Programming in hybrid constraint languages

Vineet Gupta ⋆ Radha Jagadeesan ⋆⋆ Vijay Saraswat⋆ Daniel G Bobrow⋆

Abstract. We present a language, Hybrid cc, for modeling hybrid systems compositionally. This language is declarative, with programs being understood as logical formulas that place constraints upon the temporal evolution of a system. We show the expressiveness of our language by presenting several examples, including a model for the paperpath of a photocopier. We describe an interpreter for our language, and provide traces for some of the example programs.

1 Introduction and Motivation

The constant marketplace demand of ever greater functionality at ever lower price is forcing the artifacts our industrial society designs to become ever more complex. Before the advent of silicon, this complexity would have been unmanageable. Now, the economics and power of digital computation make it the medium of choice for gluing together and controlling complex systems composed of electro-mechanical and computationally realized elements.

As a result, the construction of the software to implement, monitor, control and diagnose such systems has become a gargantuan, nearly impossible, task. For instance, traditional product development methods for reprographics systems (photo-copiers, printers) involve hundreds of systems engineers, mechanical designers, electrical, hardware and software engineers, working over tens of months through several hard-prototyping cycles. Software for controlling such systems is produced by hand – separate analyses are performed by the software, hardware and systems engineers, with substantial time being spent in group meetings and discussions coordinating different design decisions and changes, and assessing the impact these decisions will have on the several teams involved. Few, if any, automated tools are available to help in the production of documentation (principles of operation, module/system operating descriptions), or in the analysis and design of the product itself. A different team — often starting from little else than the product in a box, with little "documentation" — produces paper and computational systems to help field-service representatives diagnose and fix such products in the field.

Such work practices are unable to deal with the increasing demand for faster time to market, and for flexibility in product lines. Instead of producing one product, it is now necessary to produce a family of "plug-and-play" generic components — finishers (stackers, staplers, mailboxes), scanners, FAX modules, imaging systems, paper-trays, high-capacity feeders — that may come together for the very first time at customer-site.

⋆ Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto Ca 94304; {vgupta,saraswat,bobrow}@parc.xerox.com
⋆⋆ Dept. of Mathematical Sciences,Loyola University-Lake Shore Campus, Chicago, Il 60626; radha@math.luc.edu

The software for controlling such systems must be produced in such a way that it can work even if the configuration to be controlled is known only at run-time.

To address some of these problems, we are investigating the application of *model-based computing* techniques. The central idea is to develop compositional, declarative models of the various components of a photo-copier product family, at different levels of granularity, customized for different tasks. For instance, a marker is viewed as a transducer that takes in (timed) streams of sheets S and video images V, to produce (timed) streams of prints P. A model for the marker, from the viewpoint of the scheduler, is a set of constraints that capture precisely the triples $\langle S, V, P \rangle$ which are in fact physically realizable on this marker. The marker model is itself constructed from models for nips, rollers, motors, belts, paper baffles, solenoids, control gates etc.; the models are hooked together in exactly the same way as the corresponding physical components are linked together. Together with this, software architectures are developed for the tasks that need to be accomplished (such as scheduling, simulation, machine control, diagnostic tree generation). Finally, linking the two are special-purpose reasoners (operating at "configuration-time") that produce information of the right kind for the given task architecture, given the component models and system configuration.

We expect this approach to be useful for a variety of tasks:

– **Scheduling.** A model of the system specifies what outputs will be produced given (perhaps continuous) input and control commands. In principle, the same model can be used to search the space of inputs and control commands, given a description of system output. In practice, the design of such an inference engine is made complex by the inherent combinatorial complexity of the search process. With a formal model in hand, it becomes possible to use automated techniques to better understand the search-space and design special-purpose reasoners. This approach is currently being deployed, in collaboration with several other teams, in the development of schedulers for a new generation of products from Xerox Corporation.
– **Code generation.** A physical model of a system can be used for *envisionment* [dKB85] — studying the possible paths of evolution of the system. Since some of the parameters of a system can be controlled, it becomes possible to develop automatically controllers which would specify these parameters leading the system to a desirable state, and away from paths which lead to unsafe states.
– **Diagnostic-tree generation.** Given a model of components, their interconnection, and their correct (and possibly faulty) behavior — perhaps with other information such as prior probabilities for failure — it is possible to construct off-line repair-action procedures[FBB+94]. These can be used by service technicians as guides in making probes to determine root cause for the manifest symptom.
– **Explanation.** Given a simulation-based model, it is possible to annotate behavioral rules with text in such a way that the text can be systematically composed to provide natural-language explanations for why an observable parameter does or does not have a particular value.
– **Productivity Analysis.** Models may be analyzed to determine how corresponding product designs will perform on different job mixes (e.g., a sequence of all single-sided black-&-white jobs, followed by all double-sided, color, stapled jobs).

The model-based computing approach places a set of requirements on the nature of the modeling language. We motivate these demands by considering a concrete example — the paper path of a simple photocopier, see Figure 1.

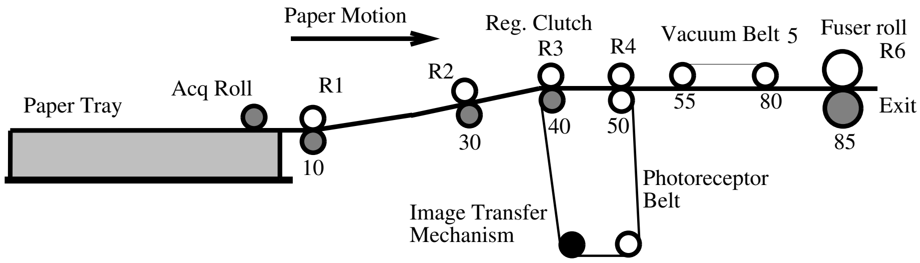

Fig. 1. A simple photocopier

Example: Paper Transportation in a Photocopier. In this photocopier, paper is loaded in a paper tray at the left of the machine. When a signal is received, the acquisition roll is lowered onto the paper and pulls the top sheet of paper towards the first set of rollers (R1). After the paper is grasped by the first set of rollers, the acquisition roll is lifted, and the rollers pull the paper forward, till it reaches the registration clutch (R3). This starts at a precise moment for perfect alignment with the toner image on the belt. The toner is transferred electrostatically onto the paper by the image transfer mechanism. The vacuum belt (5) transports it to the fuser roll (R6) which fuses the toner into the paper, and from where it exits.

Hybrid modeling. The photocopier above has a collection of components with continuous behavior, for example, the rollers and belts. The control program of the photocopier is a discrete event driven system. Therefore, the modeling language should be able to describe the interaction of the control program and the continuous components — thus, the modeling language has to fall in the framework of hybrid systems [BG90,NSY91,MMP92,GNRR93].

Executability. Given a model of the system, it should be possible to predict the behavior of the system when inputs are supplied. Thus the model should be executable, *i.e.* it should also be possible to view models as *programs*. This would allow hybrid control programs to be written using the same notation as component models. If sensors and actuators coupling the language implementation with the physical environment are provided then, in fact, it should be possible to use programs in this notation to drive physical mechanisms.

Compositional modeling. Out of considerations of reuse, it seems clear that models of composite systems should be built up from models of the components, and that models of components should reflect their physics, without reflecting any pre-compiled knowledge of the structure (configurations) in which they will be used (the "no function in structure principle", [dKB85]). Concretely, this implies that the modeling language must be expressive enough to modularly describe and support extant control architectures and techniques for compositional design of hybrid systems, see for example [HMP93, Hoo93,NK93]. From a programming language standpoint, these modularity concerns are addressed by the analysis underlying synchronous programming languages [BB91, Hal93,BG92,HCP91,GBGM91,Har87,CLM91,SJG95], (adapted to dense discrete domains in [BBG93]); this analysis leads to the following demands on the modeling language.

The modeling should support the interconnection of different components, and allow for the hiding of these interconnections. For example, the model of the copier is the composition of the the control program and the model of the paper path. The model of the paper path in turn, is the composition of the models of the rollers, clutches etc.

Furthermore, in the photocopier example, if paper jams, the control program should cause the rollers to stop. However, the model of the rollers, in isolation, does not need to have any knowledge of potential error conditions. Thus, the modeling language should support *orthogonal preemption* [Ber93] — any signal can cause preemption.

Finally, the language should allow the expression of multiple notions of *logical time* — for example, in the photocopier the notion of time relevant to the paper tray is (occurrences of) the event of removing paper from the tray; the notion of time relevant to the acquisition roll is determined by the rotation rate of the roller; the notion of time of the image transfer mechanism is the duration of the action of transferring images etc.

Thus, we demand that the modeling language be an *algebra of processes, that includes concurrency, hiding, preemption and multiform time*.

Declarative view. It must be possible for systems engineers — people quite different in training and background from software engineers — to use such a formalism. Typically, systems engineers understand the physics of the system being designed or analyzed, and are used to mathematical or constraint-based formalisms (equational and algebraic models, transfer functions, differential equations) for expressing that knowledge. This suggests that it must be possible to view a model (fragment) expressed in the language as a declaration of facts in a (real-time) (temporal) logic — see for example [MP91,AH92].

Reasoning. Given a model of the paper-path, and the control procedures governing it, it should be possible to perform a *tolerance analysis* — establish the windows on early/late arrival of sheets of papers at various sensors on the paper-path, given that a particular physical component may exhibit any behavior within a specified tolerance. This, and other such engineering tasks, suggest that the modeling language must be *amenable* to (adapting) the methodology developed in the extensive research on reasoning about hybrid and real-time systems — for example, specification and verification of properties of hybrid systems [GNRR93,dBHdRR92], qualitative reasoning about physical systems [Wd89], and envisionment of qualitative states [Kui94].

1.1 This paper

This paper describes programming in the modeling language, Hybrid cc — hybrid concurrent constraint programming. Intuitively, Hybrid cc is obtained by "freely extending" an untimed non-monotonic language Default cc over continuous time. Hybrid cc has the following key features:

- The notion of a *continuous* constraint system describes the continuous evolution of system trajectories.
- Hybrid constraint languages — developed generically over continuous constraint systems — are obtained by adding a single temporal construct, called **hence**. Intuitively, a formula **hence** A is read as asserting that A holds continuously beyond the current instant.
- Continuous variants of preemption-based control constructs and multiform timing constructs are definable in Hybrid cc.

The formal foundations of Hybrid cc are discussed in [GJSB95].

The rest of this paper is organised as follows. First, we describe the computational intuitions underlying Hybrid cc. We follow with a brief description of an interpreter for a language in the Hybrid cc framework. We then describe a series of examples, and show traces of the execution of the interpreter. These examples illustrate the programming idioms and expressiveness of Hybrid cc.

2 Hybrid cc: Computational intuitions

2.1 Background

Concurrent Constraint Programming As mentioned in the previous section, some of the crucial characteristics in a hybrid programming language are easy specification and composition of model fragments. This led us to consider Concurrent Constraint Programming languages [Sar93,SRP91] as a starting point, since these languages are built on top of constraint systems, which can be made as expressive as desired, and they have very fine-grained concurrency, so compositionality is achieved without effort. Recently, several concrete general-purpose programming languages have been implemented in this paradigm [Deb93,JH91,SHW94].

Concurrent Constraint Programming (cc) languages are declarative concurrent languages, where each construct is a logical formula. cc replaces the traditional notion of a store as a valuation of variables with the notion of a store as a constraint on the possible values of variables. Thus the store consists of pieces of information which restrict the possible values of the variables. A cc program consists of a set of agents[3] running concurrently and interacting with the shared store. Agents are of two basic kinds — *tell* agents which add information to the store (written a), and ask agents, or conditionals

[3] We use the words *agent* and *program* interchangeably in this paper, usually referring to fragments of a program as agents.

(written **if** a **then** A), which query the store about the validity of some information, and reduce to other agents if it is valid[4]. This yields the grammar:

$$A ::= a \mid \textbf{if } a \textbf{ then } A \mid \textbf{new } X \textbf{ in } A \mid A, A$$

Computation is monotonic — information can only be added to the store. Ask actions are used for synchronization — if a query is answered positively, then the agent can proceed, otherwise it waits (possibly forever) till there is enough information in the store to entail the information in the query. When no more computation is being performed (a state of *quiescence* is reached), the store is output.

The information that is added to the store consists of constraints which are drawn from a *constraint system*. Formally, a constraint system $\mathcal{C}$ is a system of partial information, consisting of a set of primitive constraints or tokens[5] D with minimal first order structure — variables and existential quantification. Associated with a constraint system is an entailment relation (denoted $\vdash_{\mathcal{C}}$) which specifies when a token a can be deduced from some others $b_1, \ldots, b_n$, denoted by $b_1, \ldots, b_n \vdash_{\mathcal{C}} a$. Examples of such systems are the system Herbrand, underlying logic programming — here tokens are equalities over terms which are finite trees with variables ranging over trees — and FD [HSD92] or finite domains, its tokens are equalities of variables and expressions saying that the range of a variable is some finite set.

A salient aspect of the cc computation model is that agents may be thought of as imposing constraints on the evolution of the system. The agent a imposes the constraint a. (A, B) imposes the constraints of both A and B — logically, this is the conjunction of A and B. **new** X **in** A imposes the constraints of A, but hides the variable X from the other agents — logically, this can be thought of as a form of existential quantification. The agent **if** a **then** A imposes the constraints of A provided that the rest of the system imposes the constraints a — logically, this can be thought of as intuitionist implication.

This declarative way of looking at programs is complemented by an operational view. The basic idea in the operational view is that of a network of agents interacting with a shared store of primitive constraints. The agent a is viewed as adding a to the store instantaneously. (A, B) behaves like the simultaneous execution of both A and B. **new** X **in** A starts A but creates a new local variable X, so no information can be communicated on it outside. The agent **if** a **then** A behaves like A if the current store entails a.

The main difficulty in using cc languages in modeling is that the cc programs can detect only the presence of information, not its absence. However, in reactive systems, it is important to handle information saying that a certain event did not occur — examples are timeouts in UNIX or the presence of jams in a photocopier (usually inferred by the paper not reaching a point by some specified time). The problem in assuming the absence of information (called negative information) during a computation is that it may be invalidated later when someone else adds that information, leading to invalidation of the subsequent computation.

[4] There are also hiding agents **new** X **in** A which hide any information about X from everyone else, these are needed for modularity.

[5] Tokens are denoted by $a, b, \ldots$ in this paper.

Our first attempt at fixing this problem was the addition of a sequence of phases of execution to the cc paradigm [SJG94b,SJG94a]. At each phase we executed a cc program, and this gave the output for the phase, and also produced the agents to be executed in subsequent phases. At the end of each phase we detected the absence of information, and used it in the *next* phase, giving us a reactive programming language, tcc. Each phase was denoted by a time tick, so we had a discrete model of time.

Default cc. The addition of time to cc however still left us with a problem—how can we detect negative information *instantaneously*? This is necessary because in some applications we found that it was not acceptable to wait until the next phase to utilize negative information [SJG95].

Since the negative information is to be detected in the same time step, it must be done at the level of the untimed language, cc. However, the cc paradigm is inherently opposed to instantaneous negative information detection. Thus in order to allow negative information to be detected within the same computation cycle, we have to modify cc — we have to use Default cc. The basic idea behind Default cc is — if the final output of a computation is known, then any negative information can be obtained from this output, and this negative information can never be invalidated, as the final output has all the information ever added. So a Default cc program executes like a cc program, except that before the beginning of the execution, it guesses the output store. All negative information requests are resolved with respect to this guess, other than that the Default cc program adds information to the store and queries the store for positive information just like a cc program. At the end of the computation, if the store is equal to the guess, then the guess was correct, and this is a valid answer. Otherwise, this branch of execution is terminated (or we can do backtracking). In an actual implementation, this backtracking is done at compile time, so at runtime we have to do a simple table lookup.

The Default cc paradigm augments the cc paradigm with the ability to detect negative information, so the only new syntax we need to add to the cc syntax is the negative ask combinator **if** a **else** A, this reduces to A if the guessed output e does not imply a. Thus the Default cc syntax is given by the grammar

$$A ::= a \mid \textbf{if } a \textbf{ then } A \mid A, A \mid \textbf{new } X \textbf{ in } A \mid \textbf{if } a \textbf{ else } A$$

The agent **if** a **else** A imposes the constraints of A unless the rest of the system imposes the constraint a — logically, this can be thought of as a *default* [Rei80].[6] (In the timed contexts discussed below, the subtlety of the non-monotonic behavior of programs — intimately related to the causality issues in synchronous programming languages — arises from this combinator.) To see this, note that A may itself cause further information to be added to the store at the current time instant; and indeed, several other agents may simultaneously be active and adding more information to the store. Therefore requiring

[6] Note that **if** a **else** A is quite distinct from the agent **if** $\neg a$ **then** A (assuming that the constraint system is closed under negation). The former will reduce to A iff the final store does not entail a; the latter will reduce to A iff the final store entails $\neg a$. The difference arises because, by their very nature, stores can contain partial information; they may not be strong enough to entail either a or $\neg a$.

that information a be absent amounts to making a demand on "stability" of negative information.

Thus we now have a language that permits instantaneous negative information detection. This enables us to write strong timeouts — if a signal is not produced at a certain time, the execution of the program can be terminated at that very time, not at some later time as we did for timed cc. Now in order to get a language for modeling discrete reactive systems, we again introduce phases in each of which a Default cc program executes. To extend Default cc across time, we need just one more construct, **hence** A, which starts a copy of A in each phase *after* the current one. We extend Default cc across real time to get a language for hybrid reactive systems in the next subsection.

2.2 Continuous evolution over time.

We follow a similar intuition in developing Hybrid cc: the continuous timed language is obtained by uniformly extending Default cc across real (continuous) time. This is accomplished by two technical developments. For the formal development, see [GJSB95].

Continuous Constraint Systems. First, we enrich the underlying notion of constraint system to make it possible to describe the continuous evolution of state. Intuitively, we allow constraints expressing initial value (integration) problems, e.g. constraints of the form $\mathtt{X} = \mathtt{0}, \mathtt{hence}\ \mathtt{dot(X)} = \mathtt{1}$; from these we can infer at time $\mathtt{t}$ that $\mathtt{X} = \mathtt{t}$. The technical innovation here is the presentation of a *generic* notion of continuous constraint system (ccs), which builds into the very general notion of constraint sytems just the extra structure needed to enable the definition of continuous control constructs (without committing to a *particular* choice of vocabulary for constraints involving continuous time). As a result subsequent development is *parametric* on the underlying constraint language: for each choice of a ccs we get a hybrid programming language. Later we will give an example of a simple constraint system and the language built over it.

Program Combinators. Second we add to the untimed Default cc the same temporal control construct: **hence** A, but now interpreting it over real time. Declaratively, **hence** A imposes the constraints of A at every *real* time instant after the current one. Operationally, if **hence** A is invoked at time t, a new copy of A is invoked at each instant in (t, ∞).

Agents	Propositions
a	a holds now
if a **then** A	if a holds now, then A holds now
if a **else** A	if a will not hold now, then A holds now
new X **in** A	exists an instance $A[t/X]$ that holds now
A, B	both A and B hold now
hence A	A holds at every instant after now

Intuitively, **hence** might appear to be a very specialized construct, since it requires repetition of the *same* program at every subsequent time instant. However, **hence** can combine in very powerful ways with positive and negative ask operations to yield rich patterns of temporal evolution. The key idea is that negative asks allow the instantaneous preemption of a program — hence, a program **hence if** P **else** A will in fact not execute A at all those time instants at which P is true.

Let us consider some concrete examples. Suppose that we require that an agent A be executed at every time point beyond the current one until the time at which a is true. This can be expressed as **new** X **in** (**hence** (**if** X **else** A, **if** a **then always** X)). Intuitively, at every time point beyond the current one, the condition X is checked. Unless it holds, A is executed. X is local — the only way it can be generated is by the other agent (**if** a **then always** X), which, in fact generates X continuously if it generates it at all. Thus, a copy of A is executed at each time point beyond the current one upto (and excluding) the time at which a is detected.

Similarly, to execute A precisely at the first time instant (assuming there is one) at which a holds, execute: **new** X **in hence** (**if** X **else if** a **then** A, **if** a **then hence** X).

In particular, the continuous version of the general preemption control construct **clock** (introduced in [SJG94b]) is definable within Hybrid cc. The patterns of temporal behavior described above are obtainable as specializations of **clock**. Further, programs containing the **clock** combinator may be equationally rewritten into programs not containing the combinator.

While conceptually simple to understand, **hence** A requires the execution of A at every subsequent real time instant. Such a powerful combinator may seem impossible to implement computationally. For example, it may be possible to express programs of the form **new** T **in** ($T = 0$, **hence** $dot(T) = 1$, **hence if** $rational(T)$ **then** A) which require the execution of A at every rational $q > 0$. Such programs are not implementable. To make Hybrid cc computationally realizable, the basic intuition we exploit is that, in general, physical systems change slowly, with points of discontinuous change, followed by periods of continuous evolution. This is captured by a *stability* condition on continuous constraint systems that guarantees that for every constraint a and b there is a neighborhood around 0 in which a either entails or disentails b at every point. This rules out constraints such as `rational(T)` as inadmissible.

With this restriction, computation in Hybrid cc may be thought of as progressing in alternating phases of computation at a time point, and in an open interval. Computation at the time point establishes the constraint in effect at that instant, and sets up the program to execute subsequently. Computation in the succeeding open interval determines the length of the interval r and the constraint whose continuous evolution over $(0, r)$ describes the state of the system over $(0, r)$.

3 The implementation

We will now present a simple implementation for Hybrid cc built on top of a simple continuous constraint system. The interpreter is built on top of Prolog, and consideration has been given to simplicity rather than efficiency. We have already identified several ways in which performance can be significantly enhanced using standard logic programming techniques.

3.1 The continuous constraint system

Basic tokens are formulas d, **hence** d or $prev(d)$, where d is either an atomic proposition p or an equation of the form $dot(x, m) = r$, for x a variable, m a non-negative integer and r a real number. Tokens consist of basic tokens closed under conjunction and existential quantification. Some rudimentary arithmetic can be done using the underlying prolog primitives. The inference relations are defined in the obvious way under the interpretation that p states that p is true and $dot(x, m) = r$ states that the mth derivative of x is r. $prev(d)$ asserts that d was true in the limit from the left, and **hence** d states that for all time $t > 0$ d holds. The inference relations are trivially decidable: the functions of time expressible are exactly the polynomials. The only non-trivial computation involved is that of finding the smallest non-negative root of univariate polynomials (this can be done using one of several numerical methods).

3.2 The basic interpreter

The interpreter simply implements the formal operational semantics discussed in detail in [GJSB95]. It operates alternately in point and interval phases, passing information from one to the other as one phase ends. All discrete change takes place in the point state, which executes a simple Default cc program. In a continuous phase computation progresses *only* through the evolution of time. The interval state is exited as soon as the status of one of the conditionals changes — one which always fired does not fire anymore, or one starts firing.

The interpreter takes a Hybrid cc program and starts in the point phase with time $t = 0$.

The point phase. The input to the point phase is a set of Hybrid cc agents and a *prev* store, which carries information from the previous interval phase — the atomic propositions and the values of the variables at the right endpoint of the interval (initially at time 0 this is empty). The outputs are the store at the current instant, which is printed out and passed on to the next interval phase as the initial store, and the agent or the *continuation*, to be executed in the interval phase.

The interpreter takes each agent one by one and processes it. It maintains the store as a pair of lists — the first list contains the atomic propositions that are known to be true at the point, and the second contains a list of variables and the information that is known about the values of their derivatives. It also maintains lists of suspended conditionals (**if** …**then** and **if**…**else** statements whose constraints are not yet known to be true or false) and a list of agents to be executed at the next interval phase.

When a tell action (a constraint a) is seen, it is added to the store and a check for consistency is made. If the constraint is of the form $dot(x, m) = r$, then it means that all derivatives of x of order $< m$ are continuous, and thus their values are copied from the *prev* store. Any conditional agents that have been suspended on a are reactivated and placed in the pool of active agents.

When any conditional agent appears, its constraint is immediately checked for validity. If this can be determined, the appropriate action is taken on the consequence of the agent — it is discarded or added to the pool of active agents. Otherwise it is added to an indexed list of suspended conditionals for future processing. A **hence** action is placed in the list of agents to be executed in the next interval.

When no more active agents are left, the interpreter looks at the suspended agents of the form **if** a **else** A, the *defaults*. On successful termination, one of two conditions must hold for each such agent: either the final store entails a, or else the final store incorporates the effect of executing A. Thus all that the interpreter has to do is to non-deterministically choose one alternative for each such default. In the current implementation, this non-deterministic choice is implemented by backtracking, with execution in the current phase terminating with the first successful branch.

The interval phase. This is quite similar to the point phase, except that it also has to return the duration of the phase. The input is a set of agents and an initial store which determines the initial values for the differential equations true in the store. The output is a set of agents to be executed in the next point phase, a store giving the values of the variables at the *end* of the interval phase, and the length of the phase. Initially we assume that the length will be infinite, this is trimmed down during the execution of the phase.

A *tell* is processed as before — it is added to a similar store, and reactivates any suspended conditionals. Conditional agents are added to the suspended list if their conditions are not known to be valid or invalid yet, otherwise the consequence of the agent is added to the active pool or discarded. In addition, if the condition can be decided, then we compute the duration for which the status will hold. For example if we have $dot(x, 1) = 3$ in the store, and $x = 4$ in the initial store, then we know that $x = 3t + 4$. Now if we ask **if** $(x = 10)$ **then** A, then we immediately know that this is not true in the current interval, so A can be discarded for now. However, we also know, by solving the equation $3t + 4 = 10$ that this status will hold for only 2 seconds, so this interval phase cannot be longer than two seconds. (This is where we need to solve the polynomial equations.)

At the end of the processing, the defaults are processed as before, and are used to trim the length of the interval phase further. Also, the remaining suspended conditionals **if** a **then** $\ldots$ are checked to when a could first become valid, to reduce the duration of the phase if necessary. Finally a computation is done to find the store at the end of the interval, and this information is passed to the next point phase. Since all agents are of the form **hence** A in an interval phase, the agent to be executed at the next point phase must be $A,$ **hence** A.

If any interval phase has no agents to execute, the program terminates.

3.3 Combinators

This basic syntax of Hybrid cc has been quite enough for us to write all the programs that we have. However, we have noticed several recurrent patterns in our programs, and some of these can be written up as definable combinators, and make the job of programming considerably easier. Note that these are defined combinators, *i.e.* they are definable in terms of the basic combinators. Thus, these defined combinators are merely syntactic sugar, and can be compiled away into basic combinators. The formal principles underlying the compilation process are discussed in a companion paper [GJSB95].

Always. **always** $A = A$, **hence** A is read logically as $\Box A$, the necessity modality. Parameterless recursion can be mimicked — replace a recursive process $P :: body$ by **always if** P **then** $body$. Parameters can be handled by textual substitution.

Waiting for a condition. **whenever** a **do** A reduces to A at the first time instant that a becomes true — if there is a well-defined notion of first occurrence of a. Note that there is no "first" occurrence of a in the situation when the a occurs in the interval $(0, t)$, as happens in the agent **hence** a. In such cases, A will not be invoked. **whenever** can be defined in terms of the basic combinators as mentioned in the previous section.

In the following discussion on definable combinators, and in the rest of this paper, we shall not repeat the caveat about the the subtlety of the well-definedness of the "first" occurrences of events.

Watchdogs. **do** A **watching** a, read logically as "A until a", is the strong abortion interrupt of ESTEREL [Ber93]. The familiar `cntrl-C` is a construct in this vein. **do** A **watching** a behaves like A until the first time instant when a is entailed; when a is entailed A is killed instantaneously. It can be defined using the basic constructs as follows.

$$
\begin{aligned}
\textbf{do } b \textbf{ watching } a &= \textbf{if } a \textbf{ else } b \\
\textbf{do if } b \textbf{ then } A \textbf{ watching } a &= \textbf{if } b \textbf{ then do } A \textbf{ watching } b \\
\textbf{do if } b \textbf{ else } A \textbf{ watching } a &= \textbf{if } b \textbf{ else do } A \textbf{ watching } a \\
\textbf{do } A, B \textbf{ watching } a &= \textbf{do } A \textbf{ watching } a, \textbf{do } B \textbf{ watching } a \\
\textbf{do new } X \textbf{ in } A \textbf{ watching } a &= \textbf{new } X \textbf{ in do } A \textbf{ watching } a \quad \text{X not free in } a \\
\textbf{do hence } A \textbf{ watching } a &= \textbf{new } X \textbf{ in } (\textbf{hence } [\textbf{if } X \textbf{ else do } A \textbf{ watching } a, \\
&\qquad\qquad\qquad \textbf{if } a \textbf{ then hence } X])
\end{aligned}
$$

Multiform time. **time** A **on** a denotes a process whose notion of time is the occurrence of a — A evolves only the store entails a. It can be defined by structural induction as above.

Implementation of the defined constructs. The laws given above suffice to implement all these constructs in the interpreter. However for the sake of efficiency, we have implemented the constructs directly, at the same level as basic constructs. The **always** construct is of course trivial, and has been implemented using the given rule. **whenever** is implemented similarly to an **if** a **then** A construct, except that if a is not entailed, then it is carried across to the next phase. It has no effect in an interval phase for the reason mentioned above.

do A **watching** a is very similar to an **if** b **else** B, and is implemented analogously. The only thing to be noted here is that whenever something from the A is to be passed on to the next phase, it must be passed inside a **do** $\ldots$ **watching** a, so that the proper termination of all spawned agents can be achieved when a becomes true. **time** A **on** a is similar — it is like an **if** a **then** A, remembering the same rule about passing spawned agents through phases.

4 Programming examples

We illustrate the programming/model description style of Hybrid cc via a few examples. In each of these examples, we write each of the conceptually distinct components separately. We then exploit the expressive power of Hybrid cc to combine the subprograms (submodels) via appropriate combinators, to get the complete program/model. We also present the trace of the interpreter on some of these examples.

4.1 Sawtooth function

We start off with a simple example, the sawtooth function defined as $f(y) = y - \lfloor y \rfloor$, where $\lfloor y \rfloor$ denotes the greatest integer in y.

As mentioned above, prev(x=1.0) is read as asserting that the left limit of x equals 1.

$\mathtt{sawtooth} ::$
$\quad \mathtt{x} = \mathtt{0},$
$\quad \textbf{hence}\ (\textbf{if}\ \mathtt{prev}(\mathtt{x} = \mathtt{1.0})\ \textbf{then}\ \mathtt{x} = \mathtt{0}),$
$\quad \textbf{hence}\ (\textbf{if}\ \mathtt{prev}(\mathtt{x} = \mathtt{1.0})\ \textbf{else}\ \mathtt{dot}(\mathtt{x}) = \mathtt{1})).$
.

The trace of this program as executed by our interpreter is seen below. The execution in the point and interval phases is displayed separately. For both phases of execution, the contents of the store are displayed. In addition, for the interval phase, the interval is also displayed, and the continuous variables are displayed as polynomials in time. In the interval, note that time is always measured from the beginning of the interval, not from 0. Also, this program has infinite output, so we aborted it after some time.

```
Time = 0.
Store contains [dot(x,0)=0].

Time Interval is (0, 1.0)
Store contains [x=1*t+0].

Time = 1.0.
Store contains [dot(x,0)=0].

Time Interval is (1.0, 2.0)
Store contains [x=1*t+0].

Time = 2.0.
Store contains [dot(x,0)=0].

Time Interval is (2.0, 3.0)
Store contains [x=1*t+0].
```

```
Time = 3.0.
Store contains [dot(x,0)=0].
Prolog interruption (h for help)? a
{Execution aborted}
| ?-
```

4.2 Temperature Controller.

We model a simple room heating system which consists of a furnace which supplies heat, and a controller which turns it on and off. The temperature of the furnace is denoted `temp`. The furnace is either on (modeled by the signal `furnace_on`) or off (modeled by the signal `furnace_off`). The actual switching is modeled by the signals `switch_on` and `switch_off`. When the furnace is on, the temperature rises at a given rate, `HeatR`. When the furnace is off, the temperature falls at a given rate, `CoolR`. The controller detects the temperature of the furnace, and switches the furnace on and off as the temperature reaches certain pre-specified thresholds: `Cut_out` is the maximum temperature and `Cut_in` is the minimum temperature.

The heating of the furnace is modeled by the following program. The multiform time construct ensures that heating occurs only when the signal `furnace_on` is present.

$$\begin{array}{l}\mathtt{furnace\_heat(HeatR)} :: \\ \quad \textbf{time}\ \ (\textbf{always}\, \mathtt{dot(temp)} = \mathtt{HeatR})\quad \textbf{on}\ \mathtt{furnace\_on}.\end{array}$$

The cooling of the furnace is modeled by the following program. The multiform time construct ensures that heating occurs only when the signal `furnace_off` is present.

$$\begin{array}{l}\mathtt{furnace\_cool(CoolR)} :: \\ \quad \textbf{time}\ \ (\textbf{always}\ \ \mathtt{dot(temp)} = -\mathtt{CoolR})\ \ \textbf{on}\ \mathtt{furnace\_off}.\end{array}$$

The furnace itself is the parallel composition of the heating and cooling programs.

$$\begin{array}{l}\mathtt{furnace(HeatR,\ CoolR)} :: \\ \quad \mathtt{furnace\_heat(HeatR),\ furnace\_cool(CoolR)}.\end{array}$$

The controller is modeled by the following program — at any instant, the program watches for the thresholds to be exceeded, and turns the appropriate switch on or off.

$$\mathtt{controller(Cut\_out,\ Cut\_in)} ::$$

$$\begin{array}{l}\textbf{always}\ (\textbf{if}\ \texttt{switch\_on}\ \textbf{then do}\ (\textbf{always}\ \texttt{furnace\_on})\ \textbf{watching}\ \texttt{switch\_off},\\ \quad\quad \textbf{if}\ \texttt{switch\_off}\ \textbf{then do}\ (\textbf{always}\ \texttt{furnace\_off})\ \textbf{watching}\ \texttt{switch\_on},\\ \quad\quad \textbf{if}\ \texttt{prev}(\texttt{temp} = \texttt{Cut\_out})\ \textbf{then}\ \texttt{switch\_off},\\ \quad\quad \textbf{if}\ \texttt{prev}(\texttt{temp} = \texttt{Cut\_in})\ \textbf{then}\ \texttt{switch\_on}).\end{array}$$

The entire assembly is defined by the parallel composition of the furnace and the controller.

$$\begin{array}{l}\texttt{controlled\_furnace}(\texttt{HeatR},\ \texttt{CoolR},\ \texttt{Cut\_out},\ \texttt{Cut\_in}) ::\\ \quad (\texttt{furnace}(\texttt{HeatR},\ \texttt{CoolR}),\\ \quad\ \ \texttt{controller}(\texttt{Cut\_out},\ \texttt{Cut\_in})).\end{array}$$

The trace of this program with parameters `HeatR = 2, CoolR = -0.5, Cut_Out = 30, Cut_in = 26` and initial conditions `temp = 26` and the signal `switch_on`, as executed by our interpreter is seen below.

```
Time = 0.
Store contains [furnace_on,switch_on,dot(temp,1)=2.0,
dot(temp,0)=26.0].

Time Interval is (0, 2.0)
Store contains [furnace_on,temp=2.0*t+26.0].

Time = 2.0.
Store contains [furnace_off,switch_off,dot(temp,1)= -0.5,
dot(temp,0)=30.0].

Time Interval is (2.0, 10.0)
Store contains [furnace_off,temp= -0.5*t+30.0].

Time = 10.0.
Store contains [furnace_on,switch_on,dot(temp,1)=2.0,
dot(temp,0)=26.0].

Time Interval is (10.0, 12.0)
Store contains [furnace_on,temp=2.0*t+26.0].
Prolog interruption (h for help)? a
{Execution aborted}
```

4.3 Cat and Mouse.

Next, we consider the example of a cat chasing a mouse. A mouse starts the origin, running at speed 10 metres/second for a hole 100 metres away. After it has traveled 50 metres, a cat is released, that runs at speed 20 metres/second after the mouse. The positions of the cat and the mouse are modeled by `c` and `m` respectively. The cat wins (modeled by the signal `wincat`) if it catches the mouse before the hole, and loses otherwise (modeled by the signal `winmouse`).

The cat and the mouse are modeled quite simply. The positions of the cat and the mouse change according to the respective velocities. Note that there is no reference to the cat or the position of the hole in the mouse program; similarly for the cat program. This is essential for *reuse* of these models.

$\mathtt{mouse} :: \mathtt{M} = \mathtt{0},\ \textbf{always}\,\mathtt{dot}(\mathtt{M}) = \mathtt{10}.$
$\mathtt{cat} :: \mathtt{C} = \mathtt{0},\ \textbf{always}\,\mathtt{dot}(\mathtt{C}) = \mathtt{20}.$

The entire system is given by the assembly of the components. The mouse wins if it reaches the hole `m=100`; the cat wins if `c=100` and the mouse has not reached the hole. The combination

$$\textbf{do}\ (\ldots)\ \textbf{watching}\,\mathtt{prev}(\mathtt{m} = \mathtt{100}), \textbf{whenever}\,\mathtt{m} = \mathtt{100}\ \textbf{do}\ \ldots$$

is used as an exception handler; it handles the case of the mouse reaching the hole. Similarly, the combination

$$\textbf{do}\ (\ldots)\ \textbf{watching}\,\mathtt{prev}(\mathtt{c} = \mathtt{100})), \textbf{whenever}\,\mathtt{prev}(\mathtt{c} = \mathtt{100})\ \textbf{do}\ \ldots$$

is used as an exception handler; it handles the case of the cat reaching hole. The higher priority of **do** `...` **watching** `prev(c=100)` relative to **do** `(...)` **watching** `prev(m=100)` is captured by lexical nesting — in this program the cat catches the mouse if both reach the hole simultaneously.

```
system_configuration ::
   do (do (mouse,
           whenever prev(m = 50) do cat)
        watching prev(m = 100),
       whenever prev(m = 100) do winmouse)
    watching prev(c = 100),
   whenever prev(c = 100) do wincat.
```

A trace of this program as executed by the interpreter is given below:

```
Time = 0.
Store contains [dot(m,1)=10,dot(m,0)=0].

Time Interval is (0, 5.0)
Store contains [m=10*t+0].

Time = 5.0.
Store contains [dot(m,1)=10,dot(m,0)=50.0,dot(c,1)=20,
dot(c,0)=0].
```

```
Time Interval is (5.0, 10.0)
Store contains [c=20*t+0,m=10*t+50.0].

Time = 10.0.
Store contains [wincat].

Termination after Time 10.0.
 Initial Conditions: [wincat].
 Store has [].

yes
```

The other alternative, *i.e.* the mouse wins if if the cat and the mouse reach the hole simultenously. is captured by reversing the lexical nesting of **do** `(...)` **watching** `prev(m=100)` and **do** `(...)` **watching** `prev(c=100)`. Concretely, the program is:

$$
\begin{array}{l}
\texttt{system\_configuration} :: \\
\quad \textbf{do}\ (\textbf{do}\ (\texttt{mouse}, \\
\qquad\qquad \textbf{whenever}\ \texttt{prev}(\texttt{m} = \texttt{50})\ \textbf{do}\ \texttt{cat}) \\
\qquad\quad \textbf{watching}\ \texttt{prev}(\texttt{c} = \texttt{100}), \\
\qquad\quad \textbf{whenever}\ \texttt{prev}(\texttt{c} = \texttt{100})\ \textbf{do}\ \texttt{wincat}) \\
\quad\ \ \textbf{watching}\ \texttt{prev}(\texttt{m} = \texttt{100}), \\
\quad \textbf{whenever}\ \texttt{prev}(\texttt{m} = \texttt{100})\ \textbf{do}\ \texttt{winmouse}.
\end{array}
$$

A trace of this program as executed by the interpreter is given below:

```
Time = 0.
Store contains [dot(m,1)=10,dot(m,0)=0].

Time Interval is (0, 5.0)
Store contains [m=10*t+0].

Time = 5.0.
Store contains [dot(m,1)=10,dot(m,0)=50.0,dot(c,1)=20,
dot(c,0)=0].

Time Interval is (5.0, 10.0)
Store contains [c=20*t+0,m=10*t+50.0].

Time = 10.0.
Store contains [winmouse].

Termination after Time 10.0.
 Initial Conditions: [winmouse].
 Store has [].
```

yes

4.4 A simple game of Billiards

We model a billiards (pool) table with several balls. The balls roll in a straight line till a collision with another ball or an edge occurs. When a collision occurs the velocity of the balls involved changes discretely. When a ball falls into a pocket, it disappears from the game. For simplicity, we assume that all balls have equal mass and radius (called R). We model only two ball collisions, and assume that there is no friction.

Impulses, denoted I, are assumed to be vectors. Velocities, positions are assumed to be pairs with an x-component and a y-component.

The structure of the program is that each ball, each kind of collision, and the check for pocketing for each ball are modeled by programs. A ball is basically a record with fields for name, position and velocity.

The agent `ball` maintains a given ball (`Ball`) with initial position (`InitPos`) and velocity (`InitVel`).[7] As the game evolves, position changes according to velocity (`dot(Ball.pos) = Ball.InitVel`) and velocity changes according to the effect of collisions. A propositional constraint `Change(Ball)`, shared between the collision and ball agents, communicates occurrences of changes in the velocity of the ball named `Ball`. The `ball` process is terminated when the ball is removed, *eg.* it falls in a pocket — as before, the higher priority of **do** `...` **watching** `pocketed(Ball)` relative to **do** `(...)` **watching** `Change(Ball)` is captured by lexical nesting.

```
ball(Ball, InitPos, InitVel) ::
   do (Ball.pos = InitPos,
       do hence Ball.vel = InitVel watching Change(Ball),
       whenever Change(Ball) do ball(Ball, Ball.pos, Ball.newvel)
       hence dot(Ball.pos) = Ball.vel)
     watching pocketed(Ball).
```

The table is assumed to start at $(0, 0)$, with length *xMax* and breadth *yMax*. If a ball hits the edge, one velocity component is reversed in sign and the other component is unchanged.

```
edge_collision(B) ::
   always (if (B.pos.x = B.r) or (B.pos.x = xMax − B.r)
             then (Change(B), B.newvel = ( − B.vel.x, B.vel.y)),
           if (B.pos.y = B.r) or (B.pos.y = yMax − B.r)
             then (Change(B), B.newvel = (B.vel.x, − B.vel.y))).
```

[7] The program is given here in a syntax that allows the declaration of procedures. The syntax $p(X_1, \ldots, X_n) :: A$ is read as asserting that for all $X_1, \ldots, X_n$, $p(X_1, \ldots, X_n)$ is equivalent to A.

Ball-ball collisions involve solutions to the quadratic conservation of energy equation. "**if** $|I| = 0$ **else** $|I| > 0$" chooses the correct solution, $I \neq 0$. The solution $I = 0$ makes the balls go through each other! We use distance(P1,P2) as short hand for the computation of the distance between the P1 and P2.

```
two_ball_collision(B1, B2) ::
   always if  (B1.pos = P1, B2.pos = P2, B1.vel = V1, B2.vel = V2
               distance(P1, P2) = 2 * R)
      then  (I_{B1, B2}.y * (P2.x - P1.x) = (P2.y - P1.y) * I_{B1, B2}.x,
             M * |V1|^2 + M * |V2|^2 = M * |B1.newvel|^2 + M * |B2.newvel|^2,
             M * V1 + I_{B1, B2} = M * B1.newvel, M * V2 - I_{B1, B2} = M * B2.newvel,
             if |I_{B1, B2}| = 0 else |I_{B1, B2}| > 0,
             change(B1), change(B2)).
```

A ball is pocketed if its center is within distance p from some pocket.

```
pocket(Ball) ::
   always if in_pocket(xMax, yMax, Ball.pos) then pocketed(Ball).
in_pocket(xMax, yMax, P) ::
   (distance(P, (0, 0)) < p or distance(P, (0, yMax)) < p or
    distance(P, (xMax, 0)) < p or distance(P, (xMax, yMax)) < p or
    distance(P, (xMax/2, 0)) < p or distance(P, (xMax/2, yMax)) < p).
```

4.5 The Copier Paper Path

We now present a larger example, which utilizes some of the programming ideas illustrated by the previous "toy" examples. We model the copier paper path that we mentioned in Section 1. As we discussed there, we will model each of the components of the paper path in Hybrid cc, and put them to to produce a model for the entire paper path. This model and the underlying intuitions are discussed in detail in [GSS95]—here we present an overview.

The constraint system we use for this model is richer than the constraint system described before — besides simple propositions and first-order formulas on them, we can assert arithmetic inequalities between variables, their derivatives and real constants. We will also allow the expression of attribute-value lists — given a list L we can use $L.a$ to refer to the value of a in the list L, if it exists. $L[a = r, b = s, \ldots]$ adds the attributes $a, b, \ldots$ to L with value $r, s, \ldots$.

The Intuitions. Our basic idea is to construct models for each of the individual components of the paperpath — the rollers and belts etc. We will also build a model for the sheets of paper going through the paperpath. In addition, we need to model the *local interaction* between a sheet of paper and a transportation element, i.e. a belt or a roller.

This interaction results in the transportation elements applying forces to the sheet, and the sheet then transmits these forces and moves under their influence. The local interactions will naturally induce a partition of the sheet into segments, which we view as individual entities, analyzing their freebody diagrams.

We make a number of modeling assumptions to simplify our model. For example, we do not model the acceleration of the sheet of paper, instead we view its velocity as changing discretely when a change takes place in the influences on it. We also assume that the sheet is homogeneous, the paperpath is straight, and some other things, these are discussed in detail in [GSS95].

The Model. The paperpath is modeled as a segment of the real line, and each component occupies a segment of the real line. All forces and velocities are oriented with respect to the paperpath. The first model fragment describes a basic transportation element—this can be further specialized into models for different types of belts and rollers. The properties and initial state of the element are specified using an attribute-value list `Init`. Each element has a nominal velocity $\mathtt{V}_{\mathtt{nom}}$, which is the velocity along the paperpath supplied by the motor. It also has an actual velocity $\mathtt{V}_{\mathtt{act}}$, which may differ from the nominal velocity, as a sheet of paper may be pulling or pushing the element. The force exerted on the transportation element is denoted $\mathtt{F}_{\mathtt{act}}$, and the force exerted by the motor is denoted $\mathtt{F}_{\mathtt{resist}}$. The freebody equation relates these two. $\mathtt{F}_{\mathtt{resist}}$ is bounded by $\mathtt{F}_{\mathtt{fast}}$ and $\mathtt{F}_{\mathtt{slow}}$. The remaining constraints show the relation between the forces and the velocities — they assert that if the element is going faster than the nominal velocity $\mathtt{V}_{\mathtt{nom}}$, then the force $\mathtt{F}_{\mathtt{act}}$ must be at its maximum value and similarly for the minimum value.

$$
\begin{array}{l}
\mathtt{transportationElement}(\mathtt{T},\ \mathtt{Init}) :: \\
\quad \textbf{always}\ (\mathtt{transport}(\mathtt{T}), \\
\qquad\quad \mathtt{T.Loc} = \mathtt{Init.Loc},\ \mathtt{T.surfacetype} = \mathtt{Init.surfacetype}, \\
\qquad\quad \mathtt{T.F}_{\mathtt{fast}} = \mathtt{Init.F}_{\mathtt{fast}},\ \mathtt{T.F}_{\mathtt{slow}} = \mathtt{Init.F}_{\mathtt{slow}}, \\
\qquad\quad \mathtt{T.F}_{\mathtt{normal}} = \mathtt{Init.F}_{\mathtt{normal}}, \\
\qquad\quad \mathtt{T.F}_{\mathtt{act}} + \mathtt{T.F}_{\mathtt{resist}} = 0, \\
\qquad\quad -\mathtt{T.F}_{\mathtt{fast}} \leq \mathtt{T.F}_{\mathtt{resist}} \leq \mathtt{T.F}_{\mathtt{slow}}, \\
\qquad\quad \textbf{if}\ \mathtt{T.V}_{\mathtt{act}} > \mathtt{T.V}_{\mathtt{nom}}\ \textbf{then}\ \mathtt{T.F}_{\mathtt{act}} = \mathtt{T.F}_{\mathtt{fast}}, \\
\qquad\quad \textbf{if}\ \mathtt{T.V}_{\mathtt{act}} < \mathtt{T.V}_{\mathtt{nom}}\ \textbf{then}\ \mathtt{T.F}_{\mathtt{act}} = -\mathtt{T.F}_{\mathtt{slow}}).
\end{array}
$$

This model is simply an enumeration of the constraints on the transportation element, these are **always** true. The value of $\mathtt{V}_{\mathtt{nom}}$ is not set here, this is done during the specialization of this generic transportation element description into various particular kinds of elements.

The next "component" of the paperpath is the sheet of paper that travels along it. This again has a number of properties, and also an initial location. Furthermore, the sheet is partitioned into various segments — the `contactSegments` are those portions of the sheet in contact with some transportation element. Between two contact segments is an `internalSegment`, whereas beyond the extreme contact segments are the `TailSegments`. In order to create these segments, we use the new operator **forall**

$X : C[X].A[X]$. This is simply a parallel composition of *finitely* many $A[X]$'s, one for each X such that $C[X]$ is true. The declarative style of programming is particularly useful here — this way of asserting the existence of segments is much simpler than the dynamic creation and destruction of segments that would have to be done in an imperative program.

```
sheet(S, Init) ::
   S.Loc = Init.Loc,
   do always
      (sheet(S),
       S.width = Init.width, S.length = Init.length,
       S.thickness = Init.thickness, S.elasticity = Init.elasticity,
       S.surfacetype = Init.surfacetype, S.strength = Init.strength,
       if ∃I.(S.I.condition = tearing) then S.torn,
       forall T : transport(T). if (T.engaged, | T.Loc ∩ S.Loc | > 0)
                                then contact(S, S.Loc ∩ T.Loc),
       forall I : contact(S, I). contactSegment(S, I),
       forall I : contact(S, I). forall J : contact(S, J).
         if I < J then if ∃K.(contact(S, K), I < K < J)
                       else internalSegment(S, (ub(I), lb(J))),
       forall I : contact(S, I). if ∃K.(contact(S, K), K < I)
                                 else leftTailSegment(S, (lb(S.Loc), lb(I))),
       forall I : contact(S, I). if ∃K.(contact(S, K), I < K)
                                 else rightTailSegment(S, (ub(I), ub(S.Loc))),
       dot(lb(S.Loc)) = S.vel.lb(S.Loc),
       dot(ub(S.Loc)) = S.vel.ub(S.Loc))
    watching S.torn.
```

For brevity, we omit the code for the segments of the sheet. The agent `interact` creates an interaction process `sheetTransportation` for each transportation element overlapping the sheet. The sheet transportation process models the interaction between the sheet and the transportation element. It asserts that the force exerted by the element on the sheet is equal and opposite to the force exerted by the sheet on the element. It gives the freebody equation for the contact segment, and then bounds the amount of force exerted by the transportation element by the amount of friction. Finally, the friction and the velocities are related.

```
interact(S) ::
   always forall T : transport(T).
     if (T.engaged, | T.Loc ∩ S.Loc | > 0)
        then sheetTransportation(S, T, T.Loc ∩ S.Loc).
```

$\mathtt{sheetTransportation(S,\ T,\ I)} ::$
  $\textbf{new}\ \mathtt{F_{friction}}\ \textbf{in}$
    $(\mathtt{I} = \mathtt{S.Loc} \cap \mathtt{T.Loc},$
    $\mathtt{F_{friction}} = \mathtt{mu(S.surfacetype,\ T.surfacetype)} \times \mathtt{T.F_{normal}},$
    $\mathtt{S.I.F_{te}} + \mathtt{T.F_{act}} = 0,$
    $\mathtt{S.(lb(I)).F_{left}} + \mathtt{S.(ub(I)).F_{right}} + \mathtt{S.I.F_{te}} = 0,$
    $|\ \mathtt{S.I.F_{te}}\ | \leq \mathtt{F_{friction}},$
    $\textbf{if}\ \mathtt{S.vel.(ub(I))} > \mathtt{T.V_{act}}\ \textbf{then}\ \mathtt{S.I.F_{te}} = -\mathtt{F_{friction}},$
    $\textbf{if}\ \mathtt{S.vel.(ub(I))} < \mathtt{T.V_{act}}\ \textbf{then}\ \mathtt{S.I.F_{te}} = \mathtt{F_{friction}}).$

Finally we provide the model for the paperpath shown in Figure 1. The various kinds of transportation elements are modeled first, followed by a typical sheet, then these are composed together to give the entire paperpath. There is one new combinator used here — **wait** T **do** A, which waits for T seconds before starting A. This can be expressed in terms of the basic combinators. A new sheet is placed at $(0, 21)$ every time a signal `Sync` is received from the scheduler.

$\mathtt{cvElement(R,\ Init)} ::$
  $\mathtt{transportationElement(R,\ Init)},\ \textbf{always}\ (\mathtt{R.V_{nom}} = \mathtt{Init.V_{nom}},\ \mathtt{R.engaged}).$
$\mathtt{roller(R,\ Init)} ::$
  $\mathtt{cvElement(R,\ Init[surfaceType} = \mathtt{rubber},\ \mathtt{F_{fast}} = 1000,\ \mathtt{F_{slow}} = 1000,$
    $\mathtt{F_{normal}} = 300]),$
  $\textbf{if}\ (\mathtt{R.F_{act}} = \mathtt{R.F_{slow}} \vee \mathtt{R.F_{act}} = -\mathtt{F_{fast}})\ \textbf{then}\ \mathtt{R.broken}.$
$\mathtt{clutchedRoller(R,\ Init)} ::$
  $\mathtt{cvElement(R,\ Init[surfaceType} = \mathtt{rubber},\ \mathtt{F_{fast}} = 3,\ \mathtt{F_{slow}} = 1000,$
    $\mathtt{F_{normal}} = 200]),$
  $\textbf{if}\ \mathtt{R.F_{act}} = \mathtt{R.F_{slow}}\ \textbf{then}\ \mathtt{R.broken}.$
$\mathtt{belt(R,\ Init)} ::$
  $\mathtt{cvElement(R,\ Init[surfaceType} = \mathtt{steel},\ \mathtt{F_{fast}} = 10000,\ \mathtt{F_{slow}} = 10000,$
    $\mathtt{F_{normal}} = 120]).$
$\mathtt{fuserRoll(R,\ Init)} ::$
  $\mathtt{cvElement(R,\ Init[surfaceType} = \mathtt{rubber},\ \mathtt{F_{fast}} = 1000,\ \mathtt{F_{slow}} = 1000,$
    $\mathtt{F_{normal}} = 1000]).$
$\mathtt{regClutch(R,\ Init)} ::$
  $\mathtt{transportationElement(R,\ Init)},\ \textbf{always}\ \mathtt{R.engaged},$
  $\textbf{always}\ (\textbf{if}\ \mathtt{R.on}\ \textbf{then}\ \textbf{do}\ \textbf{always}\ \mathtt{R.V_{nom}} = \mathtt{Init.V_{nom}}\ \textbf{watching}\ \mathtt{R.off},$
      $\textbf{if}\ \mathtt{R.off}\ \textbf{then}\ \textbf{do}\ \textbf{always}\ \mathtt{R.V_{nom}} = 0\ \textbf{watching}\ \mathtt{R.on}).$

$\mathtt{a4GlossySheet(S)} ::$
  $\mathtt{sheet(S,\ [length} = 21,\ \mathtt{width} = 29.7,\ \mathtt{thickness} = 0.015,\ \mathtt{elasticity} = 15,$
      $\mathtt{surfaceType} = \mathtt{glossy},\ \mathtt{strength} = 500,\ \mathtt{Loc} = (0,\ 21)]).$

$\mathtt{copierModel(Sync)} :: \textbf{new}\ (\mathtt{R1,\ R2,\ R3,\ R4,\ R5,\ R6})\ \textbf{in}$
  $(\mathtt{roller(R1,\ [V_{nom}} = 30,\ \mathtt{Loc} = (10,\ 10.5)]),$

```
roller(R2, [V_nom = 30, Loc = (30, 30.2)]),
regClutch(R3, [V_nom = 30, Loc = (39.9, 40.1), surfaceType = rubber,
                 F_fast = 1000, F_slow = 1000, F_normal = 300]),
clutchedRoller(R4, [V_nom = 0, Loc = (50, 50.1)]),
belt(R5, [V_nom = 30, Loc = (55, 80)]),
fuserRoll(R6, [V_nom = 35, Loc = (85, 86)]),

always (f(rubber, glossy) = 0.8, f(steel, glossy) = 0.1),
always if Sync then (new S in (a4GlossySheet(S), interact(S)),
                    R3.off, wait .8 do R3.on)).
```

The model can now be executed to simulate the photocopier. However, we would like to put to the various other uses mentioned in the introduction, by exploiting the fact that Hybrid tcc has a well-defined mathematical semantics.

One simple use for our model would be to verify whether some properties are true of it. For example we might want to assert that sheets will never tear — **always forall** `S:sheet(S)`. **if** `S.torn` **then** `ERROR`. Similarly, several other properties can be asserted, these can be now proved using various theorem-proving and model-checking techniques. We could also want to prove that successive sheets never overlap — this proof would give us a condition on how far apart the `Sync` signals must be. Similar techniques may be used for control code generation. The other applications mentioned in the introduction also make use of such formal models and their semantics.

4.6 Qualitative simulation.

Abstract interpretation of Hybrid tcc programs might provide an alternate conceptual framework for the qualitative modeling approaches of [Kui94].

Hybrid tcc can be used to model exactly systems with continuous and discrete change. For example, here is a model of a bathtub, with the tap on, and the drain unplugged [Kui94, p112].

```
bathtub(Inflow, F, Capacity) :: new (netflow, outflow, amount) in
   constant(F), monotone(F), F(0) = 0,
   constant(Inflow),
   constant(Capacity),
   do always (netflow = outflow + Inflow,
              outflow = F(amount),
              dot(amount) = netflow)
    watching (amount = Capacity ∧ dot(amount) > 0).
```

Here the first three lines assert that the three parameters do not vary with time, and that F is a monotone function in its argument, assuming the value 0 on input 0.

Essentially with this information, QSIM produces a graph that shows that from the initial state there are only three possible (qualitative) states that the program can transit to (for any possible input values of Inflow, F, and Capacity): namely those in which the tub overflows, the tub equilibrates at a level below Capacity, and the tub equilibrates at capacity. To do this reasoning, QSIM employs knowledge about the behavior of continuously-varying functions.

Since Hybrid tcc provides an exact description of the model, it should be possible to do an abstract interpretation of the program to get the qualitative reasoning of QSIM. This is useful in cases when a precise value for the inputs is not known, only a range is given. Abstract interpretation could also be useful in verification. For example, in the cat-and-mouse example, we may not know the exact velocities of the cat and the mouse, only a range of velocities. However it may still be possible to prove that the mouse always wins. The ability to run this program abstractly for all the possible values of the velocities provides an alternative way of verification of this property.

5 Future work

This paper presents a simple programming language for hybrid computing, extending the untimed computation model of Default cc uniformly over the reals. Conceptually, computation is performed at each real time point; however, the language is such that it can be compiled into finite automata of the style of [NOSY93]. We have an interpreter for the language by directly extending the interpreter for Default cc, with an integrator for computing evolution of a set of variables subject to constraints involving differentiation.

We demonstrate the expressiveness of the language through several programming examples, and show that the preemption-based control constructs of the (integer-time based) synchronous programming languages (such as "**do** ... **watching** ...) extend smoothly to the hybrid setting.

Much remains to be done. What are semantic foundations for higher-order hybrid programming, in which programs themselves are representable as data-objects? How should such a hybrid modeling methodology be combined with the program structuring ideas of object-oriented programming? What is the appropriate notion of types in such a setting? How do the frameworks for static analysis of programs (data- and control-flow analysis, abstract interpretation techniques) generalize to this setting? What are appropriate techniques for partial evaluation and program transformation of such programs? Besides these language issues, there are a number of modeling and implementation issues — What are appropriate continuous constraint systems which have fast algorithms for deciding the entailment relation? Can boundary value problems be solved in this framework? How does stepsize in numerical integration techniques affect execution and visualization of results? Some of these will be the subject of future papers.

Acknowledgements. Work on this paper has been supported in part by ONR through grants to Vijay Saraswat and to Radha Jagadeesan, and by NASA. Radha Jagadeesan has also been supported by a grant from NSF.

References

[AH92] R. Alur and T. Henzinger. Logics and models of real time: a survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *REX workshop "Real time: Theory in Practice"*, volume 600 of *LNCS*, 1992.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, September 1991.

[BBG93] A. Benveniste, M. Le Borgne, and Paul Le Guernic. *Hybrid Systems: The SIGNAL approach*. Number 736 in LNCS. Springer Verlag, 1993.

[Ber93] G. Berry. Preemption in concurrent systems. In *Proc. of FSTTCS*. Springer-Verlag, 1993. LNCS 781.

[BG90] A. Benveniste and P. Le Guernic. Hybrid dynamical systems and the signal language. *IEEE Transactions on Automatic control*, 35(5):535–546, 1990.

[BG92] G. Berry and G. Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, November 1992.

[CLM91] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9), September 1991.

[dBHdRR92] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *REX workshop "Real time: Theory in Practice"*, volume 600 of *LNCS*, 1992.

[Deb93] Saumya K. Debray. QD-Janus: A Sequential Implementation of Janus in Prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.

[dKB85] Johan de Kleer and John Seely Brown. *Qualitative Reasoning about Physical Systems*, chapter Qualitative Physics Based on Confluences. MIT Press, 1985. Also published in AIJ, 1984.

[FBB+94] M. Fromherz, D. Bell, D. Bobrow, et al. RAPPER: The Copier Modeling Project. In *Working Papers of the Eighth International Workshop on Qualitative Reasoning About Physical Systems*, pages 1–12, June 1994.

[GBGM91] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, September 1991.

[GJSB95] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel Bobrow. Computing with continuous change. Technical report, Xerox Palo Alto Research Center, May 1995. Submitted.

[GNRR93] Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors. *Hybrid Systems*. Springer Verlag, 1993. LNCS 736.

[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.

[HCP91] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, Special issue on Another Look at Real-time Systems, September 1991.

[HMP93] T. A. Henzinger, Z. Manna, and A. Pnueli. Towards refining temporal specifications into hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229, 1993.

[Hoo93] J. Hooman. A compositional approach to the design of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229, 1993.

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

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.

[Kui94] Ben Kuipers, editor. *Qualitative Simulation*. MIT Press, 1994.

[MMP92] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *REX workshop "Real time: Theory and Practice"*, volume 600 of *Lecture notes in computer science*, pages 447–484, 1992.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991. 427 pp.

[NK93] A. Nerode and W. Kohn. *Multiple Agent Hybrid Control Architecture*. Number 736 in LNCS. Springer Verlag, 1993.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. *An Approach to the Description and Analysis of Hybrid Systems*. Number 736 in LNCS. Springer Verlag, 1993.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From atp to timed graphs and hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *REX workshop "Real time: Theory and Practice"*, volume 600 of *LNCS*, 1991.

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

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

[SHW94] Gert Smolka, Henz, and J. Werz. *Constraint Programming: The Newport Papers*, chapter Object-oriented programming in Oz. MIT Press, 1994.

[SJG94a] V. A. Saraswat, R. Jagadeesan, and V. Gupta. *Constraint Programming*, chapter Programming in Timed Concurrent Constraint Languages. NATO Advanced Science Institute Series, Series F: Computer and System Sciences. Springer-Verlag, 1994.

[SJG94b] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Press, July 1994.

[SJG95] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default Timed Concurrent Constraint Programming. In *Proceedings of Twenty Second ACM Symposium on Principles of Programming Languages, San Francisco*, January 1995.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.

[Wd89] D.S. Weld and J. de Kleer. *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann, 1989.