MIDI and OSC Controller State Machines in ChucK

Electro-Music 2013

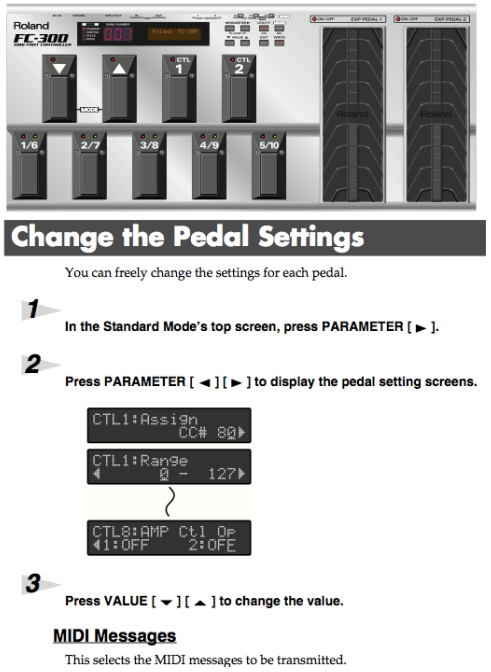
Dale E. Parson

Kutztown University of PA

3-4 PM, September 8, 2013

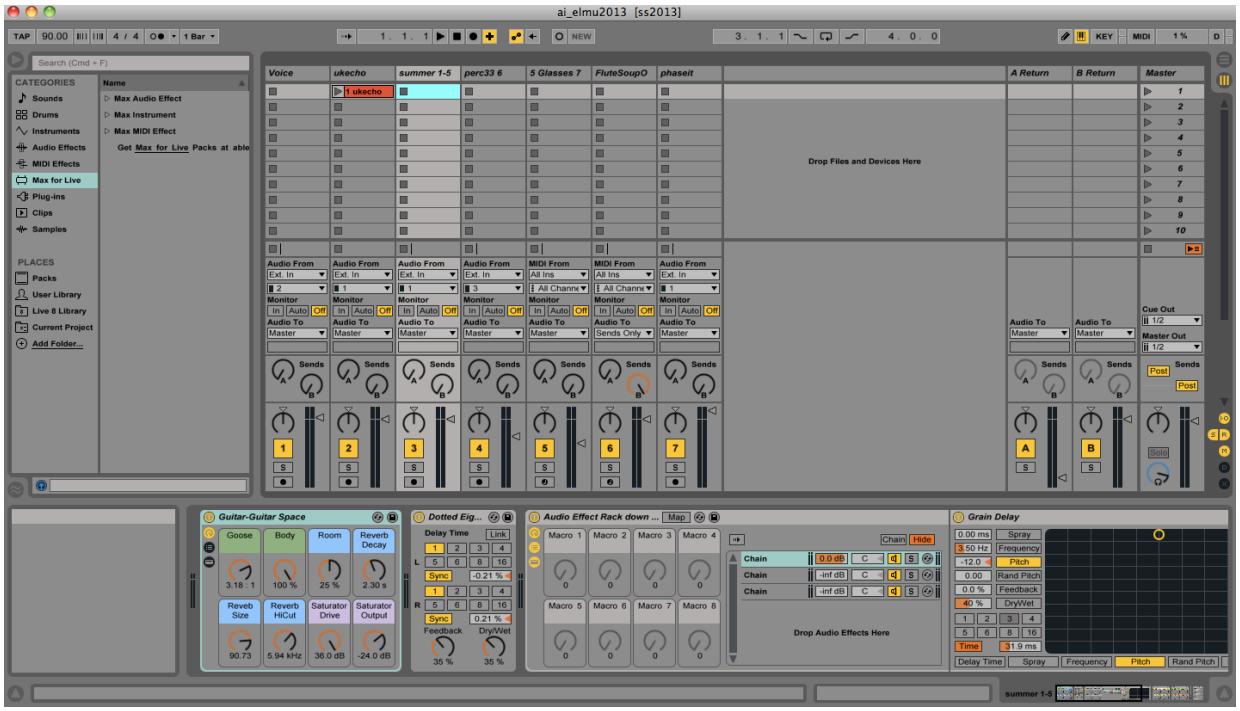
Hardware MIDI / OSC Controllers

- Configuring these things . . .



Software Synthesizer Systems

- . . . to control these things is a big pain!



Why is it a pain?

- Every hardware MIDI controller has a low-bandwidth button-to-LED user interface.
- There are no user interface standards.
    - You configure each controller differently.
- The mapping of MIDI CC messages from controllers to synths is often many-to-many.
    - Different button, sliders, etc. may affect the same synthesizer parameter.
    - A given button, etc. may affect more than 1 parameter.

One solution

- Some people like to use dedicated controllers.

Hands free controllers.

- I prefer to keep my hands on these.

Caution: Programming Ahead!

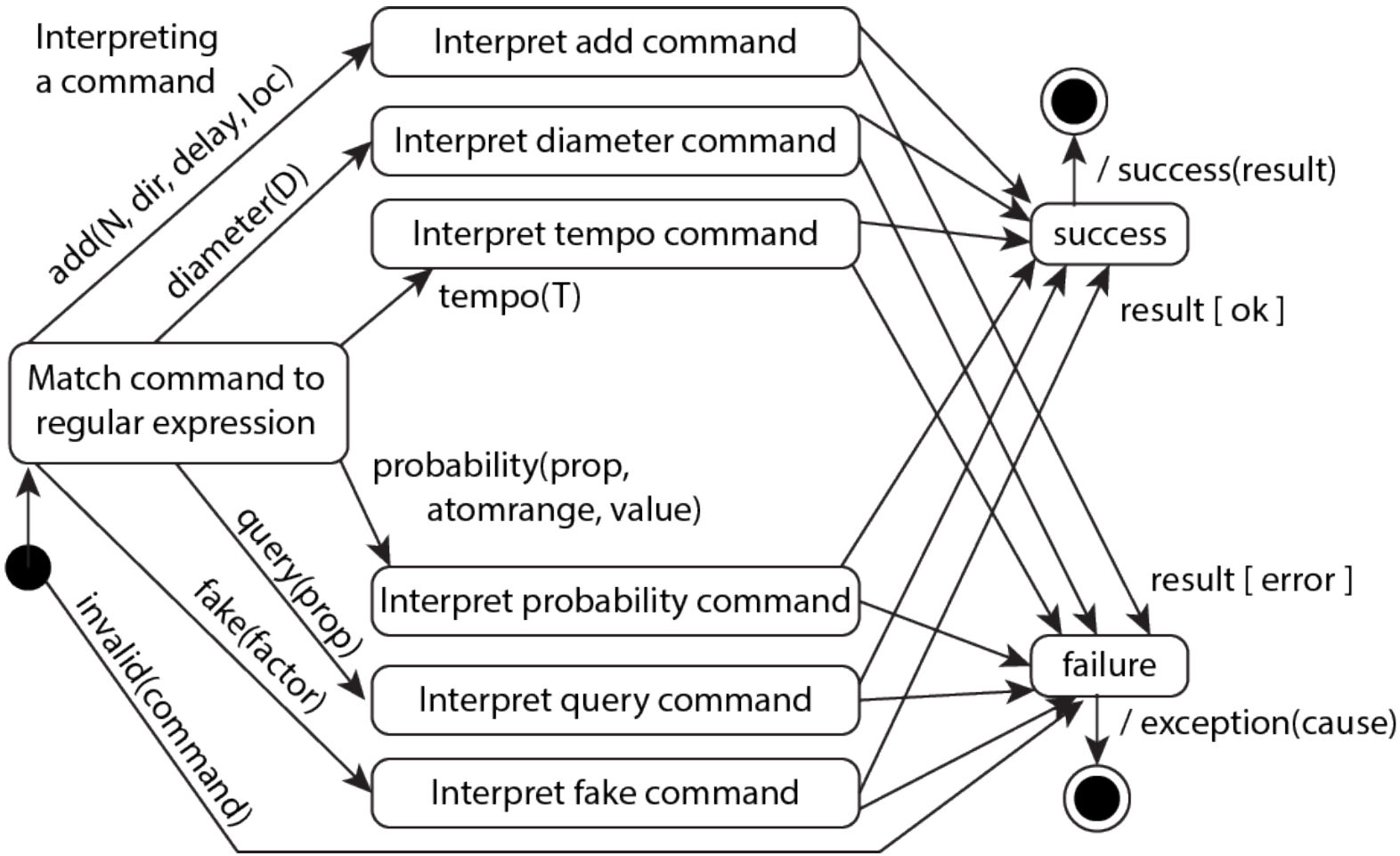
- I used to write ad hoc ChucK scripts to translate controller messages to messages that pertain to the composition.

```
while (true) {
  min => now ;
  while (min.recv(msg)) {
    if (msg.data1 == 176 && (msg.data2 == 1 || msg.data2 == 7 || msg.data2 == 95 || msg.data2 == 94)) {
      // 1 and 7 are the expression pedals, 95 and 94 are the latches.
      mout.send(msg);     // echo it to output
    } else if (msg.data1 == 192 && msg.data2 >= 0 && msg.data2 < 10) {
      // It is in range of the switches, translate to ctrlr 12.
      176 => msg.data1 ;      // chuck Controller into the field
      if (toggle[msg.data2] < 127) {  // switch is off, turn it on
        127 => msg.data3 ;
        127 => toggle[msg.data2] ;
      } else {                    // switch is on, turn it off
        0 => msg.data3 ;
        0 => toggle[msg.data2] ;
      }
      // translate the switch number to a controller number:
```

Ad hoc ChucK problems?

- Every composition duplicates a certain amount of work and ends up with somewhat irregular, composition-specific control structure.
- A better approach is to specify a State Machine, which is essentially a (software) sequencer that accepts controller input messages, emits composition output messages, and transitions to the next state in the composition.

UML State Machine Diagrams!



UML State Diagram Constructs

- *State* bubbles include normal states, a unique *start state*, and one or more *accept states*.
- *Transitions* between states S1 ➔ S2.
- An *event* triggers a transition.
  - Optional data *arguments* may accompany event arrival.
  - An optional, boolean *guard* expression determines whether to cross a transition when its event arrives.
  - An optional *activity* produces output actions and updates to state variables that are available to subsequent guards and activities.

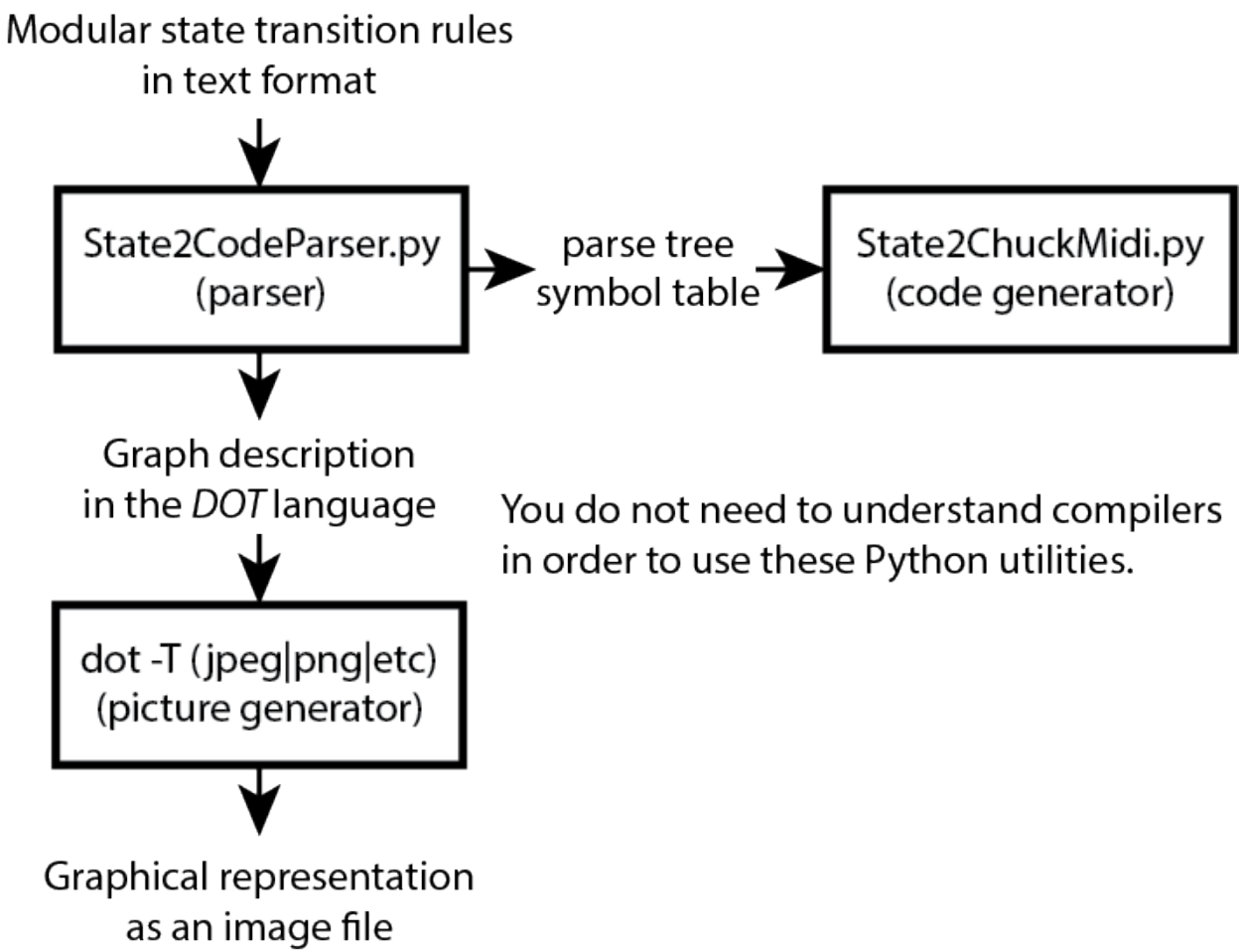
State Diagrams ➔ MIDI or OSC

- Arrival of a MIDI or OSC message from a controller constitutes an event.
  - Data fields are arguments.
  - Guards determine which messages to process.
  - Activities send outgoing messages to tools.
  - It is possible to automate much of the programming.
- Interconnected states control the sequencing of the composition / performance tools.
  - (State machines can enter sub-state machines.)

How to program a state machine?

- Open source tools tend to be over-coupled to the back end code (target machine) that they generate.
- I do not have time to create yet another graphical structure editor.
- State2CodeParser.py compiles simple state transition rules into a parse tree, symbol table & visualization file for *dot*, a *graphviz* visualization utility.
- State2ChuckMidi.py is a back end that takes output from State2CodeParser.py and produces ChucK code.

Compiling rules to state machines



You do not need to understand compilers in order to use these Python utilities.

State Transition Rules for my AI set

```
machine ai2013 {    start silent0, state theme1, state middle2, state
    himelody3,
   state highestmelody4, state doublemelody5, state chordal6,
   state theme7, accept lastnotes8 ;
   silent0 -> theme1 midi()
      [@msgin.data1==176 && msgin.data2==1 && msgin.data3==0@]
      / @1=>msgout.data2;emit(msgout);2=>msgout.data2;emit(msgout);
         6=>msgout.data2;emit(msgout);7=>msgout.data2;emit(msgout);@,
                              (More rules follow.)
```

Interpreting a rule as $state_i$-> $state_{i+1}$ event(args) [guard] / activity

The states and events are required, the arguments, guard and activity optional.

Emitted code for above

```
if (stateIndex == 0) {
    if (msgin.data1==176 && msgin.data2==1 && msgin.data3==0) {
        6=>msgout.data2;
        127=>msgout.data3;
        emit(msgout);
        7=>msgout.data2;
        emit(msgout);
        6 => stateIndex ;
    }
```

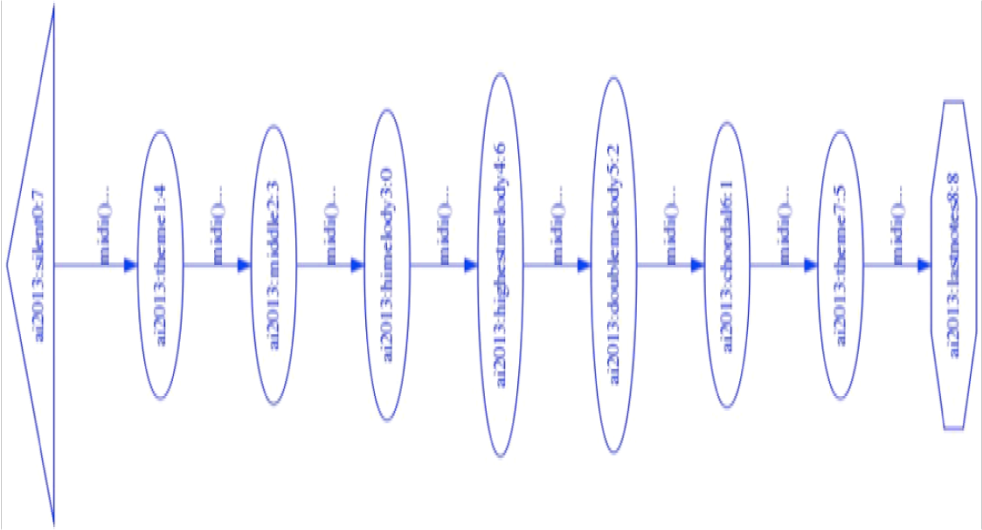
Compiling & running

- python State2CodeParser.py elmu2013_ai.txt elmu2013_ai.dot elmu2013_ai.ck State2ChuckMidi State2ChuckMidi
  - Compile state machine to ChucK and a dot graph.
- dot -Tjpeg elmu2013_ai.dot > elmu2013_ai.jpg
  - Compile the dot graph to a jpeg image.
- chuck elmu2013_ai.ck:3:1
  - Run the ChucK MIDI sequencer / translator code.

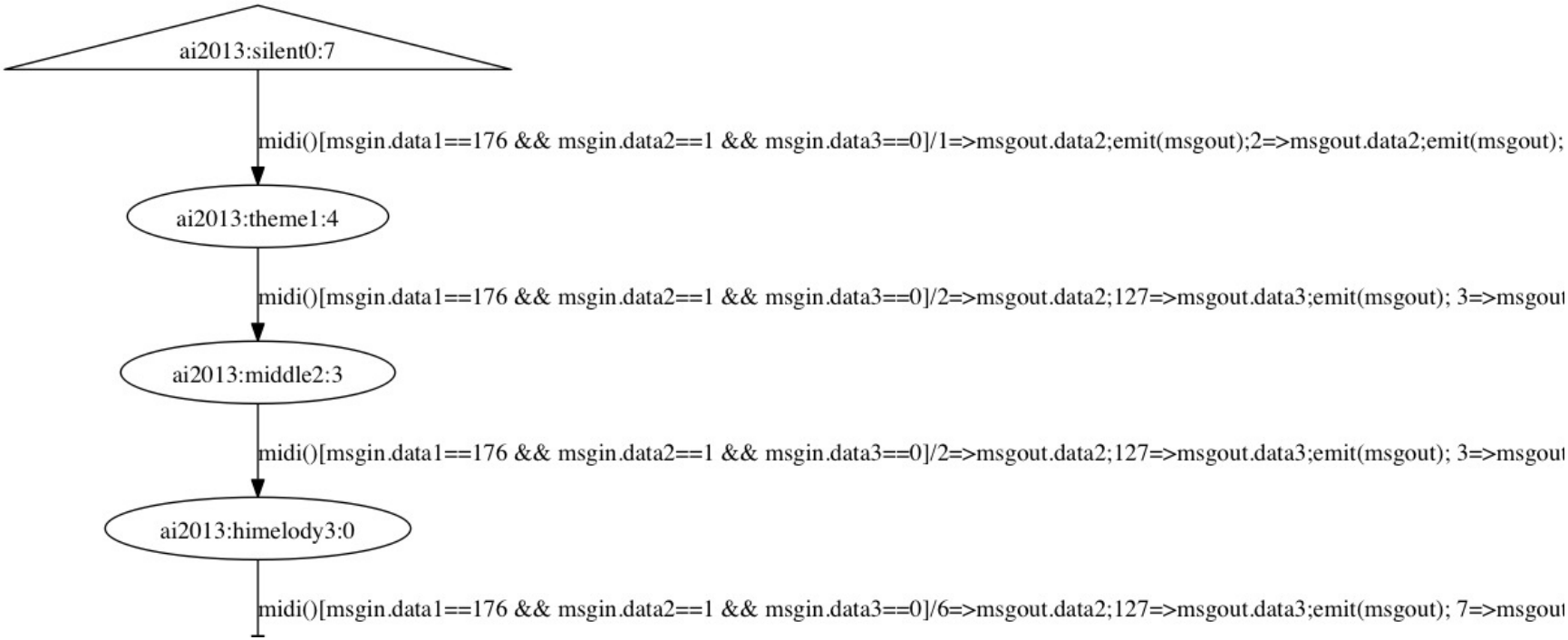
What are the advantages?

- You do not need to write an entire ChucK script for every composition.
- The boilerplate part of the ChucK script stays in a template file.
- Approach is readily portable to other event types (OSC), multiple ports, and performance languages (e.g., Supercollider, Java) by writing a new back end, no change to State2CodeParser.py.
- Approach generates a dot graph and prints out state transitions from ChucK during performance.

Dot graph for my 1:30 PM piece



Zoom in on first few states



Plans

- I am in the middle of designing a Python back end for my Operating Systems students to simulate aspects of an O.S. using state machines.
    - IT students should not have to write low-level code.
    - Code-savvy musicians are a similar population of users.
- For music I'd like to add OSC, then multiple I/O ports (more than one MIDI input or output device), then a Java back end. I'll add SC when I finally get time to learn it.

Conclusions

- I find this approach to organizing the CC messages based on the composition rather than on the controller device useful.
- It saves coding time in adapting boilerplate code.
- It generates useful dot graphs and messages.
- It adapts readily to other back end targets.
- It has uses beyond computer music.
  - Python 2.7.X http://www.python.org/getit/
  - Graphviz for dot visualization http://www.graphviz.org/
  - ChucK http://chuck.cs.princeton.edu/ (or SC or other)