Where do we go from here?

March 27	COSPAN
April 3	Statecharts and PAISLey
April 8	TTMs
April 10	Neil's presentation of PVS
April 17	Albert's presentation of HOL

Note that April 8 is an additional class. It is a Tuesday, 1-3 p.m. in LP378.

Lecture 10. Overview

- Idea of COSPAN
- S/R specifications
- Simple examples
- Concurrency example
  - View of concurrency
  - Pause states

Reading: Z. Har'El, R. Kurshan, "COSPAN User's Guide".

Overview of COSPAN

- Used to develop hardware or software
- Target system implementation is created through a formal top-down development procedure based upon successive refinement.
- Target implementation is automatically generated from the final level of definition.
- At each development level the system is symbolically analyzed for compliance with user-defined behavioral traits (*tasks*).
- Successive refinement procedure means that a task whose performance is proved at one developmental level is guaranteed performed at all successive levels.
- Programs are written in S/R - an automaton language
- Easy to specify control flow but not complex data structures - good for highly parallel hardware or communication protocols.

S/R Specification

- Each specification is a list of processes.
- Inputs to each process are outputs of several of the processes
- In order to "read" outputs of other processes, need to `import`.
- Each process has a set of *states*. Each state defines possible outputs or *selections*.
- Selections are non-deterministically chosen outputs of Mealy machine.
- *Transition predicates* describe how a process "resolves" a current "global" selection by changing state.
- Each process resolves the current global selection by moving to a state through a state transition whose associated predicate is enabled by the global selection.
- Semantics is independent of the order of declaration in S/R specification.

First Example

```
proc ALT  /* ALTERNATOR */
  selvar #:(off, on)
  stvar  $:(OFF, ON)
  init   OFF
  trans

  OFF        {off, on}
      ->ON  : # = on
      ->OFF : else;
  ON         {off, on}
      ->OFF : # = off
      ->ON  : else;
  end; /* ALT */
```


Some Syntax

- `init` - initial states of the process
- `trans` / `end` - transition relation
- `selvar` and `stvar` - process selections and states.
- # and $ typically indicate selection and state variables.
- Inside process, variables can be referenced by their local names, outside - ALT.#.
- The following are equivalent: (A: Lost_msg) and A.#=Lost_msg.
- + - addition or OR.
- * - multiplication or AND.
- Usually, process inputs consist only of selections, i.e., machine state is unobservable and the behavior is determined through machine outputs alone.

Notion of Scope

- `monitor` is a process which is allowed to use variables outside of its scope, like state variables of other processes.
- No `proc` or `monitor` may use a variable unless it is declared in an `imports` declaration. - `proc` can only use selection variables. `monitor` - selection and state variables.
- use `monitor` only for analysis rather than specification.
- scope of a processor consists of variables it imports from previous level, the selection variables of its children referenced by their extended path names, and its own variables.

Scope Example

```
x, y, z.. in the scope of this level

  proc A
    import x, y
    selvar w
    assgn  w := B.# + C.#

    proc B                  proc C
      import x, w             import y
      selvar #                selvar #
```

Scope of A has x, y, w and selection variables if its children.

Second Example

```
proc ALT
  selvar #:(0..1)
  stvar  $:(0..1)
  init   0
  trans

  true        {0,1}
      -># : #=($+1) mod 2
      ->$ : else;
  end /* ALT */

monitor TASK /* The state and selection of ALT */
             /* repeatedly coincide */
  import ALT
  stvar  $:(OK, NOK)
  recur  NOK->OK
  cyset  {OK}
  init   OK
  trans

  true  ->NOK : ALT.#<>ALT.$
        ->OK  : else;
  end /* TASK */
```


Specifying Intended Behavior

- In a special task (`TASK` in our example)
- `TASK` is not a process but an *automaton*, defining those sequences of (state, selection) "produced" by ALT which constitute performance of stated task.
- ALT is correct iff starting with some state, state = selection.
- `recur NOK -> OK` - TASK repeatedly moves from NOK to OK (*acceptance condition*)
- `cyset {OK}` - TASK stays forever within the set of states {OK} ({cycle set}).
- TASK is performed by ALT iff every sequence is accepted by some acceptance condition of TASK.

Checking Intended Behavior

- Command `cospan ex2.sr` generates a directed (reachability) graph and checks its cycle structure against `recur` and `cyset` declarations.
- Get an error report:

```
0(0)  .ALT.$=0  .TASK.$=OK
      ->@      /* self-loop */
      ->1(?)>  /* next state is 1 */
1(1)  .ALT.$=1  .TASK.$=NOK
      ->2(?)>
      ->3(?)+
2(2)  .ALT.$=0  .TASK.$=NOK
      ->0(0)+
      ->1(1)< /* back-edge */
Bad cycle at 2(2).
4 states reached, 3 states searched.
...
Task failed
```

Error report is not fully explained in the manual...

Another Example - Parameterized Processes

Problem: Test the "conjecture" that the (multiplicative) group of units in the ring $Z/2^rZ$ is cyclic.

Solution: Compute the powers of x mod 2^r for all odd values of x in the range $3 \leq x \leq 2^r - 1$

Check that for at least one such x, $x^n \equiv 1$ (mod 2^r) only if $n \equiv 0$ (mod 2^{r-1})

Check the condition for each $x = 2i + 3$ for $0 \leq i < 2^{r-1}$
and for each n, s.t. $2 \leq n < 2^{r-1}$.

Parameterized Processes (Cont'd)

Solution Details: Check prev. condition simultaneously for all x: for each x, let $A(x)$ be a process which checks successive powers of $x, x^2, x^3, \ldots$ for the property $x^n \not\equiv 1 (\text{mod} 2^r)$.

If this property is true for some n, $A(x)$ checks x^{n+1}, etc.

Otherwise, it stops checking.

Task for these processes is that in at least one $A(x)$, the value of n should become 2^{r-1}.

```
proctype A(r,x:integer)
  stvar   n:integer
  cyset   {2^(r-1)}
  init    2
  trans

  true    ->n+1  :(n<2^(r-1))*(x^n mod 2^r <> 1)
          ->n    :else;

  end /* A() */

proc B[i<2^(R-1)-1]:A(R,2*i+3)
```


Parameterized Example (Cont'd)

- This created an array of A(r, x) of dimension $2^{r-1} - 1$.

- Can test conjecture for various values of R:

`cospan -DR=2 ex3.sr` - correct
`cospan -DR=4 ex3.sr` - task fails, since all nontrivial units of $Z/16Z$ have order 2 or 4 (therefore, not 8), so the group of units cannot be cyclic.

- Error report:

```
...
ex3.an: Deadlock at 2(2).
ex3.an: Bad cycle at 2(2).
ex3.an: Task failed...
0(0)   .B[0].n=2  .B[1].n=2  .B[2-6].n=2
       ->1(?)>
1(1)   .B[0].n=3  .B[1].n=3  .B[2,3,6].n=2  B[4,5].n=3
       ->2(?)>
2(2)   .B[0].n=4  .B[1].n=4  .B[2,3,6].n=2  B[4,5].n=4
       ->@
       *D*
```

- Specified a task without specifying an extra `monitor`.

- None of the processes had selection variables, and thus there was no coordination between the processes.

Extended Example

```
/* FLOW CONTROL PROTOCOL */
type  message:(0..(2^3)-1)
macro Ready   := 0    /* null message          */
      Ack     := 1    /* ack to SENDER         */
     Lost_msg := 8    /* Code for lost message */
proctype CHANNEL(input:integer)
  import  input
  selvar  #:(0..2^3)
  stvar   $:message
  init    Ready
  trans
  Ready           {0}
        ->input   :true;
  $>0             {0: $, Lost_msg}
        ->Ready   :(#=$) + (#=Lost_msg)
        ->$       :else;
  end /* CHANNEL() */
proc SENDER
  import CH[1]
  selvar #:message
  stvar  $:(SEND,WAIT)
  init   SEND
  trans
  SEND         {0:1..(2^3)-1}
      ->WAIT   :#>0
      ->$      :else;
  WAIT         {0}
      ->SEND   :(CH[1]:Ack)
      ->$      :else;
  end /*SENDER */
```


Extended Example (Cont'd)

```
proc CH[i<2]: CHANNEL(SENDER.# ? i=0 | RECEIVER.#)
proc RECEIVER
  import CH[0]
  selvar #:(0,1)
  stvar  $:message
  init   Ready
  trans
  Ready          {0}
     ->CH[0].#    :~(CH[0]:Lost_msg)
     ->$          :else;
  $>0            {0: Ack}
     ->Ready      :#=Ack
     ->$          :else;
  end /* RECEIVER */

monitor TASK
  import SENDER, CH[0], RECEIVER
  stvar  $:(0..2^3)
  cyset  {0, 1, 2, 3, 4, 5, 6, 7}
  init   0
  trans
  0  -> SENDER.#  :true;
  $ <> Lost_msg
     -> 0         :(RECEIVER.$=$)+(CH[0]: Lost_msg)
     -> Lost_msg  :else*((RECEIVER.$>0)+
                         (CH[0]: 0)*(CH[0].$=0))
     -> $         :else;
  Lost_msg
     -> $         :true;
  end /*TASK */
```

Modeling Interprocess Communication

- This illustrates the modeling of continuous time, delay and transmission across an asynchronous interface.
- Have 4 asynchronous components:
  SENDER attempts to send non-null messages across an asynchronous interface to the outgoing CHANNEL CH[0] to the RECEIVER and waits for Ack from CHANNEL CH[1].
  CHANNEL CH[0] asynchronously propagates messages to RECEIVER or looses them.
  RECEIVER sends an Ack to CHANNEL CH[1].
- Modeling of asynchrony and time - signals are instantaneously transferred between processes
- But consider wire to be the third process - all delay is in the wire.

Pictures

Modeling delay:

Sender/Receiver protocol:

Propagation Delay - Example

FIGURE 4 on p. 19 of Manual

Some More Notation - Selection

Syntax of selection predicate has two possible forms:

- $\{x_1, x_2, ..., x_n\}$ means that any of these selections may be selected from the associated state, and as long as the process remains in that state, the selections may repeatedly (and nondeterministically) change to any of those n values.
- $\{p : x_1, ..., x_n\}$ means that upon entering a state, v, process first selects p for some nonzero amount of time, then the selection may change to any of $x_1, ..., x_n$. The selection p is no longer possible, unless it is one of the x_i's.

Cyclic States

Picture:

Sometimes want to reference the "pause" state v@ explicitly:

```
init  v@
cyset {v@}

....
GO         {p: x, y, z}
  -> STOP : #=x
  -> GO@  : #=y
  -> $    : else;
```


Verifying Extended Example

- `cospan ex4.sr` - 77 states, task is performed.
- Means that every message which is sent and is not explicitly lost or forever held by the CHANNEL is eventually received by the RECEIVER.
- Want to specify that if after some point in time the CHANNEL never loses or holds forever a message, then after that point in time, the SENDER is able to send messages to the RECEIVER without loss.
- Add liveness property: the protocol will always eventually "make progress". This would find an error even when SENDER simply "does not wish" to send any messages.
- Hm... This should tell us that "verification" and "validation" are only as good as the properties we are trying to satisfy.

Specifying Additional Properties

- Can remove `monitor TASK` and add to definition of `proc SENDER` the following line:

```
recur * -> SEND
```

i.e., resulting spec performs this new task iff the SENDER repeatedly returns to its state SEND, from which it may always send a message.
- One more problem - either CHANNEL or the RECEIVER may "pause" forever. Can explicitly state these among acceptance conditions

```
cyset {1@, 2@, 3@, 4@, 5@, 6@, 7@}
```

in `proctype CHANNEL` and `proc RECEIVER`. (This is same as `cyset {*@}`.)
- So, we were able to do without `TASK` monitor again.

Could do the same with another `TASK`

```
/* SENDER may send an unlimited number of messages, providec
/* neither CHANNEL nor the RECEIVER have infinite transit
/* delay, i.e., "pause" forever
monitor TASK2
  import SENDER, CH, RECEIVER
  stvar  $: (SENDING, PAUSING, OTHER)
  recur  *->SENDING
  cyset  {PAUSING} /* excepts cases of infinite transit dela
  init   OTHER
  trans
      ->SENDING  :SENDER.$=SEND
      ->PAUSING  :else*(+[i in 0..1](CH[i].$>0)*(CH[i]:0)+
                  (RECEIVER.$>0)*(RECEIVER: 0))
      ->OTHER    :else;
```

Can include both tasks in one file using `#ifdef`'s and then run `cospan -Dtask1 ex4.sr`.

Next Time

Executable Specifications - PAISLey and StateCharts. Readings:
- Pamela Zave, "An Operational Approach to Requirements Specifications for Embedded Systems", IEEE Transactions on Software Engineering, Vol SE-8, No. 3, May 1992, pp. 250-269.
- Pamela Zave, "An Insider's Evaluation of PAISLey", IEEE Transactions on Software Engineering, Vol. 13, No. 3, March 1991, pp. 212-225.
- David Harel, "StateCharts - A Visual Formalism for Complex Systems", Science of Computer Programming, Vol. 8, pp. 231-274, 1987.

These will be available outside my office (watch the newsgroup).

food for slide eater

food for slide eater

food for slide eater