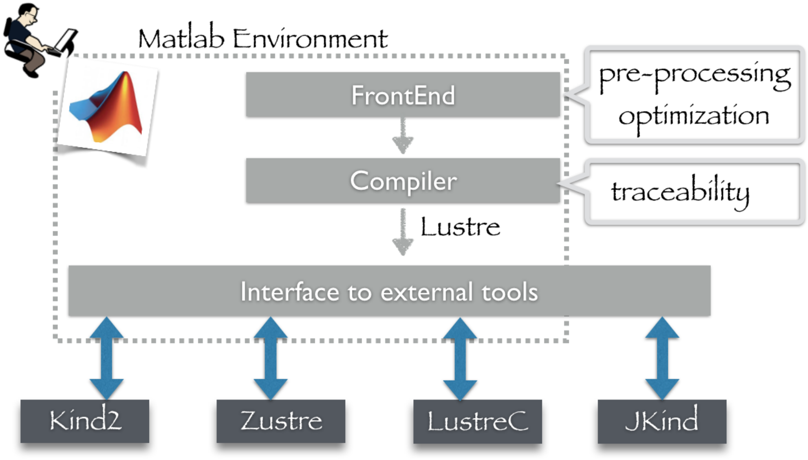
Automated Analysis of Stateflow Models

LPAR 2017, Maun, Botswana

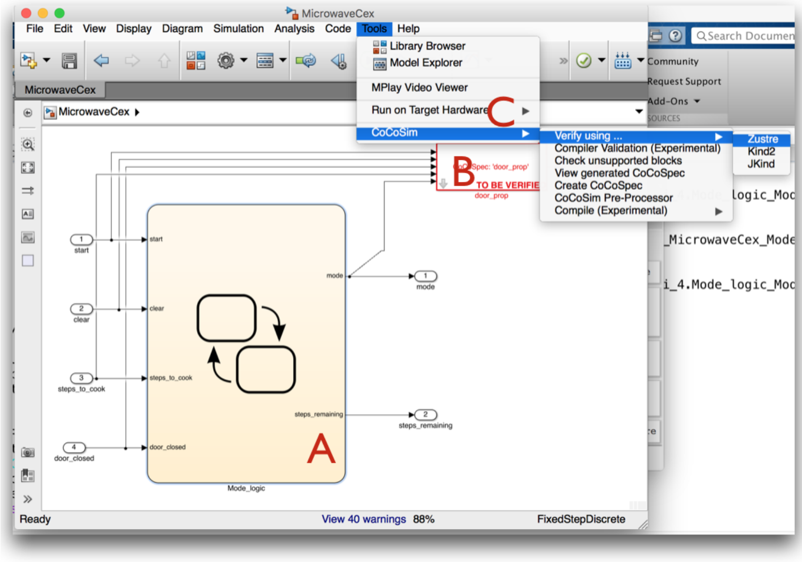
Hamza Bourbouh, Christophe Garion, Pierre-Loïc Garoche, Arie Gurfinkel, Temesghen Kahsai & Xavier Thirioux

May, 9th 2017

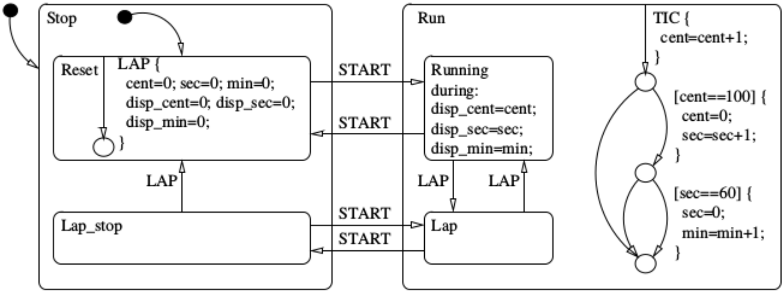
Cocosim


Matlab Environment
FrontEnd
pre-processing
optimization
Compiler
traceability
Lustre
Interface to external tools
Kind2
Zustre
LustreC
JKind

Cocosim & Stateflow



The Stopwatch Stateflow model

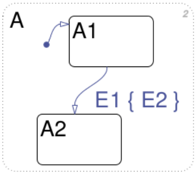


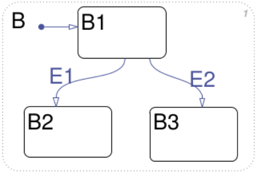
from Hamon, “A denotational semantics for stateflow”.

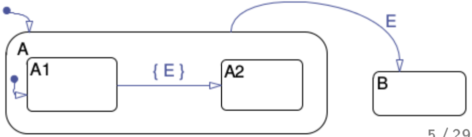
Extreme semantics

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.
   $\Rightarrow$ People are using it and asking for verification means

Motivation – Theoretical roots

1. Do we want to analyze this? Yes.
   $\Rightarrow$ 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.
   $\Rightarrow$ 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 fo
ical Statecharts-like language of the M
suite. This semantics makes use of con
even the most complex constructions c
as inter-level transitions, junctions, or
mediate application of this semantics is
scheme for the language.

Categories and Subject Descri

D.3.1 [**Programming Languages**]: F
Theory—*Semantics*; D.2.6 [**Software**
gramming Environments—*Graphical* E

General Terms

Design, Languages

Keywords

Stateflow,denotational semantics, conti

1. INTRODUCTION

As embedded systems grow in comp

International Journal on Software Tools for Technology Transfer (STTT)
Volume 9, Numbers 5-6, October 2007; Special section FASE'04/05, Pages 447–456

An Operational Semantics for Stateflow*

Grégoire Hamon and John Rushby

[1] The MathWorks, Natick, MA, USA
[2] Computer Science Laboratory, SRI International, Menlo Park CA, USA

The Stateflow Language

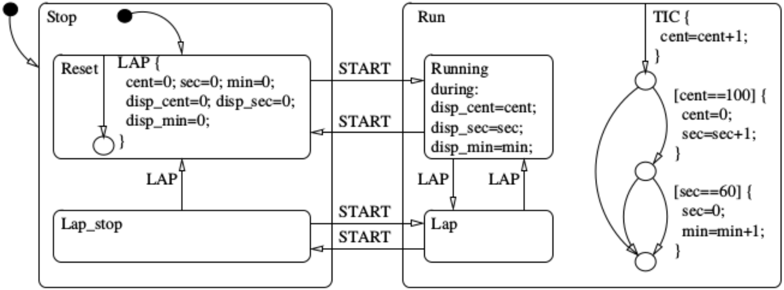
Program	P	$::=$	$(s, [src_0, \ldots, src_n])$
SrcComp	src	$::=$	$s : sd \mid j : T$
StateDef	sd	$::=$	$((a_e, a_d, a_x), T_o, T_i, C)$
Comp	C	$::=$	$Or\ (T, [s_0, \ldots, s_n]) \mid And\ ([s_0, \ldots, s_n])$
Trans	t	$::=$	$(e, c, (a_c, a_t), d)$
Dest	T	$::=$	$\emptyset \mid t.T$
TransList	d	$::=$	$p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

No dynamic execution of signals

The Stopwatch Encoding

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

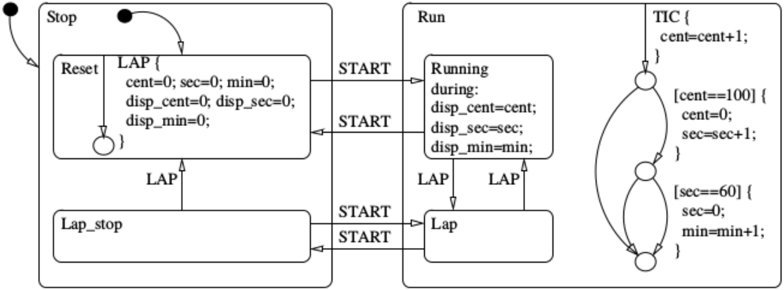
An execution of the Stopwatch model



from Hamon, "A denotational semantics for stateflow".

```
#### 1
main -> false
main.run -> false
main.run.lap -> false                  -- Event none --
main.run.running -> false              -- no action performed --
main.stop -> false
main.stop.lap_stop -> false
main.stop.reset -> false
```

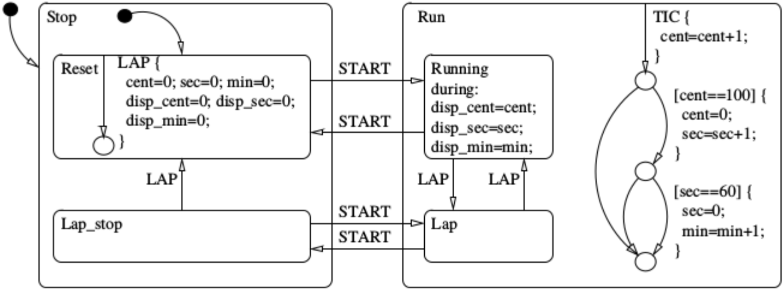
An execution of the Stopwatch model



from Hamon, "A denotational semantics for stateflow".

```
#### 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

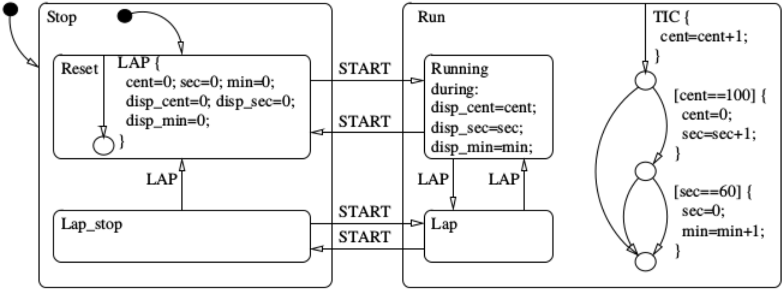


from Hamon, "A denotational semantics for stateflow".

```
#### 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)
```

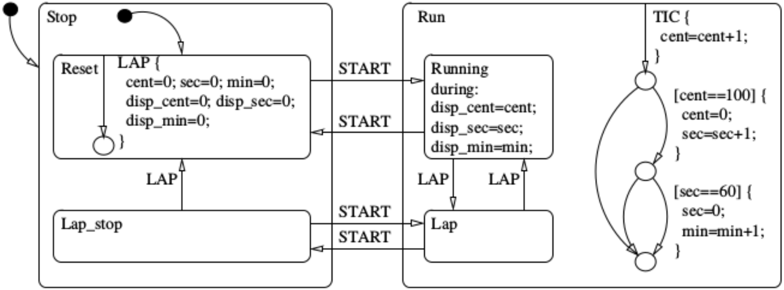
An execution of the Stopwatch model



from Hamon, "A denotational semantics for stateflow".

```
#### 4
main -> true
main.run -> true
main.run.lap -> false                    -- Event START --
main.run.running -> true                 -- no action performed --
main.stop -> false
main.stop.lap_stop -> false
main.stop.reset -> false
```

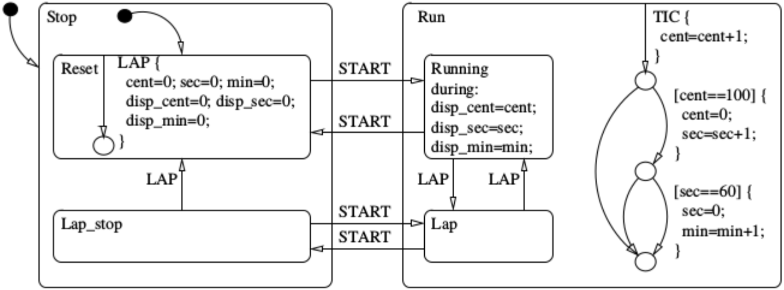
An execution of the Stopwatch model



from Hamon, "A denotational semantics for stateflow".

```
#### 5
main -> true
main.run -> false
main.run.lap -> false                    -- Event TIC --
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".

```
#### 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:

$$
\begin{aligned}
\theta : \mathsf{KEnv} \quad ::= \quad \{ \; & p_0 : (\mathcal{S}[\![p_0 : sd_0]\!]^e\ \theta, \mathcal{S}[\![p_0 : sd_0]\!]^d\ \theta, \mathcal{S}[\![p_0 : sd_0]\!]^x\ \theta) \\
& \ldots \\
& p_n : (\mathcal{S}[\![p_n : sd_n]\!]^e\ \theta, \mathcal{S}[\![p_n : sd_n]\!]^d\ \theta, \mathcal{S}[\![p_n : sd_n]\!]^x\ \theta) \\
& j_0 : \mathcal{T}[\![T_0]\!]\ \theta, \ldots, j_k : \mathcal{T}[\![T_k]\!]\ \theta \}
\end{aligned}
$$

- Dynamic environment of states/variables:

$$
\begin{aligned}
\rho : \mathsf{Env} \quad ::= \quad & \{x_0 : v_0, \ldots, x_n : v_n, \\
& \; s_0 : b_0, \ldots, s_k : b_k\}
\end{aligned}
$$

Hamon's Interpreter: Basics

- Continuations (as arguments) denote success/failure:

$$k+ : Env \rightarrow path \rightarrow Env$$
$$k- : Env \rightarrow Env$$

- Primitive operators:

$$\mathcal{A}[\![.]\!] \ : \ action \rightarrow KEnv \rightarrow Env \rightarrow Env$$
$$\mathcal{B}[\![.]\!] \ : \ condition \rightarrow KEnv \rightarrow Bool$$

- Predefined actions:

$$open\ p, \quad close\ p$$

Hamon's Interpreter: Transitions

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

$$
\begin{array}{l}
\tau[\![(\boldsymbol{e_t}, \boldsymbol{c}, (\boldsymbol{a_c}, \boldsymbol{a_t}), \boldsymbol{d})]\!]\ \theta\ \rho\ \textit{\textbf{success}}\ \textit{\textbf{fail}}\ \boldsymbol{e} = \\
\quad \texttt{if}\ (\boldsymbol{e_t} = \boldsymbol{e}) \wedge (\mathcal{B}[\![\boldsymbol{c}]\!]\ \rho)\ \texttt{then} \\
\quad\quad \texttt{let}\ \textit{\textbf{success}}' = \\
\quad\quad\quad \lambda\rho_{\boldsymbol{s}}.\lambda\boldsymbol{p}.\texttt{if}\ \boldsymbol{p} = []\ \texttt{then}\ \textit{\textbf{success}}\ \rho_{\boldsymbol{s}}\ \boldsymbol{p} \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else}\ \textit{\textbf{success}}\ (\mathcal{A}[\![\boldsymbol{a_t}]\!]\ \theta\ \rho_{\boldsymbol{s}})\ \boldsymbol{p}\ \texttt{in} \\
\quad\quad \mathcal{D}[\![\boldsymbol{d}]\!]\ \theta\ (\mathcal{A}[\![\boldsymbol{a_c}]\!]\ \theta\ \rho)\ \textit{\textbf{success}}'\ \textit{\textbf{fail}}\ \boldsymbol{e} \\
\quad \texttt{else} \\
\quad\quad \textit{\textbf{fail}}\ \rho
\end{array}
$$

Hamon's Interpreter: Transitions

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

$$
\begin{array}{l}
\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\
\quad\quad \texttt{let}\ \mathit{success}' = \\
\quad\quad\quad \lambda\rho_s.\lambda p.\texttt{if}\ p = []\ \texttt{then}\ \mathit{success}\ \rho_s\ p \\
\quad\quad\quad\quad\quad\quad \texttt{else}\ \mathit{success}\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\
\quad\quad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho)\ \mathit{success}'\ \mathit{fail}\ e \\
\quad \texttt{else} \\
\quad\quad \mathit{fail}\ \rho
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{T}[\![t.\emptyset]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \tau[\![t]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e \\
\mathcal{T}[\![t.t'.T]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{let}\ \mathit{fail}' = \lambda\rho_f.\mathcal{T}[\![t'.T]\!]\ \theta\ \rho_f\ \mathit{success}\ \mathit{fail}\ e\ \texttt{in} \\
\quad \tau[\![t]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}'\ e
\end{array}
$$

Hamon's Interpreter: Transitions

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

$$
\begin{array}{l}
\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\
\qquad \texttt{let}\ \mathit{success}' = \\
\qquad\quad \lambda\rho_s.\lambda p.\texttt{if}\ p = []\ \texttt{then}\ \mathit{success}\ \rho_s\ p \\
\qquad\qquad\qquad\qquad \texttt{else}\ \mathit{success}\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\
\qquad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho)\ \mathit{success}'\ \mathit{fail}\ e \\
\quad \texttt{else} \\
\qquad \mathit{fail}\ \rho
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{T}[\![t.\emptyset]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \tau[\![t]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e \\
\mathcal{T}[\![t.t'.T]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{let}\ \mathit{fail}' = \lambda\rho_f.\mathcal{T}[\![t'.T]\!]\ \theta\ \rho_f\ \mathit{success}\ \mathit{fail}\ e\ \texttt{in} \\
\quad \tau[\![t]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}'\ e
\end{array}
$$

- Destinations: final states p or intermediate junction j

$$
\begin{array}{l}
\mathcal{D}[\![p]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \mathit{success}\ \rho\ p \\
\mathcal{D}[\![j]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \theta^{\mathsf{j}}(j)\ \rho\ \mathit{success}\ \mathit{fail}\ e
\end{array}
$$

Hamon's Interpreter: Transitions

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

$$
\begin{array}{l}
\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\
\quad\quad \texttt{let}\ \mathit{success}' = \\
\quad\quad\quad \lambda\rho_s.\lambda p.\texttt{if}\ p = []\ \texttt{then}\ \mathit{success}\ \rho_s\ p \\
\quad\quad\quad\quad\quad\quad \texttt{else}\ \mathit{success}\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\
\quad\quad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho)\ \mathit{success}'\ \mathit{fail}\ e \\
\quad \texttt{else} \\
\quad\quad \mathit{fail}\ \rho
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{T}[\![t.\emptyset]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \tau[\![t]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e \\
\mathcal{T}[\![t.t'.T]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{let}\ \mathit{fail}' = \lambda\rho_f.\mathcal{T}[\![t'.T]\!]\ \theta\ \rho_f\ \mathit{success}\ \mathit{fail}\ e\ \texttt{in} \\
\quad \tau[\![t]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}'\ e
\end{array}
$$

- Destinations: final states p or intermediate junction j

$$
\begin{array}{l}
\mathcal{D}[\![p]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \mathit{success}\ \rho\ p \\
\mathcal{D}[\![j]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \theta^{\mathrm{j}}(j)\ \rho\ \mathit{success}\ \mathit{fail}\ e
\end{array}
$$

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)

$$
\begin{array}{l}
\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\
\quad\quad \texttt{let}\ \mathit{success}' = \\
\quad\quad\quad \lambda\rho_s.\lambda p.\texttt{if}\ p = []\ \texttt{then}\ \mathit{success}\ \rho_s\ p \\
\quad\quad\quad\quad\quad\quad \texttt{else}\ \mathit{success}\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\
\quad\quad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho)\ \mathit{success}'\ \mathit{fail}\ e \\
\quad \texttt{else} \\
\quad \mathit{fail}\ \rho
\end{array}
$$

Problems with Hamon's semantics

- transition actions executed in reverse order
  $(c_1, t_1) \to (c_2, t_2)$ should evaluate to (c_1, c_2, t_1, t_2)

$$
\begin{array}{l}
\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\
\quad\quad \texttt{let}\ \mathit{success}' = \\
\quad\quad\quad \lambda\rho_s.\lambda p.\texttt{if}\ p = []\ \texttt{then}\ \mathit{success}\ \rho_s\ p \\
\quad\quad\quad\quad\quad\quad \texttt{else}\ \mathit{success}\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\
\quad\quad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho)\ \mathit{success}'\ \mathit{fail}\ e \\
\quad \texttt{else} \\
\quad \mathit{fail}\ \rho
\end{array}
$$

- 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)

$$
\begin{array}{l}
\mathcal{T}[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ \mathit{success}\ \mathit{fail}\ e = \\
\quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\
\quad\quad \texttt{let}\ \mathit{success}' = \\
\quad\quad\quad \lambda\rho_s.\lambda p.\texttt{if}\ p = []\ \texttt{then}\ \mathit{success}\ \rho_s\ p \\
\quad\quad\quad\quad\quad\quad \texttt{else}\ \mathit{success}\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\
\quad\quad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho)\ \mathit{success}'\ \mathit{fail}\ e \\
\quad \texttt{else} \\
\quad \mathit{fail}\ \rho
\end{array}
$$

- 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

$$
\begin{array}{l}
\mathcal{C}[\![Or(T, S)]\!]^x\ \theta\ \rho\ e = \\
\quad \texttt{fold}\ (\lambda p.\lambda\rho.\ \text{if}\ \rho(p)\ \texttt{then}\ \theta^x(p)\ p\ e\ \texttt{else}\ \rho)\ S\ \rho
\end{array}
$$

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 ρ:
  - + 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 Plotkins[1] for λ-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:

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

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

---

[1] Gordon D. Plotkin. "Call-by-Name, Call-by-Value and the lambda-Calculus". In: *Theor. Comput. Sci.* 1.2 (1975), pp. 125–159.

[2] Julia L. Lawall and Olivier Danvy. "Separating Stages in the Continuation-Passing Style Transformation". In: *POPL'93*.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 2006. ISBN: 978-0-521-03311-4.

CPS semantics: Basics

- Continuations denote wrapping/success/failure:

$$\begin{aligned} w &: \mathit{path} \rightarrow \mathit{Den} \rightarrow \mathit{Den} \\ k+ &: \mathit{Den} \\ k- &: \mathit{Den} \end{aligned}$$

- Primitive operators:

$$\begin{aligned} \mathcal{A}[\![.]\!] &: \mathit{action} \rightarrow \mathit{KEnv} \rightarrow \mathit{Den} \\ \mathcal{I}te[\![.]\!] &: \mathit{condition} \rightarrow \mathit{KEnv} \rightarrow \mathit{Den} \rightarrow \mathit{Den} \rightarrow \mathit{Den} \\ \gg &: \mathit{Den} \rightarrow \mathit{Den} \rightarrow \mathit{Den} \\ \mathcal{I}d &: \mathit{Den} \end{aligned}$$

- Predefined actions/conditions:

$$\mathit{open}\ p, \quad \mathit{close}\ p, \quad \mathit{active}\ p$$

- Loose (L) or strict (S) mode
- Outer (**o**), inner (**i**) or entering (**e**) mode

CPS semantics: Transitions

- Transitions:

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

$$\begin{array}{l}
\mathcal{I}te(event(e_t) \wedge c, \\
\quad (\texttt{let}\ \ success' = success \gg (\mathcal{A}[\![a_t]\!])\ \ \texttt{in} \\
\quad\ (\mathcal{A}[\![a_c]\!]) \gg (\mathcal{D}[\![d]\!]\ \ \theta\ \ wrapper\ \ success'\ \ fail\ \ fail^{glob})), \\
\quad fail)
\end{array}$$

CPS semantics: Transitions

- Transitions:

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

$$\begin{array}{l} \mathcal{I}te(event(e_t) \wedge c, \\ \quad (\texttt{let}\ \ success' = success \gg (\mathcal{A}[\![a_t]\!])\ \ \texttt{in} \\ \quad\ (\mathcal{A}[\![a_c]\!]) \gg (\mathcal{D}[\![d]\!]\ \ \theta\ \ wrapper\ \ success'\ \ fail\ \ fail^{glob})), \\ \quad fail) \end{array}$$

- Lists of Transitions:

$$\mathcal{T}[\![t.\emptyset]\!]\ \ \theta\ \ wrapper\ \ success\ \ fail\ \ fail^{glob} = \tau[\![t]\!]\ \ \theta\ \ wrapper\ \ success\ \ fail\ \ fail^{glob}$$

$$\begin{array}{l} \mathcal{T}[\![t.T]\!]\ \ \theta\ \ wrapper\ \ success\ \ fail\ \ fail^{glob} = \\ \quad \texttt{let}\ \ fail' = \mathcal{T}[\![T]\!]\ \ \theta\ \ wrapper\ \ success\ \ fail\ \ fail^{glob}\ \ \texttt{in} \\ \quad \tau[\![t]\!]\ \ \theta\ \ wrapper\ \ success\ \ fail'\ \ fail^{glob} \end{array}$$

- Destinations:

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

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

CPS semantics: States

- Entering/exiting states (loosely or strictly):

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

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

- Computing states reactions:

$$\mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^{d}\ \theta : Den =$$

$\quad \texttt{let}\ wrapper_i = \texttt{open\_path}^{i}\ \emptyset\ p\ \texttt{in}$
$\quad \texttt{let}\ wrapper_o = \texttt{open\_path}^{o}\ \emptyset\ p\ \texttt{in}$
$\quad \texttt{let}\ fail_o =$
$\qquad \texttt{let}\ fail_i = \mathcal{C}[\![C]\!]^{d}\ p\ \theta\ \texttt{in}$
$\qquad\quad (\mathcal{A}[\![a_d]\!]\ \theta) \gg (\mathcal{T}[\![T_i]\!]\ \theta\ wrapper_i\ \mathcal{I}d\ fail_i\ fail_i)\ \texttt{in}$
$\quad \mathcal{T}[\![T_o]\!]\ \theta\ wrapper_o\ \mathcal{I}d\ fail_o\ fail_o$

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

Instanciating the CPS encoding

CPS framework fully parametric:

- Types for denotation/continuation: what do we want to build/manipulate?
- Defintition of primitive operators on the continuations:
  - `open` p, `close` p
  - Assignment: $v = expr$
  - Ite construct: $\mathcal{I}te(cond, T, E)$:
  - Composition $\gg$

Instanciations:

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

Instantiations: Interpreter

- Denotation type: $Den = Env \to Env$
- Rules:

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

Instantiations: Code Generator

- Denotation type:

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

- Rules:

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

Code Generated from Stopwatch Example

```
principal =
if Active(main)
then
  <CallD(main)>
else
  <Open(main)>;
  <Open(main.stop)>;
  <Open(main.stop.reset)>
endif
```

Code Generated from Stopwatch Example

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

Instantiation: Lustre Code Generator 1/2

- Lustre is a dataflow language with notions of automata

  $\Rightarrow$ *core language of our CocoSim toolchain*

- Denotation type: $Den = Name \to Name \to LustreAST$
- Rules:

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

Figure: Lustre instantiation

Instantiation: Lustre Code Generator 2/2

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

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

Figure: Lustre instantiation

Instantiation: Lustre Code Generator 2/2

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

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

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

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

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:

- Subtitute 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 ?