Obtaining System Equations from Hardy Cross Method

As shown in section XX, the minitown system can be described by simple linear relationships between the system demand, tank height and pump configuration. In section XX, we show how these relationships were determined by analyzing EPANET simulations. In order to further validate these equations outside of EPANET, we performed a similar analysis using the hardy cross method, an iterative algorithm for determining the flow in pipe networks. The hardy cross method makes use of principles of continuity to balance the flow through pipes in a network until reaching some acceptable measure of convergence from an initial guess. Using a python script, multiple hardy cross simulations were performed and linear regressions were calculated from the resulting data to obtain the following system equations:

(1) 1 Pump on => Qpump = 113.7 + 6.74*Demand -1.35*TankH
(2) 2 Pumps on => Qpump = 181.44 + 18.36*Demand - 2.16TankH

As compared to those obtained through EPANET simulations:

(3) 1 Pump on => Qpump = 113.6 + 8.4*Demand -1.4*TankH
(4) 2 Pumps on => Qpump = 178.4 + 20.9*Demand - 2.1*TankH

NuXmv Introduction and Overview

Nuxmv is a model checking tool that has the ability to use SMT-verification for both finite and infinite state systems, unlike FSP that only supports finite state analysis. The main draw of NuXmv is that it supports unbounded integers and reals. This allows us to directly use the real-valued functions that define our system behavior which NuXmv evaluates using fixed point arithmetic. This gives us greater flexibility than tools like FSP, which limit us to bounded integer variables and force us to define all of our state transitions explicitly.

One limitation of NuXmv (and most model checking tools) is that there is limited support for non-linear and transcendental functions. Many dynamic system equations contain trigonometric or exponential terms; in those cases, work must be done to simplify the relationships into terms that Nuxmv can evaluate (a finite difference scheme could be used, for example). As seen in previous sections, for MiniTown, we have simplified the demand flow-tank height relationship down to a set of two variable linear equations - something that NuXmv can readily handle.

Once a valid model specification has been constructed in NuXmv, various model checking engines can be used to verify assertions about its behavior. In the infinite state case, we must use bounded model checking. NuXmv will only be capable of proving that an assertion is True or False, or that a certain state is reachable, within a max bound. This bound is roughly proportional to the amount of transitions that would be needed to reach such a state. Because, in our model, we use Nuxmv's state transitions as our implied time (hours), we can think of our bounded model checking in terms of time. In other words, asking NuXmv to check an assertion with a max bound of 10, is equivalent to asking if that assertion is valid within 10 hours.

Basic NuXmv Model Structure and Finite State Model in NuXmv

We will first introduce the finite state miniTown model in NuXmv and use it to illustrate the basic structure and operation of NuXmv models.

In a broad sense, NuXmv modules define variables and constants, as well as the rules that govern their behavior during transitions from one state to the next. NuXmv models are typically organized into blocks such as DEFINE, VAR and TRANS.

In the finite state case, we open our module by defining the model variables in the VAR block. The variable name is followed by the set of possible values that it can take on:

```
18  MODULE main
19
20  DEFINE
21
22  VAR
23    pump1  : {on,off};
24    pump2  : {on,off};
25    demand : {0,25,50,75,100};
26    tank   : -10..660;
27    time   : 0..200;
28
```

Because this is a finite state model, all of our variables are restricted to bounded sets of symbolic values (such as pumps being on or off) or bounded integers (such as our tank height). An important note is that although the miniTown tank is 6.5m tall, we have multiplied everything by 100 for our NuXmv models. This is done because we are limited to using Integer variables, but would like to work in 2 decimal place precision. We also add some extra imaginary height at either end of the tank (-10 to 660). This is done because we will be checking assertions such as whether the tank ever overflows - and we want our model behavior to control that type of specification, rather than enforcing it as a definition of the variable. Time is implicit within our model as the transition between states but we introduce a time variable only so that we can later vary demand as a function of time

Following our variable declarations we insert an ASSIGN block. Here we do variable assignment for initial states with init() and simple transition rules with next() . If an initial value for a variable is not specified, NuXmv will interactively ask for one at run time if a simulation is performed, otherwise it will pick random initial values if just checking a specification.
Case statements in NuXmv are analogous to if/ if else/ else statements - we can use these to make simple rules about the behavior of some variables during transitions, such as specifying the behavior of the pumps turning on or off depending on tank height:

```
ASSIGN

init(tank) := 500; -- change this as needed
init(pump1) := on;
init(pump2) := off;
init(time) := 0;
init(demand) := 100;

-- have to do this or else nuxmv complains that time isnt defined past 200
next(time) := case

        time <= 199: time + 1;
        TRUE : time;
esac;

next(pump1) := case

        next(tank) <= 400 : on;
        next(tank) >= 630: off;
        TRUE : pump1;
esac;

next(pump2) := case

        next(tank) <= 100 : on;
        next(tank) >= 450: off;
        TRUE : pump2;
esac;

next(demand) := case

   next(time) = 1 : 100;
   next(time) = 2 : 100;
   next(time) = 3 : 75;
   next(time) = 4 : 75;
   next(time) = 5 : 50;
   next(time) = 6 : 50;
   next(time) = 7 : 75;
   next(time) = 8 : 50;
   next(time) = 9 : 50;
```

Demand in miniTown can be modeled as a fixed sequence of values (from the original paper/study) that vary with time. We discretize these into 0,25,50,75 and 100% of max demand and use a case statement that relates current time to demand. The demand pattern is specified up to 170 hours, but only a portion of the case statement is shown above.

TRANS blocks define transitions that require a more complex definition than what can be fit into a next() assignment or case statement. In this model, the TRANS statements specify the change in the tank level as a logical implication (->) that evaluates the pump configuration, demand and tank height. This illustrates another challenge in using a finite state representation - every distinct transition state must be defined explicitly - which can result in hundreds of lines as below. For MiniTown, we created a python script to automatically generate these transition statements by looping through all possible configurations and evaluating equations (1) and (2). The script also combines any transitions that can be grouped (such as distinct adjacent tank heights that still result in the same transition) to simplify the model.

```
225      next(time) = 165 : 100;
226      next(time) = 166 : 100;
227      next(time) = 167 : 100;
228      next(time) = 168 : 100;
229      next(time) = 169 : 50;
230      TRUE:100;
231
232   esac;
233
234   TRANS pump1 = off & pump2 = on & demand =  0 & tank >= 0 & tank < 507 -> next(tank) = tank - -13 ;
235   TRANS pump1 = on & pump2 = off & demand =  0  & tank >= 0 & tank < 507 -> next(tank) = tank - -13 ;
236   TRANS pump1 = off & pump2 = on & demand =  0 & tank >= 507 & tank < 639 -> next(tank) = tank - -12 ;
237   TRANS pump1 = on & pump2 = off & demand =  0  & tank >= 507 & tank < 639 -> next(tank) = tank - -12 ;

                                 ...
449   TRANS pump1 = on & pump2 = on & demand = 75  & tank >= 35 & tank < 430 -> next(tank) = tank - -4 ;
450   TRANS pump1 = on & pump2 = on & demand = 75  & tank >= 430 & tank < 648 -> next(tank) = tank - -3 ;
451   TRANS pump1 = on & pump2 = on & demand = 75  & tank >= 648 & tank < 649 -> next(tank) = tank - -2 ;
452   TRANS pump1 = on & pump2 = on & demand = 75  & tank >= 649 & tank < 650 -> next(tank) = tank - -1 ;
```

Finally, LTLSPEC statements let us define the assertions we wish to verify, the G prefix indicates that we would like to check the specification globally for all states. In this model we check various tank heights to verify if they are ever breached:

```
548   LTLSPEC G tank <= 550;
549   LTLSPEC G tank <= 600;
550   LTLSPEC G tank <= 650;
551
552   LTLSPEC G tank >= 300;
553   LTLSPEC G tank >= 200;
554   LTLSPEC G tank >= 100;
555   LTLSPEC G tank >= 0;
```

These specifications can be checked interactively through the bounded model checking command from the NuXmv command line. Because this model uses finite state variables, NuXmv can check these assertions globally without the caveat that they only apply within a max bound. This is the advantage we receive in exchange for being restricted to bounded integers and explicit definition of all state transitions.

When checking these assertions, NuXmv will display a trace (sequence of transition states) that leads to any counter-example of an assertion. Below is a snip of running this model:

```
nuXmv > check_ltlspec
-- specification  G tank <= 550  is true
-- specification  G tank <= 600  is true
-- specification  G tank <= 650  is true
-- specification  G tank >= 300  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    pump1 = on
    pump2 = off
    demand = 100
    tank = 500
    time = 0
  -> State: 1.2 <-
    tank = 489
    time = 1
  -> State: 1.3 <-
    tank = 478
    time = 2
  -> State: 1.4 <-
    demand = 75
    tank = 467
    time = 3
  -> State: 1.5 <-
    tank = 462
    time = 4
  -> State: 1.6 <-
```

Infinite State Model in NuXmv

A separate model of the miniTown system was created using infinite precision variables to test the bounded model checking features. In this model we can define our system variables as unbounded integers and reals as below:

```
49  VAR
50    pump1  : {on,off};
51    pump2  : {on,off};
52    demand : real;
53    tank   : real;
54    time   : integer;
55
```

With these variable types we can then directly use our system equations in the DEFINE block:

```
29  DEFINE
30
31  pumpFlow0 := 0;
32  pumpFlow1 := 113.699 + 6.735*demand - 1.35* (tank/100);
33  pumpFlow2 := 181.44 + 18.356*demand - 1.35*(tank/100);
34
35  tankFlow0 := demand * 208 - pumpFlow0;
36  tankFlow1 := demand * 208 - pumpFlow1;
37  tankFlow2 := demand * 208 - pumpFlow2;
38
39  dT0 := tankFlow0 * 0.004678 *100;
40  dT1 := tankFlow1 * 0.004678 *100;
41  dT2 := tankFlow2 * 0.004678 *100;
42
```

The first set of equations determine the flow through the pumps in each configuration (0,1 or 2 pumps on). The second set of equations determine the flow out of the tank as a function of the previous equations. Finally, the third set of equations determine the change in tank height as a function of the tank flow and desired time step.

Our ASSIGN block in this case is nearly identical to the finite case:

```
63  ASSIGN
64
65  init(tank) := 500; -- change this as needed
66  init(pump1) := on;
67  init(pump2) := off;
68  init(time) := 0;
69  init(demand) := 0.9;
70
71  next(pump1) := case
72
73          next(tank) <= 400 : on;
74          next(tank) >= 630: off;
75          TRUE : pump1;
76  esac;
77
78  next(pump2) := case
79
80          next(tank) <= 100 : on;
81          next(tank) >= 450: off;
82          TRUE : pump2;
83  esac;
84
85  next(time) := time + 1;
```

Demand is now a real variable so we can discretize it more finely (or not at all):

```
85   next(time) := time + 1;
86
87   -- pre-determined demand sequence
88
89   next(demand) := case
90
91       next(time) = 1 : 0.9;
92       next(time) = 2 : 0.9;
93       next(time) = 3 : 0.9;
94       next(time) = 4 : 0.7;
95       next(time) = 5 : 0.6;
96       next(time) = 6 : 0.6;
```

Should we decide to not discretize demand or constrain it to a set sequence, we could make use of an INVAR statement that places an invariant on the variable as follows:

```
270   --INVAR demand <= 1;
271   --INVAR demand >= 0;
```

Finally, we have our TRANS statements. We can specify this model in 4 lines as opposed to the hundreds of lines in the finite case:

```
279   TRANS pump1 = off & pump2 = off -> next(tank) = tank - dT0;
280   TRANS pump1 = on & pump2 = off -> next(tank) = tank - dT1;
281   TRANS pump1 = on & pump2 = on -> next(tank) = tank - dT2;
282   TRANS pump1 = off & pump2 = on -> next(tank) = tank - dT1;
```

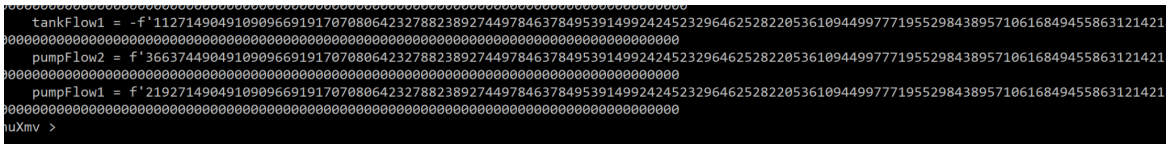
```
290   LTLSPEC G tank <= 550;
291   LTLSPEC G tank <= 600;
292   LTLSPEC G tank <= 650;
293
294   LTLSPEC G tank >= 300;
295   LTLSPEC G tank >= 200;
296   LTLSPEC G tank >= 100;
297   LTLSPEC G tank >= 0;
```

In the final section, we again define LTL-type specifications that we wish to verify with NuXmv's bounded model checker. The prefix G signifies that we wish to verify that the proceeding property holds globally for all states. As seen in figure X below, running the command

msat_check_ltlspec_bmc -k n will call the bmc checker to verify each specification up to a bound of n. As in the finite case, if a counter-example is found, NuXmv will provide a trace showing each transition that led to the undesired state.

```
nuXmv > read_model -i RealDemPattern.smv
nuXmv > go_msat
nuXmv > msat_check_ltlspec_bmc -k 20
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- no counterexample found with bound 5
-- no counterexample found with bound 6
-- no counterexample found with bound 7
-- no counterexample found with bound 8
-- no counterexample found with bound 9
-- no counterexample found with bound 10
-- no counterexample found with bound 11
-- no counterexample found with bound 12
-- no counterexample found with bound 13
-- no counterexample found with bound 14
-- no counterexample found with bound 15
-- no counterexample found with bound 16
-- no counterexample found with bound 17
-- no counterexample found with bound 18
-- no counterexample found with bound 19
```

By default , NuXmv uses a fixed point representation of decimal numbers, rather than floating point -so a demand of 0.9 will be shown as f’9/10. As the computation continues and numbers with larger decimals appear, the numerators and denominators will become increasingly large. Although this makes it harder to inspect generated traces, these numbers can be easily transformed to floating point in a post-processing step for later analysis.



Results / Testing / Comparison to FSP Results

In Progress…

In [43]:

```
print("Time----Pump1-----Pump2-----Tank")
for i in range(0,100):
    print(time[i],"----",p1[i],"----",p2[i],"----",tank[i])
```

```
Time----Pump1-----Pump2-----Tank
0.0 ---- True ---- False ---- 5.0
1.0 ---- True ---- False ---- 5.0295549035
2.0 ---- True ---- False ---- 5.058923119018806
3.0 ---- True ---- False ---- 5.088105825799056
4.0 ---- True ---- False ---- 5.117104195634523
5.0 ---- True ---- False ---- 5.145919392917168
6.0 ---- True ---- False ---- 5.174552574683898
7.0 ---- True ---- False ---- 5.2030048906630215
8.0 ---- True ---- False ---- 5.231277483320415
9.0 ---- True ---- False ---- 5.259371487905399
10.0 ---- True ---- False ---- 5.287288032496321
11.0 ---- True ---- False ---- 5.315028238045854
12.0 ---- True ---- False ---- 5.3425932184260025
13.0 ---- True ---- False ---- 5.369984080472832
14.0 ---- True ---- False ---- 5.397201924030913
15.0 ---- True ---- False ---- 5.4242478419974836
16.0 ---- True ---- False ---- 5.451122920366331
17.0 ---- True ---- False ---- 5.4778282382714
18.0 ---- True ---- False ---- 5.504364868030122
19.0 ---- True ---- False ---- 5.53073387518648
20.0 ---- True ---- False ---- 5.556936318553784
21.0 ---- True ---- False ---- 5.582973250257192
22.0 ---- True ---- False ---- 5.608845715775955
23.0 ---- True ---- False ---- 5.634554753985399
24.0 ---- True ---- False ---- 5.660101397198637
25.0 ---- True ---- False ---- 5.685486671208022
26.0 ---- True ---- False ---- 5.710711595326336
27.0 ---- True ---- False ---- 5.735777182427718
```