Semantics of Statecharts

Michael Whalen

Program Director
University of Minnesota Software Engineering Center
Statecharts

- Popular notation for implementing complex state machines
- Proposed by Harel in 1987
- Statecharts = state diagrams + depth (hierarchy) + orthogonality (parallelism) broadcast-communication
Statecharts (my history)

- 1997-99: Worked on simulation and translation tools for the Requirements State Machine Language (RSML)
- 1999-2002: Developed the semantics of the Requirements State Machine Language without Events (RSML-e) – Masters thesis
- 2002-2004: Created Compiler for RSML-e to SIMP (fully-specified subset of C) and proved its correctness – PhD thesis
  - Vowed to quit working on Statecharts 😊
- 2007-2009: Created formal analysis and compiler tools for Simulink Stateflow & worked on formal semantics
- 2010 - ??? Working with NASA Ames & Vanderbilt U on parameterized analysis of Statecharts dialects
Statecharts Formalisms

- Classical Statecharts (STATEMATE)
- Rhapsody Statecharts
- UML Statecharts
- MATLAB Stateflow
- SyncCharts (ESTEREL)
- Requirements State Machine Language (RSML)
- ...about 100 other variants
What happens when event ‘e’ occurs?

Figure from: Michelle L. Crane and Juergen Dingel, UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal, Proceedings of MoDELS2005, Montego Bay, Jamaica, October, 2005
Analyzing Statecharts

- Statecharts are used to design embedded systems
  - Sometimes safety-critical embedded software
- Some dialects are underspecified
  - UML & Rhapsody: parallel evaluation, conflicting active transitions, event ordering all underspecified
- Large projects use multiple dialects
  - NASA Constellation project: Rhapsody, UML, and Stateflow
  - Engineers familiar with different dialects read same diagram differently!
- Want to determine: when are charts “safe”?
  - Within a dialect
  - Across dialects
Syntax of Statecharts

- **States:**
  - AND (parallel)
  - OR (hierarchical)

- **Transitions**
  - Event-triggered
  - Conditional

- **Events**
  - “Basic”
  - Valued

- **Non-graphical variables**
Syntax of Statecharts

- Transition labels of the form: Event [condition] / action
  - Possible to omit one or more components
- Boundary Crossing Transitions

![Statechart Diagram]
Syntax of Statecharts

- Fork and Join: mechanisms for simplifying complex or redundant transitions

Syntax of Statecharts

- History Junctions
  - Allow restoration of child state
  - Can either be “single level” or transitive

Semantics Sketch

- All semantics have a notion of a step
  - An external event causes a chart to evaluate
    - Event can be implicit (time tick)
    - External event becomes initial active event

- Transitions are evaluated
  - Transition is enabled if
    - The source state of the transition is occupied
    - The triggering event of the transition (if any) matches an active event
    - The condition on the transition (if any) evaluates to true
Semantics Sketch

- Subset of enabled transitions *fire*
  - Change from source to destination state
  - May generate actions including additional events
    - Semantics of event propagation differ between Statecharts dialects

- Step evaluation is completed when all events have been processed
Example

1. PowerOn
   FAN1
   On [temp < 120]
   Off [temp >= 120]

2. FAN2
   On [temp < 150]
   Off [temp >= 150]

3. SpeedValue
   du: airflow = in(FAN1.On) + in(FAN2.On);

4. PowerOff
   on:
   airflow = 0;

5. SWITCH

6. SWITCH
Parallel State Evaluation

UML, Rhapsody, STATEMATE: No order specified by semantics; semantics are tool dependent in case of conflicts

Stateflow: deterministic user-specified sequential order

SyncCharts: semantics-determined partial order. Variables cannot be shared between parallel state machines, so this model would be rejected
Simultaneously Enabled Transitions

- Some dialects do not define an ordering on transitions at a particular level of hierarchy

UML, Rhapsody, STATEMATE: No order specified by semantics; semantics are tool dependent in case of conflicts

Stateflow, SyncCharts: deterministic user-specified sequential order
DISCREPANCIES BETWEEN DIALECTS
Event Processing

- Most Statecharts semantics split step into *microsteps*
  - Each microstep handles one round of event processing
  - If current round generates new events via transition actions, re-run chart until no further events are generated

- Stateflow uses *function call* semantics
  - Event action interrupts current chart processing and re-runs chart on generated event
Simultaneous Events

- Can multiple simultaneous events occur?

Classical, SyncCharts: multiple events to be “true” at the same instant

UML / Rhapsody: queue up events (in arbitrary order) and execute one at a time.

Stateflow: only one event due to function call semantics

Figure from: Michelle L. Crane and Juergen Dingel, UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal, Proceedings of MoDELS2005, Montego Bay, Jamaica, October, 2005
Transition Ordering

- How does semantics choose between simultaneously enabled transitions?

**Classical:** transitions with highest scope have highest priority. Scope is largest state that contains portion of transition arc (go to state E).

**UML / Rhapsody:** transitions in smallest substate have priority (go to state C).

**Stateflow, SyncCharts:** evaluation is “top down” based on transition source (go to state D).
Execution of Actions

**Classical:** Assignment actions within a microstep are considered simultaneous. Transition result: \( f = 2, g = 2 \)

**UML, Rhapsody, Stateflow, Esterel:** Assignment actions are sequential. Transition result: \( f = 2, g = 3 \).
UNINTENDED BEHAVIORS
Infinite Loops (Rhapsody)

- Example of GEN leading to infinite loop
- C1 queues message for C2 which queues message for C1 which ...
Bad Return Policies (Rhapsody)

- Trigger Example
- Rhapsody policy: triggered messages received while evaluating a message are dropped.
- So, no infinite loop here.
- Triggers can return values.
  - If trigger is dropped, return value is not defined by Rhapsody semantics.
Strange Charts (Stateflow)

- **Early Return Logic**
  - State On is active.
  - Off/ entry: entOff() during: durOff() exit: exitOff()

- **Infinite Event Loop**
  - On/ entry: entOn() during: durOn() exit: exitOn()
  - E_one { E_one }

- **Infinite Junction Loops**
  - A
  - / x = 1
  - [x < 10]
  - B
  - / x = x + 1

  - C
  - E {F;}
  - B

  - D
  - F
  - C

Multiple Entries (SyncCharts)

SyncCharts adds ‘strong abort’ vs. ‘weak abort’ transitions
Also ‘immediate’ vs ‘delayed’ transitions
Valued signals can be combined using commutative operator

After eval starting in s1:

v = 11,550

1. Start in s1
2. Queue transition ‘c’ (weak abort)
3. Take transition ‘a’ (v=3)
4. Take immediate transition b to s2 (v=15)
5. Take transition ‘c’ (v=105)
6. Re-enter s1 (v=210)
7. Take immediate transition b (v=1050)
8. Take immediate transition d (v=11,550) to state s3

Figure from: Charles Andre, Computing SyncCharts Reactions, *Electronic Notes in Computer Science Volume 88* (2003)
ANALYZING STATECHARTS
Stateflow Semantics

- Stateflow User Manual is 1400 pages
- Transition semantics alone is 7 pages of pseudocode
- Two attempts at formalization
  - Gregoire Hamon
    - Operational Semantics [SRI 2003]
      - Large; incomplete
    - Denotational Semantics [Chalmers 2006]
      - Based on continuations
      - Elegant, relatively complete, slightly incorrect
      - Worked with Gregoire to correct errors, complete definition
Stateflow Semantics

- Denotational semantics distills 1400 page manual into 1 ½ pages of formalism
- In: Gregoire Hamon. A Denotational Semantics of Stateflow, EMSOFT 2006
- Handful of errors in EMSOFT paper w.r.t. to boundary crossing transitions, transition actions, flowcharts
  - I fixed these and added support for a few remaining issues: history states and early return logic
  - Gregoire and I need to submit this for publication!
Stateflow Semantics

Syntax:

\[
\begin{align*}
Program \ P & ::= (s, [src_0, \ldots, src_n], I, O, L, K) \\
SrcComp \ src & ::= p : sd | j : T | f : fd \\
StateDef \ sd & ::= ((a_e, a_d, a_x), (L, K), T, C) \\
FunctionDef \ fd & ::= ((I, L), T) \\
Comp \ C & ::= Or (T, [s_0, \ldots, s_n]) | And ([s_0, \ldots, s_n]) \\
Trans \ t & ::= (e, c, (a_e, a_i), d) \\
Dest \ d & ::= p | p.j \\
TransLst \ T & ::= \emptyset | t.T \\
Path \ p & ::= \emptyset | s.p
\end{align*}
\]

Environments:

\[
\begin{align*}
Env \ \rho & ::= (I, O, K, S, V, (SI,SL,SO).L) \\
Kenv \ \theta & ::= \\
& \{ p_0 : (S[[p_0 : sd_0]]e \ \theta, S[[p_0 : sd_0]]d \ \theta, S[[p_0 : sd_0]]x \ \theta), \\
& \ldots \\
& p_n : (S[[p_n : sd_n]]e \ \theta, S[[p_n : sd_n]]d \ \theta, S[[p_n : sd_n]]x \ \theta), \\
& p_0.j_0 : T[[T_0]] \ \theta p_0, \ldots, p_k.j_k : T[[T_k]] \ \theta p_k \}
\end{align*}
\]
Stateflow Transition Semantics

\[ \tau : \text{trans} \rightarrow \text{kenv} \rightarrow \text{env} \rightarrow \text{path list} \rightarrow \text{k-} \rightarrow \text{k+} \rightarrow \text{k-} \rightarrow \text{event} \rightarrow \text{env} \]

\[ \tau \left[ \left[ \left( \text{et}, \text{c}, \left( \text{ac}, \text{at} \right), \text{d} \right) \right] \right] \theta \rho \ P \text{transact complete fail e} = \]

\[
\text{if } (\text{et} = \text{e}) \wedge (B[[\text{c}]] \rho) \text{ then}
\]

\[
\text{let } \text{transact'} = \lambda \rho_r \ . \text{transact} (A[[a_i]] \theta \rho_r) \text{ in}
\]

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

\[
\text{else}
\]

\[
\text{fail } \rho
\]

T: TransList \rightarrow KEnv \rightarrow env \rightarrow path list \rightarrow k- \rightarrow k+ \rightarrow k- \rightarrow event \rightarrow env

\[ T \left[ \left[ \emptyset \right] \right] \theta \rho \ P \text{transact complete fail e} = \text{complete } \rho \ [\ ] \]

\[ T \left[ \left[ \text{t.} \emptyset \right] \right] \theta \rho \ P \text{transact complete fail e} = \tau \left[ \left[ \text{t} \right] \right] \theta P \rho \text{transact complete fail e} \]

\[ T \left[ \left[ \text{t.t'.T} \right] \right] \theta \rho \ P \text{transact complete fail e} = \]

\[
\text{let } \text{fail'} = \lambda \rho_f . T \left[ \left[ \text{t'.T} \right] \right] \theta \rho_f P \text{transact complete fail e in}
\]

\[
\tau \left[ \left[ \text{t} \right] \right] \theta \rho \ P \text{transact complete fail’ e}
\]
Stateflow Destination / State Semantics

\(D:\) Destination \(\rightarrow KEnv \rightarrow env \rightarrow path\ list \rightarrow k- \rightarrow k+ \rightarrow k- \rightarrow event \rightarrow env\)

\(S:\) StateDef \(\rightarrow KEnv \rightarrow env \rightarrow P \rightarrow event \rightarrow env\)

open_path: \(KEnv \rightarrow env \rightarrow path\ list \rightarrow k- \rightarrow k- \rightarrow event \rightarrow env\)

\[D[[p]] \theta \rho P \ transact\ complete\ fail\ e = success\ transact\ \rho P.p\]
\[D[[v]] \theta \rho P \ transact\ complete\ fail\ e = \theta(j) P.p \rho\ transact\ complete\ fail\ e\]

\[S[[p : ((a_e, a_d, a_x), T, C)]_e \theta \rho P e = C[[C]]_e \theta (A[[a_c]] \theta (open \rho p)) P e\]

\[S[[p : ((a_e, a_d, a_x), T, C)]_d \theta \rho e =\]

\[
\begin{align*}
&\text{let} \\
&\quad \text{during} = \lambda \rho_d . (A[[a_d]] \theta \rho_d) \\
&\quad \text{fail} = \lambda \rho_f . C[[C]]_d \theta (\text{during} \rho_f) e \\
&\quad \text{complete} = \lambda \rho_c . \lambda p_c . \lambda t_c . \text{open_path} \theta \rho_c p_c t_c \text{ during fail e} \\
&\quad \text{transact} = \text{id} (* \text{identity function} *) \\
&\text{in} \\
&\quad T [[T]] \theta \rho \ transact\ complete\ fail\ e \\
&\text{end} \\
S[[p : ((a_e, a_d, a_x), T,C)]_x \theta \rho P e = close p \circ A[[a_x]] \theta \circ C[[C]]_x \theta \rho P e
\]
Implementation in Gryphon Tool Family

UMN: simulator, fault seeder, coverage measurement tool, TCG
RCI: Information Flow Modeling

Model Checkers: NuSMV, Prover, BAT, Kind, SAL

Theorem Provers: ACL2, PVS

Programming Languages: SPARK (Ada), C

Rockwell Collins/U of Minnesota
Esterel Technologies
MathWorks
Reactive Systems


CerTA FCS Phase I

- Sponsored by AFRL
  - Wright Patterson VA Directorate
- Compare FM & Testing
  - Testing team & FM team
- Lockheed Martin UAV
  - Adaptive Flight Control System
  - Redundancy Management Logic
  - Modeled in Simulink
  - Translated to NuSMV model checker

<table>
<thead>
<tr>
<th>Subsystem/Blocks</th>
<th>Charts/Transitions/TT Cells</th>
<th>Reachable State Space</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triplex voter</td>
<td>10 / 96</td>
<td>3 / 35 / 198</td>
<td>6.0 \times 10^{13}</td>
</tr>
<tr>
<td>Failure processing</td>
<td>7 / 42</td>
<td>0 / 0 / 0</td>
<td>2.1 \times 10^{4}</td>
</tr>
<tr>
<td>Reset manager</td>
<td>6 / 31</td>
<td>2 / 26 / 0</td>
<td>1.32 \times 10^{11}</td>
</tr>
<tr>
<td>Totals</td>
<td>23 / 169</td>
<td>5 / 61 / 198</td>
<td>N/A</td>
</tr>
</tbody>
</table>

... for each of ten control surfaces

Phase I Results

<table>
<thead>
<tr>
<th>Effort (% total)</th>
<th>Errors Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>60%</td>
</tr>
<tr>
<td>Model-Checking</td>
<td>40%</td>
</tr>
</tbody>
</table>
Functional Analysis of Stateflow

CerTA FCS Phase II – Verification of Stateflow Flowcharts

- Stateflow Flowcharts
  - No explicit states
  - Stateflow junctions
  - Cyclic paths
  - Transitions modify local state variables
  - Imperative programming

- Solution
  - Extension to translator to support flowcharts
  - Require a parameter that specifies the maximum times any cycle will be executed
  - This bound becomes property to check
Analysis of RCI State Machine Notation

FCS 5000 Flight Control Mode Logic

Mode Controller A

Example Requirement
Mode A1 => Mode B1

Counterexample Found in Less than Two Minutes
Found 27 Errors

Mode Controller B

Converted to Simulink
Translated to NuSMV

6.8 x 10^{21} Reachable States
RCI Stateflow analysis

- Focused on *functional analysis*
  - Prove functional and safety requirements of mixed Simulink/Stateflow models
- Based on Stateflow: deterministic notation
- Autogenerated some “well-formedness” properties
  - State consistency
  - Absence of *early return logic*
  - Junction loop bounds
New Work with NASA Ames and Vanderbilt University

- Examining well-formedness properties
  - Consistency of evaluation
    - Parallel state machines
    - Multiple enabled transitions
  - Finiteness of intra-step event graph
  - Chart state consistency

- Preservation properties across dialects
  - Creation of parameterized semantics for multiple dialects
  - Equivalence
  - Preservation of functional properties
Parallel State Consistency

- Syntactic mechanisms check disjointness of parallel charts (SyncCharts)

Example chart rejected by SyncCharts because $x$ is assigned or tested by both parallel states S1 and S2.
Semantic Parallel State Consistency

- Attempt all interleavings for given state using incremental SAT solver
  - Create next-step transition relation in parts
    - Start from “leaf” parallel machines
  - Given current state, show equivalence of parallel machines for current step
    - If we can’t show equivalence, flag an error
    - A little bit like partial order reduction
  - Choose arbitrary interleaving and compose up to next level

\[
\text{Predicate for Initial State} \land \text{Predicate for Previous Execution Steps} \implies (AB \Leftrightarrow BA)
\]
Conclusions

- Each of the examined semantics has quirks
- Be wary of assuming a particular semantics just given the visual notation
  - Bigger problem for groups that use more than one dialect (e.g. NASA) in same system
- Formal analysis is very helpful for finding latent bugs in charts
- Working on parameterized semantics for multiple dialects (derived from Hamon’s, Atlee’s work)
- Starting to explore analysis over multiple dialects
BACKUP SLIDES
Unintended Orderings (Stateflow)

- Order of evaluation of parallel charts
A little history

- Chart determinism
  - Mats Heimdahl: Completeness and Consistency of RSML [1993-1996]
    - conservative
    - Not sound in the presence of multiple simultaneous events

- Functional properties
  - William Chan: Model Checking Large Software Specifications [1996-99]
Example Chart

Stop

Reset

LAP {
  cent=0; sec=0; min=0;
  disp_cent=0; disp_sec=0;
  disp_min=0;
}

Lap_stop

LAP

START

LAP

START

START

Run

START

Running during:
  disp_cent=cent;
  disp_sec=sec;
  disp_min=min;

Lap

TIC {
  cent=cent+1;
}

[cent==100] {
  cent=0;
  sec=sec+1;
}

[sec==60] {
  sec=0;
  min=min+1;
}
Stateflow Semantic Formalization

- SRI – Operational semantics
  - Large, Complex
  - Several facets of the language not covered
- Gregoire Hamon [Mathworks] – Denotational semantics
  - Small
  - Relatively complete
  - Not quite right
- I’ve been working with Gregoire on completeness and corrections
Stateflow Semantics Problems

- Two different kinds of actions: *transition* actions and *condition* actions
  - Condition actions occur upon satisfying condition for a transition segment
  - Transition actions only occur when transition reaches an end state
- Possible to use flowcharts to create poorly structured programming language
Strange looking charts

For Loop Chart

A/
entry: entA()
during: durA()
exit: exitA()

State A is active.

E_one { i = 0 }

[i < 10] { i++; func1() }

B/
entry: entB()
during: durB()
exit: exitB()
Syntax of Statecharts

- Non-graphical variables
- Functions
  - UML: Calls to functions / methods defined in a class
  - Stateflow: *Graphical Functions*
Discrepancies: Fork and Join

- What happens when forks reference multiple events?

**Classical, SyncCharts:** multiple simultaneous events are possible, so the transitions have meaning.

**UML, Rhapsody, Stateflow:** only one event at-a-time due to queueing; the transition cannot fire.
Rhapsody Semantics

- Conditional connectors allow splitting transitions based on condition
  - If >1 condition is simultaneously true one is selected arbitrarily.
  - All guards are evaluated simultaneously prior to actions.
Rhapsody Semantics

- Statecharts embedded within classes
- Each chart is assigned a thread
  - Multiple charts can share a thread
  - Thread operates as “event dispatcher” to its objects
- Event communication has two forms
  - Asynchronous queueing: GEN method
    - Can queue to self
  - Synchronous invocation: TRIGGER method
    - Function call semantics
Problems with Rhapsody

- Several parts of semantics are unspecified (according to Harel06)
- Event queuing allows possible interleaving between “internal” and “external” events
- Ordering of evaluation on parallel state machines is undefined
Chart Transition Consistency

- **Local consistency:** can > 1 outgoing transition fire from a given state?
  - Necessary for determinism within UML, Rhapsody, STATEMATE dialects
  - Sufficient to show determinism when paired with parallel state consistency

- **Hierarchical consistency:** can > 1 outgoing transition fire from state hierarchy?
  - Necessary (but not sufficient) to show determinism between different dialects