State Notation Language Overview

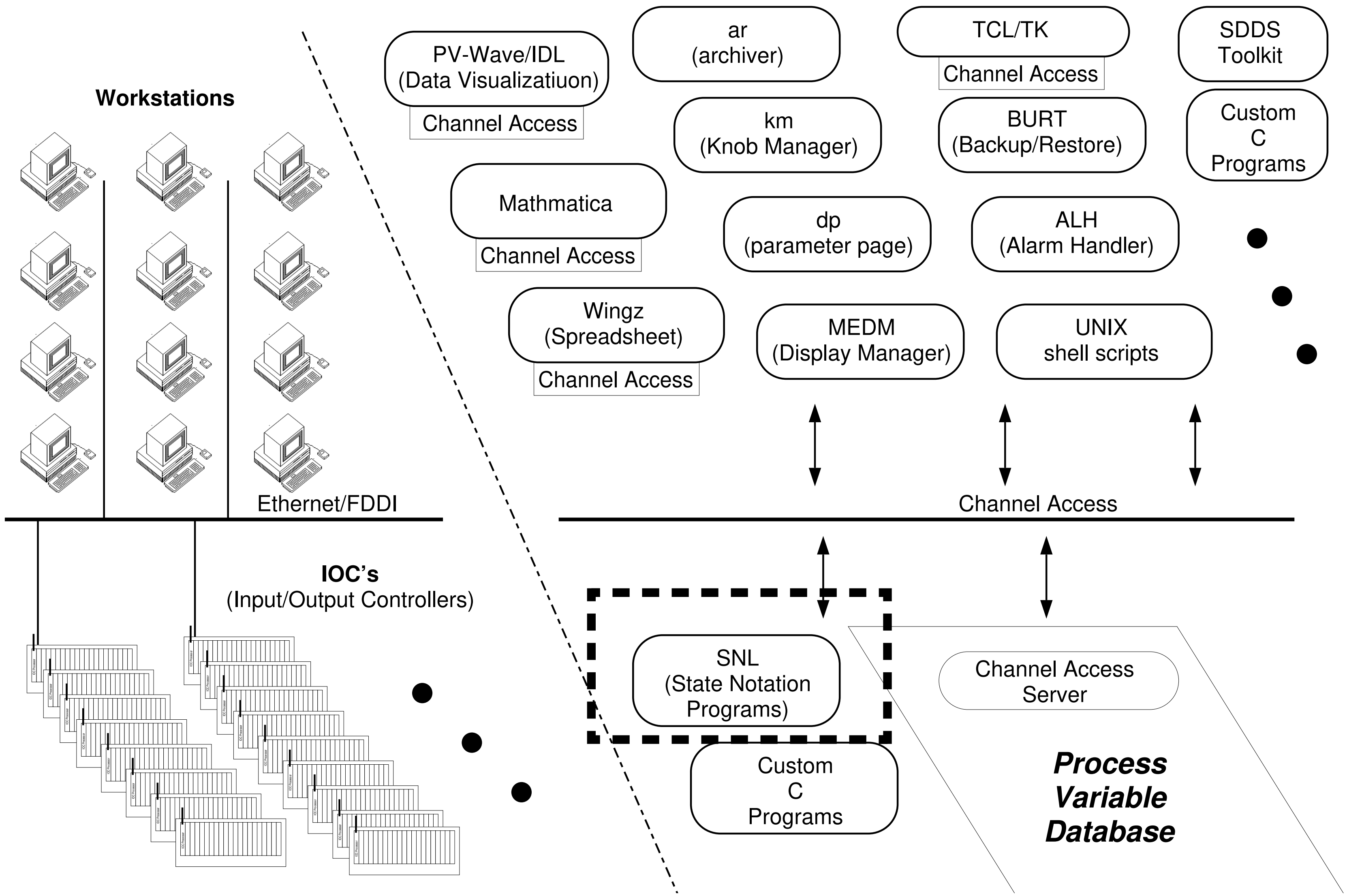
Ned Arnold

May 1996

State Notation Complier and Sequencer

- Allows programming of sequential “state-oriented” operations to run in the IOC
- The program interacts with the run-time database(s) via channel access
- Latest manual : http://www.atdiv.lanl.gov/doc/epics/sequencer/snl_1.9_man.html

EPICS Architecture and Tools



Uses

- State machines
- Startup sequences
- Enforce prudent operational procedures
- Watch for likely fault modes that are hard to detect via alarms
- Implement complex closed loop control schemes
- Coordinate control of multiple devices

Advantages

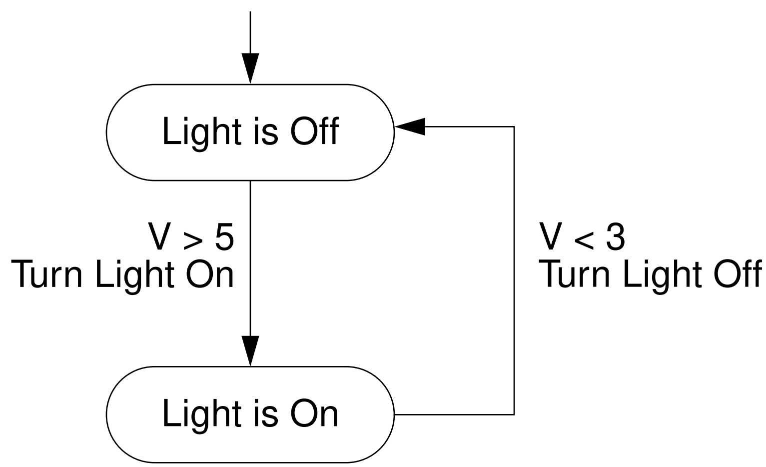
- Can implement complicated algorithms
- Can stop, reload, restart a sequence program without rebooting
- "Converse" with the operator through a string_in record
- C code can be embedded as part of the sequence
- All Channel Access "hassles" are taken care of for you
- File access can be implemented as part of the sequence

Definitions

- *SNL* : State Notation Language
- *SNC* : State Notation Compiler
- *sequencer* : The tool within the IOC that executes the compiled SNL code
- *Program* : A complete SNL application consisting of declarations and one or more state sets
- *State Set* : A set of states that make a complete state machine
- *State :* A particular “mode” of the program in which it remains until a transition statement is evaluated to be TRUE

Basics

- The SNL code structure follows a “state transition diagram” format




```
state light_off {
        when (v > 5.0) {
              light = TRUE;
              pvPut (light);
        } state light_on
}

state light_on {
        when (v < 3.0) {
              light = FALSE;
              pvPut(light);
        } state light_off
}
```

Basics (cont.)

- Once in a given state, “when” conditions are specified under which to leave that state
- Action statements are executed during the transition from one state to another
- Access to Process Variables via channel access is accomplished by simply “assigning” a PV to a sequence variable

A Complete State Program (with 2 state sets)

```
program level_check

float  v;
assign  v to "ts1:ai1";
monitor  v;
short  light;
assign  light to
"Indicator_light";

float  vout;
float  delta;
assign vout to "ts1:ai1";
```

```
ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = TRUE;
      pvPut(light);
    ) state light_on;
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = FALSE;
      pvPut (light);
    } state light_off;
  }
}
```

```
ss generate_voltage {
  state init {
    when () {
      vout = 0.0;
      pvPut(vout);
      delta = 0.2;
    } state ramp
  }

  state ramp {
    when (delay (0.1)) {
      if ((delta > 0.0 &&
         vout >= 11.0) ||
         (delta < 0.0 &&
         vout <= -11.0))
         /*change direction*/
         delta = -delta;
      vout += delta;
      pvPut(vout);
    } state ramp;
  }
}
```

More Basics ...

- A state can have multiple “when” statements. The first one to be evaluated to be TRUE will be executed. This allows “conditional branching” within the sequence program.
- When entering a state, all “when” statements are evaluated
- If no “when” is true, sequence program task “pends” until an event occurs
- It is possible to create a “loop” and consume all available CPU

Example of Multiple “whens”

```
state checks {

    when (interlockChasPwrBi == 0) {
        sprintf(seqMsg1, “Electron Gun Not Ready...”);
        pvPut(seqMsg1);
        sprintf(seqMesg2, “Gun Interlock Chassis off”);
        pvPut(seqMsg2);
    } state initialChecks

    when (gunLocal) {
        sprintf(seqMsg1, “Electron Gun Not Ready...”);
        pvPut(seqMsg1);
        sprintf(seqMesg2, “Egun in local control”);
        pvPut(seqMsg2);
    } state initialChecks

    when (gunInterlocksRdyCC == 0) {
        sprintf(seqMsg1, “Electron Gun Not Ready...”);
        pvPut(seqMsg1);
        sprintf(seqMesg2, “Interlocks not OK”);
        pvPut(seqMsg2);
    } state initialChecks
```

```
    when (interlockChasPwrBI && (gunLocal == 0) && gunInterlocksRdyCC) {
        gunAutoStart = 0;
        pvPut(gunAutoStart);
        gunAutoStop = 0;
        pvPut(gunAutoStop);
        sprintf(seqMsg1, “Push Auto-Start to begin autostart...”);
        pvPut(seqMsg1);
        sprintf(seqMesg2, “Push Auto-Stop to begin autostop...”);
        pvPut(seqMsg2);
%%      taskDelay(60);
    } state waitForRequest
}

state initialChecks {
    when (delay(2.0)) {
        sprintf(seqMsg1, “Initial Checks”);
        pvPut(SeqMsg1);
        sprintf(seqMesg2, “”);
        pvPut(seqMsg2);
%%      taskDelay(60);
    } state checks
}
```

Other Features

- Assignment of “macros” at program startup (multiple copies of same sequence)
- Arrays (each element can be assigned to a PV)
- Event Flags (used to sync state sets)
- Built-In functions (pg 18 of Manual)

Debugging

- seqShow
- seqChanShow
- printf(”Here I am in state xyz \n”);
- td xxxxxx
- ld < my_sequence_program.o
- seq &my_sequence_program.o

Examples

- Compensate for Attenuator Phase Shift when adjusted
- Automatically check timing of BPM's
- E-gun startup
- Modulator startup
- Automatic Test Sequences (240 Inputs to the MPS)
- BPM Trajectory Plot