Hybrid cc with interval constraints

Björn Carlson * Vineet Gupta**

Abstract. Hybrid cc is a constraint programming language suitable for modeling, controlling and simulating hybrid systems, i.e. systems with continuous and discrete state changes. The language extends the concurrent constraint programming framework with default reasoning and combinators for programming continuous behavior. The most important constraint systems used in Hybrid cc are nonlinear equations and ordinary differential equations over intervals. We describe the implementation of the Hybrid cc interpreter and constraint solvers, and evaluate the performance using some example programs.

1 Introduction

Hybrid cc [GJS97, GJSB95] is a compositional, declarative language, based on constraint programming, which enables modeling and simulation of hybrid systems in one framework. In Hybrid cc, a hybrid system is specified by a set of constraints on its temporal behavior. Each constraint describes an internal relationship of the system, e.g. the heat loss of a container as a function of time, or the acceleration as it depends on mass. The constraints are based on standard formalisms used in physics and engineering, such as differential equations and algebraic equations. Discrete events and state changes, such as turning on a heater when the ambient temperature drops too low, are specified using the combinators of concurrent constraint programming [Sar93] and default logic [Rei80]. The formal operational semantics of Hybrid cc is described in [GJS97].

This paper presents an implementation of Hybrid cc. We have chosen an interval constraint system for our implementation, since this gives us the ability to model some uncertainty in the parameters. The two most important classes of constraints used in our implementation are (nonlinear) algebraic and ordinary differential equations. Algebraic constraints are solved by interval propagation using indexicals, interval splitting, the Newton-Raphson method and the Simplex algorithm. Differential equations are integrated using a version of the fourth-order Runge-Kutta method with adaptive stepsize, modified for interval variables. We use constraint propagation to solve the simultaneous differential equations.

Interval constraints provide Hybrid cc with the expressive power required for many modeling problems [GSS95], where inequalities are used to express physical constraints like bounds on force magnitudes etc. In addition, many physical systems are imprecise in nature, i.e. we cannot construct a perfectly accurate model. The imprecision is

* 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

captured by interval constraints. By using constraint propagation inside the numerical integrator we are further able to strengthen the precision of the integration by adding redundant constraints, which narrow the divergence (see example in Section 3) that follows from imprecise initial conditions.

Our implementation is based on an interpreter and compiler written in C and Yacc. The compiler translates each program into a graph of expressions, which is interpreted combinator by combinator. The interpreter keeps a constraint store similar to the store of traditional constraint programming languages. The memory is managed using a conservative garbage collector for C.

The implementation is easily embeddable in other systems. For example, we have integrated Hybrid cc with Java, both as a Win32 dynamic library with an API for compiling and running Hybrid cc under Windows 95 and Windows NT, and as a remote procedure call interface for compiling and running Hybrid cc remotely. We have also developed a modeling client in Java with support for visual Hybrid cc programming, and graphical output, both as graph plots and 2.5 D animations generated by sampling variables during the execution of an Hybrid cc program.

The performance of Hybrid cc on interval constraint benchmarks shows that its interval propagation is comparable with the best interval solvers *e.g.* clp(Newton) [VMK95]. Clearly, our interval version of the Runge-Kutta method is not as fast as standard libraries for integration over real-valued variables. However, by using interval variables and constraint propagation inside the numerical solver, we get a more flexible and robust system for modeling with differential equations.

The paper is structured as follows. In Section 2 we give a brief introduction to Hybrid cc and give its operational semantics. We then give a description of the constraint solvers in Section 3. Finally we conclude with a comparison with related work and an evaluation of the performance of the interpreter.

2 Hybrid cc – the language and its use

Hybrid cc extends concurrent constraint programming with defaults, continuous combinators and objects. The basic set of combinators in Hybrid cc are as follows:

c	tell the constraint c
if d **then** A	if d holds, reduce to A
unless d **then** A	reduce to A unless d holds
A, B	parallel composition
new V **in** A	V is local to A
forall $C(X)$ **do** A	do $A[I/X]$ for each instance I of class C
hence A	execute A at every instant after now
$X(T_1, \ldots, T_k)$	execute X with parameters $T_1, \ldots, T_k$

The constraint system we have implemented is as follows.

Continuous Constraints. These constraints assert equalities and inequalities over arithmetic terms. The syntax is as follows:

$$\begin{aligned}
ContConstr &::= Term\ RelOp\ Term \mid cont(LVariable)\\
RelOp &::= = \mid >= \mid <=\\
Term &::= LVarExpr \mid Constant \mid Term\ BinOp\ Term \mid UnOp(Term)\\
&\quad \mid Term'\\
LVarExpr &::= LVariable \mid UVariable.LVarExpr\\
BinOp &::= + \mid - \mid * \mid / \mid \hat{}\\
UnOp &::= - \mid sin \mid cos \mid log \mid exp \mid prev
\end{aligned}$$

$LVariable$s are variable names which start with a lowercase character while $Constant$s are floating point numbers. $LVarExpr$s are $LVariable$s or property expressions (see paragraph below). The semantics of most constructs is as expected. For example, $exp(x)$ is the exponential function e^x, $Term'$ denotes the derivative of $Term$ with respect to the implicit variable time.

$cont(x)$ 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. Note that asserting $x' = 3$ automatically asserts that x is continuous, as differentiability implies continuity.

Ask arithmetic constraints also allow the Relops $<, >, ! =$.

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

$$\begin{aligned}
DConstr &::= UVarExpr \mid UVarExpr = DExpr\\
UVarExpr &::= UVariable \mid UVarExpr.UVarExpr\\
DExpr &::= UVarExpr \mid String \mid (VarList)[VarList]HccProg\\
&\quad \mid (VarList)HccProg \mid UVarExpr(VarList)[VarList]HccProg\\
VarList &::= UVariable \mid LVariable \mid VarList, VarList
\end{aligned}$$

A $UVariable$ is a variable name starting with an uppercase character. $UVarExpr$'s are $UVariable$s or property expressions (see below). $HccProg$ is any Hybrid cc program, defined above.

A $DConstr$ given as a $UVarExpr$ is a signal constraint. These constraints are typically used to communicate to some other statement of code that a certain state is reached or a certain property is true.

A String is any string of characters enclosed within double quotes. These are mostly used for properties of objects — i.e. `Switch = "on"`.

The constraint $X = (V_1, \ldots, V_k)HccProg$ sets up a closure definition. It defines X to behave like $\lambda V_1 \ldots \lambda V_k.HccProg$ with the exception that X can only be β-reduced when all k arguments are given. The factorial function can now be defined recursively as follows:

$$\begin{aligned}
P = (n, m, Q)\{\ &\textbf{if } (n > 0) \textbf{ then new } x \textbf{ in } \{Q(n-1, x, Q), m = x * n\},\\
&\textbf{if } (n = 0) \textbf{ then } m = 1\ \}
\end{aligned}$$

so the call $P(n, x, P)$ computes $x = n!$. Closures are first class objects, and can be passed around as data.

The constraint $C = B(V_1, \ldots, V_k)[P_1, \ldots, P_l]A$ sets up a class definition. It defines a class C, where the constructor takes k arguments, and the properties of C are named P_i, $1 \leq i \leq l$. Note that a property P_i can point to a closure, this is how methods are defined. A property is treated as a variable inside A. The functor B is optional, and if used, it must be constrained to a (base) class from which C inherits all the properties. The code in A is used for defining any instance of C (see below).

Objects are created using the same syntax as for a closure call $C(Name, t_1, \ldots, t_k)$ where C will be bound to a class definition. The argument $Name$ is mapped to a property named $Self$. A property x can be referred inside the code as x, but from outside it must be referred as $Name.x$. Any code in C is run with $Self = Name$ to initialize the object.

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 $A = (M)HccProg$ will always answer unknown.

Computational model of Hybrid cc. The computational model is based on reductions of statements. Let σ denote a variable store, i.e. a set of interval constraints $x \in [a, b]$, string, atom, closure and class constraints. An Hybrid cc system consists of a store, a set of Hybrid cc statements, and some auxiliary structures.

An Hybrid cc system alternates between being in a *point phase* and in an *interval phase*. The initial phase is a point. Let A be the initial statement to be reduced. By the semantics of the operators, defined below, a stable point is eventually reached for A (we assume throughout that no infinite sequence of reductions occurs), where all constraints have been propagated, and all reductions of statements in A have been completed.

Now, either the stable store σ is inconsistent, and the computation is aborted, or it is consistent. In the latter case, the computation enters the interval phase. The statements to be reduced in this phase consist of each B that was reduced by **hence** B in the point phase, together with the statement **hence** B itself (remember that **hence** B means that B is to be reduced continuously and forever).

Similar to the point phase, all the statements in the set described above are reduced until a stable point is reached. This determines the set of constraints that are continuously true in the current phase, and the set of statements to be reduced at the next point phase. The length of the interval phase is the longest interval during which the constraint set is unchanged — this is ensured by making sure that none of the asked constraints changes status. For example, consider the program

$$x = 0, \textbf{hence } \{x' = 1, \textbf{if } (x = 2) \textbf{ then } y = 1\} \tag{1}$$

In the interval phase following $x = 0$, x evolves continuously according to $x' = 1$, through the interval $(0, 2)$ until $x = 2$ is about to become true. At this point the set of constraints may change, so the next point phase is started.

We first describe the reduction rules for each operator of Hybrid cc, and then provide the algorithm for the interpreter. The following reduction rules apply in either

phase. Γ denotes a set of Hybrid cc program fragments, σ denotes the store, **next** the set of program fragments to be run in the next phase, and **default** a set of suspended else statements. The expression $\sigma \vdash c$ denotes entailment checking.

Tell $\quad \langle(\Gamma, c), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle \Gamma, \sigma \cup \{c\}, \textbf{next}, \textbf{default}\rangle$

Ask $\quad \dfrac{\sigma \vdash d}{\langle(\Gamma, \textbf{if } d \textbf{ then } A), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle(\Gamma, A), \sigma, \textbf{next}, \textbf{default}\rangle}$

Unless $\quad \langle(\Gamma, \textbf{unless } d \textbf{ then } A), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow$
$\langle \Gamma, \sigma, \textbf{next}, (\textbf{default}, \textbf{unless } d \textbf{ then } A)\rangle$

Par $\quad \langle(\Gamma, (A, B)), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle(\Gamma, A, B), \sigma, \textbf{next}, \textbf{default}\rangle$

Forall $\quad \dfrac{\sigma \vdash I_1, \ldots I_n \text{ are instances of } C}{\langle(\Gamma, \textbf{forall } C(X) \textbf{ do } A), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle(\Gamma, A[I_1/X], \ldots, A[I_n/X]), \sigma, \textbf{next}, \textbf{default}\rangle}$

New $\quad \langle(\Gamma, \textbf{new } X \textbf{ in } A), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle(\Gamma, A[Y/X), \sigma, \textbf{next}, \textbf{default}\rangle$ (Y new)

Call $\quad \dfrac{\sigma \vdash P = (V_1, \ldots, V_k)A}{\langle(\Gamma, P(t_1, \ldots, t_k)), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle(\Gamma, A[t_1/V_1, \ldots, t_k/V_k]), \sigma, \textbf{next}, \textbf{default}\rangle}$

The rule for **hence** A differs in point and interval phases.

Hence Point $\quad \langle(\Gamma, \textbf{hence } A), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle \Gamma, \sigma, (\textbf{next}, \textbf{hence } A), \textbf{default}\rangle$

Hence Interval $\quad \langle(\Gamma, \textbf{hence } A), \sigma, \textbf{next}, \textbf{default}\rangle \rightarrow \langle(\Gamma, A), \sigma, (\textbf{next}, A, \textbf{hence } A), \textbf{default}\rangle$

The **tell** rule propagates the effects of the constraints using the algorithms described in the next section. The combinator **forall** also suspends such that if any further instance I of C is created in the current phase, the combinator adds $A[I/X]$ to Γ. Similarly, for $\sigma \vdash c$, if c is neither detected true or false in the current store, the statement that contains c is suspended and reconsidered whenever any of the variables in c changes value (e.g. is pruned).

The algorithm for the interpreter is the same in both phases, except for the integration at the end of the interval phase. It involves the following steps:

1. Run the reduction rules on the current $\langle \Gamma, \sigma, \textbf{next}, \textbf{default}\rangle$, till no further reductions can take place.
2. If σ is inconsistent, return 0.
3. If **default** is empty, return 1.
4. Remove one statement from **default**— **unless** c **then** A. If $\sigma \vdash c$, go to step 3.
5. Add A to Γ. Run the interpreter on the current state. If the result is 1 and $\sigma \nvdash c$, return 1.

6. Undo the effects of the previous step by backtracking. Run the interpreter on the current state. If the result is 1 and $\sigma \vdash c$. return 1. Otherwise return 0.

Note that the effect of the last three steps is that a maximal set of defaults is chosen and executed (similar to the maximal extensions of [Rei80]). This is similar to the causal loops in synchronous languages [Hal93, BB91, Har87, SJG96]. There can be many different maximal sets, our interpreter chooses any one randomly. For example **unless** X **then** Y, **unless** Y **then** X can reduce to either X or Y but not both. If no maximal set exists, as for the statement **unless** X **then** X, then the computation must be aborted.

A Hybrid cc program A is run as follows.

1. Run interpreter with $\Gamma = A$, and empty σ, **next** and **default** in the point phase. If the result is 0, abort.
2. Run the interpreter in the interval phase with $\Gamma =$ **next**, as returned by the point phase. σ, **next** and **default** are again empty. If the result is 0, abort. Record all the tells, and also the ask constraints that were checked during the phase.
3. Integrate the arithmetic constraints that were told in the previous step, until one of the ask constraints changes status (i.e. goes from false to true or unknown, etc.). Go to step 1 with $\Gamma =$ **next**.

Hybrid cc also contains various constructs from synchronous programming, e.g. **do** A **watching** c, **when** c **do** A, but since their behavior can be derived from the behavior of the above constructs we omit them here.

Implementation. Our interpreter implements essentially the above algorithm, with a few changes. For example, the interpreter is not recursive, but uses stacks for managing the backtracking. The compiler of Hybrid cc straightforwardly translates each statement A into an expression graph, where each node corresponds to an operator of the language. We optimize memory by sharing code as far as possible.

We omit a detailed description of the implementation, since most of it is based on standard techniques for how a concurrent constraint language based on reductions is implemented, e.g. we have borrowed from AKL and cc(FD) [HSD92, Jan94] in how constraints and suspensions are treated, how memory is managed (using a conservative garbage collector for C), and how backtracking is implemented (using choicepoint and trail stacks).

3 The constraint solvers

3.1 Nonlinear equations

We consider in the following only constraints of the form $f(\mathbf{x}) = 0$, as all other constraints can be reduced to this form by introducing slack variables. Interval pruning is used as the basic means for constraint solving. We have implemented four pruning operators: indexicals, interval splitting, the Newton-Raphson method, and the Simplex method. The pruning is hence stated as: given $f(x) = 0$ and an interval constraint $x \in [a, b]$ for x, apply one or more operators to $f(x)$ to compute a new interval $[a_1, b_1] \subseteq [a, b]$ for x. If this fails, the constraint is deemed inconsistent.

An *indexical* is the fastest way to update the interval for x [HSD92, Car95]. Given $f(x, \mathbf{y}) = 0$, we try rewriting the constraint in an explicit form $x = g(\mathbf{y})$, for some term g. Now x is set to $[a, b] \cap g(\mathbf{I}/\mathbf{y})$, where $\mathbf{y} \in \mathbf{I}$ holds in the current store, and g is evaluated over intervals.

For example, consider $x + y = 0$, $x \in [0, 3]$, and $y \in [-1, -2]$. Then the indexical $x = -y$ is used to set x to $[1, 2]$.

Splitting of intervals is used to narrow the interval for x by splitting it recursively. Given $f(x, \mathbf{y}) = 0, x \in [a, b], \mathbf{y} \in \mathbf{I}$, we split $[a, b]$ until the smallest $a_1 \in [a, b]$ is found such that $0 \in f(a_1, I)$. Hence, if $0 \in f([a, \frac{a+b}{2}])$, then $a_1 \in [a, \frac{a+b}{2}]$, and otherwise $a_1 \in [\frac{a+b}{2}, b]$. Similarly, b_1 is computed, thus $x \in [a_1, b_1]$.

For example, given $x^2 = 1$ and $x \in [-\infty, \infty]$, it follows that $0 \in [-\infty, 0]$, but $0 \notin [-\infty, -100]$ say, so a_1 is determined to be in $[-100, 0]$. Eventually, a_1 is determined to be -1.

The third pruning method is the *Newton-Raphson* method adapted to intervals[AH83, VMK95]. As in splitting, the leftmost and rightmost zeros for $f(x)$ are computed separately. Let $f'(x) = \frac{df(x)}{dx}$, $I = [a, b]$ such that $0 \in f(I)$ and $0 \notin f'(I)$ (this guarantees that there is only one zero in I, and can be accomplished by splitting), and let $m_i \in I_i$. Let $I_0 = I$, and define $I_{i+1} = I_i \bigcap (m_i - \frac{f(m_i)}{f'(I_i)})$. Iterate until $I_i = I_{i+1}$. It follows that $0 \in f(I_i)$.

In practice, we combine splitting and the Newton-Raphson method for quick results, just as in clp(Newton). Splitting is useful in reducing the size of an interval, but is inefficient in pinning down the roots exactly. The Newton-Raphson method finds roots very quickly, if they are known to lie in a small interval. For example, given $x^2 = 1$ and $x \in I = [-\infty, \infty]$, we split I recursively until I is split down to $[-2, -1]$. By setting I_0 to I, and applying the Newton Raphson method, $I_i = [-1, -1]$ is produced. Similarly, $[1, 1]$ is produced for the rightmost zero. Hence, the final interval returned by the pruning operator is $[-1, 1]$. Note that this interval is clearly an approximation to the set of solutions, since only -1 and 1 are solutions to the equation.

The *Simplex* method is used as a global pruning method and is applied only at certain times due to its high cost, unlike the previous lightweight methods, which are applied incrementally. It is useful for detecting inconsistent conjunctions that otherwise lead to slow convergence of the propagator, a well-known problem when inequalities are used. For example, consider the conjunction $\{x \leq y - \epsilon, y \leq x - \epsilon\}$, for some small $\epsilon > 0$. This conjunction is inconsistent but the pruning algorithm reduces the size of the intervals by ϵ at each iteration, forcing a large number of iterations before an inconsistency is detected.

Given a set of constraints $C = \{c_1, \ldots, c_k\}$, we linearize them into a set of linear constraints $L = \{l_1 = 0, \ldots, l_k = 0\}$, by replacing each *nonlinear* term, e.g. xy, by a new variable z uniformly throughout the set C. The detection of common subexpressions is useful here. Hence, each l_i is of the form $a_0 + a_1 z_1 + \cdots + a_n z_n = 0$, for some constants a_j and variables z_j. Note that C is consistent implies L is consistent.

Now, we apply the Simplex method to L to check whether L is inconsistent, using standard techniques. If successful, the Simplex algorithm can be used again for pruning the original variables of C. For each such variable x, a_1 (the new minimum value of x) is computed by minimizing x, and b_1 (the new maximum) by maximizing x.

By applying the above, a conjunction such as $x + y = 3, 2x - y = 0$ produces the interval constraints $x \in [1, 1]$ and $y \in [2, 2]$ immediately, whereas the other operators produce no pruning. The conjunction $x \leq y - \epsilon, y \leq x - \epsilon$ is shown to be inconsistent.

Implementation. Internally, each constraint c is decomposed into a set of pairs, (x, f), where x is a variable in c and f is either the indexical that prunes x, derived as above, or c itself. The latter is then used for splitting and Newton-Raphson. Each variable y points to a set of such pairs, such that when y is pruned, the variables dependent on y are also pruned. Prunings are propagated by a variant of the arc-consistency algorithm (Figure 1).

We use an optimization based on the fact that decomposing a constraint as above produces equivalent variants, and hence when one variant is true the others are true too [Car95]. Hence, for each (x, f) that is dequeued, if f is marked as entailed, the pair is ignored since no more pruning is generated by f. When a pair (x, f) is enqueued, the list of variables of f is checked. If all of them (except the slack variables) are bounded by an interval $[a, a]$, f is marked entailed. All pairs generated from the same constraint share the entailment mark, hence when one is marked, they are all marked.

```
int propagate(queue) {
  while (queue is not empty) {
    (var,constraint) = dequeue(queue);
    if constraint is marked entailed continue;
    if constraint is an indexical
      interval = intersect(var,evalIndexical(constraint));
    else
      interval = project(var,constraint); // using splitting, NR
    if interval is empty return 0;
    if interval is a strict subset of var {
      var = interval;
      enqueue(var->constraints,queue);
      if all variables in constraint are determined
        mark constraint entailed;
    }
  }
  return 1;
}
```

Fig. 1. Pseudo-code for the propagator

Telling in a point or interval phase. In a point phase, if we constrain the variable x', then in addition to the propagation described above we infer that x is continuous, and set its value to the limiting value in the previous interval phase, if one exists.

In an interval phase, for an arithmetic constraint $t = 0$, for which t' is defined, the arithmetic constraint $t' = 0$ is also added (this is done recursively while the derivatives

of all the variables are defined). For example, if $x + y = 0$ is added, then $x' + y' = 0$ is added too, if x' and y' are defined in the current state. This is sound, and improves the propagation and entailment checking by adding redundant constraints.

3.2 Entailment checking.

Let the current store be σ. In the point phase, a constraint $t = 0$ is entailed if t evaluates to $[0, 0]$ in σ, where each operator is evaluated over intervals rather than points, and where a variable x is replaced by $[a, b]$, where $x \in [a, b]$ belongs to σ. Similar reasoning is applied to $t \leq 0$ and $t < 0$. We assume that each arithmetic constraint is normalized to one of the forms above.

In the interval phase, if $t = 0$ was true in the previous point store, and $t' < 0$ holds in σ, then $t < 0$ also holds in σ. Similarly, for any positive natural number n for which $t^{(n)}$ is defined, if $t^{(m)} = 0$ and $t^{(n)} < 0$ are true in σ, for all m s.t. $0 < m < n$, $t < 0$ is true in σ. The constraint $t = 0$ is consequently true in the interval phase if $t^{(m)} = 0$ is ruled true for all m for which $t^{(m)}$ is defined.

3.3 Ordinary differential equations.

We use a version of Runge-Kutta integration with adaptive step-size for integrating the differential equations numerically[PTVF92] (although it is easy to add other integration methods). The initial conditions of the integration are given by the store from the most recent point phase.

The pseudo-code for the integrator is shown in Figure 2, where we give the simpler fourth-order Runge-Kutta with no error checking to illustrate the interplay between propagation and integration.

We have made three changes to the basic Runge-Kutta algorithm. First, we use interval arithmetic throughout. This makes the system more flexible since we do not require specific initial conditions, e.g. a variable x can be constrained to $[0, 100]$ at the start of the integration. However, the interval arithmetic also introduces a divergence problem. For some examples, the solution interval for a variable x grows in size as the integration proceeds, i.e. we lose precision. We are exploring methods for improving this automatically, but meanwhile we rely on using redundant constraints to contain the divergence (see example below).

Second, each step in the integration includes propagation (but with no use of the Simplex algorithm), so that we can solve simultaneous equations of an arbitrary form. In the standard Runge-Kutta procedure, x_{n+1} is computed from x_n (its previous value) by considering the explicit equation $x' = f(x)$ and computing x_{n+1} by: $x_{n+1} = x_n + \Sigma_{i=1}^{6} c_i k_i$, and k_i is defined as: $k_1 = hf(x_n)$, $k_i = hf(x_n + b_{i1}k_1 + \cdots + b_{i(i-1)}k_{i-1})$, $1 < i \leq 6$, where b_{ij} and c_i are constants given by the Runge-Kutta formulas, and h is the time step.

In Hybrid cc we do not necessarily have the equation $x' = f(x)$, but rather one or several equations on x', e.g. $(x')^2 = x$. We thus compute k_i by first setting each dependent variable (x for which x' is constrained) to $x_n + b_{i1}k_1 + \cdots + b_{i(i-1)}k_{i-1}$, where x_n is the previous interval for x, and then propagating, in an initially empty store

```
integrate(diff_eqs,check_list) {
  let h be the initial step size (0.1);
  let V be the dependent variables of diff_eqs;
  set the initial values of V by the store from the point phase;
  propagate;
next:
  integrate V;
  if a constraint in check_list is overshot, backtrack and shrink h;
  if a constraint in check_list changes state, stop;
  go to next;
}

integrate V { // 4th order Runge-Kutta with no error-control
  compute k1 from current x' (for each x in V);
  for i in [2,4] {
    initialize a new store;
    update x (for each x in V) to compute ki;
    propagate;
    compute ki for current x' (for each x in V);
  }
  set new x = old x + 1/6*k1 + 1/3*k2 + 1/3*k3 + 1/6*k4;
  propagate; // to get values of variables after time-step
}
```

Fig. 2. Overview of the integration procedure

σ_i, the consequences of the differential equations. At quiescence, k_i is set to $[ha_i, hb_i]$, where x' is constrained to $[a_i, b_i]$ in σ_i. Afterwards, each σ_i is discarded.

Example 1. The following system of equations describes the tank temperature(t) and concentration of a substance a (c_a) in a tank being stirred (the other parameters are constants) [Kay96].

$$c_a' = \frac{c_{a_i} - c_a}{\tau} - k_0 e^{-E/t} c_a$$
$$t' = \frac{t_i - t}{\tau} - k_0 e^{-E/t} c_a$$

This system is highly nonlinear due to the exponential containing t. It diverges very quickly given an initial state such as $0.933 \leq c_a \leq 0.934, 353.358 \leq t \leq 353.36$ — after some steps t and c_a both become $[0, \infty)$. Adding the constraints $c_a \geq 0.8445$ and $t' \leq 0$ to the set of differential equations keeps the intervals for c_a and t considerably narrower (the width being of the order 10^{-2}). These constraints can be obtained automatically by using qualitative methods [Kay96].

Third, the integration is made to interrupt *exactly* at the time instant when some constraint in a given list of constraints to be checked changes its state. This is necessary to make the results of the implementation be independent of the step-size, modulo numerical errors. Thus in the program 1, $x = 2$ is false while $x \in (0, 2)$, but when

the integrator reaches $x = 2$ we must switch to the point phase. The point when the integrator stops is called the *breakpoint*.

The standard Runge-Kutta procedure however does not know about the breakpoints, so it may overshoot, e.g. in the program above, go from $x = 1.9$ to $x = 2.3$ in one step, depending upon the current integration stepsize. Hence, we must force the integrator to consider every "important" point.

We detect overshooting by recording, for each given constraint to be checked, whether it is initially true or false or neither. For each integration step, we check whether the status of any constraint changes, e.g. goes from true to false. When we detect overshooting, we backtrack, i.e. we undo the most recent integration step, and try a smaller step size to find the point when the break should happen exactly, e.g. in the program above, force the integrator to reach $x = 2$. Currently, we use a simple linear interpolation technique for computing the smaller step size, though more sophisticated techniques are possible.

4 Examples and Evaluation

We now give an idea of the performance of Hybrid cc on some representative benchmarks picked from [vHLB97]. We show the runtimes of Hybrid cc computing all the solutions to each problem on a SPARCstation 20 system. For example, the Broyden Banded functions are computed by the following constraints:

$$f_i(x_1, \ldots, x_n) = x_i(2 + 5x_i^2) + 1 - \Sigma_{j \in J_i} x_j(1 + x_j) \quad (1 \leq i \leq n)$$

where $J_i = \{j \mid j \neq i \& \max(1, i-5) \leq j \leq \min(n, i+1)\}$, which for $n = 3$ is written in Hybrid cc as:

```
0 = x1 * (2 + 5*x1^2) + 1 - x2*(x2+1),
0 = x2 * (2 + 5*x2^2) + 1 - x1*(x1+1) - x3*(x3+1),
0 = x3 * (2 + 5*x3^2) + 1 - x1*(x1+1) - x2*(x2+1),
-1 <= x1, x1 <= 1, -1 <= x2, x2 <= 1, -1 <= x3, x3 <= 1
```

For these problems, the constraint solver of Hybrid cc finds the unique solution to within 10^{-6}.

The other examples we have considered are the Moré-Cosnard nonlinear integral equations, an interval arithmetic problem (i4), and a combustion problem [vHLB97]. We give the runtimes for some different n in the case of the Broyden Banded, and the Moré-Cosnard equations.

Example	**run-time (sec)**	**Example**	**run-time (sec)**
Broyden 10	0.13	Moré-Cosnard 40	39.8
Broyden 40	0.6	Moré-Cosnard 80	436
Broyden 160	2.6	interval 4	21.2
Moré-Cosnard 10	0.4	combustion	6.2

These numbers compare with the numbers published for clp(Newton) as follows. In the Broyden and Moré-Cosnard, Hybrid cc is between 3 and 5 times as fast, taking the difference between the hardware used into account. For the interval-4 example, Hybrid cc is twice as fast as clp(Newton), and for the combustion example 50% slower.

We present a longer example illustrating the use of Hybrid cc in modeling a hybrid system. 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 `trap Pocketed in ...` — at that moment all program fragments 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. `lcont(x)` asserts that the variable `x` is left continuous, and `rcont(x)` asserts that the variable `x` is right continuous.

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 {
      cont(px), cont(py),
      lcont(vx), lcont(vy),
      if (ChangeX || ChangeY) then direction = vx^2/(vx^2 + vy^2),
      unless (ChangeX || ChangeY) then 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
      }
    } watching Pocketed
  }
```

```
},
Edges = (){
  always forall Ball(X) do {
    if (X.px = radius || X.px = xMax - radius) then {
      XEdgeCollision,
      X.ChangeX, X.vx = -prev(X.vx)
    },
    if (X.py = radius || X.py = yMax - radius) then {
      YEdgeCollision,
      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 {
Collision,
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
    if (prev(X.px)^2 + prev(X.py)^2 <= pocket^2
      ||prev(X.px)^2 + (prev(X.py)-yMax/2)^2 <= pocket^2
      ||prev(X.px)^2 + (prev(X.py)-yMax)^2 <= pocket^2
      ||(prev(X.px)-xMax)^2 + prev(X.py)^2 <= pocket^2
      ||(prev(X.px)-xMax)^2 + (prev(X.py)-yMax/2)^2 <= pocket^2
      ||(prev(X.px)-xMax)^2 + (prev(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()
```

The last few lines set up the initial configuration. We ran this program for 74 simulated time units, after which all the balls were either at rest or pocketed — the total time of execution was 0.77 seconds on an UltraSparc 2.

5 Related work

The SHIFT programming language developed at UC Berkeley [DGS96, SDG96] is also intended for simulation of hybrid systems. Programs in SHIFT are synchronous concurrent collections of hybrid automata [ACH+95]. Computations proceed in alternating point and interval phases, as in Hybrid cc. SHIFT has an object-oriented framework for constructing models, and also has constructs with side-effects which are useful in writing state-machines. However it is not a declarative language — the transitions and states have to be explicitly programmed. The interaction of concurrency and side-effects also causes semantic problems in maintaining determinism. In the current implementation of SHIFT, only fixed step-size Runge-Kutta integration is supported, and breakpoints can occur only at the end of a step.

Differential equations with intervals are an active field of research, we will not attempt to provide a survey here. Most of the research is concerned with using intervals to provide validated solutions to differential equations, *i.e.* a statement of the form that the solution must lie in a certain interval. For a starting point into the field, see the tutorial by George Corliss [Cor95].

The field of interval reasoning is even larger. Several systems for interval constraint solving have been built, one of the recent ones is clp(Newton) [VMK95, vHLB97], which uses similar propagation methods, but also exploits multiple representations of constraints. This naturally leads to better pruning of the intervals in many problems. However the above comparison shows that in many problem instances, the performance of our system is comparable to that of clp(Newton).

6 Conclusion and Future Work

We have presented an implementation of a programming language for hybrid systems, Hybrid cc. The key feature of the language, which is reflected in the implementation, is that it is constraint-based, using interval constraints. The interval constraints are necessary to make Hybrid cc applicable to many modeling problems, and the interval propagation used inside the numerical solver for differential equations improves the accuracy of the integration.

Hybrid cc will be used to construct real-life models for engineering and educational purposes. We have already started some work in this direction, and plan to use Hybrid cc for simulation of rovers and spacecraft.

References

[ACH+95] 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.

[AH83] Gotz Alefeld and Jurgen Herzberger. *Introduction to Interval Computations*. Academic Press, 1983.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[Car95] Bjorn Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University, 1995.

[Cor95] G. F. Corliss. Guaranteed error bounds for ordinary differential equations. In W.A.Light and M.Marletta, editors, *Theory of Numerics in Ordinary and Partial Differential Equations*, volume IV of *Advances in Numerical Analysis*, pages 1–75. Oxford University Press, 1995.

[DGS96] Akash Deshpande, Aleks Gollu, and Luigi Semenzato. The SHIFT programming language and run-time system for dynamic networks of hybrid automata. Technical report, UC Berkeley PATH Project, 1996. `www-path.eecs.berkeley.edu/shift/doc/ieeshift.ps`.

[GJS97] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Computing with continuous change. *Science of Computer Programming*, 1997. To appear. Available from http://ic.arc.nasa.gov/people/vgupta.

[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.

[GSS95] Vineet Gupta, Vijay Saraswat, and Peter Struss. A model of a photocopier paper path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, August 1995.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[HSD92] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University, 1992.

[Jan94] S. Janson. *AKL – A Multiparadigm Programming Language*. PhD thesis, Uppsala University, 1994.

[Kay96] Herbert Kay. *Refining Imprecise Models and Their Behaviors*. PhD thesis, University of Texas at Austin, 1996.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.

[Rei80] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81 – 132, 1980.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1993.

[SDG96] Luigi Semenzato, Akash Deshpande, and Aleks Gollu. The SHIFT reference manual. Technical report, UC Berkeley PATH Project, 1996. `www-path.eecs.berkeley.edu/shift/doc/shift.ps.gz`.

[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.

[vHLB97] Pascal van Hentenryck, Michel Laurent, and Frederic Benhamou. Newton: Constraint programming over non-linear constraints. *Science of Programming*, 1997. to appear.

[VMK95] Pascal Van Hentenryck, David McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal of Numerical Analysis*, 1995. (Accepted). (Also available as Brown University technical report CS-95-01.).