Models for concurrent constraint programming

Vineet Gupta * Radha Jagadeesan ** Vijay Saraswat*

Abstract. Concurrent constraint programming is a simple but powerful framework for computation based on four basic computational ideas: *concurrency* (multiple agents are simultaneously active), *communication* (they interact via the monotonic accumulation of constraints on shared variables), *coordination* (the presence or absence of information can guard evolution of an agent), and *localization* (each agent has access to only a finite, though dynamically varying, number of variables, and can create new variables on the fly). Unlike other foundational models of concurrency such as CCS, CSP, Petri nets and the π-calculus, such flexibility is already made available within the context of *determinate* computation. This allows the development of a rich and tractable theory of concurrent processes within the context of which additional computational notion such as indeterminacy, reactivity, instantaneous interrupts and continuous (dense-time) autonomous evolution have been developed.
We survey the development of some of these extensions and the relationships between their semantic models.

1 Introduction

Concurrent Constraint Programming (CCP) arose as a generalization of work in dataflow and concurrent logic programming languages. The central idea is to view processes as independent agents communicating by imposing constraints on shared variables. Constraints are taken to be pieces of partial information — they *constrain* the values that variables can take rather than, as in imperative languages, *fixing* the value.

The importance of the idea of constraint-based communication for computation lies in the methodology of *compositional computing* that it enables. Hitherto, the only principled method for the construction of large computational systems has been *decomposition* or modularization. The system is broken into smaller pieces, with interface variables serving as the boundary between computation internal to the module and external computation. In many settings it is desirable to have a *reusable* collection of such modules — therefore it becomes necessary to construct rather general components that capture some interesting set of aspects of the behavior, and then allow for flexible ways in which such components can be combined to achieve a range of desired functionality. For instance, when modeling a circuit one may desire to have models for components (voltage and current sources, resistors, wires, lamps) constructed independent of their context of use (the no-function-in-structure principle of [dKB85]). For this to work,

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

it must be possible for the internal structure of the component to represent the effect of all the possible networks that it may be placed in. One desires modularity, and yet *sensitivity* to the environment.

Constraint-based communication offers one such means for achieving the desired flexibility. A component $p(X, Y, Z)$ may share an interface (X) with another component $q(X, A)$ and a different interface (Y, Z) with yet another $r(Y, Z, B)$. The influence p may have with q (via X) is *symmetric*: the constraints imposed by both p and q on X are jointly visible to both p and q (and to other agents with access to X, i.e. sharing this unit of interface). Furthermore constraints are *additive*: the constraints c that p may impose on X could combine with the constraints d that q may impose on X to enable the presence of another constraint e which is not separately entailed by either c or d. This allows a component to have *non-local* influences on other components: p may impose a constraint on Y which could combine with a constraint imposed by r on (Y, B) that could affect another component s that shares B. In this way a component may have an affect on other components with which it is not connected directly and with which it even does not share any interface. Further, constraints support *dynamic interfaces* (c.f. "mobility" in the π-calculus): as p evolves it may introduce another interface variable T which may be communicated via its previous interface to a new component thus establishing a new interface. Similarly, shared interface variables may be dropped. Finally, such concurrent communication between components is *determinate* and *timing-independent* ("Church -Rosser") — what matters is not *when* a constraint is added to the shared store, but that it is.

These ideas have led to the development of several concrete modeling/programming notations e.g.cc(ccFD)[HSD92], Oz [SHW94], and notations in model-based computing (see http://www.parc.xerox.com/mbc). Underlying them, over the last several years we have developed mathematical models for programming combinators involved. The purpose of this review is to collect together in one place and summarize the basic development of these models.

In outline, the basic ideas are as follows. Communication in CCP is based on a generic, parametric notion of first-order pieces of partial information: first-order because the constraints involve *variables* over some underlying domain of discourse, partial because constraints do not necessarily completely determine the values that variables take. The development of the control constructs within CCP proceeds uniformly, given a particular choice of a constraint system. The basic paradigm is formalized by means of the language of *determinate CCP*:

$P ::= a$	*Tell* the constraint a to the store.
$\mid$ **if** a **then** P	If a can be deduced from the store, reduce to P.
$\mid$ **new** X **in** P	Introduce a new variable X in P.
$\mid P, P$	Run in parallel.

In this setup, recursive procedure calls with parameters can also be supported. Denotations of program are taken to be the "fixed points" of the program: those stores in which the program can run to quiescence without adding any more information. Thus the denotations are sets of constraints with certain properties.

Reactive computation is introduced by means of the notion that a program may interact with its environment in a sequence of discrete time steps [SJG94a]. At each time step, a determinate cc program is executed to quiescence. Two new concepts are introduced:

$P ::=$	**hence** P	Run P at every step from the next instant.
	$\mid$ **if** a **else next** P	Unless a can be deduced from the store, reduce to P at the next instant.

Recursion is now allowed to happen only across time instants. Denotations are taken to be sets of sequences of constraints with certain properties.

Instantaneous interrupts cannot be supported in the above model of concurrent computation: one can only detect the *absence* of information, or negative information, at the current instant and act on it at the *next* instant. Supporting the detection of negative information instantaneously requires extending the untimed language cc, since cc itself does not allow the detection of such unstable information — any information detected by a cc program is never invalidated, something which is not true of negative information. We did this by adding *defaults*, which allow us to detect stable negative information:

$P ::=$ **if** a **else** P	Unless a can be deduced from the store, reduce to P.

Introducing defaults poses the problem of non-monotonicity: a constraint that could be produced in the presence of a may not be produced in the presence of additional information $a \wedge b$. Nevertheless we showed [SJG] that a remarkably simple model was possible for this language that took a process to be a set of *pairs* of constraints.

Extending Default cc over discrete time provides a framework for reactive computing that supports instantaneous interrupts.

In order to model physical systems, it becomes necessary to extend these ideas to allow for continuously varying agents. This is done by extending the notion of a constraint system to a *continuous constraint system*: essentially this allows for the specification of constraints (such as those involving differential equations) which vary continuously over (real) time. No new constructs are needed – just the interpretation of time has to be taken now to be the reals rather than the integers. Denotations now become functions from the non-negative reals to Default cc observations, similar to the Timed Default cc observations which were functions from natural numbers to Default cc observations.

The embeddings between these languages are summarized in Figure 1. Note that we have systematically ignored the dimension of indeterminacy (e.g. guarded nondeterministic choice). This leads to an orthogonal development of all these ideas, which is outside the scope of this paper.

2 The cc languages

We will first give brief accounts of the various languages mentioned in Figure 1. For detailed information, we refer the reader to the appropriate papers.

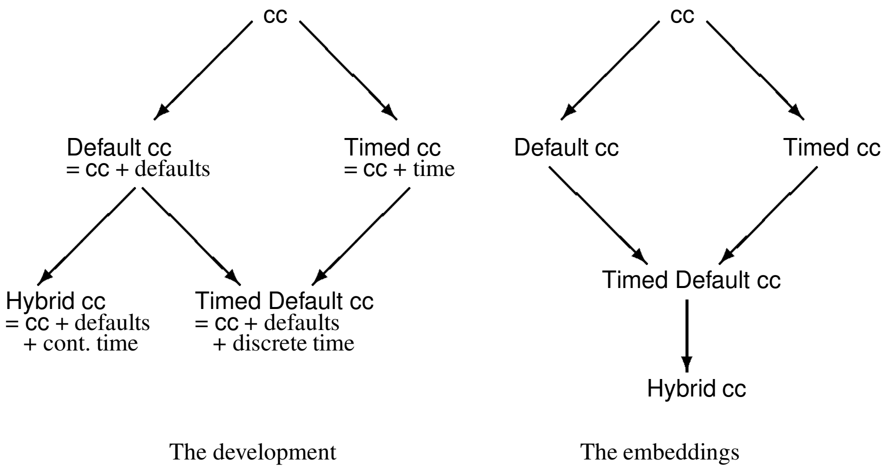

Fig. 1. The relations between the cc languages.

2.1 Constraint Systems.

All cc languages are built generically over constraint systems [Sar92, SRP91]. 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), denoted by $\vdash$, that relates tokens to tokens. We use $a, b, \ldots$ to range over tokens. Logical entailment induces through symmetric closure the logical equivalence relation, $\approx$.

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

1. D is closed under conjunction($\wedge$); $\vdash \subseteq D \times D$ satisfies:
   (a) $a \vdash a$
   (b) $a \vdash a'$ and $a' \wedge a'' \vdash b$ implies that $a \wedge a'' \vdash b$
   (c) $a \wedge b \vdash a$ and $a \wedge b \vdash b$
   (d) $a \vdash b_1$ and $a \vdash b_2$ implies that $a \vdash b_1 \wedge b_2$.
2. $\mathbf{Var}$ 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) $a \vdash \exists_X a$
   (b) $\exists_X(a \wedge \exists_X b) \approx \exists_X a \wedge \exists_X b$
   (c) $\exists_X \exists_Y a \approx \exists_Y \exists_X a$
   (d) $a \vdash b$ implies that $\exists_X a \vdash \exists_X b$.
3. $\vdash$ is decidable.

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

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 $\mathtt{true} = \{a \mid \forall b \in D.\ b \vdash a\}$ and greatest element $\mathtt{false} = D$. Reverse inclusion is written $\supseteq$. $\exists, \vdash$ lift to operations on constraints. Examples of such systems are the system Herbrand (underlying logic programming), FD [HSD92], and `Gentzen` [SJG94b].

Example 1. **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 2. **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 3. **The `Gentzen` constraint system.** For real-time computation we have found the simple constraint system ($\mathcal{G}$) to be very useful. `Gentzen` provides the very simple level of functionality that is needed to represent signals, *e.g.* as in ESTEREL and LUSTRE. 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 $b_i = \mathtt{true}$ otherwise.

In the rest of this paper 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.

2.2 Concurrent constraint programming

The model for determinate **cc** [SRP91] is based on observing for each agent A those stores u in which it is quiescent, that is those stores u in which executing A does not result in the generation of any more information. Define the predicate $A{\downarrow}^u$ (read: "A converges on u" or "A quiesces on u"). The intended interpretation is: A when executed in u does not produce any information that is not entailed by u. We then have the evident axioms for the combinators:

Tell The only inputs on which a can converge are those that already contain the information in a:

$$\frac{a \in u}{a \downarrow^u}$$

Ask The first corresponds to the case in which the ask is not answered, and the second in which it is:

$$\frac{a \notin u}{(\mathbf{if}\ a\ \mathbf{then}\ A) \downarrow^u} \quad \frac{A \downarrow^u}{(\mathbf{if}\ a\ \mathbf{then}\ A) \downarrow^u}$$

Parallel Composition To converge on u, both components must converge on u:

$$\frac{A_1 \downarrow^u \; A_2 \downarrow^u}{(A_1, A_2) \downarrow^u}$$

Hiding Information about the variable X is local to A.

$$\frac{A \downarrow^v \; \exists_X . u = \exists_X . v}{(\textbf{new } X \textbf{ in } A) \downarrow^u}$$

Note that these axioms for the relation are "compositional": whether an agent converges on u is determined by some conditions involving whether its sub-agents converge on u. This suggests taking the denotation of an agent A to be the set of all u such that $A \downarrow^u$ — thus the denotation of an agent is the set of all its quiescent stores. Because of the axioms above, the denotation is compositional.

Conversely, we would like to obtain the sets of observations which can be the denotations of agents. Such a set should have the property that above every (input) constraint a, there is a unique minimal element (the output) in the denotation. We can say this generally by requiring that the sets of constraints representing denotations be closed under glbs of arbitrary non-empty subsets.

The following definitions are evident —

$$\begin{aligned}
[\![a]\!] &= \{u \in |D| \mid a \in u\} \\
[\![\textbf{if } a \textbf{ then } B]\!] &= \{u \in |D| \mid a \in u \Rightarrow u \in [\![B]\!]\} \\
[\![A, B]\!] &= [\![A]\!] \cap [\![B]\!] \\
[\![\textbf{new } X \textbf{ in } A]\!] &= \{u \in |D| \mid \exists v \in [\![A]\!], \exists_X u = \exists_X v\}
\end{aligned}$$

Recursion is handled via least fixed points on the domain of processes ordered by reverse inclusion.

The output of a process Z on a given input i is the least o in Z that is above i.

$$Z(i) = \{o \in |D| \mid o \in Z, \forall j \in Z, j \supseteq i \Rightarrow j \supseteq o\}$$

The determinacy of cc is reflected in the fact that $Z(i)$ is always a singleton set.

2.3 The Timed cc programming language.

Our first extension of cc for specifying reactive systems was Timed cc [SJG94a]. The necessity for this arose from the fact that cc does not allow the detection of negative information, *i.e.* the absence of information. However, this information is frequently needed in a reactive system — we often want to take some action because a certain event did not occur within a certain time.

Timed cc arises from combining cc with work on the synchronous languages ([BG92], [HCP91], [GBGlM91], [Har87], [CLM91]). These languages are based on the hypothesis of Perfect Synchrony: *Program combinators are determinate primitives that respond instantaneously to input signals. At any instant the presence* and *the absence of signals can be detected.* In synchronous languages, physical time has the same status as any other external event, *i.e.* time is multiform. So combination of programs with different notions of time is allowed. Programs that operate only on "signals" can be compiled into finite state automata with simple transitions. Thus, the single step execution time of the program is bounded and makes the synchrony assumption realizable in practice.

Integrating cc with synchronous languages yields Timed cc: at each time step the computation executed is a cc program. Computation progresses in cycles: input a constraint from the environment, compute to quiescence, generating the constraint to be output at this time instant, and the program to be executed at subsequent time instants. There is no relation between the store at one time instant and the next — constraints that persist, if any, must explicitly be part of the program to execute at subsequent time instants.

The addition of time allows the detection of negative information as follows — if a has not happened at time t, we can take action based on that information at time $t+1$, since no further information could contradict the fact that a was not true at time t. Thus time-steps naturally allow us to detect and act upon such negative information. The syntax of Timed cc is built as follows:

$$\boxed{P ::= a \mid P, P \mid \textbf{if}\ a\ \textbf{then}\ P \mid \textbf{new}\ X\ \textbf{in}\ P \mid \textbf{if}\ a\ \textbf{else next}\ P \mid \textbf{hence}\ P}$$

The cc combinators behave as before. The agent **if** a **else next** P, when executed at time t, checks if a becomes true at time t. If it does, then nothing happens, otherwise P is set up to execute at the next time instant. **hence** P executes a copy of P at each time step *after* the time step at which it was activated.

Notation. Let s, s' be sequences over some domain Dom, and let d be an element of the domain. $s \cdot s'$ denotes sequence concatenation. d can be considered a one element sequence for this purpose. Given a set of sequences Z, We use Z **after** s to denote the set of sequences $\{s' \mid s \cdot s' \in Z\}$, and $Z(0)$ to be $\{u \in Dom \mid \exists s.\ u \cdot s \in Z\}$. $|s|$ denotes the length of s. s^i denotes the prefix of s of length i, while $s(i)$ denotes the i'th element. $\exists_X s$ is the string s', where $s'(k) = \exists_X s(k)$, for all $k < |s|$. ϵ denotes the empty sequence.

Model for Timed cc. Execution of a Timed cc program can be seen as a sequence of executions of cc programs. Thus an observation is a sequence of quiescent stores of each of these programs. So observations are finite sequences of constraints, and the set of all observations is denoted **TccObs**. A process Z is a set of observations satisfying the following conditions:

- ϵ, the empty sequence, is in Z.
- If $s \in Z$, then every prefix of s is also in Z.
- For every $s \in Z$, the set $(Z$ **after** $s)(0)$ is a cc process.

The last condition is due to the fact that we have a cc process at each step, so after any sequence of interactions with the environment, we have a cc process left to execute, to determine the result of the next interaction.

The definitions of the combinators are as expected:

$$
\begin{aligned}
\mathcal{T}[\![a]\!] &= \{\epsilon\} \cup \{u \cdot s \in \textbf{TccObs} \mid a \in u\} \\
\mathcal{T}[\![\textbf{if } a \textbf{ then } B]\!] &= \{\epsilon\} \cup \{u \cdot s \in \textbf{TccObs} \mid a \in u \Rightarrow u \cdot s \in \mathcal{T}[\![B]\!]\} \\
\mathcal{T}[\![A, B]\!] &= \mathcal{T}[\![A]\!] \cap \mathcal{T}[\![B]\!] \\
\mathcal{T}[\![\textbf{new } X \textbf{ in } A]\!] &= \{s \in \textbf{TccObs} \mid \exists s' \in \mathcal{T}[\![A]\!], \exists_X s = \exists_X s', \\
&\qquad \forall i < |s|.s(i) \in \textbf{new } X \textbf{ in } (\mathcal{T}[\![A]\!] \textbf{ after } s'^{i-1})(0)\} \\
\mathcal{T}[\![\textbf{if } a \textbf{ else next } B]\!] &= \{\epsilon\} \cup \{u \cdot s \in \textbf{TccObs} \mid a \notin u \Rightarrow s \in \mathcal{T}[\![B]\!]\} \\
\mathcal{T}[\![\textbf{hence } A]\!] &= \{\epsilon\} \cup \{u \cdot s \in \textbf{TccObs} \mid \forall s_1, s_2.\, [s = s_1 \cdot s_2 \Rightarrow s_2 \in \mathcal{T}[\![A]\!]]\}
\end{aligned}
$$

Note that we did not need to add recursion explicitly to the language, as the construct **hence** A enables us to define guarded parameterless recursion.

The output of a Timed cc process Z on a sequence of inputs i is a sequence o with $|i| = |o|$ and $o(k) \in ((Z \textbf{ after } o^{k-1})(0))(i(k))$.

2.4 Default cc

While the extension of cc to Timed cc allowed us to detect negative information, there was still an asymmetry between positive and negative information — negative information could not be acted upon until the next time instant. This is not acceptable in several situations, since the delays could cascade, rendering the model useless. So it is necessary to detect negative information immediately, and this requires extending the basic monotonic model of cc.

The fundamental move we now make is to allow the expression of *defaults*, after [Rei80]. We allow agents of the form **if** a **else** A, which intuitively mean that *in the absence of* information a, reduce to A. Note however that A may itself cause further information to be added to the store; and indeed, several other agents may simultaneously be active and adding more information to the store. Therefore requiring that information a be absent amounts to making an *assumption* about the future evolution of the system: not only does it not entail a now, but also it will not entail a in the future. Such a demand on "stability" of negative information is inescapable if we want a computational framework that does not produce results dependent on vagaries of the differences in speeds of processors executing the program. We call the resulting language Default cc [SJG].

The critical question in building a model for Default cc then is: how should the notion of observation be extended? Intuitively, the answer seems obvious: observe for each agent A those stores u in which they are quiescent, *given the guess v about the final result*, since once the negative information is detected with respect to the final result, it cannot be voided. 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.

Formally, we define a predicate $A \downarrow_v^u$ (read as: "A converges on u under the guess v"). 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.

We then have the evident axioms for the primitive combinators:

Tell The information about the guess v is not needed:

$$\frac{a \in u}{a \downarrow_v^u}$$

Positive Ask The first two rules cover the case in which the ask is not answered, and the third the case in which it is:

$$\frac{a \notin v}{(\textbf{if } a \textbf{ then } A) \downarrow_v^u} \quad \frac{a \notin u, A \downarrow_v^v}{(\textbf{if } a \textbf{ then } A) \downarrow_v^u} \quad \frac{A \downarrow_v^u}{(\textbf{if } a \textbf{ then } A) \downarrow_v^u}$$

Parallel Composition Note that a guess v for A_1, A_2 is propagated down as the guess for A_1 and A_2:

$$\frac{A_1 \downarrow_v^u \; A_2 \downarrow_v^u}{(A_1, A_2) \downarrow_v^u}$$

Negative Ask In the first case, the default is disabled, and in the second it can fire:

$$\frac{a \in v}{(\textbf{if } a \textbf{ else } A) \downarrow_v^u} \quad \frac{A \downarrow_v^u}{(\textbf{if } a \textbf{ else } A) \downarrow_v^u}$$

Hiding Hiding becomes considerably more complicated in this model, we refer the reader to [SJG] for details.

Again, note that these axioms for the relation are "compositional": whether an agent converges on (u, v) is determined by some conditions involving whether its sub-agents converge on (u, v). This suggests taking the denotation of an agent A to be the set of all (u, v) such that $A \downarrow_v^u$; because of the axioms above, the denotation is compositional. Formally, we define observations as a pair of constraints (u, v), where $u \subseteq v$. The intended interpretation is: (u, v) is an observation of A if when the guess v is used to resolve defaults, then executing A in u does not produce any information not entailed by u, and executing A in v does not produce any information not entailed by v.

Define **DObs** $\stackrel{d}{=} \{(u, v) \in |D| \times |D| \mid v \supseteq u\}$. A process Z is a collection of observations that satisfies the following conditions:

1. Guess convergence — $(v, v) \in Z$ if $(u, v) \in Z$. We will only make those guesses v under which a process can actually quiesce, *i.e.* executing the process in v does not produce any information not entailed by v.
2. Local determinacy — the idea is that once a guess is made, every process behaves like a cc agent. Thus, for each v such that $(v, v) \in Z$, the set $\{u \in |D| \mid (u, v) \in Z\}$ is a cc process.

The denotational definitions now follow from the model given above.

$$\begin{aligned}
\mathcal{D}[\![a]\!] &\stackrel{d}{=} \{(u, v) \in \textbf{DObs} \mid a \in u\} \\
\mathcal{D}[\![\textbf{if } a \textbf{ then } A]\!] &\stackrel{d}{=} \{(u, v) \in \textbf{DObs} \mid \; a \in v \Rightarrow (v, v) \in \mathcal{D}[\![A]\!], \\
&\qquad\qquad\qquad\qquad\quad\;\; a \in u \Rightarrow (u, v) \in \mathcal{D}[\![A]\!]\} \\
\mathcal{D}[\![\textbf{if } a \textbf{ else } A]\!] &\stackrel{d}{=} \{(u, v) \in \textbf{DObs} \mid a \notin v \Rightarrow (u, v) \in \mathcal{D}[\![A]\!]\} \\
\mathcal{D}[\![A, B]\!] &\stackrel{d}{=} \mathcal{D}[\![A]\!] \cap \mathcal{D}[\![B]\!]
\end{aligned}$$

For the formal definition of hiding, we refer the reader to [SJG]. Recursion is modeled by least fixed points on the domain of processes ordered by reverse inclusion, however we will consider Default cc without recursion. Here are some examples illustrating some denotations.

Example 4.

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

This is an example of a default theory which does not have any extensions ([Rei80]). 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. It is different from **if** b **else** b, whereas in default logic and synchronous languages both these agents are considered the same, *i.e.* meaningless, and are thrown away.

Example 5.

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

This agent 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 agent b. However, **if** a **else** b is not the same as **if** $\neg a$ **then** b, in the second case some agent must explicitly write $\neg a$ in the store, but in the first case merely the fact that no agent can write a is sufficient to trigger b. This difference is demonstrated by running both b and **if** a **then** b, **if** a **else** b in parallel with **if** b **then** a — b produces $a \sqcup b$ on $true$, while **if** a **then** b, **if** a **else** b produces no output.

The output of a Default cc process Z on input i is some resting point o, such that there is no stopping point between i and o. Thus

$$Z(i) = \{o \in |D| \mid (o,o) \in Z, \forall (j,o) \in Z, j \supseteq i \Rightarrow j \supseteq o\}$$

Note that Z is not monotone, for example $(\mathcal{D}[\![\textbf{if } a \textbf{ else } b]\!])(\texttt{true}) = \{b\}$, while $(\mathcal{D}[\![\textbf{if } a \textbf{ else } b]\!])(a) = \{a\}$. Note also that Default cc programs can have no outputs or multiple outputs on an input. For example **if** a **else** a has no output on `true`, while both a and b are possible outputs of **if** a **else** b, **if** b **else** a on `true`. Default cc programs which have unique outputs for all inputs are called *determinate*. In [SJG], we describe an algorithm for determinacy detection.

2.5 Timed Default cc

Default cc can be extended to yield a timed language just as cc was extended to Timed cc. At each time step, a Default cc program is executed to determine the output on the given input, then the program for the next step is set up, and the system becomes dormant until the next step is triggered by an input from the environment. Thus we get a reactive programming language in the family of synchronous programming languages ([BB91]), called Timed Default cc [SJG].

We need to add one more combinator to Default cc, **hence** A, which executes a new copy of A at each step after the current one.

Just as Timed cc observations were finite sequences of cc observations, similarly Timed Default cc observations are finite sequences of Default cc observations. However notice from the input-output relation of Default cc that only Default cc observations of the form (v,v) can be seen as the result of an execution. Since at any time step the previous time steps are completed, their observations must be of the form (v, v). So a Timed Default cc observation is a sequence of Default cc observations in which all elements but the last must be of the form (v, v). The definition of a process remains the same as Timed cc, with cc changed to Default cc.

The denotational definitions of processes are as before, we repeat them here for completeness.

$$
\begin{aligned}
\mathcal{P}[\![a]\!] &= \{\epsilon\} \cup \{(u,v)\cdot s \in \textbf{TDObs} \mid a \in u\} \\
\mathcal{P}[\![\textbf{if } a \textbf{ then } B]\!] &= \{\epsilon\} \cup \{(u,v)\cdot s \in \textbf{TDObs} \mid \begin{array}[t]{l} a \in u \Rightarrow (u,v)\cdot s \in \mathcal{P}[\![B]\!] \\ a \in v \Rightarrow (v,v) \in \mathcal{P}[\![B]\!]\} \end{array} \\
\mathcal{P}[\![A, B]\!] &= \mathcal{P}[\![A]\!] \cap \mathcal{P}[\![B]\!] \\
\mathcal{P}[\![\textbf{if } a \textbf{ else } B]\!] &= \{\epsilon\} \cup \{(u,v)\cdot s \in \textbf{TDObs} \mid a \notin v \Rightarrow (u,v)\cdot s \in \mathcal{P}[\![B]\!]\} \\
\mathcal{P}[\![\textbf{hence } A]\!] &= \{\epsilon\} \cup \{(u,v)\cdot s \in \textbf{TDObs} \mid \forall s_1, s_2.\, s = s_1 \cdot s_2 \Rightarrow s_2 \in \mathcal{P}[\![A]\!]\}
\end{aligned}
$$

The definition of hiding is similar to the definition in Timed cc, we refer the reader to [SJG] for details.

The input output relation is similar to that for Timed cc— the output of a Timed Default cc process Z on a sequence of inputs $i = \langle i_0, i_1, \ldots i_n\rangle$ is a sequence $o = \langle o_0, o_1, \ldots o_n\rangle$, where $o_k \in ((Z \textbf{ after } o'^{k-1})(0))(i_k)$, where $o'(k) = (o_k, o_k)$ for all $k < |o|$.

We demonstrate the expressiveness of Timed Default cc by defining a few combinators in terms of the basic ones given above.

Example 6. We can define the **next** A combinator to start a copy of A at the next time instant, in terms of **hence** A —

$$\textbf{next } A = \textbf{new } \texttt{stop} \textbf{ in hence } [\textbf{if } \texttt{stop} \textbf{ else } A, \textbf{hence } \texttt{stop}]$$

Thus if **next** A is called at time 1, then from time 2 onwards **if** `stop` **else** A is executed. At time 2, no `stop` is generated (as it is local and **hence** `stop` does not start generating it till time 3), so A is started. From 3 onwards, `stop` is generated. Thus the net effect is the execution of a copy of A at time 2.

Example 7. A useful variant of **hence** A is **always** $A \stackrel{d}{=} A$, **hence** A, it simply starts a new copy of A every time, instead of from the next time instant.

Example 8. Parameterless guarded recursion can also be defined. Consider a Timed Default cc program as a set of declarations $g :: A$ along with an agent. (Here g names a parameterless procedure.) These declarations can be replaced by the construct **always if** g **then** A. The names of the agents g can now occur in the program, and will be treated as simple propositional constraints. Note that only one call of an agent needs to be made at one time instant.

Example 9. Another useful combinator is **first** a **do** B, which starts the process B at the first time instant that a becomes true. It can be defined as

$$\textbf{first } a \textbf{ do } B = \textbf{new } \texttt{stop} \textbf{ in always } [\begin{array}[t]{l}\textbf{if } \texttt{stop} \textbf{ else if } a \textbf{ then } B \\ \textbf{if } a \textbf{ then hence } \texttt{stop}]\end{array}$$

This program keeps on executing **if** a **then** A, unless it receives the signal $\texttt{stop}$. The $\texttt{stop}$ is produced in all instants after a is true. Note the fact that this definition is identical to the definition for the continuous language Hybrid cc [GJS], this will also be the case for the other definitions given below.

Example 10. The agent **time** A **on** a denotes a process whose notion of time is the occurrence of the tokens a — A evolves only at the time instants at which the store entails a. This is definable as follows:

Given a token a and a sequence $s \in$ **TDObs**, define the subsequence of s in which $a \in \pi_1(s(i))$ as s_a. Formally, this subsequence is defined by induction on the length as follows:

$$\begin{aligned} \epsilon_a &= \epsilon \\ (s \cdot (u,v))_a &= \begin{cases} s_a \cdot (u,v), & \text{if } a \in u \\ s_a, & \text{otherwise} \end{cases} \end{aligned}$$

Now define $\mathcal{P}[\![\textbf{time}\ A\ \textbf{on}\ a]\!] \stackrel{d}{=} \{s \in \textbf{TDObs} \mid s_a \in \mathcal{P}[\![A]\!]\}$.

This combinator satisfies the following equational laws, which can be used to remove its occurrences from any program.

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

time A **on** a can be used to construct various other combinators that manipulate the notion of time ticks being fed to a process. The general schema is to time the process A on some signal $\texttt{go}$. Now another process is set up to generate $\texttt{go}$ whenever one wants A to proceed. The next few examples illustrate this.

Example 11. **do** A **watching** a is an interrupt primitive related to *strong abortion* in ESTEREL ([Ber93]). **do** A **watching** a behaves like A until a time instant when a is entailed; when a is entailed A is killed instantaneously. Using **time** this is definable as:

$$\begin{aligned} \textbf{do}\ A\ \textbf{watching}\ a = \textbf{new}\ \texttt{stop}, \texttt{go}\ \textbf{in}\ [\ & \textbf{time}\ A\ \textbf{on}\ \texttt{go}, \\ & \textbf{first}\ a\ \textbf{do}\ \textbf{always}\ \texttt{stop}, \\ & \textbf{always}\ \textbf{if}\ \texttt{stop}\ \textbf{else}\ \texttt{go}] \end{aligned}$$

Example 12. There is a related weak abortion construct ([Ber93]) —$\mathbf{do}\ A\ \mathbf{trap}\ a$ behaves like A until a time instant when a is entailed; when a is entailed A is killed from the *next* time instant. It can be defined as

$$\begin{aligned} \mathbf{do}\ A\ \mathbf{trap}\ a = \textbf{new}\ \texttt{stop}, \texttt{go}\ \textbf{in}\ [\ & \textbf{time}\ A\ \textbf{on}\ \texttt{go}, \\ & \textbf{first}\ a\ \textbf{do}\ \textbf{hence}\ \texttt{stop}, \\ & \textbf{always}\ \textbf{if}\ \texttt{stop}\ \textbf{else}\ \texttt{go}] \end{aligned}$$

Note that the signal $\texttt{stop}$ is generated from the time instant *after* a is seen.

Example 13. The Suspension-Activation primitive, $\mathbf{S}_a\mathbf{A}_b(A)$, is a preemption primitive that is a variant of *suspension* in ESTEREL ([Ber93]). $\mathbf{S}_a\mathbf{A}_b(A)$ behaves like A until a time instant when a is entailed; when a is entailed A is suspended immediately (hence the S_a). A is reactivated in the time instant when b is entailed (hence the A_b). The familiar $(\texttt{control} - \texttt{Z}, \texttt{fg})$ is a construct in this vein. This can be expressed as:

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

2.6 Hybrid cc

Timed Default cc was obtained by extending Default cc across the natural numbers. Similarly, extending Default cc across the real numbers yields Hybrid cc [GJS, GJSB95, GJS96]. Hybrid cc is intended to be a language for describing hybrid systems, which are systems that can evolve discretely as well as continuously [BG90, NSY92, MMP92, GNRR93].

We model hybrid systems that evolve in a piecewise continuous manner. Thus 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. The “extension” of Default cc over continuous time 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, the Default cc model of processes is extended over continuous time to describe Hybrid cc processes.

Continuous constraint systems(ccs) are described in detail in [GJS]. Informally, continuous constraint systems are obtained by incorporating intuitions from real analysis into constraint systems. Intuitively, continuous constraint systems express the information content of initial value problems in integration. Continuous constraint systems support an integration operation, $\int^r \texttt{init}(a) \wedge \texttt{cont}(b)^3$, which determines the effect of b holding continuously in the interval $(0, r)$, if a held at time 0. For example, $\int^7 \texttt{init}(x = 3) \wedge \texttt{cont}(dot(x) = 4) \vdash x = 31$. [GJS] describes a set of axioms for continuous constraint systems — these include intuitive properties of integration such as the monotonicity and continuity of integration, and some computability axioms to enable finite description and implementation. A continuous evolution of the ccs is a function from some prefix of the reals to $|D|$, such that $\exists a.\forall r > 0.f(r) = \int^r a$.

Now, we are ready to describe informally what Hybrid cc observations should be. First, note that we intend the execution at every time instant to be modeled by the execution in Default cc. So, we choose observations to be functions with domain a prefix of the non-negative reals and range Default cc observations[4]. Secondly, note

[3] cont and init may be thought of as labels which distinguish the constraints which hold continuously from those which hold at the initial points.

[4] Contrast this against the observations of Timed Default cc, which were functions from a prefix of the natural numbers to Default cc observations.

that we intend every function 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 continuous evolutions of the ccs. For such a function, we can partition its domain into a (possibly infinite) alternating sequence of points and open intervals, called *phases* of the observation — the point phases indicate discontinuities and the open interval phases are continuous behaviors of the ccs. Finally, as in Timed Default cc, we intend the functions to satisfy observability — *i.e.* the computation in any phase but the last is a completed Default cc computation. We use **HObs** for the set of Hybrid cc observations.

Processes are sets of observations that satisfy similar conditions as in Timed cc— the empty observation is in every process, and all processes are prefix closed. In addition, if every proper prefix of an observation is in a process, then the observation must be in the process. Finally we wish to be able to execute Default cc programs in each phase, so as before the process **after** any observation must form a Default cc process. These definitions are stated precisely in [GJS].

The only new combinator we need to add to the set of Default cc combinators is again **hence** A, which executes a new copy of A at every real time point in the *open* interval starting at the time it was called. The denotational definitions are similar to Timed Default cc. We use $\pi_1(z)$ and $\pi_2(z)$ to denote the first and second components of a pair z.

$$
\begin{aligned}
\mathcal{H}[\![a]\!] &= \{\epsilon\} \cup \{f \in \textbf{HObs} \mid a \in \pi_1(f(0))\} \\
\mathcal{H}[\![\textbf{if } a \textbf{ then } B]\!] &= \{\epsilon\} \cup \{f \in \textbf{HObs} \mid \; a \in \pi_1(f(0)) \Rightarrow f \in \mathcal{H}[\![B]\!] \\
&\qquad\qquad\qquad\qquad\qquad a \in \pi_2(f(0)) \Rightarrow (\pi_2(f(0)), \pi_2(f(0))) \in \mathcal{H}[\![B]\!]\} \\
\mathcal{H}[\![A, B]\!] &= \mathcal{H}[\![A]\!] \cap \mathcal{H}[\![B]\!] \\
\mathcal{H}[\![\textbf{if } a \textbf{ else } B]\!] &= \{\epsilon\} \cup \{f \in \textbf{HObs} \mid a \notin \pi_2(f(0)) \Rightarrow f \in \mathcal{H}[\![B]\!]\} \\
\mathcal{H}[\![\textbf{hence } A]\!] &= \{\epsilon\} \cup \{f \in \textbf{HObs} \mid \forall t > 0, f \restriction [t, \infty) \in \mathcal{H}[\![A]\!]\}
\end{aligned}
$$

where $f \restriction [t, \infty)(r) = f(t+r), 0 \leq r$. Again, we omit the definition of hiding. Guarded parameterless recursion can again be coded using **hence** . The input-output relation for a process on a piecewise continuous input trace is defined as for Timed Default cc.

Combinators. The hybrid analogs of the defined combinators given above can be defined in Hybrid cc also. In fact, none of the definitions change. For details, see [GJS].

3 The relationships between the languages

As pointed out in the introduction, the languages given above are successively more general. We will now give the embeddings that establish the partial order described there. Note that while we have omitted the definition of hiding in some of the above languages, all the embeddings hold for programs with hiding also.

3.1 Embedding cc in Default cc

cc embeds conservatively in Default cc. The embedding of cc into Default cc is immediate, since Default cc is a strictly larger language. Denotationally, given a cc process Q, the Default cc process corresponding to it is given by $\{(u, v) \in \textbf{DObs} \mid u, v \in Q\}$.

The partial converse exploits a characterization of a large class of "monotone" Default cc processes — intuitively, this class captures the processes that do not exploit the ability to detect negative information. For any determinate Default cc process P, the input output relation $P(i)$ is the graph of a monotone function, if P satisfies:

1. If $(u, v) \in P$ then $(u, u) \in P$.
2. If $(u, v) \in P, (v', v') \in P, v' \supseteq v$ then $(u, v') \in P$.

Note that the embedding of any cc process satisfies both the properties given above, and thus is monotone. For a Default cc process P satisfying the conditions of the above lemma, the fixed point set of the corresponding cc process is just $\{u \mid (u, u) \in P\}$.

Furthermore, we note that all the definitions of Default cc (including hiding) for the cc combinators also commute with this embedding, so if we take the embedding first and then use the Default cc definitions, we get the same results as using the cc definitions and then embedding in Default cc.

3.2 Embedding cc in Timed cc

Timed cc allows a different embedding of cc. This embedding exploits the time structure of Timed cc to reveal some of the operational structure of the cc program.

We can define **first** a **then next** A as

$$\textbf{new } \texttt{stop} \textbf{ in always if } a \textbf{ then } [\textbf{if } \texttt{stop} \textbf{ else next } A, \ \textbf{hence } \texttt{stop}]$$

The embedding is as follows:

$$\begin{aligned}
\mathcal{E}(a) &= a, \textbf{hence } a \\
\mathcal{E}(\textbf{if } a \textbf{ then } A) &= \textbf{first } a \textbf{ then next } \mathcal{E}(A) \\
\mathcal{E}(A, B) &= \mathcal{E}(A), \mathcal{E}(B) \\
\mathcal{E}(\textbf{new } X \textbf{ in } A) &= \textbf{new } X \textbf{ in } \mathcal{E}(A)
\end{aligned}$$

An interesting feature of this embedding is that at any time-step, the cc program to be executed is a tell process. This embedding reveals a lot more of the operational structure of the program than the usual cc semantics — informally it represents a step semantics to cc programs obtained by stratifying the addition of constraints. Note that the embedding is not fully abstract — two equivalent cc programs may be embedded differently. For example a, b will be embedded differently from a, **if** a **then** b.

It is also possible to construct a partial converse to the above embedding. Consider the monotone observations of Timed cc— those sequences s where for all $1 < i < |s|$, $s(i) \supseteq s(i-1)$. Consider any Timed cc process containing only such observations, we will call it a monotone process if it also satisfies the following condition: If $s \cdot u$ and $s \cdot u'$ are in Z, and $u' \supseteq u$, then Z **after** $(s \cdot u') \subseteq Z$ **after** $(s \cdot u)$. This is similar to the second condition in the above subsection. Then, we have:

Lemma 2. *Let Z be a monotone* Timed cc *process. Define $Z' = \{u \in |D| \mid \exists s \in Z, s(|s|-1) = u, \forall k > 0 \exists s'.(s \cdot s') \in Z, |s'| = k, \forall i < k.s'(i) = u\}$. Then Z' is a* cc *process.*

The Z' in this lemma consists of all those u under which Z reaches quiescence — that is, unless there is some input from the environment, Z will not add anything to the store. It can be alternatively defined as $\{u \in |D| \mid \exists s \in |D|^\omega, Fin(s) \subseteq Z, u = \bigsqcup_i s(i)\}$, where $Fin(s)$ is the set of finite prefixes of s.

Let A be a cc program. Let $Z = \mathcal{T}[\![\mathcal{E}(A)]\!]$ be the process obtained by embedding it in Timed cc. Let Z' be the cc process defined in Lemma 2. Then $Z' = [\![A]\!]$.

Embedding Timed cc **and** Default cc **in** Timed Default cc. Since Default cc is conservative over cc, Timed cc directly embeds in Timed Default cc. Also, since a recursion-free Default cc program embeds in any one instant of the Timed Default cc program, every recursion-free Default cc program can be understood as a Timed Default cc program, embedding Default cc in Timed Default cc.

3.3 Embedding Timed Default cc in Hybrid cc

Since Timed Default cc is a discrete language with no autonomous behavior, the continuous constraint system over which Hybrid cc must be defined to embed Timed Default cc is very simple — we have $\int^r a \wedge \mathtt{cont}(b) = a, r = 0$, and $\int^r a \wedge \mathtt{cont}(b) = b, r > 0$. Thus no continuous evolution is possible, and hence there is no possibility of autonomous change. This ccs is described in detail in [GJS].

In order to encode the **hence** A of Timed Default cc, we need to know when the next input comes in from the environment. Following discrete time synchronous languages such as `Esterel`, we attach to each discrete input a special token `tick`. Thus `tick` is a discrete signal, and whenever any input comes in from the environment, `tick` is always present. Now we can embed Timed Default cc in Hybrid cc. The only non-trivial case is **hence** of Timed Default cc. The embedding of **hence** A becomes **hence if** `tick` **then** $E(A)$, where $E(A)$ is the embedding of A. Thus at every `tick` after the current one, a copy of A is started.

As with the earlier embeddings, a partial converse can be described. Consider the subset **DHObs** of Hybrid cc observations built upon the constraint system described above, which have the property that `tick` is true in every point phase, and the pair of constraints at any point in each interval phase are (`true`, `true`). Each such observation obviously corresponds to a Timed Default cc observation, so every Timed Default cc process can be embedded into a Hybrid cc process. The following lemma states the connection between these embeddings.

Lemma 3. *Given a* Timed Default cc *program P, let its embedding in* Hybrid cc *be P'. Then the denotation $\mathcal{P}[\![P]\!]$ when embedded in* **DHObs** *is exactly $\mathcal{H}[\![P']\!] \cap \mathbf{DHObs}$.*

Acknowledgements. This work was supported by grants from ARPA and ONR, and the second author was supported by a grant from the NSF.

References

[BB91] A. Benveniste and G. Berry, editors. *Another Look at Real-time Systems*, September 1991. Special issue of the Proceedings of the IEEE.

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

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

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

[CLM91] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In Benveniste and Berry [BB91]. Special issue of the Proceedings of the IEEE.

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

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

[GBGlM91] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. le Maire. Programming real time applications with SIGNAL. In Benveniste and Berry [BB91]. Special issue of the Proceedings of the IEEE.

[GJS] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Computing with continuous change. *Science of Computer Programming*. To appear.

[GJS96] 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.

[GJSB95] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel Bobrow. Programming in hybrid constraint languages. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Sankar Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 226–251. Springer Verlag, November 1995.

[GNRR93] Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

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

[HCP91] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In Benveniste and Berry [BB91]. Special issue of the Proceedings of the IEEE.

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

[MMP92] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In de Bakker et al. [dBHdRR92], pages 447–484.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In de Bakker et al. [dBHdRR92].

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

[Sar92] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz*, 1992.

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

[SJG] 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.

[SJG94a] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, July 1994.

[SJG94b] 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.

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