Executable Specifications

- What are executable specifications?
- PAISLey
  - Design Principles
  - Examples
- StateCharts
  - Some basic constructs
  - More interesting constructs
  - Examples
- Interesting Features in PAISLey and StateCharts
  - Concurrency
  - Communication
  - Arrays
- Extended examples
- Evaluation of these tools

Where Should These Be Used?

Reactive or embedded systems.

Write requirements specifications:

1) to use them as vehicles for communication
2) change them
3) use them to constrain target systems
4) use them to accept or reject final products

PAISLey - Introduction

What is PAISLey?
- Process-oriented, Applicative, Interpretable Specification Language

Vision:
- Requirements must be understandable and modifiable
- Also, precise, unambiguous, internally consistent, and complete
- Also, minimal, as not to overconstrain the target system
- Finally, formally manipulable or testable.

Design Principles:

- Typed functional language - allows typechecking and having unspecified, uninterpreted functions, like

    `f[(g[y], h[z])]`

- Specs define a *successor function*, but each state "remembers" all its relevant history
- No unbounded recursion
- No loop constructs (only through function composition)
- Each portion of the system is specified as a *process*
- Symbols are used to indicate operations.

Example - CRT Terminal

```
terminal-cycle: DISPLAY --> DISPLAY;

terminal-cycle[d] =
  display[display-and-transact
             [(d, think-of-request[d])]];

think-of-request: DISPLAY -->REQUEST;

display-and-transact:
  DISPLAY X REQUEST -->
     DISPLAY X (RESPONSE U ERROR-MESSAGE);

display-and-transact[(d,r)] =
  display[(d,r)], transact[r]);

transact: REQUEST --> RESPONSE U ERROR-MESSAGE;

display:
  DISPLAY X (REQUEST U RESPONSE U ERROR-MESSAGE)
  --> DISPLAY.
```

"Arrays"

Can easily define several virtually the same objects:

- VECTOR = #1..10<X INTEGER >
  10-tuple of integers
- #1..3<func>[arg]
  func[func[func[arg]]]
- BIG-SET =J#1..3< ∪ LITTLE-SET-J>
  BIG-SET = LITTLE-SET-1 ∪ LITTLE-SET-2
  ∪ LITTLE-SET-3
- (k#0..9999<,terminal-k-cycle[blank-display]>,
  database-cycle[initial-database]
)
  initiates a system with 10000 terminal processes.

"Array" Example

```
k#0..9999

<;terminal-k-cycle: DISPLAY --> DISPLAY;

  terminal-k-cycle[d] =
    display
      [display-and-k-transact
        [(d, think-of-request[d])]];

  display-and-k-transact:
    DISPLAY X REQUEST
    --> DISPLAY X (RESPONSE U ERROR-MESSAGE);

  display-and-k-transact[(d,r)] =
    (display[(d,r)], k-transact[r]);

  k-transact: REQUEST --> RESPONSE U ERROR-MESSAGE;
>;

think-of-request: DISPLAY --> REQUEST;

display:
  DISPLAY X (REQUEST U RESPONSE U ERROR-MESSAGE)
  --> DISPLAY.
```


Communication between Processes

- Asynchronous interactions are specified using "exchange functions": two-way point-to-point mutually synchronized communication.
- Has one argument - a value to output, returns a value which was obtained as input (i.e., no difference between reads and writes!)
- Two processes exchange arguments
- Two attributes to be specified:
  - a type ("x", "xm", "xr")
  - a channel (Example: `xr-real-time`)
- Exchange happens only within one channel.

Communication between Processes (Cont'd)

- "x" - match with any pending exchange function on its channel or wait until there is one. If several possibilities - nondeterministically choose a match ("ensuring no starvation").
- "xm" - used to specify competition. Behave like "x"'s but cannot match with other "xm"'s and complete to match with exchange with other types.
- "xr" - specifies non-blocking communications. Will match with anything on its channel if there is anything, or else just terminate, returning its own argument as value.

Interactions Among Exchange Functions

Fig. 4 on p. 215 of Zave's 1991 paper

Example - Terminal Specification

```
transact[r] = receive-response[send-request[r]];
send-request: REQUEST --> FILLER;
send-request[r] = xm-requ[r];
receive-response: FILLER --> RESPONSE U ERROR-MESSAGE;
receive-response[null] = x-resp['null'];

database-cycle[d] =
    finalize-transaction[
       perform-transaction[(d,receive-request)]];
receive-request: --> REQUEST;
receive-request = x-requ['null'];
perform-transaction:
  DATABASE X REQUEST --> DATABASE X RESPONSE;
finalize-transaction:
  DATABASE X RESPONSE --> DATABASE;
finalize-transaction[(d,r)] =
  proj-2-1[(d,send-response[r])];
send-response: RESPONSE --> FILLER;
send-response[r] = x-resp[r].
```

Conventions:
- FILLER - an intrinsic set whose only element is the constant `'null'`.
- proj-2-1 - projects an ordered pair onto its first component

Small Examples

Real-time clock:

```
(clock-cycle[0],...);
clock-cycle: TIME --> TIME;
TIME = INTEGER;
clock-cycle[t] = proj-2-1[(increment[t],offer-time[t])];
increment: TIME --> TIME;
offer-time: TIME --> FILLER U TIME;
offer-time[t] = xr-time[t];

/* a process reading current time: */
current-time: --> TIME;
current-time = xm-time['null'];
```

Note that this is not broadcast, since there can be at most one reader!

Producer-Consumer buffer:

```
next-buffer: BUFFER --> BUFFER;
next-buffer[b] = give-to-consumer[get-from-producer[b]];
get-from-producer: BUFFER --> BUFFER;
get-from-producer[b] =
  /full[b]: b,
   'true': put-on-tail[(b,xr-prod['null'])]/;
give-to-consumer: BUFFER --> BUFFER;
  /empty[b]: b,
   'true': put-on-head[(xr-cons[first[b],rest[b]])]/.
```


How to Implement Communication?

If many "xm"'s and one "x" in different nodes in a network, then
- When "xm" is initialized, send a message carrying its argument to the node where matching "x" resides and block "xm" process.
- These messages are queued up in arrival order.
- Upon initiation of "x", wait until the queue is non-empty.
- Then remove the first entry as the "match", take the value stored as output, send a termination message containing "x"'s argument to the matching "xm" and continue.

Communication Implementation (Cont'd)

If one "xr" and either one"x" or many "xm" 's, then

- Queue is formed at the site of the "xr"
- If "xr" is initiated when the queue is empty, then it does not wait for messages.

If many "xr" 's and one "x", then

- Matching is done at the site of the "x", but no queue forms.
- If "xr" sends an initiating message but the "x" is not pending, a termination message with the original argument is sent back to "xr" immediately.

Toy Process-Control System (PCS)

```
(object-cycle[initial-object-state],
 sensor-reader-cycle[initial-sensor-reader-state],
 j#1..2< , monitor-j-cycle[initial-monitor-state] > );
initial-object-state: --> OBJECT-STATE;
object-cycle: OBJECT-STATE --> OBJECT-STATE;
object-cycle[state] =
       proj[(1, (simulate-object[state],
            offer-sensor-value[sensor[state]]))];
  simulate-object: OBJECT-STATE --> OBJECT-STATE;
  sensor: OBJECT-STATE --> SENSOR-VALUE;
  offer-sensor-value: SENSOR-VALUE --> FILLER;
  offer-sensor-value[value] = xr-sensor-value[value];
initial-sensor-reader-state: --> SENSOR-READER-STATE;
sensor-reader-cycle:
      SENSOR-READER-STATE --> SENSOR-READER-STATE;
sensor-reader-cycle[state] =
  proj[(1, (update-sensor-reader-state[(state,
                         get-sensor-value[Null])],
            offer-sensor-reader-state[state]))];
  get-sensor-value: FILLER --> SENSOR-VALUE;
  get-sensor-value[null] = x-sensor-value[null];
  update-sensor-reader-state:
     SENSOR-READER-STATE * SENSOR-VALUE
     --> SENSOR-READER-STATE;
  offer-sensor-reader-state:
     SENSOR-READER-STATE --> FILLER;
  offer-sensor-reader-state[state] =
     xr-sensor-reader-state[state];
```


PCS (Cont'd)

```
MONITOR-STATE = STATUS-INFO * CONTROL-PARAMETERS;
initial-monitor-state: --> MONITOR-STATE;
j#1..2
< ; monitor-j-cycle: MONITOR-STATE --> MONITOR-STATE;
    monitor-j-cycle[(status,control)] =
      (update-j-status-info[((status,control),
                       get-sensor-reader-state[Null])],
       install-new-control-if-any[(control
                       accept-j-control-parameters)]);
      update-j-status-info:
       MONITOR-STATE * SENSOR-READER-STATE
       --> MONITOR-STATE;
      accept-j-control-parameters: -->
         CONTROL-PARAMETERS | FILLER
>;
      get-sensor-reader-state:
         FILLER --> SENSOR-READER-STATE;
      get-sensor-reader-state[null] =
         xm-sensor-reader-state[null];
      install-new-control-if-any:
         CONTROL-PARAMETERS * (CONTROL-PARAMETERS | FILLER)
         --> CONTROL-PARAMETERS;
      install-new-control-if-any[(old,new)] =
         /is-null[new]:old, True:new/;
```


Data Flow in PCS

Fig. 3 on p. 215 of Zave's 1991 paper

Handling Performance Requirements

- Can specify reliability by giving lower bounds on a random variable.
- Can specify real-time constraints, like maximum, mean, or constant of a random variable.
- Three stages: input transmission, transaction processing, and output transmission. Failure at any stage aborts subsequent stages and propagates an appropriate error message.
- Specification is done in comments and used in simulation and analysis.

Handling Reliability

- Can be attached only to functions with ranges divided into two subsets
  `"--> SUCCESS-RESULT ∪ FAILURE-RESULT"`
- Failure - returns a value in its failure range.
- Exchange function - even when they fail, then match another exchange. If matching exchange functions both have reliability properties, they must succeed or fail together.
- This makes reliability independent of timing and functionality.
- Only primitive functions can *really* fail; others are evaluated by their definitions.
- Can be simulated by specification interpretor.
- Can be checked for internal consistency, i.e., if `f[x]` is `g[h[x]]`, then
  `u-bound(f) < u-bound(g) + u-bound(h)`

Examples

"Synchronous Closed-Loop" systems, like on-line database.
- A response-time limit of 3 s:
  `transact: ! -->"maximum = 3 sec."`
- An average load of 200 transactions/s, assuming 10,000 terminals in the system:
  `terminal-cycle: ! --> "mean = 50 sec"`
- At least 99 percent of all transactions must be processed successfully:
  `transact: % --> "prob{'success'} >=.99"`

"Asynchronous Closed-Loop" systems, like process control.

- Granularity of the discrete simulations of the machines in the environment
- A real-time limit on the fully-automatic feedback loop realized by the system
- A real-time limit on the partially manual feedback loop
- A derived performance requirement on the system's internal database component which guarantees that other systems meet their requirement.

All of these are specified as timing requirements in a single process!

Available Analysis Methods

Figure 5, p. 217 of Zave's 1991 paper

Design Principles of StateCharts

- State/Event approach
- An approach need to naturally cater for general statements, like
1) "in all airborne states, when yellow handle is pulled, the seat will be ejected" (i.e., need to cluster states into a superstate)
2) "gearbox change of state is independent of braking system" (i.e., need independence, or orthogonality)
3) "when selection button is pressed, enter selected mode" (i.e., need more general transitions than the single event-labeled arrows)
4) "display-mode consists of time-display, date-display and stopwatch-display" (i.e., need refinement of states)

Syntax

Events:

a	button a being depressed
$\hat{a}$	a being released
a in $c.b$	event a occurs while subsystem c is in state b
30 sec in b	been sitting in b for 30 seconds
battery inserted	obvious...

States:
- Hierarchies (substates)
- XOR states (can be only in one state)
- AND states (concurrent execution)
- History states

Actions and Activities

Action - a zero-time occurrence
Activity - a non-zero time occurrence, like beeping, displaying, or executing length computation (activity to action is same as condition to event)
Each activity X is associated with 2 actions and one condition:
- start(X)
- stop(X)
- active(X) - condition

Can be associated with transitions and/or states (Mealy/Moore machines):
- with transitions - action is taken when transition is taken
- with events - entry S; exit M, N; throughout X

Examples

And/OR States

Fig. 19, p. 242 of Harel

History

Fig. 11, p. 239 of Harel

Part of Citizen's Watch

Fig 13, p. 240 of Harel

And Another Part - Clearing History

Fig. 28, p. 249 of Harel

Array Examples in StateCharts

Easy

Fig. 29 on p. 259 Harel

More difficult

Fig. 40, p. 260 of Harel

Selection and Condition Examples

Condition:

Fig. 33, p. 253 of Harel

Selection - occurs when the state to be entered is determined based on outcome of a selection event.

Fig. 34, p. 254 of Harel

Performance Requirements

Specifying timing:
- `timeout(event, number)` represents an event occurring precisely when `number` of time units have elapsed from `event`
- Also has a graphic notion:

Fig. 25, p. 254 of Harel

Specifying reliability (or probability):
- Allow nondeterminism and specify a bias on the coin to be tossed when it arises.
- These are "Markov-charts"!

What is Good About StateCharts

- Visual, easy to use. Became a product heavily used in industry
- And... has formal semantics [Harel, Naamad, TOSEM'96]:
  - Analyzable for completeness and consistency [Leveson, Heimdahl, TSE'96]
  - Can be translated into other formalisms, where model-checking, preorders, bisumulations and other automatic verification strategies can be applied! [Uselton, Smolka]

What is Formal Semantics?

Finite-state machines DO NOT ALWAYS have formal semantics. Precise meaning is given to
- when the system changes state,
- when the event is triggered,
- what constitutes a "step" (i.e., do we wait until other states make changes based on the first change)
- how to deal with ambiguity and non-determinism

Example - STATEMATE semantics

1. Reactions to evternal and internal events, and changes that occur in a step, can be sensed only after completion of the step.
2. Events "live" for the duration of one step only, the one following that in which they occur, and are not "remembered" in subsequent steps.
3. Calculations in one step are based on the situations at the beginning of the step (e.g., which states the system was in, which activities were active, and the values of conditions and dataitems)
4. A maximal subset of nonconflicting transitions and static reactions is always executed)"greediness property" of the semantics.
5. The execution of a step takes zero time.

PAISley and StateCharts - What is Good?

- Both have a dataflow description and a finite-state machine
- Hierarchical development, ideas of refinement
- Ability to check if requirements correspond to user-envisioned scenarios (scenario-based requirements testing)
- Can be analyzed for type correctness
- Allow specification of real-time properties
- Were used to specify many applications:
  - PAISLey: its own GUI, a finite-element system with adaptive refinement, machine monitoring system, lift controller, secure message switch, network architecture prototype, submarine light guide system, RAPID-1 architecture
  - StateCharts: Citizens watch, Levi aircraft, TCAS II - Traffic Collision Avoidance System (in RSML)

PAISley vs. StateCharts

- PAISLey has a simpler semantics, no special cases.
- So (by authors of PAISley) "semantic complexity must have a negative impact on the ease with which a specification is constructed, understood, and validated"
- StateCharts has a wider range of applicability
- But "PAISley is a better choice because of its simplicity"

(From the editor - PAISley has always remained a one-person project, whereas StateCharts has been sold to a lot of people.)

food for slide eater

food for slide eater