \theta i(j) \; \rho \; \text{success} \; \text{fail} \; e
\]

Disclaimer: talk focuses on transitions, state opening/closing is also handled in the paper.
Problems with Hamon’s semantics

- transition actions executed in reverse order

\[(c_1, t_1) \rightarrow (c_2, t_2)\] should evaluate to \[(c_1, c_2, t_1, t_2)\]

\[
\tau[(e_t, c, (a_c, a_t), d)] \theta \rho \text{ success fail } e =
\]
\[
\text{if } (e_t = e) \land (B[c] \rho) \text{ then}
\]
\[
\text{let } success' =
\]
\[
\lambda \rho_s . \lambda p . \text{if } p = [] \text{ then success } \rho_s \ p
\]
\[
\text{else success } (A[a_t] \theta \rho_s) \ p \text{ in}
\]
\[
D[d] \theta (A[a_c] \theta \rho) success' \ fail e
\]
\[
\text{else}
\]
\[
\text{fail } \rho
\]
Problems with Hamon’s semantics

- Transition actions executed in reverse order
  
  \[(c_1, t_1) \rightarrow (c_2, t_2)\] should evaluate to \[(c_1, c_2, t_1, t_2)\]

\[
\tau[(e_t, c, (a_c, a_t), d)] \theta \rho \text{ success fail } e =
\]

\[
\text{if } (e_t = e) \land (B[c] \rho) \text{ then}
\]

\[
\text{let } \text{success}' = \lambda \rho_s. \lambda p. \text{if } p = [] \text{ then } \text{success } \rho_s \ p
\]

\[
\text{else } \text{success} (A[a_t] \theta \rho_s) \ p \text{ in}
\]

\[
D[d] \theta (A[a_c] \theta \rho) \text{ success}' \text{ fail } e
\]

- Invalid order of entering/closing actions when a transition succeeds
- Outer/inner/entering transitions don’t conform to standard
Problems with Hamon’s semantics

- Transition actions executed in reverse order
  \[(c_1, t_1) \rightarrow (c_2, t_2)\] should evaluate to \[(c_1, c_2, t_1, t_2)\]

\[
\tau[(e_t, c, (a_c, a_t), d)]] \theta \rho \text{ success fail } e =
\begin{align*}
\text{if } (e_t = e) \land (B[c] \rho) \text{ then } \\
\text{let } \text{success}' = \\
\lambda \rho s. \lambda p. \begin{cases} \\
\text{success } \rho s \ p \\
\text{else success } (A[a_t] \theta \rho s) \ p \text{ in } \\
D[d] \theta (A[a_c] \theta \rho) \text{ success'} \ fail \ e \\
\end{cases} \\
\text{else fail } \rho
\end{align*}

- Invalid order of entering/closing actions when a transition succeeds
- Outer/inner/entering transitions don’t conform to standard
- More importantly: could be made more aesthetic
  - contains a mix a continuations (denotations) and first order evaluation

\[
C[\text{Or}(T, S)]^x \theta \rho e =
\text{fold } (\lambda p. \lambda \rho. \text{ if } \rho(p) \text{ then } \theta^x(p) \ p \ e \text{ else } \rho) \ S \rho
\]
Our Proposition: a pure Continuation Passing Style (CPS) semantics

Restore Stateflow semantics

- Introduce a wrapper continuation
- Introduce a global failure continuation
- Distinguish between outer, inner and entering transitions with modes

Enlarge the Scope

- Factorize out and abstract away environment $\rho$:
  
  - enables interpreter, code generator, source-to-source transformation, etc
  - be careful with loops in junction sequences
- Introduce fine-grained memoization and modularity
CPS – Continuation Passing Style denotational semantics

- proposed in the 70s by Plotkin\(^1\) for \(\lambda\)-calculus call-by-value semantics
- developed for efficient compilation: Lawall, Danvy\(^2\) or Appel\(^3\) “offering a good format for compilation and optimization”

Plotkin’s call-by-value CPS rules:

\[
\begin{align*}
\llbracket x \rrbracket \kappa &= \kappa \ x \\
\llbracket \lambda x. e \rrbracket \kappa &= \kappa (\lambda x \cdot \lambda k \cdot \llbracket e \rrbracket \ k) \\
\llbracket e_0 e_1 \rrbracket \kappa &= \llbracket e_0 \rrbracket (\lambda v_0.\llbracket e_1 \rrbracket (\lambda v_1 \cdot v_0 \ v_0 \ v_1 \ k))
\end{align*}
\]

Associate to each function an explicit continuation \(\kappa : t \rightarrow t\), endomorphic map over \(t\) on which control is explicitly modeled.

---


\(^2\) Julia L. Lawall and Olivier Danvy. “Separating Stages in the Continuation-Passing Style Transformation”. In: POPL’93.

CPS semantics: Basics

- Continuations denote wrapping/success/failure:
  \[ w : \text{path} \rightarrow \text{Den} \rightarrow \text{Den} \]
  \[ k+ : \text{Den} \]
  \[ k- : \text{Den} \]

- Primitive operators:
  \[ \mathcal{A}[][.] : \text{action} \rightarrow \text{KEnv} \rightarrow \text{Den} \]
  \[ \mathcal{Ite}[][.] : \text{condition} \rightarrow \text{KEnv} \rightarrow \text{Den} \rightarrow \text{Den} \rightarrow \text{Den} \]
  \[ \gg : \text{Den} \rightarrow \text{Den} \rightarrow \text{Den} \]
  \[ \mathcal{I}d : \text{Den} \]

- Predefined actions/conditions:
  \[ \text{open } p, \quad \text{close } p, \quad \text{active } p \]

- Loose (L) or strict (S) mode
- Outer (o), inner (i) or entering (e) mode
CPS semantics: Transitions

- Transitions:

\[ \tau[(e_t, c, (a_c, a_t), d)] (\theta : KEnv) (\text{wrapper} : w) (\text{success} : k^+) (\text{fail} : k^-) (\text{fail}^{\text{glob}} : k^-) : \text{Den} = \]

\[ \text{Ite}(\text{event}(e_t) \land c, \]

\[ (\text{let } \text{success}' = \text{success} \Rightarrow (A[a_t]) \text{ in} \]

\[ (A[a_c]) \Rightarrow (D[d] \theta \text{wrapper success}' \text{ fail fail}^{\text{glob}})), \]

\[ \text{fail}) \]
CPS semantics: Transitions

- Transitions:

\[ \tau[(e_t, c, (a_c, a_t), d)] (\theta : KEnv) (wrapper : w) (success : k^+) (fail : k^-) (fail^{glob} : k^-) : Den = \]

\[ \text{Ite}(\text{event}(e_t) \land c, \]

\[ (\text{let } success' = success \implies (A[a_t])) \text{ in} \]

\[ (A[a_c]) \implies (D[d] \theta \text{ wrapper success' fail fail}^{glob}), \]

\[ \text{fail}) \]

- Lists of Transitions:

\[ \mathcal{T}[t.0] \theta \text{ wrapper success fail fail}^{glob} = \tau[t] \theta \text{ wrapper success fail fail}^{glob} \]

\[ \mathcal{T}[t.T] \theta \text{ wrapper success fail fail}^{glob} = \]

\[ \text{let } \text{fail}' = \mathcal{T}[T] \theta \text{ wrapper success fail fail}^{glob} \text{ in} \]

\[ \tau[t] \theta \text{ wrapper success fail'} fail^{glob} \]

- Destinations:

\[ \mathcal{D}[p] \theta \text{ wrapper success fail fail}^{glob} = \text{wrapper p success} \]

\[ \mathcal{D}[j] \theta \text{ wrapper success fail fail}^{glob} = \theta(j) \text{ wrapper success fail fail}^{glob} \]
CPS semantics: States

- Entering/exiting states (loosely or strictly):

\[
S[p : ((a_e, a_d, a_x), T_0, T_i, C)]^e_S (\theta : KEnv) (\emptyset : Path) = (C[C]^e p \theta)
\]
\[
S[p : ((a_e, a_d, a_x), T_0, T_i, C)]^e_S \theta \cdot s.p_d = (\theta_L^e(p.s) p_d)
\]
\[
S[p : ((a_e, a_d, a_x), T_0, T_i, C)]^x_S (\theta : KEnv) : Den = (C[C]^x p \theta)
\]

- Computing states reactions:

\[
S[p : ((a_e, a_d, a_x), T_0, T_i, C)]^d_S \theta : Den =
\]
\[
\begin{align*}
&\text{let } \text{wrapper}_i = \text{open\_path}^i \emptyset p \text{ in} \\
&\text{let } \text{wrapper}_o = \text{open\_path}^o \emptyset p \text{ in} \\
&\text{let } \text{fail}_o =
\end{align*}
\]
\[
\begin{align*}
&\text{let } \text{fail}_i = C[C]^d p \theta \text{ in} \\
&(\mathcal{A}[a_d] \theta) \gg (T[T_i] \theta \text{ wrapper}_i \ Id \ fail_i \ fail_i) \text{ in} \\
&T[T_o] \theta \text{ wrapper}_o \ Id \ fail_o \ fail_o
\end{align*}
\]

\[
\text{open\_path}^v \theta p p_s p_d : w =
\]
\[
\begin{align*}
\text{if } &\text{hd}(p_s) = \text{hd}(p_d) \wedge \text{hd}(p_s) \neq \emptyset \text{ then} \\
&\text{open\_path}^v \theta p . \text{hd}(p_s) \text{ t1}(p_s) \text{ t1}(p_d) \\
&\text{else match } v \text{ with} \\
&o \rightarrow \lambda \text{den}. \theta_L^e(p.\text{hd}(p_s)) \gg \text{den} \gg \theta_L^e(p.\text{hd}(p_d)) \text{ t1}(p_d) \\
i \rightarrow \lambda \text{den}. \theta_S^e(p.\text{hd}(p_s)) \gg \text{den} \gg \theta_S^e(p.\text{hd}(p_d)) \text{ t1}(p_d) \\
e \rightarrow \lambda \text{den}. \text{den} \gg \theta_L^e(p.\text{hd}(p_d)) \text{ t1}(p_d)
\end{align*}
\]
Instanciating the CPS encoding

CPS framework fully parametric:

- Types for denotation/continuation: what do we want to build/manipulate?
- Definition of primitive operators on the continuations:
  - open $p$, close $p$
  - Assignment: $v = expr$
  - Ite construct: $Ite(cond, T, E)$:
  - Composition $\Rightarrow$

Instanciations:

- Interpreter
- Imperative Code generator
- Dataflow Code Generator (Lustre)
Instantiations: Interpreter

- Denotation type: $\text{Den} = \text{Env} \rightarrow \text{Env}$
- Rules:

\[
\begin{align*}
\mathcal{A}[\text{open } p](\rho) & = \rho [p \mapsto \text{true}] \\
\mathcal{A}[\text{close } p](\rho) & = \rho [p \mapsto \text{false}] \\
\mathcal{A}[v = \text{expr}](\rho) & = \rho [v \mapsto [\text{expr}]_\rho] \\
\text{Ite}(\text{cond}, T, E)(\rho) & = \text{if } [\text{cond}]_\rho \text{ then } T(\rho) \\
 & \quad \text{else } E(\rho) \\
(D_1 \gg D_2)(\rho) & = D_2 \circ D_1(\rho) \\
\text{Id}(\rho) & = \rho \\
\bot & = \text{assert false}
\end{align*}
\]
Instantiations: Code Generator

- Denotation type:

\[
Den ::= \begin{align*}
& Den;Den \\
| & \text{if } \text{cond} \text{ then } Den \text{ else } Den \\
| & v = expr \ | \ \text{nop} \ | \ \text{assert false}.
\end{align*}
\]

- Rules:

\[
\begin{align*}
A[\text{open } p] &= p = \text{true} \\
A[\text{close } p] &= p = \text{false} \\
A[v = expr] &= v = expr \\
Ite(\text{cond}, T, E) &= \text{if } \text{cond} \text{ then } T \\
& \quad \text{else } E \\
(D_1 \gg D_2) &= D_1 ; D_2 \\
Id &= \text{nop} \\
\bot &= \text{assert false}
\end{align*}
\]
Code Generated from Stopwatch Example

principal =
if Active(main)
    <CallD(main)>
then
else
    <Open(main)>;
    <Open(main.stop)>;
    <Open(main.stop.reset)>  
endif
component CallD(main.run.lap) =
begin
  if Event(START)
  then if Active(main.run.running)
      then <Close(main.run.running)>
     else if Active(main.run.lap)
      then <Close(main.run.lap)>
     else <Nil>;
  <Close(main.run)>; <Open(main.stop)>;
  <Open(main.stop.lap_stop)>
else if Event(LAP)
  then <Close(main.run.lap)>;
  <Open(main.run.running)>
else <NIL>
end
Modularity through Memoization

- Each evaluation of denotation \( \theta^e(p) \), \( \theta^d(p) \) or \( \theta^x(p) \) may be substituted by a call to a procedure
- This is possible since all arguments are static (paths, modes)
- Denotation \( \theta^j(j) (= \mathcal{T}[[j : T]] \theta) \) could also be turned into a call
- We need first-order representations of continuation arguments, through e.g. defunctionalization
  - wrapper \( \equiv \) mode \( \times \) path, success \( \equiv \) action list, fail \( \equiv \)??
- We could then factorize out junctions occurring in many paths, avoiding combinatorial blow-ups
- And handle loops, provided no transition actions occur
Lustre is a dataflow language with notions of automata

⇒ core language of our CocoSim toolchain

Denotation type: \( Den = Name \rightarrow Name \rightarrow LustreAST \)

Rules:

\[
\begin{align*}
\mathcal{A}[\text{open } p] \text{ in out} & := \quad \overline{\text{out}} = \overline{\text{in}[\text{in}_p \mapsto \text{true}]} \\
\mathcal{A}[\text{close } p] \text{ in out} & := \quad \overline{\text{out}} = \overline{\text{in}[\text{in}_p \mapsto \text{false}]} \\
\mathcal{A}[v = \text{expr}] \text{ in out} & := \quad \overline{\text{out}} = \overline{\text{in}[\text{in}_v \mapsto [\text{expr}]]} \\
\mathcal{A}[\text{call } p] \text{ in out} & := \quad \overline{\text{out}} = \text{thetad}_p(\overline{\text{in}}) \\
(L_1 \gg L_2) \text{ in out} & := \quad (L_1 \text{ in name}_{uid}) ; \\
& \quad (L_2 \text{ name}_{uid} \text{ out}) \\
\mathcal{T}d \text{ in out} & := \quad \overline{\text{out}} = \overline{\text{in}} \\
\bot \text{ in out} & := \quad \text{assert false}
\end{align*}
\]

Figure: Lustre instantiation
node thetad_p (\text{\textit{in}} : \textit{T}_{\text{in}}) \text{ returns } (\textit{out} : \textit{T}_{\text{out}})
let (S^{d}[p] \text{ in out}); \text{tel}

\text{Ite} (\text{\textit{cond}}, T, E) \text{ in out} :=
\text{automaton name}_{\text{uid}}
\text{state Cond :}
  \text{unless } [\neg \text{\textit{cond}}]_{\text{in}} \text{ restart NotCond}
  \text{let } (T \text{ in out}); \text{tel}
\text{state NotCond :}
  \text{unless } [\text{\textit{cond}}]_{\text{in}} \text{ restart Cond}
  \text{let } (E \text{ in out}); \text{tel}

\text{Figure: Lustre instantiation}
node thetad_p (\_in : \_T_{in}) returns (\_out : \_T_{out})
  let (\_S_d[p] \_in \_out); tel

\text{Ite}(\text{cond}, T, E) \_in \_out :=
  \text{automaton name}_{uid}
  \hspace{1em} \text{state Cond}:
    \hspace{1em} \text{unless } [\neg \text{cond}] \_in \text{ restart NotCond}
    \hspace{1em} \text{let } (T \_in \_out); \text{ tel}
  \hspace{1em} \text{state NotCond}:
    \hspace{1em} \text{unless } [\text{cond}] \_in \text{ restart Cond}
    \hspace{1em} \text{let } (E \_in \_out); \text{ tel}

\textbf{Figure:} Lustre instantiation

Encoding preserves the hierarchical structure of input model
Experimentation / Implementation

- Generic CPS prototype in Ocaml
- Direct encoding of the modular compilation scheme for Lustre in CocoSim in Matlab
  - encode Stateflow constructs into Lustre + automata (while preserving structure)
- Good performances: enable compilation and verification property is valid or a counter-example is produced

<table>
<thead>
<tr>
<th>models</th>
<th># props</th>
<th># safe</th>
<th># unsafe</th>
<th># timeout</th>
<th>safe (time)</th>
<th>unsafe (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microwave</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>65.51</td>
<td>0</td>
</tr>
<tr>
<td>NasaDockingApproach</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>360</td>
<td>0</td>
</tr>
<tr>
<td>GPCA_System_Monitor</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.64</td>
<td>0</td>
</tr>
<tr>
<td>GPCA_Logging</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4.88</td>
<td>0</td>
</tr>
<tr>
<td>GPCA_Top_Level_Mode</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>GPCA_CONFIG</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>19.34</td>
</tr>
<tr>
<td>GPCA_INFUSION_MGR</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>596.51</td>
<td>0</td>
</tr>
<tr>
<td>GPCA_Alarm</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>281.12</td>
</tr>
</tbody>
</table>
Contribution

- CPS encoding of Stateflow semantics
- Instanciation as
  - interpreter
  - imperative code generator
  - Lustre code generator
- Implemented
  - in Ocaml in the general settings and
  - in Matlab in the Lustre one
- Enable code generation and model verification of general Simulink/Stateflow models

Perspectives:

- Substitute Matlab algorithm by our Ocaml generic CPS code
- Compile basic automata into more complex one
  - avoid huge number of nested binary automata
- More fine grain integration with CocoSim
  - nodes in Simulink within Stateflow nodes
  - call to external C functions (S-functions)
  - interpret counter example over Stateflow nodes
Thank you for your attention!

Any questions?