Modeling Hybrid Systems using Analytic Constraint Logic Programming

Timothy J. Hickey and David K. Wittenberg

Computer Science Department, Brandeis University
{tim|dkw}@cs.brandeis.edu

Abstract. We use an interval-based Analytic Constraint Logic Programming (ACLP) language to accurately and declaratively model Hybrid Systems. In particular, we model the continuous part of Hybrid Systems using Ordinary Differential Equation (ODE) constraints on function variables. Because we make intervals ubiquitous, error bars in measurements and ODE parameters can be modeled explicitly in a natural manner. There are four main advantages to using an ACLP language to model Hybrid systems. 1. ACLP allows one to manipulate ODEs declaratively. This eliminates any ad hoc reasoning about the "modeling" error as the constraints deal directly with the ODEs. In particular, the intervals computed by the ACLP system are guaranteed to contain all solutions to the (analytic) constraints. 2. Since there is no modeling error, the analyst can seek to improve the model by employing carefully estimated error bars for all measured quantities and for representing uncertainty anywhere within the system (e.g. in the parameters of the ODEs, in the conditions for state changes). 3. Since the program faithfully represents the Hybrid System, one can in principle use program analysis techniques to directly analyze the properties of the Hybrid System. 4. CLP(F) handles ODEs with non-linear terms, unlike many of the older Hybrid Automata models.

1 Introduction

A Hybrid System is a system composed of a digital part (typically a small computer) and an analog part (typically a physical system with sensors and actuators). All computer controlled or monitored processes in the real world are hybrid systems. Recent work on Hybrid Systems includes defining models [14] [13] [1] [15][4] and calculating the behaviour of the analog parts [8] [2].

A Hybrid Automaton [1] [6] [8] [5] [14] [13] is a formal model of a Hybrid System. Most Hybrid Automata models are expanded finite state automata with two different forms of state transition. The first (a "jump" transition), is the standard state transition in a finite state automata, which takes zero time, and changes the state of the digital part of the system. The second (a "flow" transition), is the change in the analog part of the system over time. There are many versions of hybrid automata, which differ in what they can model. Some earlier models required that the flow transitions be piece-wise linear. Many later models allow flow transitions described by ODEs, sometimes limited to linear ODEs.

One of the important uses of Hybrid Automata is to prove *safety properties* of a Hybrid System. A safety property is a guarantee that if the system is started in a reasonable state, it never enters some prohibited region of the state space. Typically that region is defined only by the values of analog components of the state space.

Interval Arithmetic is the obvious choice for modeling hybrid systems, as the interface between the analog and the digital part involves imperfect hardware whose description must include error bars. Intervals are a very clear way of explicitly expressing error bars. Because Interval Arithmetic provides constraints on the range of values that each variable can take on, it is well suited to proving that certain values are not reached (*ie.*safety properties).

There has been recent work in using Constraint Logic Programming [12] to analyze various aspects of Hybrid Systems [3], [17], [18], [16]. One problem with these conventional CLP approaches to modeling Hybrid Systems is that they must deal with the ODEs describing the continuous part of the system using some sort of approximation (such as discretization into difference equations or restriction to ODEs that have a closed form solution). This introduces a "modeling error" that must be reasoned about outside of the CLP program. In [9] CLP(F), an Analytic Constraint Logic Programming (ACLP) language over the domain of differentiable functions is defined. In this paper, we show how CLP(F) allows one to overcome this "modeling error" by allowing the ODE to be expressed explicitly as a constraint on function variables. The resulting ACLP program has the property that the results computed using the CLP(F) system are guaranteed to contain all solutions of the ODEs modeled by the constraints. The primary disadvantages of this approach are that it does not yet handle the wrapping problem for the ODEs and it is very resource intensive and hence can not currently model systems over a long modeling period.

One of the major benefits of this approach is that the problem of analyzing the hybrid system is transformed into the problem of analyzing the corresponding CLP(F) program. In principle, one should be able to apply well understood program analysis techniques to this program and directly infer provable properties of the corresponding hybrid system. In this paper we describe only the simpler types of analysis that one can do by directly solving CLP(F) constraints related to the hybrid system.

To demonstrate the ACLP approach to Hybrid System modeling, we consider the Hybrid System of a thermostat introduced in Henzinger *et al.* [7] [8]. This is a system consisting of a stirred pot of water with a temperature sensor and a heater in it. When the measured temperature goes above a threshold, the logic circuit shuts off the heater (after a small delay). Similarly, when the measured temperature goes below a threshold, the logic circuit turns on the heater (after a small delay). The safety property in question is to establish upper and lower bounds for the temperature of the water. The state diagram is given in Figure 2.

Henzinger *et al.* take a major step towards reliability of their results by using interval arithmetic in solving the differential equations which describe the system, we improve on this by modeling the system declaratively as an ACLP program (written in the CLP(F) language [11] [9]) in which the differential equations appear directly as constraints in the program and the system is modeled using intervals for all measurements (to model the inevitable error-bars of instruments) as well as to provide over-approximations to deal with rounding error.

2 CLP(F)

Hickey [9] introduced the idea of an Analytic Constraint Logic Programming (ACLP) and the language CLP(F) as an example of such a language. CLP(F) is built using metalevel techniques to extend a CLP(Intervals) system CLIP [10]. In CLP(F) the constraint domain allows one to declare variables representing various analytic values including:

- *real numbers, X*
- *infinitely differentiable functions, F, on a finite interval [a,b]*
- *vectors of numbers, functions, or vectors*

A full description of the language is available in [11] and [9]. In this section, we provide a brief overview of the language, its semantics, its implementation, and its use.

As is common in CLP languages, the constraints are enclosed in curly braces "{}". The different types of variables are declared using the `type` predicate. The CLP(F) interpreter provides answers to queries in the form of a sequence of solution sets, where each solution set provides a real interval for each of the constraint variables. The soundness property of CLP implies that every correct solution to the query must be contained in one of the solution sets (assuming that the program eventually terminates). On the other hand, not every element of the solution set is guaranteed to be a solution (and indeed, there may not be any actual solutions in any particular solution set returned by the interpreter).

2.1 Algebraic Constraints and Solvers

The CLP(F) constraint language allows you to express any algebraic equality or inequality constraint among real variables. For example,

```
| ?- {X^2=2,X>0}.
X = 1.41421356237309... ? ;
no
| ?-
```

The CLP(F) interpreter represents the interval for X in a compact form. The ellipsis "..." indicates that all shown digits are correct and hence X must lie in the interval:

```
[1.41421356237309,
 1.41421356237310)
```

Also, note that the user entered a semi-colon after the solution and the interpreter responded with "no" which indicates that there are no more solutions.

Sometimes there maybe more than one solution to a given constraint. The constraint solver will indicate this by returning an interval that contains all solutions:

```
| ?- {X^2=2}.
X = REAL(-1.4142135623730953675192267837 7,
            1.4142135623730953675192267837 7) ?
no
| ?-
```

Here, to find the discrete set of solutions one must apply a divide-and-conquer approach where one divides the interval into subintervals (here of size 10^{-6}) and searches for solutions in each one. This is done using the "queue" method of the `solve_clip` solver:

```
| ?- {X^2=2},solve_clip(queue,[X],0.000001).
X = 1.41421356237309... ? ;
X = -1.41421356237309... ?
(10 ms) no
| ?-
```

The constraint language for real variables allows any equations and inequalities constructed using the arithmetic operators and the standard mathematical functions (sin, cos, tan, asin, acos, atan, exp, log, exponentiation, $**$, integral powers X^Y, and others). There is also a moderately sophisticated solver `solve_clip(METHOD,VARS,N)` which allows you to specify the solving method, the list of variables that should be solved, and a parameter N representing how much work should be done (e.g. a maximum allowed width for intervals, or a maximum depth for a divide and conquer splitting routine). For example, the following query demonstrates the use of the forward checking solver `fwchk`, which divides the domain into a set of K boxes (initially $K = 1$) and in each step it divides each box into 2^V sub-boxes, where V is the number of variables in `VARS`, and applies the default narrowing procedure. Any boxes that are proved to contain no solutions are discarded and the result is returned as the smallest box containing all of the remaining boxes. An example of the narrowing done by this method to solve $x^x = 1 + \cos(x) \ \wedge \ x > 0$ is:

```
| ?- {X**X = 1+cos(X), X>0},solve_clip(fwchk,[X],N).
N = 0  X = REAL(0,inf) ? ;
N = 1  X = 1.247504654353... ? ;
N = 2  X = 1.24750465435333... ? ;
N = 3  X = 1.24750465435333... ?
(1720 ms) yes
| ?-
```

2.2 Analytic constraints and procedures

CLP(F) also allows one to constrain functions by functional equations involving the same arithmetic operators and mathematical functions as discussed above. In addition, one can constrain a function to take certain values at certain points and to have a range that lies within an interval. For example, the following constraint specifies that F is a function on $[0, 1]$ such that $F' = F$ and $F(0) = 1$ and $F(A) = 2$ and $F(1) = E$ and $F([0, 1] \subset [-1000, 1000]$:

```
| ?- type([F],function(0,1)), {[ ddt(F,1)=F, eval(F,0)=1,
      eval(F,A)=2, eval(F,1)=E,  F in [-1000,1000] ]}.
A = 0.6931471... E = 2.7182818...
F = ps(1,10,[(0,[1,1,1,1,1,1,1,1,1,1]), ....) ;
(760 ms) no
| ?-
```

The type predicate is used to declare that F is an infinitely differentiable function on the interval $[0, 1]$. The CLP(F) system solves analytic constraints by soundly approximating analytic functions by power series and introducing arithmetic constraints among the Taylor coefficients of the functions at the endpoints, at points in the interval, and over the entire range.

The function F is then constrained to be equal to its first derivative, and to take the value 1 at 0 and to take values in $[-1000, 1000]$ for all $x \in [0, 1]$. The variables A and E are not declared to be functions and hence are real constants by default. They are constrained so that $F(A) = 2$ and $F(1) = E$. The constraint solver finds A and E to 7 decimal digits of precision and also finds an interval for F (only partially shown here) that specifies intervals for its first 10 derivatives at 0 and 1, and for the range of its first 10 derivatives over $[0, 1]$. The number of derivatives (10) can be set to any value N (but space and time complexity grows quadratically with N).

In this paper we will use CLP(F) to define higher order constraints which specify that two points lie on a trajectory defined by on ODE. For example, in the simplest model of a thermostat we use the following CLP(F) procedure, where T0,T1 are times, A0,A1 are temperatures at those times, A is the temperature function (so $A(T0) = A0$) and Alpha,Beta are the heat loss and the heater element components of the ODE for A. The parameter I is a mandatory maximum step size needed by CLP(F) as all functions must be defined on finite intervals.

```
ode((T0,A0),[I,[Alpha,Beta]],A,(T1,A1)) :-
   type([A],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta,
    eval(A,0)=A0,   eval(A,T)=A1,
    A in [-1.0E100,1.0E100],
    T=T1-T0,   T in [0,I]   ]}.
```

In this case the ODE is $f' = af + b, f(0) = a_0, f(t) = a_1$ which can be solved exactly. Thus, in this case, we can actually define the ode procedure (almost) equivalently in CLP(Intervals) without using the sophisticated analytic constraint solving machinery, as follows:

```
odeSolveForm((T0,A0),[I,[Alpha,Beta]],_,(T1,A1)) :-
  {A1 = C1*exp(Alpha*(T1-T0)) - C2,
   C1=(A0 + Beta/Alpha),
   C2= Beta/Alpha}.
```

The difference between these two procedures is that the first one returns the solution function A of the ODE, which can be used to compute f at any point in the region, or to return the range, or to further constrain the trajectory, while the second merely returns the range for T.

The CLP(F) solver can also handle very complex non-linear differential equations as it based on a "brute force" reduction of the analytic constraints into arithmetic constraints which are solved with a simple interval arithmetic constraint solver. For example, in the more complex examples we model a system consisting of a fluid with temperature $A(t)$ which is heated by a heating element whose temperature $B(t)$ has a non-linear component $\sin(B(t))$ in its defining ODE. This system is modeled by the following procedure:

```
ode2((T0,A0),[I,[Alpha,Beta,Gamma,Delta]],A,(T1,A1)) :-
   type([A,B],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta + Gamma*B,
    ddt(B,1) = Delta*(B + 0.1*sin(B)),
    eval(A,0)=A0,   eval(A,T)=A1,
    eval(B,0)=1,
    A in [-1.0E100,1.0E100],
    B in [-1.0E100,1.0E100],
    T=T1-T0,   T in [0,I]
]}.
```

We are unaware of a closed form for the solutions of this ODE and yet the CLP(F) system is easily able to use this definition to compute $(T1, A1)$ from $(T0, A0)$, or, as we will see below, to use this procedure to find values of the parameters `Alpha, Beta, Gamma, Delta` which make the system behave in some desired fashion.

2.3 Programs

CLP(F) programs are Prolog programs in which the bodies of rules may contain constraints on functions and their derivatives. CLP(F) provides the full power of Prolog in addition to the power of the underlying constraint solver and both are combined within a single logical semantics. Moreover, by the soundness and completeness of CLP semantics, if a CLP interpreter returns with N solution sets $C_1, \ldots, C_n$ for a query $Q(X, F)$, then all solutions of the query Q consisting of a real vector X and a vector F of real-valued functions, are contained in the union of the solution sets C_i.

For example, the program in Figure 1 is one way of implementing a general hybrid system simulator in CLP. The first parameter of the `evolve` procedure is the initial state of the hybrid system, which consists of a discrete state S and a continuous state X. The second parameter is a list of values used to specify the particular hybrid system. The third parameter is the final (or ending) state of the hybrid system. The last parameter is a list of the discrete state changes needed to go from the initial to the final state. The continuous part of these states are the values of the continuous variables at the moment of state change.

```
evolve(H,C,H,[]).

evolve((S0,X0),C,(S,X),[(S1,X1,Range)|R]) :-
   statechange((S0,X0),C,S1),
   in_trajectory((S1,X0),Range,C,X1),
   evolve((S1,X1),C,(S,X),R).
```

Fig. 1. A general simulator for hybrid systems

Observe that the procedure is relatively simple. It looks for a state change from `(S0,X0)` to state `S1`. Such a change may constrain the values of `X0` to lie within narrow intervals. Then it looks for a new trajectory represented by the a continuous variable `X1`. Typically, the continuous state will be represented by a pair of dependent variables `(T,A)` where A is the value of some ideal system sensor at time T. The `in_trajectory` procedure looks up the ODE, C, that should hold in this state and applies that ODE to the initial values `X0` to get the new values `X1`. The range `R` of the continuous variable during this trajectory is also returned. This is used for determining safety properties. In general other ”range” data could also be returned (e.g. the range of the derivative of the temperature or of some other system parameter).

3 Henzinger's Model and Analysis

In this section, we present the model of a thermostat with a delay in switching used by Henzinger *et al.* [8]. Henzinger's model consists of a finite state controller with an analog input measuring the temperature in the tank. The controller has a 1-bit output to control a heater in the tank. The tank always loses heat at a rate directly proportional to the temperature, and, while the heater is on, is heated at 4 degrees/second. Mathematically, after the heater reaches equilibrium in the on position $A' = -A + 4$ and at equilibrium in the off position $A' = -A$. The controller switches the heater off within one second of the temperature going above a pre-set value, and turns the heater on within one second of the temperature dropping below another threshold. Note that this model assumes that the thermometer is perfect, the heater produces a constant and perfectly known heat output, the element heats and cools instantly and the physics of the tank are perfectly modeled by the ODE. Given those assumptions, they then use interval techniques to eliminate round off errors in proving safety properties.

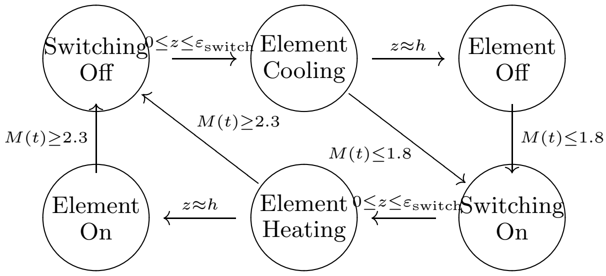

Fig. 2. State model allowing thermostat to shut off before element is warm

Henzinger *et al.* [7] analyzes reachability by using over approximation. Consider the space defined by the real variables. All calculations are done on rectangles in that space, such that the rectangle completely contains the region that it is modeling. By maintaining a *frontier* (a list of areas which have not been explored, but which border on the areas which are reachable) and a list of areas which are reachable, one can explore areas to determine what areas can be reachable, and not have to consider any area more than once.

Later, Henzinger *et al.* [8] made a significant advance over previous approaches because the use of interval arithmetic algorithms allowed them to use a more realistic model for the analog part of a hybrid system. The earlier models used piece-wise linear approximations to the analog system in polygonal regions and the provability was guaranteed by using infinite precision rational arithmetic to analyze the resulting model. The transition to Interval Arithmetic allowed the analog part of the model to be represented by ordinary differential equations and the correctness was then guaranteed by the conservative nature of the interval arithmetic ODE solving algorithms, in particular with respect to the handling of rounding errors. Nevertheless, the program which analyzes the model must still be implemented in a conventional programming language (e.g. C or Fortran or Matlab) in which the ODE solving is done by calling interval arithmetic-based ODE solving library routines.

The approach we propose in this paper is to model the hybrid system by an ACLP program in which the ODEs can be expressed directly as constraints on function variables. This will allow the program to directly represent the hybrid system in the sense that one can analyze the hybrid system by simply analyzing the corresponding program.

4 CLP(F) Model of the Thermostat

In this section we present two models of a thermostat. The first simple model demonstrates the key ideas. The second illustrates how one can easily extend the simple model to a model that more faithfully represents the real hybrid system by more closely approximating the physics of the system. In section 5, we discuss adding error bars to all measured values in the model.

4.1 The Simple Model

Our first CLP(F) model of a thermostat is shown in Figure 3. To clarify the key concepts, this first model assumes there are only two states: on and off. When the system state is on, the ODE governing the temperature A is $A' = -A + 4$. When the system state is off, the ODE is $A' = -A$. The system switches from on to off when the temperature rises above 2.3 and it switches from off to on when it drops below 1.8. The `in_trajectory` procedure models the trajectory by looking up the proper ODE for the current state and then calling the ODE procedure to constraint the new state variables `(T1,A1)`. It also, adds the constraint that the temperature range is contained in `[-1000, 2.3]` (resp. `[1.8,1000]`). This is not needed for our simple example because the temperature rises monotonically and then falls monotonically and then rises again. With more complex models, the temperature might not behave so nicely and this constraint states that no point in the trajectory has passed the threshold for switching. The `statechange` procedure simply indicates the condition that signals a state change and provides the new state. The `ode` procedure models the specified ODE as we have described above. Finally the `test` procedure shows how this program can be used to model the behavior of the system. It initializes the list describing the system to be analyzed and then invokes the `evolve` procedure. The result of a simple query (at what times is the temperature 2) to this system is shown in Figure 4.

One subtle point about this model is that the CLP(F) solver will only work effectively if a finite step size is explicitly given (this is the `I` parameter appearing in the `in_trajectory` and `ode` procedures. If the step size is too large, then the CLP(F) solver will return very wide, unhelpful intervals for all variables. One approach to handling this is to introduce pseudo states `(on,n), (off,n)`, where n is an integer representing the number of full steps that have been taken on the current trajectory in the current state. The continuous part can be modeled as `(t,a,z)` where t is the total elapsed time, a is the temperature at time t, and z is the time relative to the current step. Such an extension of the current techniques is straightforward and we do not show it here due to space limitations.

4.2 A More Realistic Model

In the example shown in Figure 5, we refine the previous model by using six states `on,sw0,cooling,off,sw1,heating` corresponding to the states in Henzinger's model. The model also represents the continuous state as a triple `T,A,Z` where T is the total elapsed time, A is the temperature at time T, and Z is the time since

```
evolve(H,C,H).

evolve((S0,X0),C,(S,X)) :-
   statechange((S0,X0),C,S1),
   in_trajectory((S1,X0),C,X1),
   evolve((S1,X1),C,(S,X)).

in_trajectory((S0,(T0,A0)),[I, Min, Max, ODEs],(T1,A1)) :-
  member(S0=ODE,ODEs), {T=T1-T0,T=<I},
  ode((T0,A0),[T,ODE],A,(T1,A1)),
  ( (S0=on,       {[A in [-1000,Max] ]});
    (S0=off,      {[A in [Min,1000]]})).

statechange((S0,(T0,A0)),[_I, Min, Max, ODEs],S1) :-
  ( (S0=on,      {A0= Max},             S1=off);
    (S0=off,     {A0=Min },             S1=on) ).

ode((T0,A0),[I,[Alpha,Beta]],A,(T1,A1)) :-
   type([A],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta,
    eval(A,0)=A0,   eval(A,T)=A1,
    A in [-1.0E100,1.0E100],
    T=T1-T0,   T in [0,I]
]}.
% call as ?- test(S,(T,A)),{A=2}. to find full/half cycle times.
test(S,X) :-
     % Step, Min, Max,    ODES for on/off states
   C=[ 2.0,   1.8, 2.3,  [on=[-1,4],off=[-1,0]]],
   in_trajectory((on,(0,2)),C,X0),
   evolve((on,X0),C,(S,X)).
```

Fig. 3. Simplest CLP(F) model of a thermostat

```
| ?- test(S,(T,A)),{A=2}.
A = 2,  S = on,  T = 0 ?
A = 2,  S = off, T = 0.3022808718... ? ;
A = 2,  S = on,  T = 0.5029515673... ?
```

Fig. 4. Query to Simple Model - When is Temp = 2?

```
evolve(H,C,H,[]).
evolve((S0,X0),C,(S,X),[(S1,X1,Range)|R]) :-
   statechange((S0,X0),C,S1),
   in_trajectory((S1,X0),Range,C,X1),
   evolve((S1,X1),C,(S,X),R).

in_trajectory((S0,(T0,A0,Z0)),R,[Step, Min, Max, Delay, Stime,
ODEs],(T1,A1,Z1)) :-
  member(S0=ODE,ODEs), {Z1=T, T=T1-T0, T=<Step},
  ode((T0,A0),[T,ODE],A,(T1,A1)),
  ((S0=on,     {[A in [-1000,Max] ]});
  (S0=sw0,     {[T =<  Delay]});
   (S0=cooling,  {[T <   Stime, A in [Min,1000] ]});
   (S0=off,      {[A in [Min,1000] ]});
   (S0=sw1,      {[T =<  Delay]});
   (S0=heating,  {[T <   Stime, A in [-1000,Max]  ]})),
   {[V in [0,Z1], eval(A,V)=R]}.
statechange((S0,(T0,A0,T)),[Step, Min, Max, Delay,Stime,ODEs],S1) :-
  ((S0=on,      {A0=Max},       S1 = sw0);
   (S0=sw0,     {T=Delay},      S1=cooling);
   (S0=cooling, {T=Stime},      S1=off);
   (S0=cooling, {A0=Min},       S1=sw1);
   (S0=off,     {A0=Min },      S1=sw1);
   (S0=sw1,     {T=Delay},      S1=heating);
   (S0=heating, {T=Stime},      S1= on);
   (S0=heating, {A0=Max},       S1= sw0)).
ode((T0,A0),[I,[Alpha,Beta,Gamma,Delta]],A,(T1,A1)) :-
   type([A,B],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta + Gamma*B,
    ddt(B,1) = Delta*(B + 0.1*sin(B)),
    eval(A,0)=A0,   eval(A,T)=A1,
    eval(B,0)=1,
    A in [-1.0E100,1.0E100],
    B in [-1.0E100,1.0E100],
    T=T1-T0,   T in [0,I]
]}.

test(S,X,[(on,X0,Range)|R],D) :-
     % Step, Min, Max,  state0 temp0    ODES for on/off states
   C=[ 2.0,   1.8, 2.3, 0.05, 0.1,
       [on=[-1,4,0,1],off=[-1,0,0,1],
        sw0=[-1,4,0,1],sw1=[-1,0,0,1],
        heating=[-1,4,-4,D],cooling=[-1,0,4,D]
        ]],
   in_trajectory((on,(0,2,0)),Range,C,X0),
   evolve((on,X0),C,(S,X),R).
```

Fig. 5. More Complete Model of Thermostat

the system entered the current state. The Z parameter is needed to implement the "switching" specification which states that the system waits some amount of time after the threshold is passed before switching on/off the heating element. Likewise, the time in which the system is heating/cooling before it "jumps" to the maximum/minimum value is given by a time unit. This represents a discontinuity in the model since the heating temperature is assumed to immediately rise to the maximum at the end of the element-heating period.

The `sw0,sw1` states are when the system is waiting before switching the heating element on or off. The `heating,cooling` states are when the element is warming up or cooling down. The `on,off` states are when the element is fully on or off. Observe that the ODEs for each state are specified in the variable `C` of the test procedure. Also, observe that the switching conditions are given declaratively in the `statechange` procedure. Finally, note that the system is assumed to be modeled by the following more complex family of ODEs, where the parameters $(\alpha, \beta, \gamma, \delta)$ vary from state to state:

$$\begin{gathered} A' = \alpha A + \beta + \gamma B \\ B' = \delta(B + 0.1\sin(B)) \\ T = T1 - T0,\ \ 0 \le T \le I\ \ ,\quad A(0) = A0,\ \ A(T) = A1,\ \ B(0) = 1 \\ A([0,I]), B([0,I]) \subset [-10^{100}, 10^{100}] \end{gathered}$$

The variable B represents the temperature of the heating element and we assume that the rate at which it heats and cools depends on its temperature in a non-linear fashion. The test query in Figure 6 demonstrates the use of this model to determine the value of the length of one complete cycle.

```
| ?- test2(S,(T,A,T0),H,-10),{A=2,T>0},S=on.
H = [
%state    T         A         Z          range of A in state
(on,     (0.1625...,2.300*,    0.1625...,[2.0000*, 2.3000*])),
(sw0,    (0.2125...,2.3829..., 0.050*   ,[2.2999*, 2.3829*])),
(cooling,(0.3125...,2.3858..., 0.100*   ,[2.3827*, 2.4904*])),
(off,    (0.5942...,1.8000*,   0.2817...,[1.7999*, 2.3858*])),
(sw1,    (0.6442...,1.7122..., 0.050*   ,[1.7122*, 1.8000*])),
(heating,(0.7442...,1.7002..., 0.100*   ,[1.6042*, 1.7123*])),
(on,     (0.8839...,2.000,     0.1396   ,[1.7002*, 2.0000*]))]
S = on
T =  0.8839...
A = 2
T0 = 0.1396... ?

(25080 ms) yes
| ?-
```

Fig. 6. Example of a more complex model

The following code shows a more interesting example in which the modeler is used to find all values of the ODE parameter δ in the range $[-8, -7]$ for which the system evolves to the state with S=off and $A = 2$ in exactly 0.5 seconds.

```
| ?- {D in [-8,-7]}, test2(S,(T,A,Z),_H,D),{T=0.5},narrow_all(1000000).
A = 2
D = -7.6651...
S = off
T = 0.500*
Z = 0.1874810705022... ?
(14280 ms) yes
```

5 Refining the Model

The model as presented above could be refined in several ways to more accurately reflect the real Hybrid System. In this section we discuss possible refinements.

5.1 Accurately modeling measured data

The first refinement is to observe that in order to model the physical uncertainties, we could include in our model error terms for each physical measurement. In most cases the size of the error terms will be given by the manufacturer's specification. Interval arithmetic makes it straightforward to explicitly model the error-bars in various physical measurements. We simply consider each of the physical measurements to give an interval result rather than an absolute number.

5.2 Measuring Temperature

We use $A(t)$ to describe the actual average temperature of the water in the tank. Note that this is a platonic ideal, and cannot be measured at all. We could use $M(t)$ to denote the temperature measured by the thermometer and converted by the A/D converter. We would then assume that for all $t : |M(t) - A(t)| \leq \varepsilon_{\text{temp}}$. The value of $\varepsilon_{\text{temp}}$ could be estimated from the specifications of the thermometer, and knowledge of the efficiency of the stirrer and the assumption that the temperature does not change too rapidly. To make this change in our code, we simply add the constraint above, and have jump transitions (which depend on measured values) use $M(t)$, while the flow transitions continue to use $A(t)$.

5.3 Heater Element Description

Henzinger *et al.* consider a heating element, and assume that the power transfered from the element to the water is a constant while the element is on. This is only a valid approximation when the water temperature is more or less constant.

Our model provides an improved approximation in that it includes the temperature of the heating element as part of the model. There is, of course, some variation in the heater output, and the actual power should be described by $g(t)$, which (after the power has been in the "on" position long enough to achieve equilibrium) differs from G by less than some small constant $\varepsilon_{\text{heat}}$.

We could alternatively have improved Henzinger's model by starting with a model of the heater being powered up or powered down by constraining the time $h_{\max}$ it takes the heater to get to within $\varepsilon_{\text{heat}}$ of G when turning on, or within $\varepsilon_{\text{heat}}$ of 0 when turning off, and noting that during that time, $-\varepsilon \leq g \leq G + \varepsilon$.

We could then refine this model by using a linear change from one state to the other with a sufficient error bar, so if the element was turned on at time t_0, $g(t) \in [-\varepsilon + ((t - t_0) * G - \varepsilon_{\text{heat}}), \varepsilon + ((t - t_0) * G + \varepsilon_{\text{heat}})]$ and we could also add similar error bars to the exponential decay model of the heating element which we use above. We could also use error bars for the heat loss and other parameters describing the Hybrid System.

6 Conclusions

A novel aspect of the ACLP approach to Hybrid System analysis is that it establishes a close correspondence between the semantics of a particular class of constraint programs and the behavior of systems providing several advantages:

- CLP(F) handles non-linear ODEs.
- Error bars on physical measurements are handled simply and naturally by interval techniques.
- There is no "translation" from ODEs to CLP(F) making programming simpler and less error prone.

And opening up possibilities for research in several directions:

- developing more realistic models of hybrid systems using this approach
- developing analysis techniques for such systems by adapting CLP programming analysis techniques to ACLP
- developing more efficient interpreters for CLP(F) that can handle very large complex systems and which deal with the wrapping problem
- extending this work to hybrid systems where the sensors are governed by PDEs rather than ODEs.
- developing primitive implementations of the `ode` procedures so that one does not need to use the full power of ACLP (and its accompanying inefficiencies).

This paper has demonstrated that Analytic Constraint Logic Programming provides a promising approach to modeling Hybrid Systems by providing a program whose semantics precisely match the behavior of the Hybrid System. Further research is needed to see if such an approach can be scaled up to real-life systems.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

2. A. Balluchi, L. Benvenuti, M. D. D. Benedetto, G. M. Miconi, U. Pozzi, T. Villa, H. Wong-Toi, and A. L. Sangiovanni-Vincentelli. Maximal safe set computation for idle speed control of an automotive engine. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control (HSCC 2000)*, volume 1790 of *LNCS*, pages 32–44. Springer Verlag, 2000.
3. A. E. Ciarlini and T. Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. In *ACM SIGSIM Conference on AI, Simulation and Planning (AIS '2000).*
4. V. Gupta, R. Jagadeesan, V. Saraswat, and D. G. Bobrow. Programming in hybrid constraint languages. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of *LNCS*, pages 226–251. Springer Verlag, 1995.
5. T. A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.
6. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(?):110–122, 1997.
7. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.
8. T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HyTech: Hybrid systems analyis using interval numerical methods. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control (HSCC 2000)*, volume 1790 of *LNCS*, pages 130–144. Springer Verlag, 2000.
9. T. J. Hickey. Analytic constraint solving and interval arithmetic. In *POPL'00 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 27, pages 338–351, 2000.
10. T. J. Hickey. CLIP: A CLP(Intervals) dialect for metalevel constraint solving. In E. Pontelli and V. S. Costa, editors, *Practical Aspects of Declarative Languages: PADL2000*, volume 1753 of *LNCS*, pages 200–214. Springer Verlag, 2000.
11. T. J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001. http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2001/S01-02/JFLP-A01-07.pdf.
12. J. Jaffar and J. Lassez. Constraint logic programming. In *Proceedings 14th ACM Symposium on the Principles of Programming Languages*, pages 111–119, 1987.
13. N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata revisited. In M. D. D. Benedetto and A. Sangiovanni-Vincentelli, editors, *HSCC*, number 2034 in LNCS, pages 403–417. Springer Verlag, 2001.
14. N. Lynch, R. Segala, F. W. Vaandrager, and H. Weinberg. Hybrid I/O automata. Technical Report CSI-R9907, Computing Science Institue Nijmegen; Faculty of Mathematics and Informatics; Catholic Univversity of Nijmegen, Toernooivveld 1; 6525 ED Nijmegen; The Netherlands, Apr 1999.
15. O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. de Bakker, C. Huizing, W. de Roever, and G. Rozenberg, editors, *Real-Time Theory in Practice*, volume 600 of *LNCS*, pages 447–484. Rex Workshop, Springer Verlag, 1991.
16. A. Podelski. Model checking as constraint solving. In *Proceedings of SAS'2000: Static Analysis Symposium.*
17. L. Urbina. Analysis of hybrid systems in CLP($\mathcal{R}$). In E. C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, volume 1118 of *LNCS*, pages 451–467. Springer, Aug 1996.
18. L. Urbina. The generalized railroad crossing: Its symbolic analysis in CLP($\mathcal{R}$). In E. C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, volume 1118 of *LNCS*, pages 565–567. Springer, Aug 1996.