Computing with continuous change

V. Gupta [a], R. Jagadeesan [b] and V. A. Saraswat [a]

[a] *Xerox PARC, 3333 Coyote Hill Road, Palo Alto Ca 94304*

[b] *Dept. of Mathematical and Computer Sciences,Loyola University–Lake Shore Campus, Chicago, Il 60626*

Abstract

A central challenge in computer science and knowledge representation is the integration of conceptual frameworks for continuous and discrete change, as exemplified by the theory of differential equations and real analysis on the one hand, and the theory of programming languages on the other.

We take the first steps towards such an integrated theory by presenting a recipe for the construction of continuous programming languages — languages in which state dynamics can be described by differential equations. The basic idea is to start with an untimed language and extend it uniformly over dense (real) time.

We present a concrete mathematical model and language (the Hybrid concurrent constraint programming model, Hybrid cc) instantiating these ideas. The language is intended to be used for modeling and programming *hybrid systems*. The language is declarative — programs can be understood as formulas that place constraints on the (temporal) evolution of the system, with parallel composition regarded as conjunction. It is expressive — it allows the definition of continuous versions of the preemption control constructs.

The language is obtained by extending the general-purpose computational formalism of (default) concurrent constraint programming (Default cc) with a single temporal construct, called **hence** — **hence** A is read as asserting that A holds *continuously* beyond the current instant. Various patterns of temporal activity can be generated from this single construct by use of the other combinators in Default cc. We provide a precise operational semantics according to which execution alternates between (i) points at which discontinuous change can occur, and (ii) open intervals in which the state of the system changes continuously. Transitions from a state of continuous evolution are triggered when some condition starts or stops holding. We show that the denotational semantics is correct for reasoning about the operational semantics, through an adequacy theorem.

[1] Supported by grants from the National Science Foundation and the Office for Naval Research.

1 Introduction

A core divide lies at the heart of computing. Most of the physical sciences, concerned with the conceptual analysis of everyday phenomena, have relied on the development of continuous mathematics. Quantities of interest are represented as functions over the reals, and their variation over time described by systems of differential equations. The work in this area relies on three centuries of continuous development in classical mathematics. However, traditional computer science deals with discrete state spaces: finite variables holding concrete values drawn from discrete domains, such as the integers, or set-theoretic structures. Change arises through the discrete change of values for variables; the current state of a system may have no structural relationship to the past – for instance, there is no analog for the mean value theorem. The only notions of continuity have to do with formalizing the intuitive meaning of recursion as the taking of limits in some "information" space. There have not been well-developed notions of continuous programming languages, with a concomitant theory of continuous control structures.

1.1 Hybrid systems

A fundamental need to reconcile these two approaches arises in *hybrid systems*, the natural continuous extensions of *reactive systems* [25,7,22]. Reactive systems react with their environment at a rate controlled by the environment. Execution in such a system proceeds as bursts of activity. In each phase, the environment stimulates the system with an input, obtains a response within a bounded amount of time, and may then be dormant for an arbitrary period of time before initiating the next burst. Thus, the notion of time in such systems is discrete, *i.e.* time is accurately modeled by the natural numbers. In [7] determinate synchronous programming is identified as the appropriate paradigm for the modeling and programming of such systems. Examples of synchronous programming languages are CSML, Esterel, Lustre, SCCS, Signal, Statecharts and Timed default concurrent constraint programming [4,10,23,18,24,12,39,11,45,22]. This methodology has been successfully applied to actual systems — for example, telecommunication applications [31,30,40], communication protocols [9] and robotics [13].

Continuous or analog systems are those in which the system has the potential of evolving autonomously and continuously. The description of this behavior is usually in the form of differential equations that arise naturally in the description and modeling of the behavior of mechanical and physical components. For example, in the animation language TBAG [16], the motion of a ball under the influence of various forces is described directly by the equations induced by the laws of motion.

In contrast to reactive systems, the appropriate notion of time in such systems is dense, *i.e.* an accurate modeling of time requires a dense domain such as the rationals or the reals. However, the ideas of synchronous programming are still relevant to dense time domains [5].

Complex applications typically require the combination of both these ideas. Consider for example, virtual prototyping — the substitution of computer models for physical products, processes, and systems. Intuitively, the aim of virtual prototyping is to make extensive use of software models to reduce the number of physical prototypes that must be constructed. Examples include simulators for aspects of electronic or telecommunications systems, and visualizations of mechanical, chemical or civil systems. Consider a concrete example — the paper path of a simple photocopier, see Figure 1.

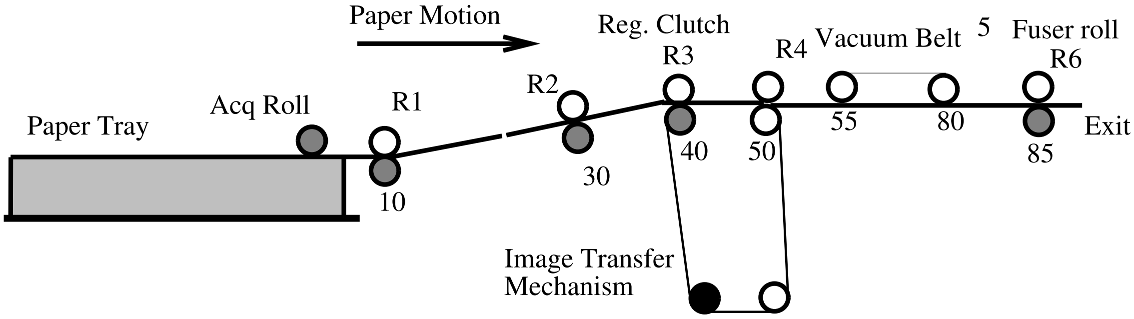

Fig. 1. A simple photocopier

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, and the image is transferred onto the paper by the image transfer mechanism. The vacuum belt (5) transports it to the fuser roll (R6), from where it exits.

The photocopier above has a collection of components with continuous behavior, for example, the rollers and belts. However, the control program — the program that controls the actions in the physical system above — of the photocopier is a reactive 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* [6,42,35,17]

1.2 The research program

This paper is concerned with executable specification/programming languages for hybrid systems. Executability ensures that given a model of the system, it is possible to predict the behavior of the system when inputs are supplied. 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.

The primary issues that arise in programming such systems are time-criticality, reliability and maintainability in the face of change. For example, in the realm of virtual prototyping, the models must be an accurate representation and rendering of objects; they must permit changes of components and must be amenable to reasoning about the actual underlying physical processes.

The focus of this paper is the design and investigation of a paradigm, Hybrid Concurrent Constraint programming or Hybrid cc, for the modeling, programming and analysis of hybrid systems. We intend to establish the viability of the paradigm by subjecting it to the following tests. These criteria are motivated by the desirable properties expected by a "user" of the paradigm.

- Is Hybrid cc declarative? Can programs be understood as formulas that place constraints on the (temporal) evolution of the system and be viewed as a declaration of facts in a (real-time) (temporal) logic [36,1]?

  This criterion can be viewed as a measure of the ease of use of the paradigm by systems engineers — people quite different in training and background from software engineers. 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.
- Is Hybrid cc powerful enough to encompass extant special purpose languages in the hybrid genre — say for example, the animation language TBAG?

  This criterion can be viewed as one measure of the expressive power of Hybrid cc. One way to demonstrate that Hybrid cc satisfies this criterion would be to show that Hybrid cc can be used as the metalanguage for a semantics of the special purpose languages.
- Does Hybrid cc support modularity, *i.e.* support for hierarchical construction of programs/specifications to aid in maintainability?

  This criterion can be viewed as a measure of the expressive power of Hybrid cc from a programming language viewpoint and can be rephrased as: Are there enough combinators — ways of building programs — in Hybrid cc?

- Does Hybrid cc specialize easily and usefully to particular domains?
  This criterion can be viewed as a measure of the utility of the insights gained by the research. One way to demonstrate this criterion would be design and implement a language in the Hybrid cc paradigm to program problems in a particular domain of hybrid systems.
- Does the paradigm integrate smoothly with extant techniques/tools, for the specification, verification and reasoning about properties, including real time constraints, of hybrid systems?
  This criterion ensures that existing technology on the analysis of hybrid systems can be reused in the Hybrid cc paradigm. Thus, Hybrid cc 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 [17,14], qualitative reasoning about physical systems [50], and envisionment of qualitative states [34].

From a designer point of view, we intend to achieve the above criterion by ensuring that Hybrid cc has good formal properties. Concretely, the technical questions addressed by research program will include the following:

- Axiomatization of the notion of a *continuous* constraint system to describe the continuous evolution of system trajectories. Design of Hybrid constraint languages — developed generically over continuous constraint systems. [2]
- Description of formal operational, denotational and logical semantics.
- Characterization of combinators definable in Hybrid cc and comparison of expressive power with synchronous programming languages.
- Compilation of Hybrid cc programs to hybrid automata and integration with existing verification techniques and tools. Applications of semantics based analysis (such as abstract interpretation and constraint based program analysis) to analysis of Hybrid cc programs.
- Typing and type checking of Hybrid cc programs — types as succinct representations of the interface presented by the component — this interface will identify the assumptions expected of the environment in which the component will work, and the guarantees provided by the component.
- Modeling of a real problem (such as the paperpath of a photocopier above) and analysis using the methods developed in the above items.

1.3 What have we done?

This paper describes and studies some of the aspects of Hybrid cc — hybrid concurrent constraint programming.

[2] Similar to the way concurrent constraint programming is developed generically over a notion of a constraint system.

– We introduce the notion of a *continuous* constraint system to describe 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.
– We describe operational and denotational semantics, and show that the denotational semantics is correct for reasoning about the operational semantics. The operational semantics has been implemented to yield an interpreter for Hybrid cc.
– We show that continuous variants of preemption-based control constructs and multiform timing constructs are definable in Hybrid cc.
– We describe a few programming examples, and the trace of an interpreter on these examples. These examples illustrate the synthesis, in Hybrid cc, of intuitions from concurrent constraint programming, synchronous programming and extant models of hybrid systems.

1.4 Organization of paper

We proceed as follows. First, we review the idea of compositional modeling — our analysis is motivated by the synchronous programming paradigm. Next, we describe the basic computational intuitions underlying Hybrid cc. Then, we begin our formal development. We start off with a review of constraint systems and our earlier work on Default cc. Next, we introduce Hybrid cc via its denotational semantics — this denotational model formalizes the "programs as constraints" idea and associates with each process the collection of its observations. The denotational model of Hybrid cc is an "extension" of Default cc over continuous time. This extension proceeds in two stages. First, we introduce the notion of a *continuous* constraint system — a real-time extension of constraint systems — to describe the continuous evolution of system trajectories. Next, we extend the Default cc model of processes over continuous time to describe Hybrid cc processes. We illustrate the language and the denotational semantics by describing a variety of combinators that are definable in the language. We then describe the operational semantics; we also describe an interpreter we have built that realizes the operational semantics and show that the denotational semantics is correct for reasoning about the operational semantics. We describe a couple of programming examples, and show traces of our interpreter on these examples. We conclude with a discussion of the issues that remain to be addressed.

1.5 Compositional modeling revisited

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 configuration in which they will be used (the "no function in structure principle", [15]). 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 [27,29,41]. From a programming language standpoint, these modularity concerns are addressed by the analysis underlying synchronous programming languages [4,22,10,23,18,24,12,45], (adapted to dense discrete domains in [5]); this analysis leads to the following demands on the modeling language.

Consider a simple protocol that implements the controller of a paper-tray of a photo-copier. The controller switches the paper tray motor on and off, always trying to keep the top of the stack of paper next to the feeder. There are two sensors, P and E, which are set to 1 when the height of the paper is OK. Whenever the paper level falls, *i.e.* one of the sensors becomes zero, the motor is activated to push up the paper stack. This protocol can be construed as a finite state machine `PerfectP` with two states as in Figure 1. This finite state machine captures the structure of the implementation of this protocol in a sequential language.

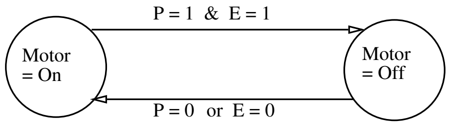

Fig. 2. Automaton for a perfect paper-tray

However, sequential programs (and hence the automaton above) do not have parallel structure; consequently, small and succinct changes in the specification can lead to global changes in the automaton [40]. For example, let us say that we want the controller to also be aware of the fact that the sensors may be broken. In this case, it should stop the motor after a certain delay, in order to prevent it from damaging the copier. An automaton for this protocol is as in Figure 2. This automaton is merely a representation of the structure of the implementation of this protocol in a sequential language.

Note that there is no *structural* relationship between the two automata. This makes the maintenance of such code through changes in requirements and specification an arduous task. Logical concurrency and preemption allow us to achieve modularity. In the above example, with a preemption combinator and parallel composition (written as `||`), the program for the second paper-

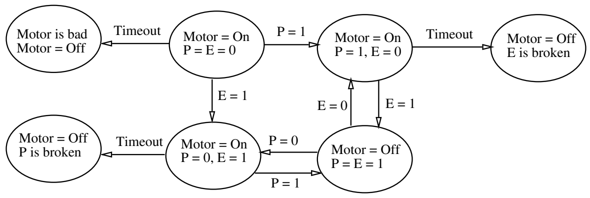

Fig. 3. Automaton for a paper-tray with failure modes

tray controller is written in terms of PerfectP as follows:

```
[do
     PerfectP                          ||   Set-Sensor-broken
 watching sensor-broken]
```

The *do ... watching* statement terminates the program PerfectP when the signal "sensor-broken" is emitted. The procedure Set-sensor-broken is responsible for emitting the signal "sensor-broken".

Intuitively, logical concurrency/parallelism plays a role in determinate reactive system programming analogous to the role of procedural abstraction in sequential programming — the role of matching program structure to the structure of the solution to the problem at hand. Furthermore, preemption — the ability to stop a process in its tracks — is a fundamental programming tool for such systems [8]; note the use of the watchdog preemption in the above example. Examples of preemption include interrupts, process suspension and process abortion. Consequently, we demand that preemption constructs have the same status as concurrency. 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 rate of rotation of the belt etc.

Thus, we demand that Hybrid cc be an *algebra of processes, that includes concurrency, hiding, preemption and multiform time.*

2 The underlying computational intuition.

Hybrid cc is a language in the concurrent constraint programming framework, augmented with a notion of continuous time and defaults. The (concurrent) constraint (cc) programming paradigm [48] 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. Computation progresses by accumulating constraints in the store, and by checking whether the store entails constraints. Recently, several concrete general-purpose programming languages have been implemented in this paradigm [33,49].

A salient aspect of the cc computation model is that programs may be thought of as imposing constraints on the evolution of the system. Default cc [45] provides five basic constructs: (tell) a (for a a primitive constraint), parallel composition (A, B), positive ask (**if** a **then** A), negative ask (**if** a **else** A), and hiding (**new** X **in** A). The program a imposes the constraint a. The program (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 programs — logically, this can be thought of as a form of existential quantification. The program **if** a **then** A imposes the constraints of A provided that the rest of the system imposes the constraint a — logically, this can be thought of as intuitionist implication. The program **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* [44]. Note that **if** a **else** A is quite distinct from the program **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 stores now contain partial information; they may not be strong enough to entail either a or $\neg a$. 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 programs may simultaneously be active and adding more information to the store. Therefore requiring that information a be absent amounts to making a demand on "stability" of negative information.

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 programs interacting with a shared store of primitive constraints. The program a is viewed as adding a to the store instantaneously. The program (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 program **if** a **then** A behaves like A if the current store entails a. The program **if** a **else** A behaves like A if the current store on quiescence does *not* entail a.

The cc paradigm has no concept of timed execution. For modeling discrete, reactive systems, [45] introduced the idea (from synchronous programming) that the environment reacts with a system (program) at discrete time ticks.

At each time tick, the program executes a cc program, outputs the resulting constraint, and sets up another program for execution at the next clock tick. Concretely, this led to the addition of two control constructs to the language **next** A (execute A at the next time instant), and **always** A (execute A at every time instant). Thus, intuitively, the discrete timed language was obtained by uniformly extending the untimed language (cc or Default cc) across (integer) time.

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.

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) problem, e.g. constraints of the form $\texttt{init}(X = 0), \texttt{cont}(dot(X) = 1)$ (read as follows: the initial value of X is 0; the first derivative of X is 1); from these we can infer at time $\texttt{t}$ that $\texttt{X} = \texttt{t}$. 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.

Second we add to the untimed Default cc a single temporal control construct: **hence** A. Declaratively, **hence** A imposes the constraints of A at every 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	there is 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[3] of a program — hence, a program **hence** (**if** a **else** A) will in fact not execute A at all those time instants at which a is true.

Let us consider some concrete examples. Let **always** $A \stackrel{d}{=} (A, \textbf{hence}\ A)$. Suppose that we require that a program A be executed at every time point until a is true. This can be expressed as **new** X **in** (**always** (**if** X **else** A, **if** a **then always** X)). Intuitively, at every time point, 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 program (**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 until 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 always** (**if** X **else if** a **then** A, **if** a **then hence** X).

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** $(\texttt{init}(T=0), \textbf{hence}\ dot(T) = 1, \textbf{hence if}\ rational(T)\ \textbf{then}\ A)$ which require the execution of A at every rational $q > 0$. Such programs are not implementable, because rationals (irrationals) are everywhere dense as a subset of the reals. We show that in fact Hybrid cc is computationally realizable. The basic intuition we exploit to handle the first problem is that most physical systems change "slowly", with points of discontinuous change, followed by periods of continuous evolution. Technically, we introduce a *stability* condition for continuous constraint system that guarantees that for every pair of constraints a and b there is a neighborhood around 0 in which a entails b either everywhere or nowhere. This rules out constraints such as *rational(T)*.

With these restrictions, computation in Hybrid cc may be thought of as progressing in phases of computation — alternating between point phases and open interval phases. Computation at a 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 $(0, r)$.

One further problem remains, arising from the interaction of existentials (new variable creation) and continuous evolution of the system. For example, we

[3] Instantaneous preemption allows a program to be aborted immediately, rather than after executing a little longer like the control-C of Unix. See [8].

may have programs of the form **hence** (**new** T **in** ...) which seem to require the creation of a copy of the new variable T for every real instant in the continuous evolution in $(0, \ldots)$. To handle this problem, we restrict the variables on which existential quantification can be performed to those variables for which *one* copy of the variable suffices for a continuous evolution. Technically, we emulate the flexible existential quantification of temporal logic, by making $\exists_X$ commute with the temporal constructions in the ccs.

3 Background

We first review Default cc [45], an extension of cc programming. For technical convenience, the presentation of constraint systems here is a slight variation of an earlier published presentation [47]. [4]

3.1 Constraint systems.

A constraint system $\mathcal{D}$ is a system of partial information, consisting of a set of primitive constraints (first-order formulas) or *tokens* D, closed under conjunction and existential quantification, and an inference relation (logical entailment) $\vdash$ that relates tokens to tokens. We use $a, b, c, \ldots$ to range over tokens. $\vdash$ induces through symmetric closure the logical equivalence, $\approx$. Formally,

Definition 1 *A* constraint system *is a structure* $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ *such that:*

- *D is closed under conjunction($\wedge$); $\vdash \subseteq D \times D$ satisfies:*
  - $a \vdash a$
  - $a \vdash a'$ *and* $a' \wedge a'' \vdash b$ *implies that* $a \wedge a'' \vdash b$
  - $a \wedge b \vdash a$ *and* $a \wedge b \vdash b$
  - $a \vdash b_1$ *and* $a \vdash b_2$ *implies that* $a \vdash b_1 \wedge b_2$.
- **Var** *is an infinite set of* variables*, such that for each variable* $X \in \mathbf{Var}$*,* $\exists_X : D \to D$ *is an operation satisfying usual laws on existentials:*
  - $a \vdash \exists_X a$
  - $\exists_X (a \wedge \exists_X b) \approx \exists_X a \wedge \exists_X b$
  - $\exists_X \exists_Y a \approx \exists_Y \exists_X a$
  - $a \vdash b$ *implies* $\exists_X a \vdash \exists_X b$.
- $\vdash$ *is decidable.*

The last condition is necessary to have an effective operational semantics.

[4] [45] extends the conference version in POPL 95 with hiding.

A *constraint* is an entailment closed subset of D. For any set of tokens S, we let $\overline{S}$ stand for the constraint $\{a \in D \mid \exists \{a_1, \ldots, a_k\} \subseteq S.\ a_1 \wedge \ldots \wedge a_k \vdash a\}$. For any token a, $\overline{a}$ is just the constraint $\overline{\{a\}}$.

The set of constraints, written $|D|$, ordered by inclusion($\subseteq$), forms a complete algebraic lattice with least upper bounds induced by $\wedge$, least element $\texttt{true} = \{a \mid \forall b \in D.\ b \vdash a\}$ and greatest element $\texttt{false} = D$. Reverse inclusion is written $\supseteq$. $\exists, \vdash$ lift to operations on constraints. Examples of such systems are the system Herbrand (underlying logic programming), FD [26](finite domains), and `Gentzen` [46].

Example 2 The Herbrand constraint system. Let L be a first-order language L with equality. The tokens of the constraint system are the atomic propositions. Entailment is specified by Clark's Equality Theory, which include the usual entailment relations that one expects from equality. Thus, for example, $f(X, Y) = f(A, g(B, C))$ must entail $X = A$ and $Y = g(B, C)$.

Example 3 The FD constraint system. Variables are assumed to range over finite domains. In addition to tokens representing equality of variables, there are tokens that that restrict the range of a variable to some finite set.

Example 4 The `Gentzen` constraint system. For real-time computation we have found the simple constraint system ($\mathcal{G}$) to be very useful. The primitive tokens a_i of `Gentzen` are atomic propositions $X, Y, Z \ldots$. These can be thought of as signals in a computing framework. The entailment relation is trivial, i.e. $a_1 \wedge \ \ldots \wedge a_n \vdash_{\mathcal{G}} a$ iff $a = a_i$ for some i. Finally $\exists_X(a_1 \wedge \ \ldots \wedge a_n) = b_1 \wedge \ \ldots \wedge b_n$ where $b_i = a_i$ if $a_i \neq X$ and $b_i = \texttt{true}$ otherwise.

In the rest of this section we will assume that we are working in some constraint system $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$. We will let $a, b, \ldots$ range over D. We use $u, v, w \ldots$ to range over constraints.

3.2 Default cc

We provide here a brief review of our earlier work on Default cc. More details and motivation for the definitions can be found in [45].

Denotational semantics of Default cc. We extend the idea of cc, that the denotation of a process consists of all observations. The critical question is: what is an observation? Observe for each agent A those stores u in which they are *quiescent*[5], *given the guess v about the final result.* Note that the guess v must always be stronger than u — it must contain at least the information on which A is being tested for quiescence. The intended interpretation is: if the guess v is used to resolve defaults, then executing A in u does not produce any information not entailed by u.

Definition 5 (Default cc **Observations**)
DObs $= \{(u, v) \in |D| \times |D| \mid v \supseteq u\}$.

To describe processes and the denotation of combinators, we need some notation – for a set S of constraints, and constraint v, (S, v) stands for the set $\{(u, v) \mid u \in S\}$. $\sqcap S$ stands for the greatest lower bound of S.

A process is a collection of observations that satisfy two "closure" conditions:

- *Local determinacy* — the idea is that once a guess is made, every process behaves like a determinate and monotone (with respect to the partial order on constraints) program.
- *Guess convergence* — we will only make those guesses under which a process can actually quiesce. This is needed to eliminate spurious guesses, which could never be correct as they are not quiescent stores.

Definition 6 (Process) *A process Z is a subset of* **DObs** *satisfying:*

- $(\sqcap S, v) \in Z$ *if* $S \neq \emptyset$ *and* $(S, v) \subseteq Z$
- $(v, v) \in Z$ *if* $(u, v) \in Z$.

Note that under this definition, Default cc processes are closed under arbitrary intersections.

The denotational semantics of the combinators can now be described. The information about the guess v is not needed for the tell or ask combinators. An observation of a parallel composition must be a quiescent point of both programs simultaneously note that a guess v for A, B is propagated down as the guess for A and B. Note the crucial use of the guess/default in the definition for **if** a **else** A — the guess is used to determine if A is initiated.

$$\begin{aligned}
\mathcal{P}[\![A, B]\!] &\stackrel{d}{=} \mathcal{P}[\![A]\!] \cap \mathcal{P}[\![B]\!] \\
\mathcal{P}[\![a]\!] &\stackrel{d}{=} \{(u, v) \in \mathbf{DObs} \mid a \in u\} \\
\mathcal{P}[\![\mathbf{if}\ a\ \mathbf{then}\ A]\!] &\stackrel{d}{=} \{(u, v) \in \mathbf{DObs} \mid \ a \in v \Rightarrow (v, v) \in \mathcal{P}[\![A]\!], \\
&\qquad\qquad\qquad\qquad\qquad\ \ a \in u \Rightarrow (u, v) \in \mathcal{P}[\![A]\!]\}
\end{aligned}$$

[5] A quiescent store u is one in which executing A will not result in the generation of any more information.

$$\mathcal{P}[\![\mathbf{if}\ a\ \mathbf{else}\ A]\!] \stackrel{d}{=} \{(u,v) \in \mathbf{DObs} \mid a \notin v \Rightarrow (u,v) \in \mathcal{P}[\![A]\!]\}$$

Our presentation of Default cc here does not have recursion as recursion in Hybrid cc is handled by guarded recursion over time.

Hiding. Intuitively, the process **new** X **in** A is supposed to behave like the process $A[Y/X]$, where Y is some new variable distinct from any variable occurring in the environment. Somewhat surprisingly, the definition of hiding in the model is subtle and involved. The reason is that the union of two processes is not a process. Therefore, the "internal choice" (or "blind" choice) combinator $A \sqcap B$ of Hoare[6] is not expressible in the model.

Hiding, can, however, mimic internal choice, in the presence of defaults. Consider the process $A \stackrel{d}{=} (\mathbf{if}\ X = 1\ \mathbf{else}\ (Y = 1, X = 2), \mathbf{if}\ X = 2\ \mathbf{else}\ (Z = 1, X = 1))$. These are two conflicting defaults. The process contains in its denotation the observations $((Y = 1, X = 2), (Y = 1, Z = 1, X = 2))$, and $((Z = 1, X = 1), (Y = 1, Z = 1, X = 1))$. However, no information about X can appear in the denotation of the process **new** X **in** A. Consequently, one would expect **new** X **in** A to exhibit the observations $(Y = 1, (Y = 1, Z = 1))$ and $(Z = 1, (Y = 1, Z = 1))$. If **new** X **in** A is to be a process, however, it must be locally determinate: it must also exhibit the glb of these two observations, namely $(\mathtt{true}, (Y = 1, Z = 1))$. However, it cannot do that, since it must either produce $Y = 1$ or produce $Z = 1$. Thus, the straightforward definition of **new** X **in** A cannot be a process.

Our pathway for describing the denotational semantics of hiding and resolving the above problems is as follows — Let Z be any process, we want to define the process $\mathbf{new}_X Z$.

- Recall that the variable X is local to Z. Thus, default assumptions (guesses) about the variable X must be reasonable, *i.e.* there must be some evolution of Z that generates the default assumptions on X — restricting Z to such defaults gives us a subset of Z, call it Z_1.
- Identify the "maximal determinate subprocess" of Z_1 — call it Z_2. This eliminates the possibility of locally indeterminate processes, as was the case above.
- Finally, we follow intuitions from cc to obtain the definition. Consider the behavior of **new** X **in** A on an input a. a may constrain X; however this X is the "external" X which the process must not see. Hence, to obtain the behavior on a, we should observe the behavior on $\exists_X a$. However, the

[6] $A \sqcap B$ behaves like either A or B, and the choice cannot be influenced by the environment.

result, say b, may constrain X, and this X is the "internal" X. Therefore, the result seen by the environment must be $a \sqcup \exists_X b$.

Given a process Z, we build the denotation $\mathbf{new}_X Z$ in three stages, corresponding to the intuitive steps outlined above. First some notation. If S is a set of constraints, define $\exists X.S \stackrel{d}{=} \{u \in |D| \mid \exists u' \in S.\exists_X.u = \exists_X.u'\}$. For any process Z and $(v,v) \in Z$, let $Z_v \stackrel{d}{=} \{u \in |D| \mid (u,v) \in Z\}$.

- Define $Z_1 \stackrel{d}{=} \bigcup\{(Z_v, v) \subseteq Z \mid \forall u \in Z_v, u \supseteq \exists_X v \Rightarrow u = v\}$.
- Define $Z_2 \stackrel{d}{=} \bigcup\{(Z_v, v) \subseteq Z_1 \mid \forall v' \in |D|, (v', v') \in Z_1, \exists_X v = \exists_X v' \Rightarrow \exists_X Z_v = \exists_X Z_{v'}\}$.
- Now $\mathbf{new}_X Z \stackrel{d}{=} \bigcup\{(S, v) \subseteq \mathbf{DObs} \mid \exists v'[(v', v') \in Z_2, \exists_X v = \exists_X v', \exists_X S = \exists_X Z_{v'}]\}$.

So, we have: $\mathcal{P}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] = \mathbf{new}_X \mathcal{P}[\![A]\!]$

Definition 7 *A process Z is X–determinate if $Z_1 = Z_2$, where Z_1, Z_2 are as above.*

With the above definitions, we can work out the denotation of any Default cc process. Here we consider two interesting examples.

Example 8

$$\mathcal{P}[\![\mathbf{if}\ a\ \mathbf{else}\ a]\!] = \{(u,v) \in \mathbf{DObs} \mid a \in v\}$$

This is an example of a default theory which does not have any extensions [44]. However, it does provide some information, it says that the quiescent points must be greater than a, and it is necessary to keep this information to get a compositional semantics.

Example 9

$$\mathcal{P}[\![\mathbf{if}\ a\ \mathbf{then}\ b, \mathbf{if}\ a\ \mathbf{else}\ b]\!] = \\ \{(u,v) \in \mathbf{DObs} \mid b \in v, ((a \notin v) \vee (a \in u)) \Rightarrow b \in u\}$$

This program is "almost" like "if a then b else b", and illustrates the basic difference between positive and negative information. In most semantics, one would expect it to be identical to the program b. However the fact that b is produced by a default makes it different — this is demonstrated by running both the programs b and **if** a **then** b, **if** a **else** b in parallel with **if** b **then** a. b, **if** b **then** a produces $a \sqcup b$ on `true`, while **if** a **then** b, **if** a **else** b, **if** b **then** a produces no output. On the other hand it is not the same as **if** a **then** b, **if** $\neg a$ **then** b — in this program to produce b, either a or $\neg a$ must be present in the store, whereas the original program will produce b if nothing is present in the store.

Input-output behavior. Given a process Z, we can define its input-output behavior as follows —

$$Z(a) = \{b \in D \mid b \vdash a, (\bar{b}, \bar{b}) \in Z, (\forall u)(u, \bar{b}) \in Z, a \in u \Rightarrow u = \bar{b}\}$$

Thus the output on a is any guess b such that there is no quiescent point between a and b — once a is added to the store, under the final assumption b, Z will quiesce only on b. Z may be extended to constraints.

Remark 10 *Note the role of observations of (u, v) with $u \neq v$. Such observations play a crucial role in the determination of the input-output behavior of the process. However, any output v will perforce occur in the denotation of the process as (v, v).*

Example 11 In the program **if** a **else** a, there is no possible output above the input `true`. Thus running such a program would produce an error. On the other hand, there can be multiple outputs — both a and b are possible outputs on input `true` for the program **if** a **else** b, **if** b **else** a.

This leads us to the definition of determinacy.

Definition 12 *A process Z is called determinate if its input-output behavior is a function with domain D.*

Clearly, a is determinate for all a, and it can be shown that if A is determinate, then **if** a **then** A and **new** X **in** A are determinate. However, **if** a **else** A and A, B may be indeterminate even if A and B are determinate.

In [45], we provide an algorithm to detect when any program is indeterminate — we briefly sketch the algorithm later in the section. This semantic notion of indeterminacy captures the essence of the "causality cycles" in imperative synchronous programming languages.

Note the if Z is determinate, then Z is X-determinate for all variables X ([45]).

Operational semantics. A configuration Γ is a multiset of programs — to be thought of as the parallel composition of the programs. $\sigma(\Gamma)$ is the conjunction of the tokens in the tell agents in Γ. We define binary transition relations $\longrightarrow_a$ on configurations; thus, the transition relation is indexed by the "guessed output token" a that will be used to evaluate defaults.

$$\frac{a \not\vdash b}{\Gamma, \textbf{if } b \textbf{ else } B \longrightarrow_a \Gamma, B}$$

The other transition rules are as in cc — thus reinforcing the intuition that given a guess, a Default cc program essentially becomes a cc program.

$$\frac{\sigma(\Gamma) \vdash b}{\Gamma, \textbf{if } b \textbf{ then } B \longrightarrow_a \Gamma, B}$$

$$\Gamma, \textbf{new } X \textbf{ in } A \longrightarrow_a \Gamma, A[Y/X] \quad (Y \text{ not free in } A, \Gamma)$$

$$\Gamma, (A, B) \longrightarrow_a \Gamma, A, B$$

Execution of an program A corresponds to finding a terminal configuration Γ such that the "guessed output constraint" is actually achieved — see remark 10

$$\frac{\exists a \in D \quad \Gamma \longrightarrow_a^* \Gamma' \not\longrightarrow_a \quad \sigma(\Gamma') \approx a}{\Gamma \longrightarrow \Gamma'}$$

The output of the program is $\exists_{\vec{\mathbf{Y}}} a$ where $\vec{\mathbf{Y}}$ are the new variables in Γ' introduced by the operational semantics. The transition relation can easily be extended to constraints instead of tokens — indeed, this is the form which we use to show its equivalence to the denotational semantics.

Correspondence theorems. We can now show that this semantics corresponds to the denotational semantics closely. Define from the operational semantics

$$\begin{aligned}\mathcal{O}[\![A]\!] = \{(u, v) \in \mathbf{DObs} \mid (\exists v')(A, u) \longrightarrow_{v'}^* A' \not\longrightarrow_{v'}, u = \exists_{\vec{\mathbf{Y}}} \sigma(A'), v = \exists_{\vec{\mathbf{Y}}} v', \\ (A, v) \longrightarrow_{v'}^* A'' \not\longrightarrow_{v'}, v = \exists_{\vec{\mathbf{Y}}} v', v' = \sigma(A'')\}\end{aligned}$$

$\mathcal{O}[\![A]\!]$ defines the set of observations that can be made from the transition relation — recall that an observation consisted of a quiescent store, together with the guess about the final result. So $(u, v) \in \mathcal{O}[\![A]\!]$, if under guess v, u is a quiescent point, and v is itself a possible quiescent point. Note that the actual guess used is v', which contains in addition to v information on the new variables introduced in the derivation — this information is eliminated later by existential quantification. The following lemma shows that this intuition about observations is correct.

Lemma 13

$$\begin{aligned}\mathcal{O}[\![a]\!] &= \mathcal{P}[\![a]\!] \\ \mathcal{O}[\![A, B]\!] &= \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!]\end{aligned}$$

$$
\begin{aligned}
\mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!] = \{(u,v) \in \textbf{DObs} \mid\ & a \in v \Rightarrow (v,v) \in \mathcal{O}[\![A]\!], \\
& a \in u \Rightarrow (u,v) \in \mathcal{O}[\![A]\!]\} \\
\mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] = \{(u,v) \in \textbf{DObs} \mid\ & a \notin v \Rightarrow (u,v) \in \mathcal{O}[\![A]\!]\} \\
\mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] = \textbf{new}_X \mathcal{O}[\![A]\!],\ & \textit{if } \mathcal{O}[\![A]\!] \textit{ is } X - \textit{determinate}
\end{aligned}
$$

PROOF. The proof follows extant proofs [47,32] for languages in the cc paradigm, and is presented in the appendix A.

Now we can show the following theorem. We say that P satisfies the X–determinacy condition if whenever **new** X **in** A is a subprogram of P, then $\mathcal{O}[\![A]\!]$ is X–determinate.

Theorem 14 *If P satisfies the X–determinacy condition for all X, $\mathcal{P}[\![P]\!] = \mathcal{O}[\![P]\!]$.*

PROOF. A simple structural induction using the above lemma 13 yields the required result.

Determinacy detection. The above theorem allows us to use the operational semantics for determinacy detection. Let P be a program. The basic idea is simple — use the operational semantics to check that $(\forall a)\ [|\mathcal{O}[\![P]\!](a)| = 1]$. Notice that there is a quantification on a, and implicitly, one on guessed constraints. Our earlier work [45] shows that it suffices to consider only a carefully chosen finite set for these.

4 Hybrid cc— Language and Denotational Semantics

Hybrid cc is intended to be a language for describing hybrid systems. Recall that a run of a hybrid system consists of an alternating sequence of points and open intervals — open intervals of continuous evolution connected by points of discrete change where discontinuities can occur.

In this section, we describe Hybrid cc as an "extension" of Default cc over continuous time. This extension proceeds in two stages. First, we introduce the notion of a *continuous* constraint system — a real-time extension of constraint systems — to describe the continuous evolution of system trajectories alluded to above. Next, we extend the Default cc model of processes over continuous time to describe Hybrid cc processes.

Notation. We use $\mathbb{R}^+$ for the set of non-negative real numbers, and will always mean $\mathbb{R}^+$ when we say "reals". We use standard notation for intervals of real numbers: $(t1, t2)$ for the open interval, $[t1, t2)$ for the left closed and right open interval etc. We will use `Int` for the interior operator on intervals. *e.g.* $\mathtt{Int}([t1, t2]) = (t1, t2)$. We will be working with partial functions on the reals—their domains will be initial segments of $\mathbb{R}^+$, *i.e.* $[0, r)$ or $[0, r]$ for some $r \in \mathbb{R}^+$ (where $[0, 0) = \emptyset$). We use $\mathtt{dom}(f)$ for the domain of definition of f.

We define restriction, $f \restriction I$ where I is a left closed interval, as $(f \restriction I)(t) = f(t1 + t)$ where $t + t1 \in \mathtt{dom}(f), t \in I$, and the left endpoint of I is $t1$. Given two partial functions f, g, we say f is a *prefix* of g if $\mathtt{dom}(f) \subseteq \mathtt{dom}(g)$ and $\forall t \in \mathtt{dom}(f)[f(t) = g(t)]$. f is a *proper prefix* of g if f is a prefix of g and $f \neq g$.

If f is a prefix of g, we define the function g **after** f as: $(g \textbf{ after } f)(t) = g(t + r)$, where $\mathtt{dom}(f) = [0, r)$ or $[0, r]$. This is extended to sets of partial functions S in a natural way: $S \textbf{ after } f \stackrel{d}{=} \{g \textbf{ after } f \mid g \in S, f \text{ is a prefix of } g\}$. We also define $S(0) = \{g(0) \mid g \in S\}$.

We use π_1, π_2 for the first and second projections on pairs. Function composition is indicated as $f \circ g$.

4.1 Continuous constraint systems

Continuous constraint systems (ccs) augment constraint systems with the notion of a constraint holding continuously over a period of time. Intuitively, we proceed by adding structure to constraint systems to capture the information content of "initial value problems" in integration. This is done via the following mechanisms:

- For every token a, we have a token $\mathtt{cont}(a)$. Intuitively, such a token is an *activity condition.* If it holds at time $t \in \mathbb{R}^+$, it means that a is in effect at every instant $r > t$.
- For every token a, we have a token $\mathtt{init}(a)$. Intuitively, this token is an *initial condition* at time $t \in \mathbb{R}^+$ for the integration problem to be solved in the open interval starting at t.
- We require to be given, for every $r \in \mathbb{R}^+$, a relation $\vdash_r$. This relation describes what tokens (the *r-projection*) must follow at time r given some initial and activity conditions — such a family of relations captures the information content of "initial value problems" in integration.
- Finally, for every token a, we have a token $\mathtt{prev}(a)$. Intuitively, $\mathtt{prev}(a)$ holds at $t > 0$ if a holds as time tends to t in the interval $(t - \epsilon, t)$. This information is redundant in the context of the continuous evolution being

captured by a ccs [7]. However, it plays a crucial role in later sections, particularly in the programming examples, by allowing discontinuities in the evolution of processes in Hybrid cc.

Formally, we proceed as follows. Let $\langle \mathtt{Base}_D, \vdash_{\mathtt{Base}_D}, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ be a constraint system. Define the constraint system $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ as follows: The tokens, D, are given inductively by:

$$\begin{aligned}
&\mathtt{Base}_D \subseteq D \\
&a \in \mathtt{Base}_D \Rightarrow \mathtt{init}(a), \mathtt{prev}(a) \in D \\
&a, b \in D \Rightarrow a \wedge b, \exists_X a, \mathtt{cont}(a) \in D
\end{aligned} \tag{1}$$

We call a token *instantaneous* if it is built out of tokens from $\mathtt{Base}_D$, $\mathtt{init}(\mathtt{Base}_D)$ and $\mathtt{prev}(\mathtt{Base}_D)$, *i.e.* it does not contain an activity condition. We use $\mathbf{a}, \mathbf{b}$ etc. for general tokens of D; we use a, b etc for instantaneous tokens. We use ID for the set of instantaneous tokens.

The entailment relation, $\vdash$, is obtained by augmenting $\vdash_{\mathtt{Base}_D}$:

$$\begin{aligned}
&a \vdash_{\mathtt{Base}_D} b \Rightarrow \begin{cases} a \vdash b \\ \mathtt{init}(a) \vdash \mathtt{init}(b) \qquad \mathtt{prev}(a) \vdash \mathtt{prev}(b) \end{cases} \\
&\mathbf{a} \vdash \mathbf{b} \Rightarrow \mathtt{cont}(\mathbf{a}) \vdash \mathtt{cont}(\mathbf{b})
\end{aligned} \tag{2}$$

We demand that $\langle ID, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ is a sub–constraint system of$\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$, *i.e.*

$$a \vdash \mathbf{b} \Rightarrow \ \mathbf{b} \text{ is an instantaneous token.} \tag{3}$$

We denote the set of constraints associated with $\langle ID, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ by $|ID|$; we denote the set of constraints associated with $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ by $|D|$. As with any constraint system, $|ID|$ ((resp. $|D|$) ordered by inclusion is a complete algebraic lattice. We use $u, v, \ldots$ for instantaneous constraints and $\mathbf{u}, \mathbf{v}, \ldots$ for constraints of D.

[7] After all, for a continuous function, the left limit equals the right limit which in turn equals the value of the function at the point

The constructions $\texttt{init}, \texttt{prev}, \texttt{cont}$ commute with the logical connectives.

$$
\begin{array}{ll}
\texttt{init}(a \wedge b) \approx \texttt{init}(a) \wedge \texttt{init}(b) & \texttt{init}(\exists_X a) \approx \exists_X(\texttt{init}(a)) \\
\texttt{prev}(a \wedge b) \approx \texttt{prev}(a) \wedge \texttt{prev}(b) & \texttt{prev}(\exists_X a) \approx \exists_X(\texttt{prev}(a)) \\
\texttt{cont}(\mathbf{a} \wedge \mathbf{b}) \approx \texttt{cont}(\mathbf{a}) \wedge \texttt{cont}(\mathbf{b}) & \texttt{cont}(\exists_X \mathbf{a}) \approx \exists_X(\texttt{cont}(\mathbf{a}))
\end{array} \tag{4}
$$

Our final condition on $\vdash$ is that $\texttt{cont}$ is idempotent.

$$\texttt{cont}(\texttt{cont}(a)) \approx \texttt{cont}(a) \tag{5}$$

We now describe the conditions on the relation $\vdash_r$. $\vdash_r \subseteq D \times ID$ — intuitively $\mathbf{a} \vdash_r b$ holds if the "integration" of the activity conditions in $\mathbf{a}$ with the initial conditions in $\mathbf{a}$ for length r yields b. Indeed, $\vdash_r$ induces *integration* operators $\int^r$, for every $r \in \mathbb{R}^+$ are generated by $\vdash_r$:

$$\int^r \mathbf{b} \stackrel{d}{=} \{a \in ID \mid \mathbf{b} \vdash_r a\}$$

All operations on tokens, $\int^r, \wedge, \exists_X, \vdash, \vdash_r, \texttt{prev}, \texttt{cont}, \texttt{init}$, described above lift to operations on the set of constraints in the constraint system induced by $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$.

Formally, we require the following conditions on $\vdash_r, \int^r$: It is not the case that for instantaneous tokens a that $a \vdash_r a$, unless of course $r = 0$.

$$a \vdash_0 a \tag{6}$$

$\texttt{cont}(a)$ maintains a for all $r > 0$.

$$\texttt{cont}(a) \vdash_r a, \text{ if } r > 0 \tag{7}$$

Only the initial conditions, *i.e.* tokens of the form $\texttt{init}$(a), and the activity conditions, *i.e.* tokens of the form $\texttt{cont}$(a) matter on the left hand side.

$$\frac{(\forall c)\ [\mathbf{b} \vdash \texttt{init}(c) \ \Leftrightarrow\ \mathbf{b}' \vdash \texttt{init}(c),\ \mathbf{b} \vdash \texttt{cont}(c) \ \Leftrightarrow\ \mathbf{b}' \vdash \texttt{cont}(c)]}{(\forall r > 0)\ [\mathbf{b} \vdash_r a \Leftrightarrow \mathbf{b}' \vdash_r a]} \tag{8}$$

For all $r > 0$, evolution is continuous, *i.e.* the left limit, the "value" at r and the right initial value all agree.

$$\mathbf{b} \vdash_r a \ \Leftrightarrow\ \mathbf{b} \vdash_r \texttt{prev}(a) \ \Leftrightarrow\ \mathbf{b} \vdash_r \texttt{init}(a) \tag{9}$$

Transitivity must be preserved on the left and the right, with respect to $\vdash$:

$$\frac{\mathbf{b}' \vdash \mathbf{b} \qquad \mathbf{b} \vdash_r a' \qquad a' \vdash a''}{\mathbf{b}' \vdash_r a''} \tag{10}$$

The *duration* of integration is important, not *when* it was started.

$$\frac{\int^r [u \wedge \mathtt{cont}(\mathbf{b})] = v \quad \int^s [v \wedge \mathtt{cont}(\mathbf{b})] = w}{\int^{r+s} [u \wedge \mathtt{cont}(\mathbf{b})] = w} \tag{11}$$

To prevent constraints like $\mathtt{rational}(\mathrm{x})$, we require a neighborhood of 0, where $\vdash_r$ information is "stable". Define:

$$\mathbf{a} \vdash^r b \stackrel{d}{=} \forall t \in (0, r)[\mathbf{a} \vdash_t b]$$
$$\mathbf{a} \not\vdash^r b \stackrel{d}{=} \forall t \in (0, r)[\mathbf{a} \not\vdash_t b]$$

Then we require:

$$(\forall \mathbf{a} \in D, b \in ID)(\exists r > 0) \ [(\mathbf{a} \vdash^r b) \vee (\mathbf{a} \not\vdash^r b)] \tag{12}$$

Finally, we demand that $\int^r$ commutes with $\exists_X$.

$$\int^r \exists_X \mathbf{b} = \exists_X \int^r \mathbf{b} \tag{13}$$

Definition 15 (Continuous Constraint System) *A continuous constraint system (ccs) built on a constraint system* $\langle \mathtt{Base}_D, \vdash_{\mathtt{Base}_D}, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ *is a tuple* $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}, \{\vdash_r \mid r \in \mathbb{R}^+\}\rangle$ *that satisfies equations (1)–13.*

Computability: The following are (sufficient) conditions to make the operational semantics effective.

(i) $\vdash, \vdash_r, \vdash^r, \not\vdash^r$ are decidable.
(ii) For all tokens $\mathbf{a}$ and for all real numbers r, there exists a token b such that $\bar{b} = \int^r \mathbf{a}$.
(iii) For all tokens $\mathbf{a}$, there are tokens b, d such that:

$$\bar{b} = \{\mathtt{init}(c) \mid \mathbf{a} \vdash \mathtt{init}(c)\}, \bar{d} = \{\mathtt{prev}(c) \mid \mathbf{a} \vdash \mathtt{prev}(c)\}$$

(iv)

$$\frac{a, \mathbf{b} \vdash_r c \quad r > 0}{\exists_X a, \mathbf{b} \vdash_r c}$$

This condition says that existentially quantifiable variables cannot pass information across time. Together with condition 13, we can now prove that for all $r > 0$:

$$\int^{r} \exists_X(a \wedge \texttt{cont}(\mathbf{b})) = \int^{r} \exists_X(a) \wedge \ \exists_X(\texttt{cont}(\mathbf{b}))$$

This makes the existential quantification similar to the flexible quantification of temporal logic (see [36]). The combination of these conditions allows the implementor of the ccs to use *one* copy of the quantified variable for a given evolution of the ccs.

Example 16 Let $\langle \texttt{Base}_D, \vdash_{\texttt{Base}_D}, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ be a constraint system. Then, the "free" ccs $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}, \{\vdash_r \ \mid r \in \mathbb{R}^+\}\rangle$ has $\vdash_r$ defined as follows.

$$\mathbf{b} \vdash_0 a \text{ iff } \mathbf{b} \vdash a$$

$$\mathbf{b} \vdash_r a \text{ if } \mathbf{b} \vdash \texttt{cont}(a),\ r > 0$$

We note that the integration operators $\int^r \mathbf{b}$ are quite trivial— intuitively, at $r = 0$, we get the instantaneous constraints in $\mathbf{b}$, and for $r > 0$, we get the constraints a such that $\mathbf{b} \vdash \texttt{cont}(a)$. In particular if $r > 0$, $\int^r \mathbf{b}$ is independent of the instantaneous constraints in b — thus this ccs does not allow any "autonomous" evolution.

Example 17 Let $\langle \texttt{Base}_D, \vdash_{\texttt{Base}_D}, \emptyset, \emptyset\rangle$ be the constraint system whose basic tokens are formulas of the form $dot(x, m) = r$ or $x = r$, for x a symbol representing a real variable, m a non-negative integer and r a real number. The inference relation $\vdash_{\texttt{Base}_D}$ is defined in the obvious way under the interpretation that $dot(x, m) = r$ states that the mth derivative of x is r.

Now, consider the ccs $\langle D, \vdash, \emptyset, \emptyset, \{\vdash_r \ \mid r \in \mathbb{R}^+\}\rangle$ generated as follows. The token $\texttt{cont}(dot(x, m) = r)$ is interpreted as saying that that for all time $t > 0$ the mth derivative of x is r; and the intention is that $\int^r \mathbf{b}$ is standard integration — a definite integral between endpoints 0 and r of the activity conditions in $\mathbf{b}$ with initial conditions given by the tokens of form $\texttt{init}(x = c)$. The token $\texttt{prev}(x = c)$ holds at time r if the left limiting value of x is c.

The inference relations are trivially decidable: the functions of time expressible are exactly the polynomials. The $\vdash_r, \not\vdash_r, \vdash^r, \not\vdash^r$ relations are expressible parametrically in r; the only non-trivial computation involved is that of finding the smallest non-negative root of univariate polynomials (this can be done using numerical integration).

Finally note that in this constraint system, under the intended interpretation, the real variables do not satisfy the computability condition (iv) — hence, existential quantification of these is not allowed (indicated by $\mathbf{Var} = \emptyset$).

Trace of a ccs. In the rest of this paper, we will take a more extensional view of the continuous evolution captured by a ccs — as functions from reals to the instantaneous constraints entailed at each point. To this end, we first characterize the functions that describe evolutions of a ccs. Let $f : \mathbb{R}^+ \to |ID|$ be a partial function whose domain is a prefix of the positive reals, and whose range is instantaneous constraints. Let

$$\texttt{Active}(f) = \{a \in ID \mid (\forall r \in \texttt{dom}(f) - \{0\})\ [f(r) \vdash a]\}$$

Definition 18 *$f : \mathbb{R}^+ \to |ID|$ is a trace of the ccs if:*

$$(\forall r \in \texttt{dom}(f))\ [\int^{r} [f(0) \wedge \texttt{cont}(\texttt{Active}(f))] = f(r)]$$

Thus, if f is a trace, the value of f at r is obtained by "integrating" the activity conditions of f — namely $\texttt{Active}(f)$ — with initial conditions given by $f(0)$.

4.2 Denotational Model of Hybrid cc

In the rest of this section we will assume that we are working in some continuous constraint system $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}, \{\vdash_r \mid r \in \mathbb{R}^+\}\rangle$ built on a constraint system $\langle \texttt{Base}_D, \vdash_{\texttt{Base}_D}, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$. Let **DObs** be the set of Default cc observations over the constraint system ID: $\mathbf{DObs} = \{(u, v) \in |ID| \times |ID| \mid v \supseteq u\}$.

Observations. An observation, a run of the hybrid system, is a tracing of the system trajectory over time — open intervals of continuous evolution connected by points of discrete change.

What should an observation be?

First, we intend the execution at every time instant to be modeled by the execution in Default cc. So observations are functions $f : \mathbb{R}^+ \to \mathbf{DObs}$, such that $\texttt{dom}(f) = [0, r)$ or $[0, r]$, a prefix of the non-negative reals.

Secondly, we intend f to satisfy *piecewise continuity* — *i.e.* for any point in the interior of its domain, both components of the behavior of f on some open interval to the right arise from traces of the ccs. So, we demand that

$$\forall t \in \texttt{dom}(f) \exists \epsilon > 0 \text{ such that } \pi_1 \circ f \restriction [t, t+\epsilon), \pi_2 \circ f \restriction [t, t+\epsilon) \text{ are ccs traces.}$$

For such an f, we can partition its domain into a (possibly infinite) alternating collection of points and open intervals, called the *phases* of the observation.

Let $t \in \mathtt{Int}(\mathtt{dom}(f))$ be such that there is no neighborhood $[t-\epsilon, t+\epsilon)$ around t, such that $\pi_1 \circ f \upharpoonright [t-\epsilon, t+\epsilon)$ and $\pi_2 \circ f \upharpoonright [t-\epsilon, t+\epsilon)$ are traces of the ccs. This indicates that there is a "discontinuity" at t, and t is called a *point phase* of f. 0 is defined to be a point phase. Between two successive point phases, there is an *(open) interval phase*, where evolution proceeds via the ccs.

Finally, we intend f to satisfy *observability* — *i.e.* the computation at any time point in $\mathtt{dom}(f)$ is a completed Default cc computation — a computation in which the guess was attained(recall remark 10). Thus, we want to ensure that if $f(r) = (u, v), u \neq v$, then r is in the terminal phase of f. Formally, we have two cases depending on whether the last phase is a point phase — (1) r is the least real at which $\pi_1 \circ f, \pi_2 \circ f$ disagree –

$$(\forall r' < r)[\pi_1(f(r')) = \pi_2(f(r'))] \wedge \ \pi_1(f(r)) \neq \pi_2(f(r)) \Rightarrow \mathtt{dom}(f) = [0, r]$$

or an interval phase (2) $\pi_1(f(r)) \neq \pi_2(f(r))$ and there is no minimal such r, then r must be in the last continuous phase –

$$\pi_1(f(r)) \neq \pi_2(f(r)) \Rightarrow \pi_1 \circ f \upharpoonright [r, \infty), \pi_2 \circ f \upharpoonright [r, \infty) \text{ are ccs traces}$$

Definition 19 (Observations) **HObs** *consists of functions* $f : \mathbb{R}^+ \to$ **DObs** *such that* $\mathtt{dom}(f) = \emptyset, [0, r)$ *or* $[0, r]$ *for some* $r \in \mathbb{R}^+$ *and* f *satisfies* piecewise continuity *and* observability*.*

For $f \in$ **HObs**, define for every $t \in \mathtt{Int}(\mathtt{dom}(f)) \cup \ \{0\}$, $f(t^+) = (\mathtt{Active}(\pi_1 \circ f \upharpoonright [t, t+\epsilon)), \mathtt{Active}(\pi_2 \circ f \upharpoonright [t, t+\epsilon)))$, where $\epsilon > 0$ and $\pi_1 \circ f \upharpoonright [t, t+\epsilon), \pi_2 \circ f \upharpoonright [t, t+\epsilon)$ are traces of the ccs. Extend this to sets of observations by $S(t^+) = \{f(t^+) \mid f \in S, t \in \mathtt{dom}(f)\}$.

Processes. Analogous to Default cc, a process is a collection of observations. What closure properties should a process satisfy?

The first two closure conditions are quite general. The future cannot alter the past — thus, we demand prefix closure. Also, all effective computational systems satisfy a limit closure property — if every proper prefix of an observation f is in the process, then so is f.

The other closure conditions arise from the fact that we intend the Hybrid cc processes to be Default cc *generable.* Thus, we intend the execution at every time instant to be modeled by the execution in Default cc.

For all f in P $(\forall t \in \mathtt{dom}(f))$ $(P$ **after** $f \upharpoonright [0, t))(0)$ is a Default cc process.

Furthermore, we intend the continuous behavior to be generated by Default cc processes.

For all f in P $(\forall t \in \mathtt{dom}(f))(P$ **after** $f \upharpoonright [0,t])(0^+)$ is a Default cc process.

Definition 20 *A process P is a non-empty, prefix closed subset of* **HObs** *satisfying limit closure and* Default cc *generability.*

Combinators. a, **if** a **then** A, **if** a **else** A, (A, B) are inherited from Default cc and their denotations are induced by their Default cc definitions. We emphasize that here the tokens a are the instantaneous tokens of the ccs. ϵ represents the function with empty domain.

$$
\begin{aligned}
\mathcal{H}[\![a]\!] &\stackrel{d}{=} \{\epsilon\} \cup \{f \in \mathbf{HObs} \mid f(0) = (u,v), a \in u\} \\
\mathcal{H}[\![\mathbf{if}\ a\ \mathbf{then}\ A]\!] &\stackrel{d}{=} \{\epsilon\} \cup \{f \in \mathbf{HObs} \mid f(0) = (u,v), a \in u \Rightarrow f \in \mathcal{H}[\![A]\!], \\
&\qquad\qquad a \in v \Rightarrow (v,v) \in \mathcal{H}[\![A]\!]\} \\
\mathcal{H}[\![\mathbf{if}\ a\ \mathbf{else}\ A]\!] &\stackrel{d}{=} \{\epsilon\} \cup \{f \in \mathbf{HObs} \mid f(0) = (u,v), a \notin v \Rightarrow f \in \mathcal{H}[\![A]\!]\} \\
\mathcal{H}[\![A, B]\!] &\stackrel{d}{=} \mathcal{H}[\![A]\!] \cap \mathcal{H}[\![B]\!]
\end{aligned}
$$

Here when we say $(v,v) \in \mathcal{H}[\![A]\!]$, we mean that the function f with $\mathtt{dom}(f) = \{0\}$ and $f(0) = (v,v)$ is in $\mathcal{H}[\![A]\!]$. **new** X **in** A imposes the constraints of A, but hides the variable X from the other programs. Here the variable X must be one for which $\exists_X$ is allowed in the ccs. Every observation $f \in \mathcal{H}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$ is induced by an observation $g \in \mathcal{H}[\![A]\!]$ with the same domain and satisfying:

- For every $t \in \mathtt{dom}(f) = \mathtt{dom}(g)$, $\exists_X \pi_1(f(t)) = \exists_X \pi_1(g(t))$ and $\exists_X \pi_2(f(t)) = \exists_X \pi_2(g(t))$. We write this as $\exists_X f = \exists_X g$.
- For every $t \in \mathtt{dom}(f)$, $f(t)$ must equal the result of hiding X in the Default cc process given by A at time t after history $g \upharpoonright [0,t)$. A similar condition must hold for $f(t^+)$.

Formally,

$$
\begin{aligned}
&\mathcal{H}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] \stackrel{d}{=} \\
&\quad \{f \in \mathbf{HObs} \mid \exists g \in \mathcal{H}[\![A]\!].\exists_X f = \exists_X g, \\
&\qquad (\forall t \in \mathtt{dom}(f))\ [f(t) \in \mathbf{new}_X(\mathcal{H}[\![A]\!]\mathbf{after}\ g \upharpoonright [0,t))(0)], \\
&\qquad (\forall t \in \mathtt{Int}(\mathtt{dom}(f)) \cup \{0\})\ [f(t^+) \in \mathbf{new}_X(\mathcal{H}[\![A]\!]\mathbf{after}\ g \upharpoonright [0,t])(0^+)]\}
\end{aligned}
$$

We define the process $\mathbf{new}_X Z$ by replacing $\mathcal{H}[\![A]\!]$ with Z in the right sides of the above definition.

The definitions for **hence** is as expected — observations have to "satisfy" A everywhere after the first instant. Thus, intuitively, **hence** is a specialized form of "infinite" parallel composition of time shifted copies of A.

$$\mathcal{H}[\![\textbf{hence } A]\!] \stackrel{d}{=} \{f \in \textbf{HObs} \mid (\forall t \in \texttt{dom}(f) - \{0\})\ [f \restriction [t, \infty) \in \mathcal{H}[\![A]\!]]\}$$

Example 21 Consider the ccs of example 17. Consider the program $P = \texttt{init}(x = 0), \textbf{hence } (\texttt{dot}(x, 1) = 1)$. The denotation of P consists of all observations f that satisfy: $\pi_1(f(0)) \vdash \texttt{init}(x = 0)$, $(\forall r \in \texttt{dom}(f))\ [\pi_1(f(r)) \vdash \texttt{dot}(x, 1) = 1]$. From this, we deduce that for all r in an interval phase, (r', r'') of f, we have $\pi_1(f(r)) \vdash x = r + c$, where $\texttt{init}(x = c)$ holds in the prior point phase $f(r')$. In particular, for all r in the first interval phase of f, we have $\pi_1(f(r)) \vdash x = r$, Note however, that the semantics allows discontinuities in the value of x — thus at a point phase at r, $\texttt{init}(x = r)$ does not necessarily hold.

Equational laws. The above combinators satisfy the following equational laws.

All combinators commute with parallel composition.

$$\begin{aligned}
\textbf{hence } (A, B) &= \textbf{hence } A, \textbf{hence } B \\
\textbf{if } a \textbf{ then } (A, B) &= \textbf{if } a \textbf{ then } A, \textbf{if } a \textbf{ then } B \\
\textbf{if } a \textbf{ else } (A, B) &= \textbf{if } a \textbf{ else } A, \textbf{if } a \textbf{ else } B \\
(\textbf{new } X \textbf{ in } A), B &= \textbf{new } X \textbf{ in } (A, B), \text{ if } X \text{ not free in } B
\end{aligned}$$

hence is idempotent: **hence hence** A = **hence** A.

The order of conditions does not matter.

$$\begin{aligned}
\textbf{if } a \textbf{ else if } b \textbf{ then } A &= \textbf{if } b \textbf{ then if } a \textbf{ else } A \\
\textbf{if } a \textbf{ else if } b \textbf{ else } A &= \textbf{if } b \textbf{ else if } a \textbf{ else } A \\
\textbf{if } a \textbf{ then if } b \textbf{ then } A &= \textbf{if } (a \sqcup b) \textbf{ then } A
\end{aligned}$$

Finally, existential variables do not carry information across time

$$\begin{aligned}
\textbf{new } X \textbf{ in } a &= \exists_X a \\
\textbf{new } X \textbf{ in hence } a &= \textbf{hence new } X \textbf{ in } a \\
\textbf{new } X \textbf{ in } (a, \textbf{hence } A) &= \textbf{new } X \textbf{ in } a, \textbf{new } X \textbf{ in } (\textbf{hence } A)
\end{aligned}$$

4.3 Defined combinators

We illustrate the power of the above combinators by defining some combinators to succinctly write various common patterns of temporal activity. In the following, `stop, go` will stand for signal variables/constraints as described in the `Gentzen` constraint system (see example 4). Thus the agent `stop` will mean the signal `stop` is present. We illustrate the behavior of the combinators via timing diagrams.

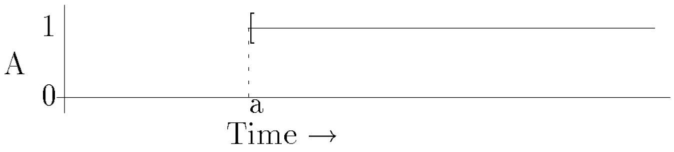

Fig. 4. Time diagram for **first** a **then** A. 1 means A is active, 0 means it is inactive.

Example 22 **first** a **then** A reduces to A at the first time a is true, if there is such an instant, as shown in Fig. 4. If there no well-defined notion of first occurrence of a, A will not be invoked. For example, there is no "first" occurrence of a in some observations of $\mathcal{H}[\![\textbf{hence } a]\!]$, even though all of them have occurrences of a. The following definition captures these intuitions.

$$\mathcal{H}[\![\textbf{first } a \textbf{ then } A]\!] \stackrel{d}{=} \{f \in \textbf{HObs} \mid \ (\exists r \in \mathtt{dom}(f))\ [(\forall r' < r) a \notin \pi_1(f(r')),$$
$$a \in \pi_1(f(r))] \Rightarrow f \restriction [r, \infty) \in \mathcal{H}[\![A]\!]\}$$

It can be expressed in terms of the basic combinators as

$$\textbf{first } a \textbf{ then } A = \textbf{new } \mathtt{stop} \textbf{ in always } (\textbf{if } \mathtt{stop} \textbf{ else if } a \textbf{ then } A,$$
$$\textbf{if } a \textbf{ then hence } \mathtt{stop})$$

where **always** $A \stackrel{d}{=} A,$ **hence** A.

Note that the above definition satisfies the caveat on first occurrences of a. For example, if a occurs for the first time throughout an open interval of continuous execution, then **if** a **then hence** x will make **hence** x be true in the entire open interval, so x will be true in the entire interval also. This will prevent the triggering of the A in the interval.

Example 23 An important construct for reactive programs is multiform time — **time** A **on** a — it allows us to pass to a process only certain fractions of the real line. This construct is characteristic of synchronous programming languages — it enables us to write a process without referring to real time — we can write it with reference to the relevant notion of "logical time", which is the time at which a holds. The process A runs only during the times a holds, as shown in Fig. 5. Formally, we proceed as follows.

Let $f \in \mathbf{HObs}$. Define $\mathtt{dom}_a(f) = \{t \in \mathtt{dom}(f) \mid a \in \pi_1(f(t))\}$. We say that f is a–good, if the set $\mathtt{dom}_a(f)$, ordered by $\leq$, is composed of a series of left-closed right-open intervals, except the last which may be right closed. Intuitively, f is a–good if the logical time given by a can be substituted for real time, *i.e.* the set of reals at which a holds can be glued together to get a prefix of the reals. If f is a–good, we define $f \upharpoonright_a$ as the pasting together of the phases of f in which a holds. Now, we can define:

$$\mathcal{H}[\![\textbf{time } A \textbf{ on } a]\!] = \{f \in \mathbf{HObs} \mid f \upharpoonright_a \in \mathcal{H}[\![A]\!], f \text{ is } a\text{–good}\}$$

This combinator satisfies the following equational laws. These laws can be used to remove occurrences of the combinator.

$$\begin{aligned}
\textbf{time } b \textbf{ on } a &= \textbf{first } a \textbf{ then } b \\
\textbf{time } (\textbf{if } b \textbf{ then } B) \textbf{ on } a &= \textbf{first } a \textbf{ then if } b \textbf{ then time } B \textbf{ on } a \\
\textbf{time } (\textbf{if } b \textbf{ else } B) \textbf{ on } a &= \textbf{first } a \textbf{ then if } b \textbf{ else time } B \textbf{ on } a \\
\textbf{time } (A, B) \textbf{ on } a &= (\textbf{time } A \textbf{ on } a), (\textbf{time } B \textbf{ on } a) \\
\textbf{time new } x \textbf{ in } A \textbf{ on } a &= \textbf{new } x \textbf{ in time } A \textbf{ on } a, (x \text{ not free in } a) \\
\textbf{time } (\textbf{hence } B) \textbf{ on } a &= \textbf{first } a \textbf{ then } [\textbf{hence } (\textbf{if } a \textbf{ then time } B \textbf{ on } a)]
\end{aligned}$$

Proof are once again omitted, as they involve simple manipulations of definitions. Only the last law needs some explanation — intuitively we start off a copy of B each time a becomes true after the first occurrence of a (this is caused by the **hence** B). However, since B itself may have extended behavior over time, it should also be in the scope of a **time** B **on** a, since it should be active only when a is true.

Now, we synthesize a variety of combinators using the above two combinators as building blocks.

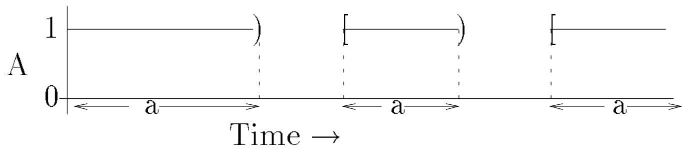

Fig. 5. Time diagram for **time** A **on** a.

Example 24 The watchdog construct **do** A **watching** a is necessary for writing reactive programs modularly. This construct allows us to terminate the execution of A whenever a occurs, as shown in Figure 6. For example, it is used in the billiards ball program later to terminate the movement of a ball when it falls into a pocket — it allows us to write the motion of a ball without referring to the pockets.

Semantically, an observation of **do** A **watching** a can arise as follows:

- A is executed in the interval $[0, r)$; a is not entailed in $[0, r)$ and a is entailed at r. In this case the process A is aborted at instant r (in particular, A is not executed at r). There are no restrictions imposed by **do** A **watching** a in $[r, \infty)$.
- A is executed in the interval $[0, r]$; a is not entailed in $[0, r]$ and a is entailed in some open interval starting at r. In this case the process A is aborted after instant r, and there are no restrictions imposed by **do** A **watching** a in (r, ∞).

The following definition captures these intuitions. If in any trace, a is not true until some time t, then the portion upto t has to be derived from a trace in A.

$$
\begin{aligned}
&\mathcal{H}[\![\textbf{do } A \textbf{ watching } a]\!] \stackrel{d}{=} \\
&\quad \{f \in \textbf{HObs} \mid \text{Let } T = \{t \in \mathbb{R}^+ \mid \forall r \in \mathtt{dom}(f), r \leq t \Rightarrow a \notin \pi_2(f(r))\}, \\
&\qquad\qquad\quad \text{then } f \restriction (T) \in \mathcal{H}[\![A]\!]\}
\end{aligned}
$$

The combinator can be defined using the time construct as follows:

$$
\begin{aligned}
\textbf{do } A \textbf{ watching } a = \textbf{new } \mathtt{stop}, \mathtt{go} \textbf{ in } (\, & \textbf{time } A \textbf{ on } \mathtt{go}, \\
& \textbf{always if } \mathtt{stop} \textbf{ else } \mathtt{go}, \\
& \textbf{always if } a \textbf{ then always } \mathtt{stop})
\end{aligned}
$$

Proofs are routine, and have been omitted. Analogous to watching, we can define another construct **do** A **while** a, which keeps A going only as long as

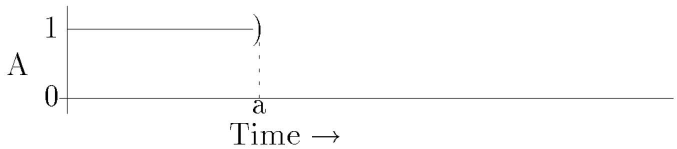

Fig. 6. Time diagram for **do** A **watching** a.

a holds.

Example 25 **do** A **trap** a is similar to **do** A **watching** a. Semantically, an observation of **do** A **trap** a arises in the following way: A is executed in the interval $[0, r]$; a is not entailed in $[0, r)$ and a is entailed at r. Process A is aborted after r, thus **do** A **trap** a imposes no restrictions on the behaviour of the system in (r, ∞). This construct allows A to respond to a before getting terminated — also a may be generated by A (unlike in **do** A **watching** a, where A generating the a could result in an indeterminate program). As in the **first**, the behavior of this construct relies on the existence of well–defined first occurrences of a. Formally,

$$\begin{aligned}
&\mathcal{H}[\![\mathbf{do}\ A\ \mathbf{trap}\ a]\!] \stackrel{d}{=} \\
&\quad \{f \in \mathcal{H}[\![A]\!] \mid (\forall r \in \mathtt{dom}(f)) a \notin \pi_2(f(r))\} \\
&\bigcup \{f \in \mathbf{HObs} \mid \exists r \in \mathtt{dom}(f), f\restriction[0,r] \in \mathcal{H}[\![A]\!], a \in \pi_2(f(r))\}
\end{aligned}$$

$$\begin{aligned}
\mathbf{do}\ A\ \mathbf{trap}\ a = \mathbf{new}\ \mathtt{stop}, \mathtt{go}\ \mathbf{in}\ (\ &\mathbf{time}\ A\ \mathbf{on}\ \mathtt{go}, \\
&\mathbf{always\ if}\ \mathtt{stop}\ \mathbf{else}\ \mathtt{go}, \\
&\mathbf{always\ if}\ a\ \mathbf{then\ hence}\ \mathtt{stop})
\end{aligned}$$

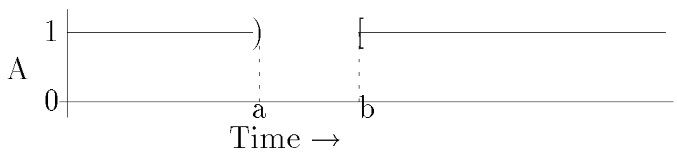

Fig. 7. Time diagram for $S_a A_b(A)$.

Example 26 Using multiform time, we can write a construct to suspend a process and later activate it. This is similar to the familiar ($\mathtt{control} - \mathtt{Z}, \mathtt{fg}$), and is illustrated in Figure 7. $\mathbf{S}_a\mathbf{A}_b(A)$ behaves like A until the first instant when a is entailed; when a is entailed A is suspended from then on (thus, the S_a). A is reactivated in the first time instant when b is entailed (thus, the A_b).

$$\begin{aligned}
\mathbf{S}_a\mathbf{A}_b(A) = \mathbf{new}\ \mathtt{stop}, \mathtt{go}\ \mathbf{in}\ (\ &\mathbf{time}\ A\ \mathbf{on}\ \mathtt{go}, \\
&\mathbf{always\ if}\ \mathtt{stop}\ \mathbf{else}\ \mathtt{go}, \\
&\mathbf{first}\ a\ \mathbf{then\ do}\ (\mathbf{always}\ \mathtt{stop})\ \mathbf{watching}\ b)
\end{aligned}$$

Again proper behavior of this construct depends on the existence of "first" occurrences of a, b. Repeated suspension–reactivation is done similarly:

$$\begin{aligned}\mathrm{RS}_a\mathbf{A}_b(A) = \mathbf{new}\ \mathtt{stop}, \mathtt{go}\ \mathbf{in}\ (\,&\mathbf{time}\ A\ \mathbf{on}\ \mathtt{go},\\ &\mathbf{always}\ \mathbf{if}\ \mathtt{stop}\ \mathbf{else}\ \mathtt{go},\\ &\mathbf{always}\ \mathbf{if}\ a\ \mathbf{then}\ \mathbf{do}\ (\mathbf{always}\ \mathtt{stop})\ \mathbf{watching}\ b)\end{aligned}$$

Thus, there is a general pattern of writing such constructs — one can always time A on go, and then precisely control the stop's and go's.

4.4 Input-output behavior.

Given a process Z, we can define its input-output behavior after a history f, with $\mathtt{dom}(f) = [0, r)$, as the i/o behavior of the Default cc process $(Z\ \mathbf{after}\ f)(0)$. Similarly, the i/o behavior after g with $\mathtt{dom}(g) = [0, s]$ is the i/o behavior of the Default cc process $(Z\ \mathbf{after}\ g)(0^+)$. This leads us to the definition of determinacy.

Definition 27 *A process Z is called determinate if*

$$\begin{aligned}(\forall f \in Z)\ [\,&\mathtt{dom}(f) = [0, r) \Rightarrow (Z\ \mathbf{after}\ f)(0)\ \textit{is determinate,}\\ &\mathtt{dom}(f) = [0, r] \Rightarrow (Z\ \mathbf{after}\ f)(0^+)\ \textit{is determinate.}]\end{aligned}$$

Similarly, the definition of X-determinate for Hybrid cc processes is obtained by extending the definition for Default cc — we again note that determinate processes are X-determinate for all variables X.

Definition 28 *A* Hybrid cc *process Z is X-determinate if for all f in Z*

- $(\forall t \in \mathtt{dom}(f))\ (Z\ \mathbf{after}\ f\restriction[0, t))(0)$ *is X-determinate*
- $(\forall t \in \mathtt{dom}(f))\ (Z\ \mathbf{after}\ f\restriction[0, t])(0^+)$ *is X-determinate*

Determinacy detection algorithm. The determinacy detection algorithm for Default cc lifts to a determinacy detection algorithm for Hybrid cc.

Given a Hybrid cc program P arising out of the syntax, we associate two sets of Default cc processes P_{inst} — the set of all possible instantaneous Default cc processes — and P_{cont} — the set of all possible Default cc processes causing continuous evolution — that can arise in the evolution of P. We provide finite and conservative estimates of these sets — thus reducing determinacy detection of Hybrid cc to determinacy detection of Default cc.

Process	P_{inst}	P_{cont}
c	$\{c, \texttt{true}\}$	$\{\texttt{true}\}$
if c **then** A	$\{\textbf{if } c \textbf{ then } P \mid P \in A_{inst}\} \cup A_{inst}$	$A_{cont} \cup \{\texttt{true}\}$
if c **else** A	$\{\textbf{if } c \textbf{ else } P \mid P \in A_{inst}\} \cup A_{inst}$	$A_{cont} \cup \{\texttt{true}\}$
(A_1, A_2)	$\{(P_1, P_2) \mid P_i \in A_{i(inst)}\}$	$\{(P_1, P_2) \mid P_i \in A_{i(cont)}\}$
new X **in** A	$\{\textbf{new } X \textbf{ in } P \mid P \in A_{inst}\}$	$\{\textbf{new } X \textbf{ in } P \mid P \in A_{cont}\}$
hence A	$\{(P_1, \ldots, P_n) \mid P_i \in A_{inst}\}$	$\{(P_1, \ldots, P_n) \mid P_i \in A_{inst} \cup A_{cont}\}$

where in the last case the P_i's are distinct.

The above test is conservative but not exact — an exact test can be obtained by compiling Hybrid cc programs to *finite* Hybrid automata [20], and checking determinacy of the Default cc programs in the states.

5 Operational Semantics and Correspondence Theorems

In this section, we describe how Hybrid cc is realized computationally. We describe a formal operational semantics and relate it to the denotational semantics presented above. The operational semantics forms the basis of our implementation, which we describe in the next section.

5.1 Operational semantics

The operational semantics for Hybrid cc is built on the operational semantics for Default cc.

We assume that the program is operating in isolation — interaction with the environment can be represented as an observation and run in parallel with the program. We use $\Gamma, \Delta, \ldots$ for multisets of programs; $\sigma(\Gamma)$ is defined as before — the tell tokens in Γ.

Configurations can be *point* or *interval* configurations. A point configuration is a Default cc program that is executed instantaneously (at a real time instant) — all discrete changes happen at point states. This execution results in three pieces of information: the token b on quiescence, a token of the form

$\texttt{init}(a)$ that is derived from b and is the initial condition for the subsequent interval state, and finally the "continuation" — the program to be executed at subsequent times.

Interval configurations are triples (a, Γ, Δ) and model continuous execution. a is the initial token, this is similar to the initial conditions in a differential equation. Γ consists of the programs active in the interval configuration. Computation progresses *only* through the (continuous) evolution of the store as captured by the passage of time. In particular, the interval state is exited as soon as the status of any of the conditionals changes — one which always fired does not fire anymore, or one starts firing. Δ accumulates the "continuation".

First, we describe transitions from point to interval configurations. $\longrightarrow$ is the transition relation of Default cc, and $\delta(\Gamma')$ is the sub-multiset of programs of the form **hence** A in Γ'.

$$\frac{\Gamma \longrightarrow \Gamma'}{\Gamma \xrightarrow{\texttt{hcc}} (\sigma(\Gamma')_{\texttt{init}}, \delta(\Gamma'), \emptyset)}$$

where $\sigma(\Gamma')_{\texttt{init}} = \{\texttt{init}(a) \mid \sigma(\Gamma') \vdash_0 \texttt{init}(a)\}$. The computability conditions on ccs guarantee that $\sigma(\Gamma')_{\texttt{init}}$ can be represented by a single token.

In the interval configuration (a, Γ, Δ) we are going to execute the program Γ "once", and make sure that the status of the conditionals remains constant throughout the interval $(0, r)$. Condition 12 on continuous constraint systems, and the finiteness of the multisets ensures the existence of such an $r > 0$. The conditions on $\exists_X$ in the ccs ensure that only *one* copy of the new variables needs to be made for the entire interval phase.

The derivation relation is indexed by the "guessed output constraint" b (used to evaluate defaults as in the operational semantics of Default cc) and the length of the interval r.

$$\frac{a, \texttt{cont}(\sigma(\Gamma)) \vdash^r a'}{(a, (\Gamma, \textbf{if } a' \textbf{ then } B), \Delta) \xrightarrow{\texttt{hcc}}_{b,r} (a, (\Gamma, B), \Delta)} (then)$$

$$\frac{a, \texttt{cont}(b) \not\vdash^r a'}{(a, (\Gamma, \textbf{if } a' \textbf{ else } B), \Delta) \xrightarrow{\texttt{hcc}}_{b,r} (a, (\Gamma, B), \Delta)} (else)$$

$$(a, (\Gamma, \textbf{hence } A), \Delta) \xrightarrow{\texttt{hcc}}_{b,r} (a, (\Gamma, A), (\Delta, A, \textbf{hence } A))(hence)$$

$$(a, (\Gamma, (A, B)), \Delta) \xrightarrow{\texttt{hcc}}_{b,r} (a, (\Gamma, A, B), \Delta) \quad (Par)$$

$$(a, (\Gamma, \textbf{new } X \textbf{ in } A), \Delta) \xrightarrow{\texttt{hcc}}_{b,r} (a, (\Gamma, A[Y/X]), \Delta) \quad (new)$$

$$(Y \text{ not free in } a, A, \Delta, \Gamma)$$

The transition from interval to point configurations, $\xrightarrow{\texttt{hcc}}$, is defined from $\xrightarrow{\texttt{hcc}}_{b,r}$ — verify that the guessed output constraint is achieved and verify that the residual conditionals were not enabled at any intermediate time.

$$\frac{\exists b \in ID \exists r > 0 \quad (a, \Gamma, \Delta) \xrightarrow{\mathtt{hcc}}{}^{*}_{b,r} (a, \Gamma', \Delta') \quad b = \sigma(\Gamma') \quad \Gamma' \downarrow_r^{b,b}}{(a, \Gamma, \Delta) \xrightarrow{\mathtt{hcc}} \Delta'}$$

The output for any time t in this phase is $\exists_{\vec{\mathbf{Y}}} f^t(a \wedge \mathtt{cont}(b))$, where $\vec{\mathbf{Y}}$ are the variables introduced by the operational semantics in this or a previous phase. $\Gamma' \downarrow_r^{c,d}$ verifies that the remaining conditionals in Γ' were not enabled at any time during the open interval $(0, r)$.

$$a \downarrow_r^{c,d}, \ \frac{\Gamma \downarrow_r^{c,d} \quad A \downarrow_r^{c,d}}{(\Gamma, A) \downarrow_r^{c,d}}, \ \frac{c \not\vdash^r a}{(\textbf{if } a \textbf{ then } A) \downarrow_r^{c,d}}, \ \frac{d \vdash^r a}{(\textbf{if } a \textbf{ else } A) \downarrow_r^{c,d}}$$

5.2 Correspondence theorems

The denotational semantics is correct for reasoning about the operational semantics. We first define $\mathcal{O}[\![P]\!]$, the operational semantics defined from the transition relations given above. The operational semantics is generalized to work over constraints, rather than tokens, in the usual way. The definition essentially states that each phase of each observation must arise from the transition relation, similar to the definition for Default cc.

Definition 29 *$\mathcal{O}[\![P]\!]$ consists of all observations $f \in \mathbf{HObs}$ of P. An observation f of P satisfies the following :*

(i) *If $\mathtt{dom}(f) = \emptyset$ i.e. $f = \epsilon$, then f is an observation of P.*
(ii) *If $0 \in \mathtt{dom}(f)$ then there is a $v \in |ID|$ such that*
- $P, \pi_1(f(0)) \longrightarrow_v^* P' \not\longrightarrow_v$. $\exists_{\vec{\mathbf{Y}}} \sigma(P') = \pi_1(f(0)), \exists_{\vec{\mathbf{Y}}} v = \pi_2(f(0))$.
- $P, \pi_2(f(0)) \longrightarrow_v^* Q \not\longrightarrow_v$. $\sigma(Q) = v$.

(iii) *If $\mathtt{dom}(f) \supset \{0\}$, let $a = \sigma(P')_{\mathtt{init}}$. Then there is a $v' \in |ID|$ such that:*
- $(a, (\delta(P'), \pi_1(f(0^+))), \phi) \xrightarrow{\mathtt{hcc}}_{v',r} (a, Q', P'')$ *with* $\exists_{\vec{\mathbf{Y}'}} \sigma(Q') = \pi_1(f(0^+))$ *and* $\exists_{\vec{\mathbf{Y}'}} v' = \pi_2(f(0^+))$ *and* $Q' \downarrow_r^{\sigma(Q'),v'}$.
- $(a, (\delta(P'), \pi_2(f(0^+))), \phi) \xrightarrow{\mathtt{hcc}}_{v',r} (a, Q'', P''')$ *with* $\sigma(Q'') = v'$ *and* $Q'' \downarrow_r^{v',v'}$.
- $\pi_1 \circ f \restriction [0, r)$ *and* $\pi_2 \circ f \restriction [0, r)$ *are traces of the ccs.*
- $f \restriction [r, \infty)$ *is an observation of* **new** $\vec{\mathbf{Y}'}$ **in** P''.

Lemma 30

$$\begin{aligned}
\mathcal{O}[\![a]\!] &= \{f \in \mathcal{H}[\![a]\!] \mid f \text{ has finitely many phases}\} \\
\mathcal{O}[\![A, B]\!] &= \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!] \\
\mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!] &= \{\epsilon\} \cup \{f \in \mathbf{HObs} \mid f(0) = (u, v), a \in u \Rightarrow f \in \mathcal{O}[\![A]\!], \\
&\qquad a \in v \Rightarrow (v, v) \in \mathcal{O}[\![A]\!]\}
\end{aligned}$$

$$\mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] = \{\epsilon\} \cup \{f \in \textbf{HObs} \mid f(0) = (u, v), a \notin v \Rightarrow f \in \mathcal{O}[\![A]\!]\}$$

$$\mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] = \textbf{new}_X \mathcal{O}[\![A]\!] \textit{ if } \mathcal{O}[\![A]\!] \textit{ is } X\textit{-determinate}$$

$$\mathcal{O}[\![\textbf{hence } A]\!] = \{f \in \{f \in \textbf{HObs} \mid (\forall t > 0) f \restriction [t, \infty) \in \mathcal{O}[\![A]\!], \; f \textit{ has finitely many phases}\}$$

PROOF. A proof sketch is provided in the appendix B.

The proof proceeds by induction on the number of phases in the observation f. It involves a proof of correctness of the transition system; (1) in point (resp. interval) phases, the transition system captures the correct observations, and (2) the transition system passes the correct "continuation" to the succeeding interval (resp. point) phase.

Now we can show the following theorem:-

Theorem 31 *If P is a* Hybrid cc *program which satisfies the X-determinacy condition for all X and f is a* Hybrid cc *observation with finitely many phases, then $f \in \mathcal{H}[\![P]\!]$ iff $f \in \mathcal{O}[\![P]\!]$.*

PROOF. A simple structural induction using the above lemma 30 yields the required result.

Note that this theorem cannot give us full abstraction because of Zeno processes. For example, the above operational semantics run on a Hybrid cc program of a typical Zeno system — say, a bouncing ball, with a coefficient of restitution $e < 1$ — does not progress to and beyond the finite real limit point. However, the denotational semantics can potentially have further information — it can have observations with infinitely many phases, which can never be observed by the operational semantics. In the parlance of programming language theory [38,43,37], this means that the denotational semantics is adequate (sound for reasoning about programs) but not fully abstract (sound and complete for reasoning about programs). Note however that if $\mathcal{O}[\![A]\!] \neq \mathcal{O}[\![B]\!]$, then there is a context P such that the output of running A, P is different from the output of B, P — this proof is the same as the proof for Default TCC [45], and is omitted.

6 An interpreter for Hybrid cc

We present a prototype implementation of a concrete language in the Hybrid cc family. The implementation is based on the operational semantics described above — however, at this point, it does not perform "determinacy checks" on the program. In particular, if there are multiple possible outputs, only one is output by our current implementation (if there is no input, an error is reported). Ideally we would like to output all, or report an indeterminacy, but at this point, this has not been implemented. It has been used to run several programs [19]. The implementation is on top of Sicstus Prolog.

The maintain combinator. Hybrid cc allows the transmission of information from point to the succeeding interval phases via the constraints of form $\mathtt{init}(\cdot)$. But, there is no mechanism to transmit information from interval to the succeeding point phases. In some programming examples, we have found it convenient to carry forward to $r+\epsilon$ the knowledge of the the constraints that hold in the right limit of $(r, r+\epsilon)$. The following definition captures this notion of transmitting information:

Definition 32 *f transmits information if $\forall r \in \mathtt{dom}(f), \forall \epsilon > 0$ such that $[r, r+\epsilon] \subseteq \mathtt{dom}(f)$ and $\pi_1 \circ f \restriction [r, r+\epsilon)$ and $\pi_2 \circ f \restriction [r, r+\epsilon)$ are traces of the ccs, we have for $i = 1, 2$ and for all tokens $a \in \mathtt{Base}_D$*

$$\int^{\epsilon} \pi_i(f(r)) \wedge \mathtt{cont}(\pi_i(f(r^+))) \vdash a \;\Rightarrow\; \pi_i(f(r+\epsilon)) \vdash \mathtt{prev}(a)$$

Now, we define a 0-ary combinator:

$$\mathtt{maintain} \;=\{f \in \mathbf{HObs} \mid f \text{ transmits information}\}$$

maintain is not definable using the other combinators of Hybrid cc. It satisfies the following laws:

$$\begin{aligned}&\mathtt{maintain} = \mathtt{maintain}, \mathtt{maintain}\\ &\mathtt{maintain} = \mathbf{hence}\ \mathtt{maintain}\end{aligned}$$

Combined with the equational laws discussed earlier, the above laws ensure that a single top level occurrence of the combinator maintain suffices for variables that are not existentially quantified. For the programs we write in the concrete language, we leave this single top level occurrence of the combinator implicit.

The continuous constraint system. The continuous constraint system we have built for our implementation separates the set of constraints that can be added to the store (the *tell* constraints) from the set whose validity can be inferred from the store (the *ask* constraints). The set of *ask* constraints is a proper superset of the set of *tell constraints*. This has been done to simplify the implementation, while retaining as much expressivity as possible. Also, our system is not complete with respect to the inferences that can be made — doing so would in fact entail solving arbitrary non-linear programming problems.

The tell constraints can be of three different kinds:

- Atoms, which are uninterpreted and simply added to the store — these are `Gentzen` style signals (see example 4).
- Constraints involving continuously varying variables like `dot(x,2) = 7` — we only allow one expression of the kind `dot(x,` n`)` on the left, and only expressions evaluating to constants on the right.
- Some other expressions like inequalities, equations etc.

Ask constraints include more general expressions, including multiplication etc. Complete details are given in the documentation that comes with the implementation.

The entailment relations are based on the entailment relations of example 17; augmented to allow for `Gentzen` style signals/constraints. In addition, the constraint system enforces a family of rules of the form

$$\mathtt{dot}(\mathtt{x},1) = \mathtt{r}, \mathtt{prev}(\mathtt{x} = \mathtt{s}) \vdash \mathtt{x} = \mathtt{s}$$

$$a \vdash \mathtt{init}(a)$$

The first set of rules enforces the semantic content of existence of derivatives of real–valued functions — namely, left limit equals the value of the function at the point. The second set makes every variable right-continuous, this is implemented for convenience. Consider the following example. (contrast with example 21).

Example 33 Consider the program $P = \mathtt{maintain}, \mathtt{x} = 0, \mathbf{hence}\ (\mathtt{dot}(\mathtt{x},1) = 1)$. The denotation of P consists of all observations f that satisfy: $\pi_1(f(0)) \vdash \mathtt{x} = 0,\ (\forall r \in \mathtt{dom}(f))\ [\pi_1(f(r)) \vdash \mathtt{dot}(\mathtt{x},1) = 1 \wedge \mathtt{x} = \mathtt{r}]$.

The control constructs module. The interpreter works alternately in point phases and interval phases. It initially starts in a point phase with time $t = 0$. We briefly present some implementation details.

The point rules. These are the Default cc rules. Any constraint is added to the store, and also wakes up any suspended asks. If a constraint c is determined to be valid by the constraint system, **if** c **then** A reduces to A as specified by the **then** rule. Otherwise it is added to the list of suspended asks. An else statement is always suspended initially, when there are no more active agents to process, it is processed as described below. To process **new** X **in** A, every occurrence of X in A is replaced by an internally generated symbol. Parallel composition is done by flattening lists of agents.

After no more active agents are left (other than **hence** A agents), the interpreter deals with the elses. For **if** a **else** A, there are two possibilities. Either a will be entailed by the final store and the agent stays suspended (in practice we delete it), or it will not be entailed, in which case A is added to the list of agents. The interpreter systematically searches these choices for each else, backtracking if a set of choices does not lead to a store that is consistent with the choices. As soon as a consistent store is obtained it is output as the output of the point phase.

The `init` constraint entailed by the store is now passed to the interval phase along with the **hence** agents.

The interval rules. In the interval phase, a Default cc program is once again executed to obtain the store. At the same time, the length of the interval phase is determined as the longest interval during which the status of the asks in the Default cc program does not change. Since we want to have the largest possible interval to make the maximum amount of progress, we initially assume that the length of the interval will be infinity. Each rule is then implemented as follows.

If the interval store, with initial conditions from the previous point store entails a, then **if** a **then** B reduces to B. In addition, the constraint system computes an r for which this deduction is valid. The new interval length is the minimum of this r and the previous length. **if** a **else** B is implemented similar to the point rules. The constraint system provides the r for which a constraint will not be entailed — this is the smallest r at which it will be entailed. For **new** X **in** A, once again we replace the X by a new internally generated symbol. For **hence** A, both A and **hence** A are added to the list of agents for the next point phase. Parallel composition is implemented as a flattening of lists.

Finally, the transition is made to the next point phase. Any suspended **then**'s or **else**'s are now examined for further limiting the extent of the interval phase (in the interpreter, this process is carried on simultaneously with the above steps). The limiting store at the end of the interval phase is passed as a `prev` constraint to the point phase along with the list of agents. Formally, this means that the transition rule from interval to point states of the operational semantics is changed to (notation is the same as in the operational semantics section):

$$\frac{\exists B \in ID \exists r > 0 \quad (a, \Gamma, \Delta) \xrightarrow{\texttt{hcc}}{}^{*}_{b,r} (a, \Gamma', \Delta') \quad b = \sigma(\Gamma') \quad \Gamma' \downarrow_r^{b,b}}{(a, \Gamma, \Delta) \xrightarrow{\texttt{hcc}} [\int^r a \wedge \texttt{cont}(b)]_{\texttt{prev}}, \Delta'}$$

where $a_{\texttt{prev}} = \{\texttt{prev}(b) \mid b \in \texttt{Base}_D, a \vdash b\}$. This change reflects the effect of the combinator `maintain`.

Procedure calls and recursion. We have implemented a simple syntax for procedure calls in the interpreter — a call to $p(A, B, C)$ is replaced by the body of $p(X, Y, Z)$, with a textual substitution of the variables. We can implement recursive calls this way — note the recursion for parameter free procedures can be written in the basic syntax using **hence** .

7 Hybrid cc— Programming Examples

We now present a couple of programs to illustrate programming in Hybrid cc.

7.1 *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.

```
furnace_heat(HeatR)::
    time (always {dot(temp) = HeatR}) on furnace_on.
```

The cooling of the furnace is modeled similarly.

```
furnace_cool(CoolR)::
    time (always {dot(temp) = -CoolR}) on furnace_off.
```

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

```
furnace(HeatR, CoolR) :: furnace_heat(HeatR), furnace_cool(CoolR).
```

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.

```
controller(Cut_out, Cut_in)::
    always [if switch_on then
                (do (always {furnace_on}) watching switch_off),
            if switch_off then
                (do (always {furnace_off}) watching switch_on)],
    always [if (prev(temp) = Cut_out) then {switch_off},
            if (prev(temp) = Cut_in) then {switch_on}].
```

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

```
controlled_furnace(HeatR, CoolR, Cut_out, Cut_in, Init_temp)::
    {switch_on}, {temp = Init_temp},
    furnace(HeatR, CoolR),
    controller(Cut_out, Cut_in).
```

The trace of this program with initial temparature 26 and parameters `HeatR = 2, CoolR = -0.5`, `Cut_Out = 30, Cut_in = 26`, as executed by our interpreter is seen below. Since the program runs forever, we aborted it after some time. 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, time is always measured from the beginning of the interval, not from 0.

```
| ?-  hcc(controlled_furnace(2, 0.5, 30, 26, 26)).

Time = 0.
Atoms  :  [furnace_on,switch_on]
Variables  :  [dot(temp,1)=2,temp=26]

Time Interval is (0, 2.0)
Atoms  :  [furnace_on]
Variables  :  [temp=2*t+26]
```

```
Time = 2.0.
Atoms  :  [furnace_off,switch_off]
Variables  :  [dot(temp,1)= -0.5,temp=30.0]

Time Interval is (2.0, 10.0)
Atoms  :  [furnace_off]
Variables  :  [temp= -0.5*t+30.0]

Time = 10.0.
Atoms  :  [furnace_on,switch_on]
Variables  :  [dot(temp,1)=2,temp=26.0]

Time Interval is (10.0, 12.0)
Atoms  :  [furnace_on]
Variables  :  [temp=2*t+26.0]
```

7.2 A billiards table simulation

We present a program modeling a simple billiards(pool) table with several balls. The table initially has several balls on it, with various velocities. The balls keep rolling in a straight line until they hit another ball or hit the edge of the table, or they come to a halt due to friction.

Each ball is modeled by an agent `ball(B, PosX, PosY, VelX, VelY)`, where `B` is the name of the ball, `(PosX, PosY)` is the initial position and `(VelX, VelY)` is the initial velocity. The initial configuration also gives the dimensions of the table — `(xMax, yMax)`, the radius of the balls, the size of the pockets and the deceleration due to friction. It activates the agents for computing edge collisions, ball-ball collisions and pocketing.

```
initial_config::
    always {xMax = 150, yMax = 300, radius = 3, pocket = 7, fric = 1},
    ball(b1, 10, 10, 25, 25),
    ball(b2, 20, 11, -35, 55),
    ball(b3, 80, 51, -15, 49),
    edge_collision,
    two_ball_collision,
    pocketing.
```

The ball agent sets up the initial position and velocity. It also computes the direction of motion of the ball, and records it as $\cos^2\theta$, where θ is the direction of movement. This is used to compute the effect of friction on each of the two components of the velocity. The constraint `ball(B)` asserts that B is a ball. The constraint `change(B)` is used to assert that ball B has had a collision. If there is no collision, then the ball continues rolling in its original direction, and its velocity decreases according to the deceleration due to friction. If there is a collision, the new direction is computed, and change in position is linked to the new velocity. The exception handler `do ... watching pocketed(B)` allows us to handle pocketing of balls. `a:b` is an uninterpreted symbol, it is used here as a pairing construct.

```
ball(B, PosX, PosY, VelX, VelY)::
    {(B:pos:x) = PosX},{(B:pos:y) = PosY},
    {(B:vel:x) = VelX}, {(B:vel:y) = VelY},
    {dot(B:pos:x) = VelX}, {dot(B:pos:y) = VelY},
    {(B:direction) = VelX*VelX/(VelX*VelX + VelY*VelY)},
    always {ball(B)},
    do hence [if change(B) else roll_ball(B),
              if change(B) then
                  [{dot(B:pos:x) = (B:vel:x)},
                   {dot(B:pos:y) = (B:vel:y)},
                   {(B:direction) = (B:vel:x)*(B:vel:x)/
                                 ((B:vel:y)*(B:vel:y)+
                                  (B:vel:x)*(B:vel:x))}]]
     watching pocketed(B).

roll_ball(B)::
    {dot(B:direction) = 0},
    {dot(B:pos:x) = (B:vel:x)},
    {dot(B:pos:y) = (B:vel:y)},
    if (prev(B:vel:x) > 0) then
        {dot(B:vel:x) = -fric*sqrt(B:direction)},
    if (prev(B:vel:x) < 0) then
        {dot(B:vel:x) = fric*sqrt(B:direction)},
    if (prev(B:vel:x) > 0; prev(B:vel:x) < 0) else
        {dot(B:vel:x) = 0},
    if (prev(B:vel:y) > 0) then
        {dot(B:vel:y) = -fric*sqrt(1 - (B:direction))},
    if (prev(B:vel:y) < 0) then
        {dot(B:vel:y) = fric*sqrt(1 - (B:direction))},
    if (prev(B:vel:y) > 0;prev(B:vel:y) < 0) else
        {dot(B:vel:y) = 0}]
```

The edge collision agent just keeps checking if any ball has hit the edge — i.e. its distance from the edge is its radius. In that case, it reverses the appropriate component of the velocity, while retaining the other. The construct **forall** C **do** A is used here as a shorthand for parallel composition — for each instance of C, it creates an instance of A to be run in parallel with all the other instances.

```
edge_collision::
    always [forall ball(B) do
```

```
[if (prev(B:pos:x) = radius;
     prev(B:pos:x) = xMax - radius) then
   ({change(B)}, {(B:vel:x) = -prev(B:vel:x)},
    {(B:vel:y) = prev(B:vel:y)}),
 if (prev(B:pos:y) = radius;
     prev(B:pos:y) = yMax - radius) then
   ({change(B)}, {(B:vel:y) = -prev(B:vel:y)},
    {(B:vel:x) = prev(B:vel:x)})]].
```

The edge collision agent computes for each pair of balls if they are currently undergoing a collision, i.e. the distance between their positions is twice the radius. The ask `(B1 @< B2)` uses the Prolog term ordering to make sure that this work is done only once for each pair of different balls. Now there are two cases. If the balls have the same x-coordinate, they simply exchange their y velocities. Otherwise, we compute $c = \tan\theta$, the direction of the line joining their centers (the first case was necessary to deal with $\theta = \pi/2$). Now by solving the equations of conservation of energy and momentum, we get the x-component of the impulse transmitted between them. The y-component is obtained by multiplying with c, since the transfer of impulse is along the radii (we assume frictionless collisions). This gives us the required velocities. Note that in true declarative programming, here we would have just stated the laws of conservation of energy and momentum. However, our constraint solver currently does not allow us to solve quadratic equations in the constraints, so we provided the solutions ourselves.

```
two_ball_collision::
    always [forall ball(B1) do forall ball(B2) do
        if (B1 @< B2) then
            if ((prev(B1:pos:x) - prev(B2:pos:x))*
                 (prev(B1:pos:x) - prev(B2:pos:x))
               +(prev(B1:pos:y) - prev(B2:pos:y))*
                 (prev(B1:pos:y) - prev(B2:pos:y)) = 4*radius*radius)
        then [{change(B1)}, {change(B2)}, compute_velocity(B1,B2)]].

compute_velocity(B1,B2)::
    if (prev(B1:pos:x) = prev(B2:pos:x)) then
        [{(B1:vel:x) = prev(B1:vel:x)},
         {(B2:vel:x) = prev(B2:vel:x)},
         {(B1:vel:y) = -prev(B2:vel:y)},
         {(B2:vel:y) = -prev(B1:vel:y)}],
    if (prev(B1:pos:x) = prev(B2:pos:x)) else
        [new c in new ix in
         [{c = (prev(B1:pos:y) - prev(B2:pos:y))/
                  (prev(B1:pos:x) - prev(B2:pos:x))},
          {ix = (prev(B2:vel:x)-prev(B1:vel:x)+
                  c*(prev(B2:vel:y)-prev(B1:vel:y)))/(1+c*c)},
          {(B1:vel:x) = prev(B1:vel:x) + ix},
```

```
          {(B2:vel:x) = prev(B2:vel:x) - ix},
          {(B1:vel:y) = prev(B1:vel:y) + c*ix},
          {(B2:vel:y) = prev(B2:vel:y) - c*ix}]].
```

Finally, the pocketing agent determines when any ball is going to fall into a pocket.

```
pocketing::
  hence [{halfyMax = yMax/2},
    forall ball(B) do
      if (prev(B:pos:x)*prev(B:pos:x)  %(0,0)
          +prev(B:pos:y)*prev(B:pos:y)
            = pocket*pocket;
          prev(B:pos:x)*prev(B:pos:x)  %(0,yMax/2)
          +(prev(B:pos:y)-halfyMax)*(prev(B:pos:y)-halfyMax)
            = pocket*pocket;
          prev(B:pos:x)*prev(B:pos:x)  %(0,yMax)
          +(prev(B:pos:y)-yMax)*(prev(B:pos:y)-yMax)
            = pocket*pocket;
          (prev(B:pos:x) - xMax)*(prev(B:pos:x)-xMax)  %(xMax,0)
          +prev(B:pos:y)*prev(B:pos:y)
            = pocket*pocket;
          (prev(B:pos:x)-xMax)*(prev(B:pos:x)-xMax)  %(xMax,yMax/2)
          +(prev(B:pos:y)-halfyMax)*(prev(B:pos:y)-halfyMax)
            = pocket*pocket;
          (prev(B:pos:x)-xMax)*(prev(B:pos:x)-xMax)  %(xMax,yMax)
          +(prev(B:pos:y)-yMax)*(prev(B:pos:y)-yMax)
            = pocket*pocket)
        then {pocketed(B)}].
```

We omit the trace of this program for brevity.

8 Work in progress/What remains to be done?

Modeling Physical systems. We have started our first major effort at modeling a real physical system in Hybrid cc [21]. In that paper we developed a compositional model of the simple photocopier paperpath described in the introduction. The model consists of about 130 lines of Hybrid cc code. Each transportation element (belt, roller etc.) is modeled by an agent which describes the effects of the various external forces on this component. A sheet of paper is modeled by a separate agent. Each sheet is under the influence of several transportation elements, and is consequently partitioned into segments. These segments, which are dynamically created and destroyed, are modeled by agents, transmit forces from one end to another, and compute the state of the sheet — buckled, straight, etc. Interaction processes are set up to make the segments interact with the transportation elements, in much the same way as the ball collision processes described above.

The fact that Hybrid cc is an appropriate modeling framework for such systems is made quite apparent by this model — the model just consists of all these elements running in parallel. The declarative nature of Hybrid cc is particularly useful, as for each model fragment, we simply stated the laws of physics applicable — equilibrium laws, boundary conditions etc. The constructs of Hybrid cc were used to create and destroy segments dynamically, without the code for each segment being aware of the creation or destruction.

Hybrid cc, Hybrid automaton and program verification. Automated verification procedures of finite state systems in general, and of hybrid systems in particular, operate on automata based representations of programs and models. Thus, it is desirable to have a compilation of the abstract higher level representations provided by Hybrid cc into an automaton form. In addition, this would establish Hybrid cc as a high-level programming notation for hybrid automata — much as synchronous programming languages are high level notation for discrete automata.

In [20], for any Hybrid cc program, we build an automaton whose valid runs are precisely execution traces of the program. This makes Hybrid cc amenable to extant automatic verification tools such as HyTech [28,2]. We intend to explore reasoning about physical systems such as the paperpath model above using these verification tools in combination with the automaton compilation algorithm.

Types for Hybrid cc. We intend to investigate types for Hybrid cc. Our motivation to type Hybrid cc programs arises from the potential of using type checking to restrict the use of program combinators. For a concrete example, recall the combinator **first** a **then** A used in programs earlier — this program reduces to A at the first time instant that a becomes true — if there is a well-defined notion of first occurrence of a. Now the denseness of the reals allows occurrences of a to exist without a "first" occurrence — for example, when a occurs in the interval $(0, t)$. A type system can be used to encode this assumption on occurrences of the event a in the type of the program.

References

[1] R. Alur and T. Henzinger. Logics and models of real time: a survey. In de Bakker et al. [14].

[2] R. Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *14th Annual IEEE Real-time Systems Symposium*, 1993.

[3] A. Benveniste and G. Berry, editors. Another Look at Real-time Systems. Special issue of *Proceedings of the IEEE*, September 1991.

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In Benveniste and Berry[3].

[5] A. Benveniste, M. Le Borgne, and Paul Le Guernic. Hybrid systems: The signal approach. In Grossman et al. [17].

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

[7] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11 – 17. Elsevier Science Publishers B.V. (North Holland), 1989.

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

[9] G. Berry and G. Gonthier. Incremental development of an HDLC entity in ESTEREL. *Computer Networks and ISDN Systems*, 22:35–49, 1991.

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

[11] G. Berry, S. Ramesh, and R.K. Shyamsunder. Communicating reactive processes. In *Proceedings of Twentieth ACM Symposium on Principles of Programming Languages*, pages 85 – 98, 1993.

[12] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In Benveniste and Berry [3].

[13] E. Coste-Maniere. Utilisation d'esterel dans un contexte ansynchrone: une application robotique. Technical report, INRIA, December 1989.

[14] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *REX workshop "Real time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[15] 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.

[16] C. Elliott, G. Schechter, R. Young, and S. Abi-Ezzi. Tbag: A high level framework for interactive, animated 3d graphics application. In *Proceedings of the ACM SIGGRAPH conference*, 1994.

[17] Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[18] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. In Benveniste and Berry [3].

[19] V. Gupta, R. Jagadeesan, V. A. Saraswat, and D. Bobrow. Programming in hybrid concurrent 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, 1995.

[20] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Hybrid cc, hybrid automata and program verification. In Alur, Henzinger, and Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science. Springer Verlag, 1996. To appear.

[21] 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.

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

[23] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In Benveniste and Berry [3].

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

[25] D. Harel and A. Pnueli. *Logics and Models of Concurrent Systems*, volume 13, chapter On the development of reactive systems, pages 471–498. NATO Advanced Study Institute, 1985.

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

[27] T. A. Henzinger, Z. Manna, and A. Pnueli. Towards refining temporal specifications into hybrid systems. In Grossman et al. [17].

[28] T.A. Henzinger and P.-H. Ho. Hytech : The cornell hybrid technology tool. In Grossman et al. [17].

[29] J. Hooman. A compositional approach to the design of hybrid systems. In Grossman et al. [17].

[30] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Proceedings of the Annual conference on Computer aided verification*, Lecture Notes in Computer Science, 1995.

[31] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Formal Methods in System Design*, 8(2):123–152, March 1996.

[32] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully-abstract semantics for a first order functional language with logic variables. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

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

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

[35] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In de Bakker et al. [14].

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

[37] A. Meyer. Semantical paradigms. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, 1988.

[38] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–23, 1977.

[39] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[40] G. J. Murakami and R Sethi. Terminal call processing in ESTEREL. Technical report, AT&T Bell Laboratories, January 1990. Abridged version appeared as *Parallelism as a Structuring Technique: Call Processing using the* ESTEREL *Language* in IFIP Transactions in Computer Science and Technology, 1990.

[41] A. Nerode and W. Kohn. Multiple agent hybrid control architecture. In Grossman et al. [17].

[42] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In de Bakker et al. [14].

[43] G. Plotkin. LCF considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[44] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13, 1980.

[45] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation.* To appear. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[46] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Programming in timed concurrent constraint languages. In B.Mayoh, E.Tougu, and J.Penjam, editors, *Constraint Programming*, volume 131 of *NATO Advanced Science Institute Series F: Computer and System Sciences*, pages 367–413. Springer-Verlag, 1994.

[47] 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.

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

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

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

A Proof of Default cc Lemma

Lemma 34 *The relation $\longrightarrow_v$ satisfies:*

Confluence: *If $A \longrightarrow_v^* A'$ and $A \longrightarrow_v^* A''$ and there is no clash in variable names between the two derivations then there is a B such that $A' \longrightarrow_v^* B$ and $A'' \longrightarrow_v^* B$.*

Monotonicity: $A \longrightarrow_v^* A' \Rightarrow\ A, B \longrightarrow_v^* A', B$.

Extensivity: $A \longrightarrow_v^* A' \Rightarrow\ \sigma(A') \supseteq \sigma(A)$.

Idempotence: $A \longrightarrow_v^* A' \not\longrightarrow_v \Rightarrow\ A, \exists_{\vec{\mathbf{Y}}}\sigma(A') \longrightarrow_v^* A' \not\longrightarrow_v$, *where $\vec{\mathbf{Y}}$ are the new variables introduced in the derivation.*

Thus, the relation $\longrightarrow_v$ satisfies the characteristic properties of the transition relation of cc languages.

Lemma 35 *For all* Default cc *programs A and B,*

$$\begin{aligned}
\mathcal{O}[\![a]\!] &= \mathcal{P}[\![a]\!]\\
\mathcal{O}[\![A, B]\!] &= \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!]\\
\mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!] &= \{(u,v) \in \mathbf{DObs} \mid\ a \in v \Rightarrow (v,v) \in \mathcal{O}[\![A]\!],\\
&\qquad\qquad\qquad\qquad\quad a \in u \Rightarrow (u,v) \in \mathcal{O}[\![A]\!]\}\\
\mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] &= \{(u,v) \in \mathbf{DObs} \mid a \notin v \Rightarrow (u,v) \in \mathcal{O}[\![A]\!]\}\\
\mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] &= \mathbf{new}_X \mathcal{O}[\![A]\!],\ \textit{if } \mathcal{O}[\![A]\!]\ X - \textit{determinate}
\end{aligned}$$

PROOF. As indicated by lemma 34, the proof of the lemma follows extant proofs for languages in the cc paradigm. Detailed proofs for all cases can be found in [45]. Here, we merely sketch a couple of cases to expose the flavor of the proof and some salient features of Default cc.

The case for parallel composition is described to show the similarity of the proof to the proof for cc languages.

- Let $(u,v) \in \mathcal{O}[\![A_1, A_2]\!]$. Then $u, A_1, A_2 \longrightarrow^*_{v'} B \not\longrightarrow_{v'}$, and $\exists_{\vec{\mathbf{Y}}}\sigma(B) = u$. We show that $(u,v) \in \mathcal{O}[\![A_1]\!]$. Let $u, A_1 \longrightarrow^*_{v'} A_1' \not\longrightarrow_{v'}$. Using lemma 34, $u \subseteq \sigma(A_1')$. Using lemma 34 again, $u, A_1, A_2 \longrightarrow^*_{v'} A_1', A_2$. Using confluence of $\longrightarrow_{v'}$, $A_1', A_2 \longrightarrow^*_{v'} B$. Using lemma 34, $\sigma(A_1') \subseteq \sigma(B)$. Thus $\exists_{\vec{\mathbf{Y}}}\sigma(A_1') = u$ and we can deduce $(u,v) \in \mathcal{O}[\![A_1]\!]$. A symmetric argument shows that $(u,v) \in \mathcal{O}[\![A_2]\!]$. Thus, $\mathcal{O}[\![A_1, A_2]\!] \subseteq \mathcal{O}[\![A_1]\!] \cap \mathcal{O}[\![A_2]\!]$.
- Let $(u,v) \in \mathcal{O}[\![A_1]\!] \cap \mathcal{O}[\![A_2]\!]$. Then, $u, A_1 \longrightarrow^*_{v'} A_1' \not\longrightarrow_{v'}$, and $\exists_{\vec{\mathbf{Y}}}\sigma(A_1') = u$; also, $u, A_2 \longrightarrow^*_{v'} A_2' \not\longrightarrow_{v'}$, and $\exists_{\vec{\mathbf{Y}}}\sigma(A_2') = u$. Note that in general $\longrightarrow_v$ satisfies:
$$A_1' \not\longrightarrow_v, A_2' \not\longrightarrow_v, \sigma(A_1') = \sigma(A_2') \Rightarrow A_1', A_2' \not\longrightarrow_v$$
Thus, we deduce that $u, A_1, A_2 \longrightarrow^*_{v'} A_1', A_2' \not\longrightarrow_{v'}$. Since $\exists_{\vec{\mathbf{Y}}}\sigma(A_1', A_2') = u$ (the new variables in the two derivations are disjoint), $(u,v) \in \mathcal{O}[\![A_1, A_2]\!]$. Thus, $\mathcal{O}[\![A_1, A_2]\!] \supseteq \mathcal{O}[\![A_1]\!] \cap \mathcal{O}[\![A_2]\!]$.

Next, we handle the key elements of the proof for the case of new variables. We reproduce the definition of the denotation for new variables for convenience. We are simplifying the definition by using the fact that $\mathcal{O}[\![A]\!]$ is X–determinate.$\mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] = \mathbf{new}_X \mathcal{O}[\![A]\!]$, where

- Define $Z_1 \stackrel{d}{=} \bigcup\{(Z_v, v) \subseteq Z \mid \forall u \in Z_v, u \supseteq \exists_X v \Rightarrow u = v\}$.
- $\mathbf{new}_X Z \stackrel{d}{=} \bigcup\{(S,v) \subseteq \mathbf{DObs} \mid \exists v'[(v',v') \in Z_1, \exists_X v = \exists_X v', \exists_X S = \exists_X Z_{v'}]\}$.

Below, we sketch the proof that $\mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] \subseteq \mathbf{new}_X \mathcal{O}[\![A]\!]$. The key case of the proof is to show that $(v,v) \in \mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$ implies that $(v,v) \in \mathbf{new}_X \mathcal{O}[\![A]\!]$.

Let $(v,v) \in \mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$. Then, $\exists v'$ such that $\mathbf{new}\ X\ \mathbf{in}\ A, v \longrightarrow^*_{v'} A'' \not\longrightarrow_{v'}, v = \exists_{newX}\exists_{\vec{\mathbf{Y}}} v', v' = \sigma(A'')$, where $newX$ is the new variable introduced for X, and $\vec{\mathbf{Y}}$ are the *other* new variables (apart from $newX$) introduced in the derivation.

Let $v'' = (\exists_{\vec{\mathbf{Y}}} v')[X/newX]$. Since $\exists_X v = \exists_X v''$, it suffices to show:

- $(v'', v'') \in \mathcal{O}[\![A]\!]$. This is a simple fact about renaming, and standard cc style proofs for the $\mathbf{new}\ X\ \mathbf{in}\ \ldots$ combinator show that $(v'', v'') \in \mathcal{O}[\![A]\!]$.

– $(\forall (w, v'') \in \mathcal{O}[\![A]\!])\,, w \supseteq \exists_X v'' \Rightarrow w = v''$. We note that from the monotonicity of the $\longrightarrow$. relation, and the fact that $\exists_{\vec{\mathbf{Y}}} v'[X/newX] = v''$, it suffices to show:

$$(A, \exists_X v'') \longrightarrow^*_{v'[X/newX]} A''[X/newX] \not\longrightarrow_{v'[X/newX]}$$
$$v'[X/newX] = \sigma(A''[X/newX])$$

This follows from

$$(A[newX/X], \exists_X v'') \longrightarrow^*_{v'} A'' \not\longrightarrow_{v'}, v' = \sigma(A'')$$

which in turn follows from $(\mathbf{new}\ X\ \mathbf{in}\ A, v) \longrightarrow^*_{v'} A'' \not\longrightarrow_{v'}, v' = \sigma(A'')$ since the X part of the information in v could not have played any role in the derivation, and $\exists_X v = \exists_X v''$.

B Proofs of Hybrid cc Lemmas

The following lemma captures the essence of the evolution in the point phases in Hybrid cc — the first consequence says that the correct Default cc observation is captured in the point phase, and the next says that the correct "continuation" is passed to the interval phase.

Lemma 36 *Let $\mathcal{H}[\![\Gamma]\!]$ satisfy the X-determinacy condition for all variables X. Let $\Gamma \longrightarrow^*_b \Gamma' \not\longrightarrow_b$, with $\vec{\mathbf{Y}}$ being the new variables introduced by the transition system. Let f be such that* $\mathtt{dom}(f) = \{0\}, f(0) = (\sigma_{\Gamma'}, b)$. *Let $\exists_{\vec{\mathbf{Y}}} f$ stand for the function such that* $\mathtt{dom}(\exists_{\vec{\mathbf{Y}}} f) = \{0\}, \exists_{\vec{\mathbf{Y}}} f(0) = (\exists_{\vec{\mathbf{Y}}} \sigma_{\Gamma'}, \exists_{\vec{\mathbf{Y}}} b)$. *Then,*

– $\exists_{\vec{\mathbf{Y}}} f \in \mathcal{H}[\![\Gamma]\!]$
– *If $b = \sigma(\Gamma')$, $\mathcal{H}[\![\Gamma]\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f = \mathcal{H}[\![$**new** $\vec{\mathbf{Y}}$ **in** $\delta(\Gamma')]\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f$*

PROOF. The first proof is essentially a statement about Default cc — it follows by induction on the number of rules in $\Gamma \longrightarrow_b \Gamma'$.

The second proof follows from the following intermediate facts:

– If $b = \sigma(\Gamma')$, then $\mathcal{H}[\![\Gamma']\!]$ **after** $f = \mathcal{H}[\![\delta(\Gamma')]\!]$ **after** f. Proof is a routine structural induction on Γ'.
– If $b = \sigma(\Gamma')$, then $\mathcal{H}[\![\Gamma]\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f = \mathcal{H}[\![$**new** $\vec{\mathbf{Y}}$ **in** $\Gamma']\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f$. Proof follows by induction on the number of transitions in $\Gamma \longrightarrow^*_b \Gamma'$.

The next lemma is the interval counterpart of the previous lemma — the first consequence says that the correct observation is captured in the interval phase, and the next says that the correct "continuation" is passed to the point phase.

Lemma 37 *Let* $(a, \Gamma, \emptyset) \xrightarrow{\texttt{hcc}\,*}_{b,r} (a, \Gamma', \Delta') \not\xrightarrow{\texttt{hcc}}_{b,r}$. *Let* $\Gamma' \downarrow_r^{(\sigma(\Gamma'),b)}$. *Let* $\vec{\mathbf{Y}}$ *be the new variables introduced in the transitions. Let* $f \in \mathbf{HObs}$, $\mathtt{dom}(f) = [0, r)$ *be such that:*

$$\pi_i(f(0))_{\mathtt{init}} = a, i = 1, 2$$

$$(\forall 0 < t < r)\pi_i(f(t)) = \begin{cases} \int^t a \wedge \mathtt{cont}(\sigma(\Gamma')), & i = 1 \\ \int^t a \wedge \mathtt{cont}(b), & i = 2 \end{cases}$$

Let $\exists_{\vec{\mathbf{Y}}} f$ *stand for the function* $\mathtt{dom}(\exists_{\vec{\mathbf{Y}}} f) = \mathtt{dom}(f), (\forall 0 \leq t < r)$ $(\forall i = 1, 2)$ $[\pi_i(\exists_{\vec{\mathbf{Y}}} f(t)) = \exists_{\vec{\mathbf{Y}}} \pi_i(f(t))]$. *Then*

- $\exists_{\vec{\mathbf{Y}}} f \in \mathcal{H}[\![\Gamma]\!]$
- *If* $b = \sigma(\Gamma')$ $\mathcal{H}[\![\Gamma]\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f = \mathcal{H}[\![$**new** $\vec{\mathbf{Y}}$ **in** $\Delta']\!]$

PROOF. The proof has very similar structure to the proof of the previous lemma 36. This is because execution in the interval phase is in effect achieved by execution of a Default cc program.

For the first item, first note that a structural induction on Δ' shows that:

$$(\sigma(\Gamma'), b) \in \mathcal{H}[\![\Delta']\!](0^+)$$

An induction on the number of the rules in the transition from Γ to Γ' yields: $(\exists_{\vec{\mathbf{Y}}} \sigma(\Gamma'), \exists_{\vec{\mathbf{Y}}} b) \in \mathcal{H}[\![\Delta']\!](0^+)$. $\exists_{\vec{\mathbf{Y}}} f \in \mathcal{H}[\![\Gamma]\!]$ follows.

The second proof follows from the following intermediate facts:

- If $b = \sigma(\Gamma')$, then $\mathcal{H}[\![$**hence** $\Gamma']\!]$ **after** $f = \mathcal{H}[\![\Delta']\!]$. Proof is a routine structural induction on Γ'.
- If $b = \sigma(\Gamma')$, then $\mathcal{H}[\![\Gamma]\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f = \mathcal{H}[\![$**new** $\vec{\mathbf{Y}}$ **in hence** $\Gamma']\!]$ **after** $\exists_{\vec{\mathbf{Y}}} f$. Proof follows by induction on the number of rules in $\Gamma \longrightarrow_b \Gamma'$.

Lemma 38

$$\begin{aligned} \mathcal{O}[\![a]\!] &= \{f \in \mathcal{H}[\![a]\!] \mid f \textit{ has finitely many phases}\} \\ \mathcal{O}[\![A, B]\!] &= \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!] \\ \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!] &= \{\epsilon\} \cup \{f \in \mathbf{HObs} \mid f(0) = (u, v), a \in u \Rightarrow f \in \mathcal{O}[\![A]\!], \\ &\qquad a \in v \Rightarrow (v, v) \in \mathcal{O}[\![A]\!]\} \end{aligned}$$

$$
\begin{aligned}
\mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] &= \{\epsilon\} \cup \{f \in \mathbf{HObs} \mid f(0) = (u,v), a \notin v \Rightarrow f \in \mathcal{O}[\![A]\!]\} \\
\mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] &= \mathbf{new}_X \mathcal{O}[\![A]\!] \textit{ if } \mathcal{O}[\![A]\!] \textit{ is } X\textit{-determinate} \\
\mathcal{O}[\![\textbf{hence } A]\!] &= \{f \in \mathbf{HObs} \mid (\forall t > 0) f \upharpoonright [t,\infty) \in \mathcal{O}[\![A]\!], \\
&\qquad\qquad f \textit{ has finitely many phases}\}
\end{aligned}
$$

PROOF. We prove by induction on the number of phases of $f \in \mathbf{HObs}$ that for all programs A that satisfy the X–determinacy condition:

$$
\begin{aligned}
f \in \mathcal{O}[\![a]\!] &\Leftrightarrow f \in \mathcal{H}[\![a]\!] \\
f \in \mathcal{O}[\![A, B]\!] &\Leftrightarrow f \in \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!] \\
f \in \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!] &\Leftrightarrow f = \epsilon \text{ or } f(0) = (u,v), a \in u \Rightarrow f \in \mathcal{O}[\![A]\!], \\
&\qquad a \in v \Rightarrow (v,v) \in \mathcal{O}[\![A]\!] \\
f \in \mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] &\Leftrightarrow f = \epsilon \text{ or } f(0) = (u,v), a \notin v \Rightarrow f \in \mathcal{O}[\![A]\!] \\
f \in \mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] &\Leftrightarrow f \in \mathbf{new}_X \mathcal{O}[\![A]\!] \\
f \in \mathcal{O}[\![\textbf{hence } A]\!] &\Leftrightarrow (\forall t > 0) f \upharpoonright [t,\infty) \in \mathcal{O}[\![A]\!]
\end{aligned}
$$

For the combinators from Default cc, the second part of lemma 36, attesting to the correctness of the continuations, allows us to use the inductive hypothesis on the remainder of f.

For **hence** A, lemma 37 allows us to conclude that the operational and denotational semantics agree on the first interval phase of f. The second part of lemma 37, attesting to the correctness of the the continuations after the first interval phase of f, allows us to use the inductive hypothesis on the remainder of f.