Bjorn Carlson
Netscape Communications
bjorn@netscape.com

Vineet Gupta
NASA Ames Research Center
vgupta@ptolemy.arc.nasa.gov
ic.arc.nasa.gov/people/vgupta

Declarative programming: Programs are assertions about physical behaviours. Useful for specification -- engineers can describe individual aspects of a system to get an executable model.

Compositional: Supports logical concurrency for easy program composition, multiform time and orthogonal preemption construction as in Esterel. Allows separate programs for modules, plug-and-play design.

Expressive: Implementors can tailor constraint system to their needs. Constraints can include ODE's, transfer functions etc., allowing models to be expressed in a language familiar to engineers.

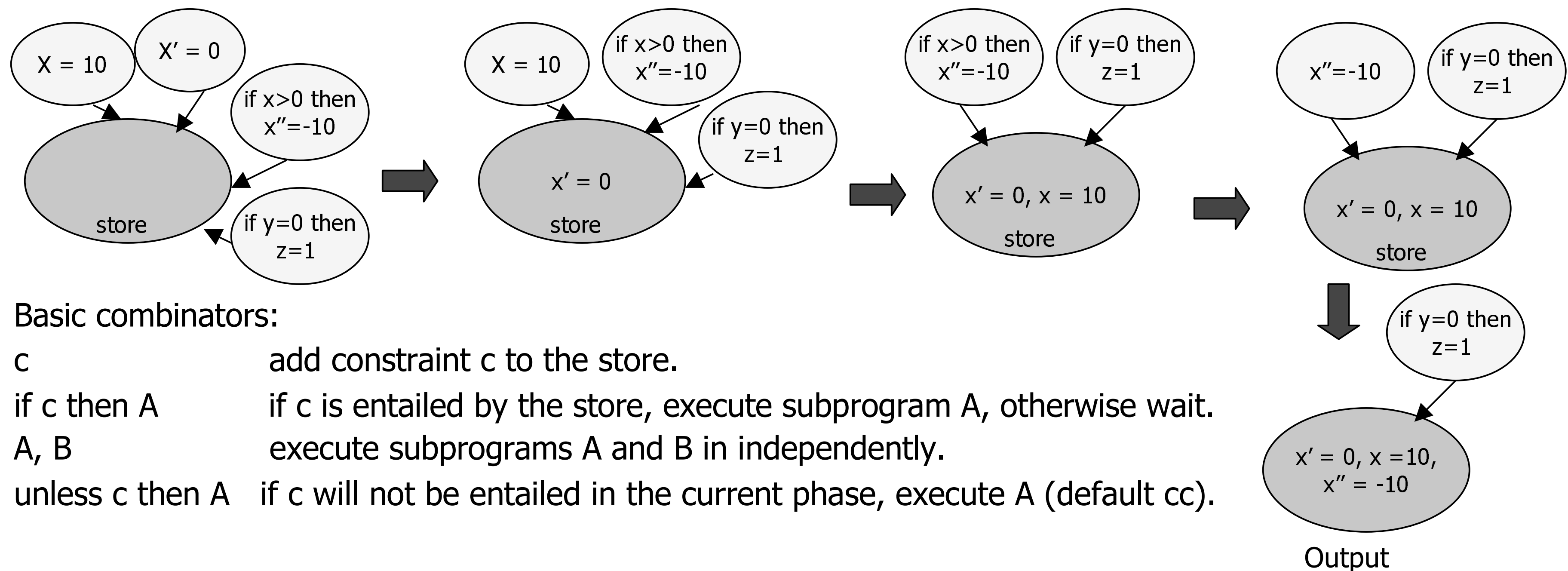
Well-developed semantics: Allows various kinds of reasoning with programs. Programs can be compiled into hybrid automata, which can be used for verification and control code synthesis.

Background: Concurrent Constraint Programming

A constraint expresses information about the possible values of variables. E.g. $x = 0$, $7x + 3y = 21$.

A cc program consists of a store, which is a set of constraints, and a set of subprograms independently interacting with it. A subprogram can add a constraint to the store. It can also ask if a constraint is entailed by the store, and if so, reduce to a new set of subprograms. The output is the store when all subprograms are quiescent.

Example program: x = 10, x′ = 0, if x > 0 then x″ = -10



Basic combinators:

c	add constraint c to the store.
if c then A	if c is entailed by the store, execute subprogram A, otherwise wait.
A, B	execute subprograms A and B in independently.
unless c then A	if c will not be entailed in the current phase, execute A (default cc).

Extending cc to Hybrid cc

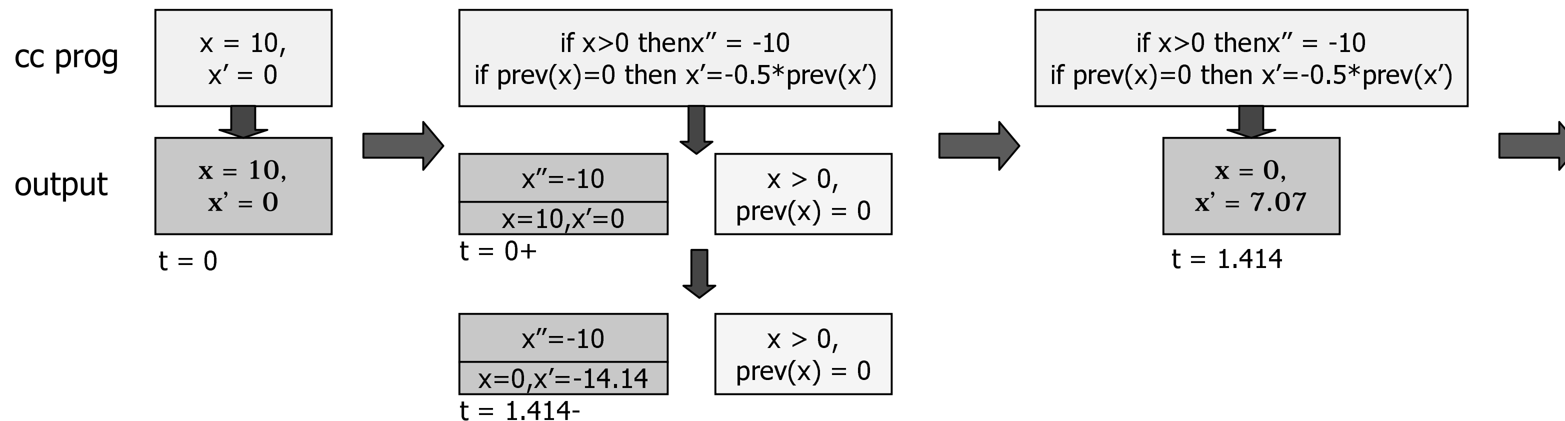
Basic assupmtion: The evolution of a system is piecewise continuous. Thus, a system evolution can be modeled a sequence of alternating point and interval phases.

Constraints will now include time-varying expressions e.g. ODE's.

Execute a cc program in each phase to determine the output of that phase. This will also determine the cc program to be run in the next phase.

In an interval phase, any constraints asked of the store are recorded as transition conditions. Integrate the ODE's in the store to evolve the time dependent variables, using the store in the previous point phase to determine the initial conditions. The phase ends when any transition condition changes status. The values of the variables at the end of the phase can be used by the next point phase.

Example program: x=10,x'=0, hence {if x>0 then x"=-10, if prev(x)=0 then x'=-0.5*prev(x')}



New combinator: hence A execute a copy of A in each phase (except the current point phase, if any)

Hybrid cc with interval constraints

- Arithmetic variables are interval valued. Arithmetic constraints are non-linear algebraic equations over these, using standard operators like +, *, ^, etc. Users can easily add their own operators as C libraries (useful for connecting with external C tools, simulators etc.).
- Object-oriented system with methods and inheritance. Methods and class definitions are constraints and can be changed during the course of a program. Recursive functions are allowed.
- Various combinators defined on the basic combinators e.g.

  do A watching c --- execute A, abort it when c becomes true

  when c do A --- start A at the first instant when c becomes true

  wait N do A --- start A after N time units

  forall C(X) do A(X) --- execute a copy of A for each object X of class C
- Arithmetic expressions compiled to byte code and then machine code for efficiency. Common subexpressions are recognized.
- Copying garbage collector speeds up execution, and allows taking snapshots of states.
- API from Java/C to use Hybrid cc as a library. System runs on Solaris, Linux, SGI and Windows NT.

The Arithmetic Constraint System

Constraints are used to narrow the intervals of their variables. For example, $x^2 + y^2 = 1$ reduces the intervals for x and y to [-1,1] each. Further adding x >= 0.5 reduces the interval for x to [0.5, 1], and for y to [-0.866, 0.866]. Various interval pruning methods prune one variable at a time.

- Indexicals: Given a constraint $f(x,y) = 0$, rewrite it as $x = g(y)$. If $x \in I$ and $y \in J$, then set $x \in I \cap g(J)$. Note: y can be a vector of variables.
- Interval splitting: If $x \in [a, b]$, do binary search to determine minimum c in [a,b] such that $0 \in f([c,c], J)$, where $y \in J$. Similary determine maximum such d in [a,b], and set $x \in [c,d]$.
- Newton Raphson: Get minimum and maximum roots of $f(x,J) = 0$, where $y \in J$. Set x as above.
- Simplex: Given the constraints on x, find its minimum and maximum values, and set it as above. Non-linear terms are treated as separate variables.

These methods can be combined to increase efficiency. For example, we use Splitting only to reduce the size of the interval of x, then use Newton Raphson to get the root quickly.

Integrating the differential equations

Differential equations are just ordinary algebraic equations relating some variables and their derivatives e.g. f = m * a″, x″ + d*x′ + k*x = 0.

We provide various integrators --- Euler, 4th order Runge Kutta, 4th order Runge Kutta with adaptive stepsize, Bulirsch-Stoer with polynomial extrapolation. Others can be added if necessary.

All integrators have been modified to integrate implicit differential equations, over interval valued variables.

Exact determination of discrete changes (to determine the end of an interval phase) is done using cubic Hermite interpolation. For example, in the example program we need to check if x = 0. We use the value of x and x′ at the beginning and end of an integration step to determine if x = 0 anywhere in this step. If so, the step is rolled back, and a smaller step is taken based on the estimate of the time when x = 0. This is repeated till the exact time when x = 0 is determined.

Example - A railroad crossing

```
always Train = (ipos, vel)[pos]{
    pos = ipos,
    always pos' = vel
},
Sensor = (pos, Sig)[]{
    always forall Train(N) do if N.pos = pos then Sig
},
Controller = (App, Lower, Exit, Raise){
    always {
        if App then wait 5 do Lower,
        if Exit then do wait 5 do Raise watching App
    }
},

Gate(), Sensor(S1, 1000, App), Sensor(S2, -100, Exit),
Controller(App, Lower, Exit, Raise),
Train(T1, 10000, -50),
wait 220 do Train(T2, 10000, -50),
```

```
Gate = (){
    g = 90,  // initial position of gate
    always {
        cont(g),
        if (g=0 || g=90) then   // steady state
            do always g' = 0 watching (Raise || Lower),
        if Raise then           // raising
            do always g' = 10 watching (Lower || g=90),
        if Lower then           // lowering
            do always g' = -10 watching (Raise || g=0)
    }
}
/*  program for railroad crossing, see Alur et al LNCS
    736 for description.  Trains T1 and T2 are instances
    of class Train, which has two arguments and one
    property, pos. T1.pos denotes the position of the
    first train etc.  S1 and S2 are two sensors
    indicating approach and exit of the trains.  The
    controller reads the sensors and sends messages
    to the gate.  The gate position varies between 0
    and 90 */
```

Example: A Schmitt trigger

```
a = 100, a' = -5,
if (a >= 12) then St1(),
if (a <= -12) then Stn1(),
if (a <= 10 && a >= -10) then St0(),

/* the state definitions */
always {
    St0 = (){
        do always t = 0     /* the activity condition  */
        watching (a >= 12 || a <= -12),
        /* the exit conditions */
        when (a >= 12 || a <= -12) do
            /* the transitions */
            if a >= 12 then St1() else Stn1()
    },
    St1 = (){
        do always t = 1 watching (a <= 10),
        when (a <= 10) do St0()
    },
    Stn1 = (){
        do always t = -1 watching (a >= -10),
        when (a >= -10) do St0()
    }
},

always {a'' = -t}


/*  This program illustrates a Schmitt trigger used in
    bang-off-bang control.   It is a three state
    automaton,  which corresponds to the functions
    St0, St1 and Stn1 above.  This example is given to
    show how an automata style program can be
    written in Hybrid cc */
```

Example: The Solar System

```
Planet = (m, initpx, initpy, initpz, initvx, initvy,initvz) [px, py, pz,
      mass]{
   px = initpx, py = initpy, pz = initpz,
   px' = initvx, py' = initvy, pz'=initvz,
   always {
     mass := m,
      px'' := sum(g * P.mass * (P.px - px)/((P.px -px)^2 + (P.py -
       py)^2 + (P.pz -pz)^2)^1.5, Planet(P), P != Self),
      py'' := sum(g * P.mass * (P.py - py)/((P.px -px)^2 + (P.py -
       py)^2 + (P.pz -pz)^2)^1.5, Planet(P), P != Self),
      pz'' := sum(g * P.mass * (P.pz - pz)/((P.px -px)^2 + (P.py -
       py)^2 + (P.pz -pz)^2)^1.5, Planet(P), P != Self)
   }
},
always pTg := 8.88769408e-10,
//Coordinates, velocities on 1998-jul-01 00:00
Planet(Sun, 332981.78652,
    -0.008348511782195148, 0.001967945668637627,
      0.0002142251001467145, -0.000001148114436325, -
      0.000008994958827348018, 0.00000006538635311283),
Planet(Mercury, 0.0552765501,
    -0.4019000379850893, -0.04633361689674035,
      0.03239207992750,-0.002423875040887606, -
      0.02672168963230259, -0.001959654820981497),
```

```
Planet(Venus, 0.8150026784,
      0.6680247657009936, 0.2606201175567890,
      -0.03529355196193388,  -0.007293563117650372,
      0.01879420958390879, 0.0006778739390714113),

Planet(Earth, 1.0,
      0.1508758612501242, -1.002162526305211,
      0.0002082851504420832, 0.01671098890724774,
      0.002627047365383169, -0.0000004771611907632339),

/*    A  fragment of a model for the Solar system.  The remaining
      lines give the coordinates and velocities of the other planets
      on July 1, 1998.  The class planet implements each planet as
      one of n bodies, determining its acceleration to be the sum of
      the accelerations due to all other bodies (this is defined by
      the sum constraint).  Units are Earth-mass, Astronomical
      units and Earth days.*/
```

Results of simulation:

Simulated time - 3321 units (~9 years).

CPU time = 55 s.

Accuracy: Mercury < 4°, Venus < 1°, other < 0.0001° away from actual positions after 9 years.