The hcc Programmer's Manual

Björn Carlson *   Vineet Gupta†

Contents

*Netscape Communications, 650 E. Middlefield Road, Bldg.1, Mountain View CA 94043; bjorn@netscape.com
†Caelum Research Corporation, NASA Ames Research Center, M/S 269-2, Bldg 269, Rm 127, Moffett Field CA 94035, vgupta@ptolemy.arc.nasa.gov

1 Introduction

This manual describes an implementation of the hcc programming and modeling language, and provides several examples of hcc programs. A description of the details of the implementation is given in [CG98].

hcc is a language for modeling and reasoning with hybrid systems, which are systems that exhibit discrete and continuous behavior. A typical example of such a system is the heating system of a room. The temperature of a room changes continuously, according to some differential equations governing the heat inflow and outflow. However, an action like someone opening a window, the thermostat switching off the furnace, or someone leaving the room, changes the equations in an instant (idealized). A good model of such a system captures both the continuous and the discrete behaviors, which is why a language such as hcc is useful.

Section 2 provides a quick tutorial to hcc, to enable the reader to start writing simple hcc programs. Later sections discuss the syntax and semantics, together with more elaborate programming examples. Also, the debugger for hcc is described. Finally, we list the Java API for hcc, and the distribution packages provided.

hcc is still under active evolution, please send any problems to `vgupta@ptolemy.arc.nasa.gov`, we will try to respond very quickly.

2 A hcc Tutorial

The simplest possible hcc program is one which only states some constraints. The constraints in hcc are of two kinds: arithmetic and symbolic. The first group are based on interval variables, and interval arithmetic, so, for example, a constraint $x \geq 4$ adds the interval constraint $x \in [4, \infty]$. Expressions like $x + y$ are evaluated using interval addition applied to the current intervals of x and y.

The symbolic constraints include constraints like $X ='' abc''$, and $X = (x, y)x = y^2$, where the latter defines X to be a lambda-abstraction such that $X(t_1, t_2)$ adds the constraint $t_1 = t_2^2$.

For example, here is a program which defines a unit circle:

```
x^2+y^2=1
```

As can be seen, the conventional programming syntax for equations is used, where `x` and `y` are variables. Given the above program, and the constraint `x=1`, `y` is bounded to 0, i.e. for any value of `x` (or `y`), the corresponding value of the other variable is computed, such that the coordinate pair lies on the unit circle. If an impossible value is given one of the variables, `x=10` say, the program fails.

Given a set of constraints like the above, one can check whether a constraint is implied. Hence, in the above program, one can check for when `y` is bounded by the upper semicircle by:

```
x^2 + y^2 = 1,
if (y>= 0 && y<=1) then A
```

where `A` is some statement that is hence issued as soon as y lies between 0 and 1. The convention in hcc is that numerical variables begin with a lowercase letter, and other variables begin with an uppercase letter. Note that by the circle equation alone, `y` is bounded between -1 and 1, so the test can be simplified to `y>=0`.

So far, we have only discussed the constraint part of hcc. Behavior extended over time is generated by differential equations and operators that enforce constraints continuously.

Suppose we want a simple program to simulate temperature, `t`. We first have to give an initial value for the temperature. Thus, the code snippet

```
t = 65
```

is added.

This sets the value of `t` to 65 at the first instant of time. It does not say how `t` will change. Note in particular that it does not say that `t` will remain 65 after the first instant. In fact, `t` will not have any value, unless some other hcc statement sets it.

Now we make `t` increase at the rate of 1 degree per time unit.

```
t = 65,
hence t' = 1
```

The `hence` construct sets up its argument to be executed now and forever on. In this case it sets up the derivative of `t` to be 1, forever! Thus, after 1000 time units, we will have `t = 1065`. This is unrealistic, so instead we say that when the temperature reaches 75, it stays constant.

```
t = 65,
do hence t' = 1 until t = 75,
when t = 75 do hence t' = 0
```

The second line of the program is an instance of the construct **do** A **until** c — this allows A to run as long as c is not true. Thus, in this case the differential equation `t' = 1` is active only as long as `t = 75` is not true. When it becomes true, the statement `hence t' = 1` is killed, and the constraint on `t'` is removed. The third line waits for the constraint `t = 75` to become true, and sets t' to be 0 from then onwards.

If we sample the values of `t` by adding the statement `sample(t)`, here is what we obtain (we set the program to abort after 100 time units).

```
0.0   65.0
0.1   65.1
0.6   65.6
3.1   68.1
10.0  75.0
10.0  75.0
10.1  75.0
10.6  75.0
13.1  75.0
25.6  75.0
88.1  75.0
100.0  75.0
Elapsed Time:
100.0
```

The interval between sample points is defined by the step length of the integrator, but can be controlled by setting the maximum step length the integrator is allowed to take.

As another example, consider an ecology of predators and prey. There are two sets of equations, one for controlling the prey, and another for controlling the predators. Also, we add a discontinuity by changing the differential equation for predator growth when the ratio between prey and predators is too small.

The program sets up the initial state to contain 8 (or 8000, say) prey and 2 (2000) predators. The population of prey is allowed to evolve by using a single differential equation. The predator population, on the other hand, evolves more aggressively when the ratio between the prey and predators is more than 2. Hence, the initial state is set by the equations:

```
py=8,
pd=2
```

The evolution is defined by two standard differential equations, describing the interaction between the two species.

```
py' = 0.08*py - 0.04*py*pd,        // prey growth
pd' = 0.2,
cont(pd),
if (pd >= 0.5*py) then // predator growth
   pd' = -0.1*pd+0.02*py*pd
else
   pd' = -0.06*pd+0.02*py*pd
```

The conditional statement checks at any moment in time which of the two differential equations to use for the predators, which thereby simulates discrete changes in the population growth.

Since `pd` is not set continuously by a single differential equation we have to declare the variable to be continuous, using `cont(pd)`, meaning that even though, temporarily, there are no equations for `pd`, there will be, hence the limiting value of `pd` from the previous interval is the current value for the variable, i.e. assume the second equation is used initially. Then at some point in the future, when there is only twice as many prey as predators, the other equation takes over. But then, for a moment, there is no value for `pd`, since the equations are swapped, and we thus have to declare `pd` as a continuously changing variable, even though we discretely changes its equations.

The `sample(pd)` statement tells the interpreter to sample the values for `pd` throughout the computation. These values can then be printed from the debugger or accessed through the API (see the section on Java). The final program is thus:

```
py=8,  // prey
pd=2,   // predator
pd'=0.2,
always py' = 0.08*py - 0.04*py*pd,  // prey growth
always {
  cont(pd),                         // predator growth
  if (pd >= 0.5*py) then
     pd' = -0.1*pd+0.02*py*pd
  else
     pd' = -0.06*pd+0.02*py*pd
},
sample(pd), sample(py)
```

where the `always` is used to issue the differential equations at every point in time.

For an example of how the two populations fluctuate see the plotting in Figure 1. The local valleys of the curves are consequences of the equation switching.

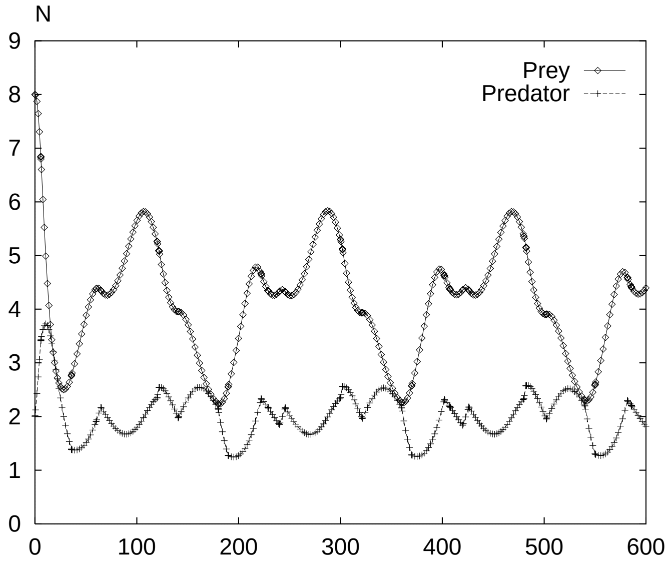

Figure 1: Example run of an ecology using nonlinear differential equations

3 The hcc Programming Language

The basis of hcc[GJS98, GJSB95] is Default cc[SJG96], which is a simple untimed language obtained by extending cc with defaults. A Default cc program consists of a set of program statements (formulas). The smallest statement is a constraint, $c(x_1, \ldots, x_k)$, which forces a relationship to be true between the variables x_i, $1 \leq i \leq k$. Typical constraints are linear or nonlinear arithmetic constraints over the reals, such as $x^2 + y^2 = 1$, which initially bounds x and y to be between -1 and 1, or atomic constraints such as X or $X = "cheerio"$. Also, in hcc, ordinary differential equations define the derivatives wrt time, i.e. $x' = x$ is equivalent to $\frac{dx}{dt} = x$.

Computation in Default cc is done by statements adding constraints to the *store*. Flow control is done by the conditional statements **if** c **then** A and **unless** c **then** A. The first of these, checks whether c is true in the current store, and if so it reduces to A. If c is false, the statement is ignored, and otherwise the statement is suspended.

The second of these reduces to A only if c is never going to hold in the current store. Hence, if c is true, the statement is ignored, and if c is false, it reduces to A immediately. However, in the case when it cannot be determined if c is true or false, a branching point is made, where it is either assumed that c will never become true, or it is assumed that c will indeed become true eventually. Thus, the statement **unless** c **then** A may lead to nondeterminism, and possible future failure (when, for example, under the assumption that c will not become true, c in fact becomes true).

So for example let the store consists of the following constraints:

$$x \in [3, 7], y \in [4, \infty], z \in [4, 4].$$

Then the statement **if** $x > 2$ **then** $y = 7$ reduces to $y = 7$, which is added to the store to yield $x \in [3, 7], y \in [7, 7], z \in [4, 4]$. The statement **unless** $y = 3$ **then** $m = 1$ now reduces to $m = 1$, because it is clear that y can never be 3. However, there is a subtle point here — the statement **unless** $x = 4$ **then** $m = 1$ can also reduce to $m = 1$, *if there are no other statements around to supply any additional information making* $x = 4$. Thus, **unless** c **then** A is *not* the same as **if** $\neg c$ **then** A — the second statement *requires* the presence of $\neg c$ to reduce to A, while the first can reduce to A in the absence of c. Hence, **unless** $\mathtt{B}ulb = "on"$ **then** $\mathtt{B}ulb = "off"$ says that unless some statement explicitly says `Bulb = "on"`, we have `Bulb = "off"`, i.e. the default is `Bulb = "off"`, which is why the name Default cc is used.

3.1 Computational model

Intuitively, hcc extends Default cc across time. We can think of a hcc program as determining some Default cc program to be run at each real time point, based on the statements in the hcc program, and the history of the computation. A hcc program therefore contains code to be run at points and continuous code that brings the computation from one point to the next relevant point.

A hcc program starts at some (arbitrary) point, computing the initial state of the execution, which is done by executing a Default cc program (which adds constraints, checks them, and runs default rules). When the point program reaches a stable point, the execution prepares for an interval computation by executing all the statements that are to be executed continuously (in a fresh store). This is again a Default cc computation. However, this time all differential equations are collected, as are all the constraints that, if they change status in the future, may make the program behave differently.

At the stable point of the continuous statements, a simultaneous integration of the differential equations is begun which is only terminated when either one of the constraints changes status (e.g. goes from false to maybe true), some constraint inconsistency occurs, or when the maximum amount of simulation time has elapsed (a given parameter). When the integration terminates, a new point computation is started, and the cycle is repeated.

Hence, the model is best seen as follows. We are given a program P (or a subset thereof) to be executed at every point on the real line. In any point computation, the program that is executed is considered to be done so instantaneously. There is no immediate next point after the current one, and we need some way to proceed with the computation. But since P is at a stable point, P does not actually produce any actions until some point in the future is reached. The idea is then to integrate the differential equations to the nearest point in the future where something different *may* occur in P. This is the *interval phase*. The interval phase thereby defines a segment of the real line where some of the dependent variables (the ones with a constrained derivative) are open. The same variables are closed when the next point is entered.

For example, immediately following a point in which the equations $x = 0, x' = 1$ are true, $x > 0$ is true, since no matter how small the step is x will be greater than 0 because of $x' = 1$, i.e. $x \in (0, \infty)$ is true. Suppose further that the program takes action when $x \geq 4$. Then the interval phase integrates x until $x = 4$.

The two most important means in hcc to create behavior over time are the use of the combinator **hence** , and the use of differential equations. By issuing a statement A as **hence** A, the statement will intuitively be run continuously and forever, starting in the open interval following the current point. By asserting **hence** $x' = 1$, for example, x will always be equal to the current time (simulation time, that is). The statement **hence** $x = 1$, on the other hand, sets x to be 1 forever. It follows that **hence** $\{x' = 1, x = 1\}$ fails.

4 Constraints

We now give the syntax and semantics of the constraint system implemented in our system, and also give an idea of the kind of inferences made.

Note that our inference algorithm is *not* complete! There are plenty of obvious deductions it will not make, e.g. the vocabulary here is sufficiently expressive to allow arbitrary differential equations, which we cannot solve analytically but only numerically. Therefore, we cannot decide whether two variables are equal given only their differential equations, e.g. in the program:

```
y = 1, always y' = y,
x = 0, always {x' = 1, z = exp(x)}
```

we know that `y = z` is always true, but this will not be inferred.

There are several types of constraints in hcc. In some cases the constraints that can be added to the store are different from the constraints for which the store can be queried, and we will point out the difference.

4.1 Continuous Constraints

This is the class of constraints which is used most often. These constraints assert arithmetic equalities and inequalities over terms. The syntax is as follows:

$$
\begin{array}{rcl}
ContConstr & ::= & Term\ RelOp\ Term \mid cont(LVariable) \mid lcont(LVariable) \\
RelOp & ::= & = \mid >= \mid <= \mid := \mid :== \\
Term & ::= & LVariableExpr \mid Constant \mid Term\ BinOp\ Term \mid UnOp(Term) \\
 & & \mid dot(Term, Num) \mid Term' \mid (Term) \\
LVariableExpr & ::= & LVariable \mid UVariable.LVariableExpr \\
BinOp & ::= & + \mid - \mid * \mid / \mid \hat{} \\
UnOp & ::= & - \mid log \mid exp \mid prev
\end{array}
$$

LVariables are strings which start with a lowercase character. There is no limit on the length of variable names. *Constants* are floating point numbers, and any floating point number accepted by C is accepted. An *LVariableExpr* that is not an *LVariable* is a property expression (see Section 4.2). The semantics of most constructs are as expected. For example, $exp(x)$ is the exponential function e^x, $Term'$ denotes the derivative of $Term$, and $dot(Term, n)$ denotes the nth derivative of $Term$, where n is a natural number.

The relop := is a variant of =, it assumes that the constraint will have no effect on the variables on the right side of the equality, and it does not propagate the effects of the constraint on these variables. Thus it behaves like an assignment statement. It is provided mainly for efficiency reasons. It is also useful in default resolution so sometimes programs which take a long time to run can be sped up considerably by changing a = to a :=.

A variant of this is the :== operator — if the left hand side of a :== is variable expression, then this variable expression is substituted for by the right hand side in any other constraint, allowing better constraint propagation. For example, if we have $x^2 = y, y = x$, then no pruning happens. However, if we have $x^2 = y, y := x$, then this produces the equation $x^2 = x$, which makes x be positive (note that Infinity is a "solution" to this equation). If the left hand side is not a variable expression, these two behave identically. Note that we often prefer := to :==, since the substitution of the right hand side means that it must be evaluated repeatedly, whereas := evaluates it once.

`cont(x)` asserts that x is continuous. Thus, *always* $cont(x)$ asserts that x is always continuous. The effect of asserting $cont(x)$ in a point phase is that the value of x in the point phase is set to the value of x at the end of the previous interval phase. $lcont(x)$ does the same, but for intervals — it sets the value of x at the beginning of the interval phase to be the same as in the point phase. Note that asserting $x' = 3$ automatically asserts that x is continuous, as differentiability implies continuity.

Some other useful declarations are as follows:

- `realvar(x, y, ...)` lets the interpreter know that $x, y, \ldots$ are real-valued rather than interval valued. The arithmetic evaluation of expressions containing these variables can take advantage of this fact, by doing much simpler calculations. Note that this declaration should appear before $x, y, \ldots$ occur in any arithmetic expressions, for maximum speedup (otherwise the prior expressions will continue to treat them as interval valued).

- `#define x e` asserts that x is a macro with the value e, where e is any $Term$. This has the effect of textually substituting e for x everywhere in the program.

- `#domain x [l, u]` makes the initial value of x be $[l, u]$ in every phase, it can be pruned further by constraints imposed in the phase.

The variables in these declarations should be global variables, as these assertions are done at compile time, whereas scopes inside functions and classes are resolved at runtime.

Ask constraints are slightly different from tell constraints:

$$
\begin{array}{rcl}
AContConstr & ::= & Term\ ARelOp\ Term \\
ARelOp & ::= & = \mid >= \mid <= \mid < \mid > \mid !=
\end{array}
$$

All variables referred to in continuous contraints are stored as closed intervals, representing their upper and lower bounds. The initial value is $[-\infty, \infty]$. Thus the constraint $x \geq 3$ would make $x \in [3, \infty]$. We currently use linear programming, interval splitting and interval-based Newton-Raphson to propagate the effects of imposing constraints. Note that the constraint engine in not complete — thus if the constraint $x = y$ is imposed, then this will set the values of x and y to be the intersection of the two previous values. However, if the previous values were $[-\infty, \infty]$, then this will not have any effect. Now if we ask $x = y$, this will be answered as unknown. Later if we set $x = 5$, then y will also be set to 5, and the asking $x = y$ will be answered positively.

4.2 Non-arithmetic Constraints

These are constraints on non-arithmetic variables — these variables do not change their values continuously. The syntax for such constraints is given by

$$
\begin{array}{rcl}
DConstr & ::= & UVariableExpr \mid UVarExpr = DExpr \\
UVariableExpr & ::= & UVariable \mid UVariableExpr.UVariableExpr \\
DExpr & ::= & UVariableExpr \mid prev(UVariableExpr) \mid String \\
 & & \mid (VarList)HccAgent \mid (VarList)[VarList]HccAgent \\
 & & \mid UVariableExpr(VarList)[VarList]HccAgent \\
VarList & ::= & UVariable \mid LVariable \mid VarList, VarList
\end{array}
$$

A $UVariable$ is a string starting with an uppercase character. A $UVariableExpr$ that is not a $UVariable$ is a property expression (see below). $HccAgent$ is any hcc agent, defined below. A $prev(UVariableExpr)$ is similar to the corresponding expression for arithmetic variables — it gives the value of the $UVariableExpr$ in the previous phase.

A $DConstr$ given as a $UVariableExpr$ is a signal constraint. These constraints are typically used to communicate to some other statement of code that a certain state is reached, and to assert properties, i.e. to let other pieces of programs know that a certain property is true.

A String is any string of characters enclosed within double quotes. There is no limit on the length of variable names or strings. These are mostly used for properties of objects — i.e. `Switch = "on"`.

The constraint $X = (A, x)Agent$ sets up a closure definition. Its effect is the same as setting $X = \lambda A.\lambda x.Agent$. Note that the $Agent$ is activated only when the closure is called. Closures can be recursive — the following is the definition of the factorial function:

```
 P = (n,m,Q){
  if n > 0 then
     new m1 in {Q(n-1, m1, Q), m := m1*n}
  else
     m = 1
},
Fact = (n,m) { P(n,m,P) },
Fact(10, m)
```

The constraint $X = (m, n)[A, m]Agent$ sets up a class definition. The first VarList is the list of parameters, whereas the second is the list of properties. These properties are defined for each object of the class, and can be used in other expressions, as described above. A property of a class can be constrained by a closure definition as described above, this is then a method for the object.

The creation of objects is described in the next section. Each object has a name, which is mapped to a property named $Self$. A property x can be referred inside the Agent as x, but from outside it must be referred as $Name.x$. $Self.x$ always refers to x in the Agent. If the optional form $D = B(\ldots)[\ldots]A$ is used, the class D is derived from B by implicitly appending at the beginning all the constructor arguments and properties of B to D, and conjoining the body of B to A.

Ask constraints for the above have a similar syntax, except that the Relop $! =$ is also allowed. Note that asks do not make sense for closure and class definitions, as we do not perform any unification on these. Thus asking $A = (M)Agent$ will always answer unknown.

We also provide a new kind of ask constraint $UVariableExpr <|>|>=|=< UVariableExpr$. This merely checks the identities of these variables, and the ordering on these is some arbitrary but total ordering. This is useful for meta-constraints and forall expressions, described below.

Meta-constraints are constraints which reduce at runtime to continuous constraints. These are given by

$$\begin{aligned} MetaConstr &::= Term = MetaOp(Term, UVariable(UVariable), AskConstraint) \\ MetaOp &::= sum \mid prod \end{aligned}$$

A meta-constraint of the form $totalwt = sum(wt, HeavyObj(X), X.Loc = “Here'')$ will create the expression $totalwt = N_1.wt + N_2.wt + \ldots$, where $N_1, N_2, \ldots$ are all the objects of class $HeavyObj$ which satisfy the constraint $X.Loc = “Here''$. This expression is created once in each phase. We delay the formation of this expression as long as possible, to allow the creation of all the objects. This means that in the execution, we postpone these constraints until the agent queue becomes empty, but before any default resolution is done.

4.3 Boolean Constraints

We also allow boolean combinations of ask constraints:

$$\begin{aligned} BoolConstr &::= Atom \mid StrConstr \mid ContConstr \mid BoolConstr \,\&\&\, BoolConstr \\ &\quad\mid (BoolConstr) \\ ABoolConstr &::= Atom \mid StrConstr \mid AContConstr \mid ABoolConstr \,\&\&\, ABoolConstr \\ &\quad\mid ABoolConstr \,||\, ABoolConstr \mid (ABoolConstr) \end{aligned}$$

These work as expected. Evaluation is from left to right.

5 The Control Constructs and Execution of hcc

hcc operates alternately in point and interval phases. It starts off in a point phase. When all the computation it can do in this point phase is completed, it starts the interval phase. During the point phase, along with the other computation, a set of programs is collected to be executed in the succeeding interval phase. This program is executed in the interval phase.

When all computation has been completed in the interval phase (and the programs to be executed in the next point have been collected), the interpreter collects all the continuous constraints imposed in the interval phase, and simulates them numerically (using some integration method, for example Euler or 4th order Runge Kutta with adaptive stepsize). The initial conditions for this simulation are given by the values of the variables in the previous point phase. It also collects all the asks that were asked in the interval computation, and stops the computation when the status of any of them changes (i.e. one goes from "false" to "possibly true"). Then the point phase is started.

The following control constructs are currently available in hcc. This list will continue to grow.

c. This construct tells a constraint c to the store. The only valid constraint relations for c at this point are $=$ and $\geq$. If adding c causes an inconsistency, then this is treated as a failure, and will cause backtracking to happen (see defaults).

$X(t_1, \ldots t_n)$. This is a function call or object creation, depending on whether X is bound to a closure or a class definition. t_i are the arguments, they can be any terms of the correct type (i.e. arithmetic terms if the corresponding parameter is an Lvariable, DExpr's if it is a UVariable). If X is a class definition, then the first argument must be a UVariable, this is the name of the created object, and inside the body of the object is bound to the property Self. Thus, in this case the number of arguments is one more than the number of parameters, i.e. a constraint $X = ()[]A$ is used for object creation as $X(I)$, where I is thus bound to an instance of X, and A added to the runtime queue.

The method of variable passing is textual substitution — t_i replaces each occurrence of the parameter p_i.

A, B. Any set of statements separated by a "," are treated as a parallel composition of statements. Each statement acts independently. No assumption should be made about the order of execution of these statements.

if c then A. This statement asks the store if the ask constraint c is true in the store. If so, this statement immediately reduces to A. Otherwise it is suspended. Interval constraints can use $>$ and $\neq$ for the relation as well.

In a point phase, a constraint $c(x_1, \ldots, x_k)$ is ruled true if any consistent assignment to x_i, $1 \leq i \leq k$, makes c true. For example, $x = y$ is ruled true iff $x, y \in [a, a]$ holds in the current store, for some real number a, and $x \geq 3$ is ruled true iff $x \in [a, b]$ holds, where $a \geq 3$.

In an interval phase, a constraint $c(x_1, \ldots, x_k)$ is ruled true if it is true in the current point, *and*, if c is differentiable, and c', c'', ..., are all ruled true. For example, the constraint $x = y$, where $x, y \in [1, 1]$ and $x', y' \in [1, 1]$ hold, is ruled true, but the constraint $x = y$, where $x, y \in [1, 1]$, $x' \in [1, 1]$, and $y' \in [2, 2]$ hold is ruled false (assuming no constraints on x'', y'', etc.

unless c **then** A**.** This statement is the default assumption statement from default logic. It has two execution paths. The first path is executed assuming that c will not be true in this phase. Then this statement behaves like A. If at some point during the phase c becomes true, then this path is dismissed, and the execution backtracks to the point where the choice was made, and the other path is taken. The assumption is that c will become true on this path. This statement itself does nothing. If c becomes true, then this path remains, otherwise the statement has no possible behavior, and it fails (leading to more backtracking).

For example, the statement `unless A then A` has no execution path, and thus always fails. On the other hand `unless A then B, unless B then A` has two execution paths, and the interpreter nondeterministically selects one of them.

Note that **unless** c **then** A is not equivalent with **if** c_n **then** A, where $c_n \equiv \neg c$. For example, the program `if x > 3 then` A will do nothing, if no other programs are present. However, `unless x =< 3 then` A will do A under the same circumstances. It is recommended that programmers always use `if` c `then` A whenever that will suffice. Backtracking is computationally expensive.

new X **in** A**.** This statement creates a new variable X and this is used wherever X occurs in A. This is useful for local scoping. Thus in the program `new x in x=0, new x in x=1` the two x's refer to different variables.

hence A**.** All the previous combinators act only in the current phase. **hence** A is the main combinator used to extend the behavior of a program across time. **hence** A in a point phase does nothing, except adding itself to the set of statements to be executed in the next interval phase. In an interval phase, **hence** A executes (a new copy of) A in the current phase, and adds both A and **hence** A to be executed in the next point phase. Thus once **hence** A is activated, A will be executed in all future phases. (Note that **hence hence** A has the same behavior as **hence** A.

always A**.** A minor variant of **hence** A, it also executes A in the point phase in which it is started.

when c **do** A**.** This construct waits for the first instant when c becomes true, and then executes A in that phase. Note that the first instant when c is true is well-defined only if it is a point phase — if c is true for the first time in an interval phase, then this construct does not do anything.

do A **watching** c**.** This construct behaves like the program A, until c becomes true. In the first phase when c becomes true, it stops A or whatever statements have been added to the current set of programs because of the execution of A. A variant of this is **do** A **hencewatching** c, which behaves exactly the same except that it does not check for c at the first instant when it is executed.

do A **until** c**.** This is almost equivalent with **do** A **watching** c — it however also does A in the first phase in which c becomes true. Thus it allows c to be generated by the A (this would cause a causal loop in watching). It is also more efficient — A is immediately started, and at the end it can be determined whether the agent A needs to survive into the next phase. A variant of this is **do** A **henceuntil** c, which behaves exactly the same except that it does not check for c at the first instant when it is executed.

while c **do** A**.** This is the dual of the previous construct, behaving like A while c remains true.In the first phase when c is no longer true, it stops A or whatever statements have been added to the current set of programs because of the execution of A. Note that this is a positive expression — it fires as soon as c become true in the current phase — so it is more efficient than the watching construct, which may have to go through the default resolution.

if c **then** A **else** B**.** This is equivalent with **if** c **then** A, **if** $not(c)$ **then** B.

time c **on** A **or time** A **on** c**.** This is the construct for undersampling time. It keeps running A while c is true, but in all the phases where c is not true, it suspends the execution of A. It is not the same as **always if** c **then** A, as the latter starts a new copy of A while "time" does not start a new copy.

wait N **do** A . This construct waits exactly N time units and then starts A. Note that the N time units are this constructs notion of time — thus if it is inside a **time** construct, then this will mean N time units during which c is true.

evalexpr $Variable = Expression$ **in** A**.** This computes the value of the $Expression$, and replaces every occurrence of $Variable$ in A with this value. Note that $Variable$ is a new variable, and its scope is restricted to A, just as in **new**. If there is any variable in $Expression$ that has not been instantiated, it suspends on it. This is useful when we want to get the value of a variable: thus a counter could be set up as follows — this allows us to set up a variable, and at any time later increment its value. Note that having `val = prev(val) + 1` is incorrect, as this will increment the counter in every point phase, and will actually produce an inconsistency in an interval phase (where $prev(x)$ is treated as x).

```
always CounterClass = (init)[val, Inc]{
   new Changed in {
        do always val = init
        watching Changed,
        always Inc = (){
            Changed,
            evalexpr tmp = prev(val) in
                do always val = tmp + 1
                hencewatching Changed
        }
   }
}
```

forall $C(X)$ **do** A**.** This is the universal quantifier — it creates an instance of A for each object of the class C. X is bound to the name of that instance, and X can be used in A.

sample$(x_1, \ldots, x_n)$ **and labeling**$(x_1, \ldots x_n)$**.** **sample**$(x_1, \ldots, x_n)$ sets up $x_1, \ldots, x_n$ to be sampled, and their values are then reported. The sampling frequency is dependent upon a constant set up in the runtime environment (see below). **labeling**$(x_1, \ldots x_n)$ narrows down the intervals of x_i to points, and can be used to enumerate all solutions of an equation. $x_1, \ldots, x_n$ can be LVariableExprs (see above).

realvar$(x_1, \ldots, x_n)$. Normally all continuous variables in hcc are interval valued. This introduces an inefficiency in programs where intervals are not needed. This declaration makes these variables be real valued, saving up half time in arithmetic. $x_1, \ldots, x_n$ can be LVariableExprs (see above).

6 Example programs in hcc

6.1 The many body problem

This program simulates the many body scenario. The 2-d space contains many planets, which interact with each other via gravitational forces.

```
Planet = (initvx, initvy, initpx, initpy, m)[px, py, mass]{
        px = initpx, py = initpy,
        px' = initvx, py' = initvy,
        always {
                mass = m,
                px'' := sum(F.x, Force(F), F.Target = Self),
                py'' := sum(F.y, Force(F), F.Target = Self)
        }
},
always Force = (S, T)[Target, x, y]{
        Self.Target = T,
        new tmp in {
                tmp := g * S.mass /((S.px - T.px)^2 + (S.py -T.py)^2)^1.5,
                x := tmp * (S.px - T.px),
                y := tmp * (S.py - T.py)
        }
},
always forall Planet(X) do forall Planet(Y) do
        if (X != Y) then new F in Force(F, X, Y)
```

The first statement defines the class Planet. The initial velocity and position are parameters, as is the mass. The position and velocity of the planet are determined by the forces of gravity acting on the planet – there is one such force for each other planet in the system, as asserted by the forall statement. Each force sets its Target parameter, and the planets then sum up all the forces whose target is themselves. Note the use of the assignment statement — in this example we can get a speedup of 10 times by using assignment instead of equality.

We can now add to this system any number of planets, to get a simulation. Thus the following simulates the solar system.

```
always g = 0.00011792,
Planet(Sun, 0, 0.00205455, 0, 0, 332948.34),
Planet(Mercury, 0, 8.1987, 0.4666, 0, 0.055271),
Planet(Venus, 0, 7.3365, 0.72794, 0, 0.81476),
Planet(Earth, 0, 6.179, 1.0167, 0, 1),
Planet(Mars, 0, 5.006367, 1.6657754, 0, 0.10745),
Planet(Jupiter, 0, -2.8928, -4.9505, 0, 317.94),
Planet(Saturn, 0, 1.9244, 10.0695, 0, 95.181),
Planet(Uranus, 0, 1.36838, 20.088, 0, 14.535),
```

```
Planet(Neptune, 0, 1.13635, 30.3155, 0, 17.135),
Planet(Pluto, 0, 0.77826, 49.33957, 0, 0.0021586)
```

If we wanted to create a check for the numerical accuracy of such a system, we could do this by making sure that momentum and energy are conserved. For momentum, we add the following lines of code:

```
always {
        totalxmomentum = sum(P.mass * P.px', Planet(P), 1=1),
        totalymomentum = sum(P.mass * P.py', Planet(P), 1=1)
},
sample(totalxmomentum), sample(totalymomentum)
```

Now we can look at the sampled values to check that the momentum remains constant. For energy, it is more convenient to add a property to each planet giving its energy. This can then be defined as

```
energy - 0.5*mass*((px')^2 + (py')^2) =
      sum(-g * P.mass * mass/((P.px - px)^2 + (P.py - py)^2)^0.5,
          Planet(P), P > Self)
```

The `P > Self` avoids counting the potential energy of each pair of planets twice. Now we add at the end:

```
always totalenergy = sum(P.energy, Planet(P), 1=1),
sample(totalenergy)
```

If we wanted to compute the total distance moved by the earth, we could add the lines

```
earthdist = 0,
always earthdist' = ((Earth.px')^2 + (Earth.py')^2)^0.5,
sample(earthdist)
```

Note that the effect of the always statement is that $earthdist(t) = \int_0^t earthvel(t)dt$, where $earthvel = \sqrt{(Earth.px')^2 + (Earth.py')^2}$. One could add a similar expression to check Kepler's second law, that planets sweep out equal areas in equal times, by integrating an expression for the rate of area swept out.

6.2 Multiple modeling — Falling ball

We show how `hcc` can be used to create multiple models of a system. These can be used to model the system at various levels of detail. The system we wish to model is a ball bouncing on a table. The ball starts at a height of 10 meters, and falls freely under gravity. When its height is 0 (`ht` is the height of the center of ball – radius of ball), it bounces. There are two possible models of bouncing — an instantaneous inelastic collision with the table, where the ball loses some energy and bounces right back, or a spring-like model of the ball when it touches the table, in this case we have a damped harmonic oscillation.

```
ht = 10, ht' = 0,
always {g = 10, k = 10000, e = 10},
always {
     Compress,      // or NoCompress, depending on model
```

```
      cont(ht),
      if ht > 0 then
        ht'' = -g   // free fall
      else
        Bounce()
},
always {
    Bounce = (){
       if Compress then SpringBounce(),
       if NoCompress then InelasticBounce()
    },
    InelasticBounce = (){
       ht' = - 0.5 * prev(ht')
    },
    SpringBounce = (){
       ht'' + e * ht' + k * ht + g = 0
    }
}
```

The two models produce quite different results — in an instantaneous inelastic collision model the ball executes an infinite number of bounces in a finite amount of time — our interpreter just fails to notice this because of a finite step size. In the spring model, there are a finite number of bounces, and then the ball just oscillates on the ground.

6.3 Billiards table

The scenario modeled is a pool table with several balls rolling on it with various initial velocities. The balls keep rolling in a straight line until they hit another ball or the edge of the table or fall into a pocket, or come to rest due to friction.

The class `Ball` defines a ball with initial parameters giving its position and velocity. Its properties are its position and velocity, and some signals to notify changes in its velocities etc. The initial velocity and position is set up. The direction of motion of the ball is also computed as $\cos^2\theta$, where θ is the direction of motion of the ball. The ball is active until it falls into a pocket, indicated by `do ... until ...` — at that moment all program statements associated with the ball are terminated. While the ball is rolling, its velocity decreases according to friction. If `ChangeX` or `ChangeY` become true, then the program issuing the Change signal computes the new velocity.

The closures `Edge` and `Collisions` keep checking if the balls collide with the edge of the table or with each other. In each case a new velocity is computed for the ball(s) involved, according to the standard laws of kinematics. The closure `Pocketed` checks if any ball has fallen into a pocket, and issues the appropriate signal to terminate the ball's existence.

```
Ball = (initpx, initpy, initvx, initvy)[px, py, vx, vy, ChangeX, ChangeY, Pocketed] {
    px = initpx, py = initpy, vx = initvx, vy = initvy,
    new direction in {
        direction = vx^2/(vx^2 + vy^2),
        do always {
            lcont(vx), lcont(vy),
            if (ChangeX || ChangeY) then direction = vx^2/(vx^2 + vy^2),
```

```
            else direction' = 0,
            px' = vx, py' = vy,
            unless ChangeX then {
                cont(vx),
                if (vx < 0) then vx' = fric * direction^0.5,
                if (vx > 0) then vx' = -fric * direction^0.5,
                if (vx = 0) then vx' = 0
            },
            unless ChangeY then {
                cont(vy),
                if (vy < 0) then vy' = fric * (1 - direction)^0.5,
                if (vy > 0) then vy' = -fric * (1 - direction)^0.5,
                if (vy = 0) then vy' = 0
            }
        } until Pocketed
    }
},
Edges = (){
    always forall Ball(X) do {
        if (X.px = radius || X.px = xMax - radius) then
            X.ChangeX, X.vx = -prev(X.vx),
        if (X.py = radius || X.py = yMax - radius) then
            X.ChangeY, X.vy = -prev(X.vy)
    }
},
Collisions = (){
    always forall Ball(A) do forall Ball(B) do
        if (A < B) then                          //not the same ball
            if ((B.px - A.px)^2 + (B.py - A.py)^2 = 4*radius^2) then {
                if (A.px = B.px) then {
                    A.ChangeY, B.ChangeY,
                    A.vy = prev(B.vy), B.vy = prev(A.vy)
                },
                if (A.px < B.px) then {
                    A.ChangeX, B.ChangeX, A.ChangeY, B.ChangeY,
                    new c in new ix in {
                        c = (A.py - B.py)/(A.px - B.px),
                        ix = (prev(B.vx - A.vx) + c*prev(B.vy - A.vy))/(1+c^2),
                        B.vx := prev(B.vx) - ix,
                        B.vy := prev(B.vy) - c*ix,
                        A.vx + B.vx = prev(A.vx + B.vx),   // X-momentum
                        A.vy + B.vy = prev(A.vy + B.vy)   // Y-momentum
                    }
                }
            }
},
Pockets = (){
    always forall Ball(X) do
```

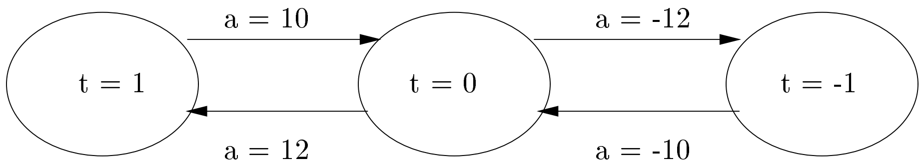

Figure 2: An automaton for the Schmitt trigger

```
        if ((X.px)^2 + (X.py)^2 <= pocket^2
            || (X.px)^2 + (X.py - yMax/2)^2 <= pocket^2
            || (X.px)^2 + (X.py - yMax)^2 <= pocket^2
            || (X.px - xMax)^2 + (X.py)^2 <= pocket^2
            || (X.px - xMax)^2 + (X.py - yMax/2)^2 <= pocket^2
            || (X.px - xMax)^2 + (X.py - yMax)^2 <= pocket^2) then
        X.Pocketed
},

always { radius = 3, xMax = 150, yMax = 300, pocket = 7, fric = 1},
Ball(B1, 10, 10, 25, 25),
Ball(B2, 20, 11, -35, 55),
Ball(B3, 80, 51, -15, 49),
Edges(), Collisions(), Pockets()
```

6.4 Writing Hybrid Automata — the Schmitt trigger

Some systems are naturally thought of as automata. An example is the Schmitt trigger often used in control theory to prevent frequent switching and wastage of resources (also known as bang-off-bang control). Such a scheme can be illustrated as follows: there is a plant represented by a, and it is controlled by a single variable u which has the values 0, 1 or -1, via the equation $a'' = -u$. u is set according to the automaton in Figure 2

The following program captures this automaton almost exactly:

```
/* initial conditions and determining the initial state */
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 in each state */
        watching (a >= 12 || a <= -12),  /* the exit conditions */
        when (a >= 12 || a <= -12) do {  /* the transitions */
            if a >= 12 then St1(),
            if a <= -12 then 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},
sample(a),
sample(t)
```

The following declarative program also has the same behavior:

```
a = 100, a' = -5,
always {
    if (a <= 10 && a >= -10) then
        do always t = 0 watching (a >= 12 || a <= -12),
    if (a >= 12) then do always t = 1 watching (a <= 10),
    if (a <= -12) then do always t = -1 watching (a >= -10)
},
always {a'' = -t}
```

6.5 Pitfalls in using hcc

hcc is still an evolving language, hence there are many pitfalls in its usage. Some of these will get eliminated as time goes on. Until then, whenever you have a program which does not have the desired behavior, send it to vgupta@ptolemy.arc.nasa.gov for debugging! Note: Any program that looks right to you, but does not produce the desired output, is regarded as a *bug in the language design* — so please do not hesitate to send it.

- **Program does not have a well defined output.** This is a frequent problem: a program looks reasonable, but upon closer inspection, it is clear that it does not have a reasonable output.

  ```
  x = 10,
  always {
    cont(x),
    if (x >= 20) then x' = -1,
    if (x < 20) then x' = 1
  }
  ```

  There is no mathematical function of time which can serve as the value of x. Thus to write such behavior, one should use a Schmitt trigger like program.

- **Program does not constrain variables enough.** The hcc constraint solver is incomplete! Often it will not make deductions that a user can make. Here is an example:

```
always w = 100,
always t = l - w,
l = 100,
always l' = 3.21,
always p = 90,
always if t = p then y = 1
```

In this case, at time 28.03738, hcc enters a point phase, as `t=p` may become true. Sure enough, that happens, and `y=1`. But in the succeeding interval phase, hcc deduces that $t = p$, because neither t nor p have derivatives, and the engine is set up to assume derivative 0 if none exists (this is correct almost every time!). The sollution is to add the redundant constraint `always t' = l' - w'`. Then hcc infers that $w' = 0, l' = 3.21$, so $t' = 3.21$. p' is still 0, so $t! = p$.

- **The lack of continuity constraints.** In the billiards ball program above, if we omit `lcont(vx), lcont(vy)`, the program does not work. This is because whenever a collision happens, vx', vy' may be undefined. Thus at the next interval phase, the ask constraints `vx = 0` etc. cannot be answered, this is then ensured by `lcont(vx)`, which asserts that vx is continuous from the left. Note that as we have asserted `always px' = vx`, we do not need to assert the continuity of px, as differentiability implies continuity. Of course having redundant constraints like this one cannot hurt.

- **Linear constraint evaluation.** hcc currently does not handle linear constraints very well, *unless Simplex is turned on.* So if your program has a lot of linear constraints, it may be a good idea to turn it on, by saying `#DO_LINEAR 1`.

- **Call-by-value.** All parameters in hcc functions and classes are call-by-reference. To create call by value parameters use **evalexpr**. Thus to have a call by value function write:

  ```
  Test = (a, b){
    evalexpr ta = a in evalexpr tb = b in Code
  }
  ```

  This is important for cases when you want to use the value of a variable to initialize another variable. For instance $x = 0, F = (var, val)\{\textbf{always}\ var = val\}, F(y, x)$ will produce the unintuitive result that y will be undefined after the first instant. Replace by $x = 0, F = (var, val)\{\textbf{evalexpr} v = val\ \textbf{inalways}\ var = v\}, F(y, x)$ to get the expected behavior.

We will add more items to the above list as time goes on (and hopefully remove some too!). Meanwhile, it is always advisable to run your program with the "-v" option turned on, often this will give a good idea of where the problem lies. And in the output for this option, any variable set to $[-inf, inf]$ is an automatic warning signal, as is the repeated reduction of the integration step size!

7 Controlling the runtime behavior of hcc

From the program it is possible to set some of the parameters which affect the runtime behavior of the interpreter. The general form of these is

$\#\langle constant\_name\rangle\ \langle constant\_value\rangle$

ans usually several of these will occur at the beginning of a file.

The following constants can be modified. Most however will never need to be modified, the default values should just work.

`EPSILON`: The value of this is any positive real number. Any two real numbers less than `EPSILON` apart are not distinguished by `hcc`. A smaller value will make the output more accurate, a bigger value will increase the speed. The default is 10^{-8}.

`ROUNDOFF`: The value of this is any positive integer. Its purpose is to quash roundoff errors in the numerical integration. At the end of every phase, it checks if the values of any variable (after multiplying by ROUNDOFF) is close to an integer. If it is, it is replaced by that integer/ROUNDOFF. E.g. if $x = 0.1999999$, and $ROUNDOFF = 10$, then we set $x = 0.2$. The default is 0 — no rounding is done.

`INTEGRATOR`: Chooses the type of the integrator. Currently it can have the values `EULER, RK4, RKQC, BS`, which respectively do Euler integration, 4th order Runge-Kutta, 4th-5th order Runge Kutta with adaptive stepsize and Bulirsch-Stoer with polynomial Richardson extrapolation. RKQC is the default: it is the most accurate, but can become very slow sometimes. EULER is the least accurate but the fastest. RK4 is useful in applications where accuracy is not so important, but values are needed regularly, for example in plotting or animation. BS is useful for integrating smooth equations, then it is faster and more accurate the RKQC.

`INTEGRATION_INIT`: Its value is a positive real number. It determines the initial step-size of the integrators. For Euler and RK4, this remains the step-size of integration, RKQC and BS modify the stepsize according to the equations being integrated. Very small values result in more steps (and higher accuracy for the the first 2 integrators) but larger values may lead to inaccuracy for the first two, and some inefficiency for RKQC. The default is 0.1.

`SAMPLE_STYLE`: The value is 0, 1 or 2. Style 0 dumps the sampled values for each variable as they occur, thus it saves memory. 1 and 2 write them out at the end of the run. Style 1 writes them by time instant, thus it writes a line containing the values for all variables for each time the samples were taken. Style 2 writes them by variable (this is used in the emacs mode), thus for each variable it lists the times and the sampled values (upper and lower bounds). More styles can be added by changing samples.c.

`SAMPLE_INTERVAL_MIN`: The value is a positive real. It determines the minimum distance between two successive samples. Useful if RKQC is taking too many small steps — normally samples are collected after every integration step, but this may produce too many samples. Default value is 0.

`SAMPLE_INTERVAL_MAX`: Its value is a positive real. Determines the maximum time elapsed between two successive samples. Useful for animations etc. when RKQC wants to take very large steps, this limits the maximum step size. Note that in the current implementation it will result in more steps, thus causing inefficiency. We can change this so that intermediate values are given by interpolation. Its default value is infinity.

`ALARM_TIMER`: Its value is a positive integer. It terminates the interpreter when `ALARM_TIMER` seconds have elapsed. Currently it is a nuisance, but its default value is 500000 (= 8 days), so it does not bother us.

`TRAIL_CHUNK`: This is the number of new trail blocks allocated each time we run out. A ver big value will slow things down initially as a lot of memory will be allocated, but small values are much worse as they cause reallocation of memory again and again. So set the largest possible value which does not delay things too much.

The constants below are rarely used.

`DO_LINEAR`: Its value is 0 or 1. It turns Simplex off or on. Simplex is often not very useful and is expensive, so we usually keep it off. However, once in a while we may need to turn it on.

`NR_WIDTH`: This determines when the propagator switches to Newton Raphson over splitting. It is needed because Newton Raphson converges very fast for small intervals, but not at all for large ones. When the size of an interval for a variable is less than `NR_WIDTH`, we apply NR to get quick convergence. Currently set to 0.1, but we do not know its effect on program performance.

`MIDPT_CUTOFF`: When we split an interval during constraint propagation, we need to take the mid-point. If we used only the midpoint, a lot of time would be spent examining the larger values of the variable, which are usually not where the answers lie. So if the width of an interval is more than this cutoff, we take geometric mean of the two endpoints, and if it is smaller we take the midpoint.

`MEMORY_CHUNK`: The value of this is any positive integer. This is the size of a block of memory allocated by hcc at a time. A larger value will cause the program to eat more memory initially, which can slow it down a bit. A smaller value will be inefficient, since many small blocks will be allocated. Note that eventually if the memory footprint of the program is almost constant, this will not matter since the garbage collector will allocate exactly that amount of memory and no more.

`CP_CHUNK`: Similar to `TRAIL_CHUNK`, for the choice point stack. This is almost never an issue.

`SYMBOL_TABLE_SIZE`: Sets the initial size of the symbol tables allocated during parsing. Currently set to 127. Since the memory allocated for the symbol tables is anyway freed during garbage collection, this is not very critical.

8 Including other hcc files and C functions

The following line in the program will append the hcc code in file *name* with the current code. It is equivalent in every way to copying over the code from the other file, but is useful for utility packages. Each file is appended only once, even if it is included from several files. Note: any #defines and library include files are local to the files where they were defined or included, so must be repeated in each file. We expect to change this in a future version.

#INCLUDE ⟨*name*⟩

It is possible to define additional C functions to add to the list of predefined functions. Examples are sin, cos etc. This is also useful for doing things which hcc is not very good at, like maintaining large amounts of data. For example, if there is a large terrain map given as a matrix, then we may want a function called **`height`**, which takes two arguments, and returns the height of the point.

The following function does the job. It takes an array of two intervals as an argument, and calls a standard C function **`heightfn`** to get the height. It returns an interval.

```
#include "../imath.h"
HInt
height(HInt *args) {
  HInt res;
  double x, y;

  x = (args[0].lower +  args[0].upper)/2;
  y = (args[1].lower +  args[1].upper)/2;
  res.lower = res.upper = heightfn(x, y);
  return res;
}
```

In order to have such a function, we define the wrapper as shown above in a C file. The C file is compiled as usual to get a `.so` file. In the hcc program using this function we add a declaration at the beginning saying

#LIBRARY ⟨*name*⟩

where *name.so* is the file generated above. (It may be useful to give complete path names: we will address this issue.) Now we can use this function like any other function:

```
always a := height(x,y),
```

For a function fn with one argument, we can provide its derivative and inverse also. These are defined as above, however the derivative must be named dfn, and the inverse named $invfn$. There are ways to sidestep this naming convention, involving more work on part of the programmer.

9 Debugging hcc

The debugger for hcc is based on the traditional step/continue style of C-debuggers. It can either be invoked from a shell or from Emacs. Suppose `hybrid` is the name of the executable for hcc. Then, a file `f.hcc` is debugged as `hybrid -g f.hcc`. The debugger proceeds by dequeueing statements and switching modes. There are five modes: **point**, **interval**, **default**, **trace**, and **meta**.

The debugger is in point-mode (interval-mode) when in a point (interval) phase, and dequeueing statements that are to be immediately executed. The debugger is in default-mode when dequeueing a statement **unless** c **then** A on which a branching is made (first assuming c does not become true, and if this leads to failure, assuming that c will become true). The debugger is in trace-mode when the integration is active, and, finally, when in meta-mode, the debugger first instantiates a meta-constraint $x = sum(X.y, C(X), c(X))$, say, to a constraint $x = X_1.y + \ldots + X_k.y$, where X_i is an instance of C, for which $c(X_i)$ is true, $1 \leq i \leq k$, and then adds the constraint to the queue.

If stepping, the debugger will stop at each dequeueing in any mode. If asked to continue, the debugger will return to the user only when a breakpoint fires. There is an assortment of breakpoints (see below), including breaking on lines of the program, and on mode-switches.

The commands of the debugger are as follows:

step steps to the next point where a statement is dequeued.

continue continues until a breakpoint fires.

help prints some online help.

break sets a breakpoint. These include breaking on line numbers (`> break 10`), queue entries (`> break #3`), object creations (`> break create Sun`), mode switches (`> break trace`), rule firing (`> break wake`) – it has to be issued when dequeueing a conditional statement (such as **if** c **then** A), and fires when the body of the statement (A) is dequeued – and time limits (`> break @23.3`).

print prints parts of the state. This includes printing variables (`> print Sun`), the queue (`> print queue`), the store (`> print store`), the default stack (`> print default`), the context (current variable mapping) (`> print context`), and the samples (`> print samples`).

cond adds a condition to a breakpoint (`> cond 0 x>=2`). A condition is a constraint that must be true at the time of breaking.

add adds a piece of code to the queue (`> add x<=99.99`). This is only accepted in point-mode.

check checks if a given constraint is currently true (`> check x>=2`). If no constraint is given, and the current statement is a conditional, the constraint of the conditional is used instead.

quit exits the debugger.

enable enables a breakpoint.

disable disables a breakpoint.

For example, suppose the file is:

```
py=8,  // prey
pd=2,   // predator
hence py' = 0.08*py - 0.04*py*pd,  // prey growth
hence {
  if (prev(pd) >= 0.5*prev(py)) then
     pd' = -0.1*pd+0.02*py*pd,
  else
     pd' = -0.06*pd+0.02*py*pd
},
sample(pd), sample(py)
```

and we load it into the debugger. Then, a step command will produce the following output:

```
debug>step
DEBUG: Point
pd = 2.00000
LINE: 2
debug>
```

where the first statement to be executed is on line 2. Suppose now we break on line 6 and continue;

```
debug>break 6
BREAKPOINT: 0
debug>cont
```

```
BREAKPOINT: 0
DEBUG: Interval
if prev(pd) >= 0.50000*prev(py) then (pd)' = (-(0.10000*pd))+0.02000*
py*pd else (pd)' = (-(0.06000*pd))+0.02000*py*pd
LINE: 6
debug>
```

The debugger thus stops in the next phase. The store and queue at that point are:

```
debug>print store
pd  [-Inf, Inf], prev  [2.000000e+00, 2.000000e+00]
py  [-Inf, Inf], prev  [8.000000e+00, 8.000000e+00]
py'  [-Inf, Inf], prev  [-Inf, Inf]
debug>print queue
Entry 0:
    if prev(pd) >= 0.50000*prev(py) then (pd)' = (-(0.10000*pd))+0.02000*
    py*pd else (pd)' = (-(0.06000*pd))+0.02000*py*pd
debug>
```

Commands can be issued using their prefix, as long as the prefix makes the command unique and is longer than 1 character. As a special case, 'co' is taken to be 'continue'. So for example, `debug> pr sa` prints the current samples. Hence, the following breaks the execution when 'pd' has passed a certain value.

```
debug>br 3
BREAKPOINT: 0
debug>cond 0 prev(pd)>=2.1
debug>cont
BREAKPOINT: 0
DEBUG: Point
(py)' = 0.08000*py-0.04000*py*pd
LINE: 3
debug>pr stor
pd  [-Inf, Inf], prev  [3.417958e+00, 3.417958e+00]
py  [-Inf, Inf], prev  [6.835915e+00, 6.835915e+00]
py'  [-Inf, Inf], prev  [-3.877215e-01, -3.877215e-01]
pd'  [-Inf, Inf], prev  [2.622199e-01, 2.622199e-01]
debug>
```

Note, that the conditional will not force the interpreter to stop exactly when the constraint becomes true, but only when it is true at the time of a dequeueing.

References

[CG98] Bjorn Carlson and Vineet Gupta. Hybrid CC and interval constraints. In Thomas A Henzinger and Sankar Sastry, editors, *Hybrid Systems 98: Computation and Control*, Lecture notes in computer science, to appear. Springer Verlag, April 1998.

[GJS98] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1-2):3–50, 1998.

[GJSB95] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel Bobrow. Programming in hybrid constraint languages. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Sankar Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture notes in computer science*. Springer Verlag, November 1995.

[SJG96] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, November/December 1996. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

A The Java interface to hcc

The Java interface is based on a class for creating, running, and manipulating hcc. The class implements Runnable which means that the hcc-computation can be done in a separate thread. NOTE! At this point, there can only be one hcc-state per Java-process. There are three sets of methods in the class: the methods for controlling the thread running hcc, the methods for creating and initializing the state, and the methods for reading the sample data generated by hcc. The hcc class also defines the Serializable interface, meaning that methods for streaming the hcc state into and out of Java streams can be done.

The main methods in each class are as follows (see also the html-documentation, which is part of the distribution).

A.1 Thread control

The following methods are important. Also, note that we separate between the hcc being suspended as a thread and being stopped. The latter puts hcc in a state where it can be manipulated by other primitives.

isRunning() returns true if the hcc thread is running.

isStopped() returns true if the hcc thread has stopped.

isSuspended() returns true if the hcc thread is suspended.

isWaiting() returns true if the hcc thread is waiting for space to be freed in the sample buffer.

isStopped() returns true if the hcc thread has stopped.

resume() resumes the hcc thread.

rewind() rewinds the hcc state to time 0.

run() starts the hcc execution.

start() restarts a previously stopped hcc execution.

stop() stops the hcc execution, meaning that hcc will return out from C and that the state can be manipulated.

suspend() suspends the hcc thread.

A.2 State manipulation

The state can be manipulated by adding code, deleting it, and setting runtime parameters. Also, the methods used in streaming the object are defined.

dispose deletes the whole state of hcc.

setMaxStep sets the maximum step length that the integrator is allowed to take.

setMinStep sets the minimum step length that the integrator is allowed to take.

setMethod sets the integration method.

setRunStep sets the length (in simulation time) of each call to hcc, before the interpreter returns.

uploadCode adds a string of code to the queue of the interpreter.

readObject reads back an hcc state from an input stream.

writeObject writes an hcc state to an output stream.

A.3 Sampling

The hcc execution can produce samples, which are time sequences of values for one or several hcc variables. These values can be read by and stored in Java.

getSampledVars returns a list of names for those variables that are sampled in the current hcc state.

sample returns a vector of sample points (one for each variable) that are generated at a given time, where the first element in the vector is the time at which the sample was made. The call suspends if the time which is asked for hasn't been reached yet.

skipTo skips ahead in time, without generating samples.

A.4 Example

Here is an example of how to use hcc from Java (see also the file `HySimulator.java` in the distribution). The program uploads a code string, and prints 100 sample points. Note that to get the sampled variables, the interpreter takes a tiny step in `uploadCode()`, so that the first sampled value will be made at a time > 0.

```
public class Test {
  public static void main(String[] args) {
    float t = 0F;
    RunHcc rc;

    try {
      rc = new RunHcc(10);
      rc.uploadCode("sample(x),x=0,hence x'=1");
      rc.start();
```

```
      for (int i=100; i>0; i--) {
        float[] v = rc.sample(t);
        t = v[0];
        System.out.println("time: " + t + " x: " + v[1]);
      }
      rc.dispose();
    }
    catch (Exception _e) { }
  }
}
```

For more examples on how to call methods in RunHcc see the file `Shell.java` in the distribution. This class uses `HySimulator.java`, which embeds RunHcc in a more general simulation class.

B The hcc distribution

We include two versions of hcc. One that runs on SunOS/Solaris, and one that runs on Windows 95/NT. Both versions are packaged by 'gzip' and 'tar', so to install them one needs those two programs (which exist under both platforms). The C and Java sources are almost identical between the two. The only difference being that in the Solaris version we include a generational garbage collector (the files 'malloggc.h' and 'malloggc.c', which replace 'mallogc.h' and 'mallogc.c') to be used by the brave.

Also included are libraries for programming hcc from Java and from C. These are shipped as '.so' or '.dll', depending on the platform. The libraries can be used without recompiling the source. Some Java source is included, which supports the basic API described above, plus sprite animations driven by hcc computations. We don't further describe how to run and install Java on these platforms. Instead we refer to `http://www.javasoft.com`. Finally, documentation is included, which consists of a documentation tree for the Java code, and the programmer's manual.

B.1 The Solaris distribution

The following is a listing of the files included in the Solaris distribution. A short description follows each file.

```
JNIHybrid.cc // C++ interface to hcc
Makefile     // Makefile for SunOS and Solaris
alloc.c      // allocation of solver data
backtrack.c  // routines for making the backtracking efficient
debug.c      // hcc debugger
debug.h
deriv.c      // support for symbolic differentiation
hccutil.c    // printing, allocation and misc. for interpreter
hybrid.h     // main include file for hcc
hybrid_hcc_RunHcc.h // h-file for Java library
imath.c      // interval arithmetic
interpreter.c // hcc interpreter
labeling.c   // labeling of interval variables
libJNIHybrid.so // shared library for Java and others
```

```
linear.c     // interface to Simplex
linear.h
main.c       // main file for hybrid
mallogc.c    // memory manager
mallogc.h
mt_server.c  // multi-threaded hcc server based on text-protocol
mt_server.o
norm.c       // normalization of constraints
parser.y     // parser and compiler
runhcc.c     // interface file into hcc (API)
sample.c     // code for outputting sample values
simplex.c    // adapted Simplex code from Numerical Recipes
solve.c      // constraint solvers
trace.c      // constraint integrators
trace.h
util.c       // printing, backtracking, and misc for solver
y.tab.c      // output from bison
```

The make-file expects `gcc` and `bison`, but should run equally well with `cc`/`acc` and `yacc`. So to install hcc, simply extract the files above from the archive ('hcc.tar') into a directory, and run 'make' in that directory. The files should compile with no warnings under Solaris and SunOS.

B.2 The Windows distribution

The Windows distribution contains Visual C++ projects for running hcc standalone (Hcc), or as a dll (HccDll), as well as the JNI-based interface to hcc from Java (JNIHybrid). The '.c' files are identical, as are the '.h' files and '.java' files with the files in the Solaris distribution.

To simply use hcc as a library, place the file `HccDll.dll` in the path where Windows looks for '.dll' files (on NT, this includes the directory `\winnt\system32`). If Visual C++ exists the '.mak' files should run as they are, i.e. to compile a version of the distribution, run `nmake Hcc.mak` for example, which compiles the standalone version of hcc. This produces 'hcc.exe', which runs from the shell.

The Visual C++ projects have the following structure:

```
    0 Sep 22 19:57 1997 Hcc/
13848 Sep 19 11:40 1997 Hcc/Hcc.mak
38912 Sep 19 17:49 1997 Hcc/Hcc.mdp
28704 Sep 19 17:49 1997 Hcc/Hcc.ncb
    0 Sep 19 12:08 1997 Hcc/Debug/
    0 Aug 11 14:48 1997 Hcc/Release/
    0 Aug 12 13:20 1997 Hcc/test/
    0 Sep 19 11:43 1997 HccDll/
14686 Sep 19 11:43 1997 HccDll/HccDll.mak
38400 Sep 19 11:43 1997 HccDll/HccDll.mdp
73056 Sep 19 11:43 1997 HccDll/HccDll.ncb
    0 Sep 19 11:42 1997 HccDll/Debug/
    0 Sep 19 11:42 1997 HccDll/Release/
    0 Sep 22 19:57 1997 JNIHybrid/
32768 Sep 19 17:25 1997 JNIHybrid/JNIHybrid.dll
```

```
 3490 Sep 19 17:25 1997 JNIHybrid/JNIHybrid.exp
16320 Sep 19 17:25 1997 JNIHybrid/JNIHybrid.lib
 6892 Sep 18 15:31 1997 JNIHybrid/JNIHyBrid.mak
34816 Sep 19 10:44 1997 JNIHybrid/JNIHyBrid.mdp
58368 Sep 19 10:44 1997 JNIHybrid/JNIHyBrid.ncb
 7405 Sep 19 17:25 1997 JNIHybrid/JNIHybrid.obj
    0 Sep 18 13:46 1997 JNIHybrid/Debug/
    0 Sep 18 13:44 1997 JNIHybrid/Release/
```

The Java files are best put in their own directory. Make sure 'JNIHybrid.dll', 'HccDll.dll', and 'HccDll.lib, are placed in the dll-path of the operating system. Otherwise, 'System.loadLibrary("JNIHybrid")' will complain when run from Java. The following directories are included:

```
    0 Sep 19 13:47 1997 java/
 1340 Sep 19 14:17 1997 java/Hba.class
 1178 Sep 19 13:47 1997 java/Hba.java
    0 Sep  9 09:23 1997 java/hybrid/
    0 Sep 19 13:50 1997 java/hybrid/anim/
    0 Sep  9 09:23 1997 java/hybrid/gui/
    0 Sep 17 10:09 1997 java/hybrid/hcc/
   59 Sep  9 09:23 1997 java/index.html
  104 Sep  9 09:23 1997 java/start.bat
    0 Sep 19 13:50 1997 java/demo/
    0 Sep  9 14:25 1997 java/demo/solar/
     0 Sep 19 14:13 1997 java/demo/ball/
337920 Sep 11 15:09 1997 java/doc.tar
     0 Sep 11 15:28 1997 java/doc/
     0 Sep  3 09:38 1997 java/doc/images/
```

where 'Hba.java', and the directories 'hybrid/anim' and 'hybrid/gui' contain an example of how to build animation software on top of hcc. The directory 'hybrid/doc' contains the documentation for the animation and the hcc packages. The RunHcc class is stored in 'hybrid/hcc', as is some code for using RunHcc.

At this point, the best documentation we offer for using hcc from Java, or any other language, is by referring to the examples above. In particular, it is worthwhile looking at the files 'java/Hba.java', 'java/hybrid/anim/BallAnimation.java', and 'java/hybrid/hcc/Shell.java' for typical usage.

C The Emacs interface to hcc

We include an Emacs-interface to hcc in the file 'hcc.el'. The interface supports running hcc from Emacs, viewing the sample output, plotting the samples (if 'gnuplot' exists on the machine on which Emacs is running), and running the debugger from Emacs (this only works on Emacs on Unix, since for some reason a subprocess reading from stdin/the keyboard under NT freezes the interaction in Emacs).

The 'hcc-mode' of Emacs supports indentation, "fontifying" keywords, jumping to the beginning and end of definitions (these are closure or lambda constraints), and debugging.

When debugging (in 'hcc-debug-mode') some shortcuts (based on modified keys) are defined to simplify issuing debugger commands. The mode is based on 'comint-mode', which thus is required.

When viewing (in 'hcc-sample-mode') some one-key shortcuts are defined, which actually are the only keys that are accepted as input, as the buffer is otherwise read-only. Plotting only works if 'gnuplot' is around. Currently, we use the text output from gnuplot, but this can easily be changed to use the Postscript output instead (see the code for 'hcc-sample-plot').