Default Timed Concurrent Constraint Programming

Vijay A. Saraswat	Radha Jagadeesan	Vineet Gupta
Systems and Practices Lab.	Dept. of Math. Sciences	Systems and Practices Lab.
Xerox PARC	Loyola University	Xerox PARC
Palo Alto, Ca 94304	Chicago, Il 60626	Palo Alto, Ca 94304

October 25, 1994

Extended Abstract[1]

Abstract

We extend the model of [SJG94b] to express strong time-outs (and pre-emption): if an event A does not happen through time t, cause event B to happen at time t. Such constructs arise naturally in practice (e.g. in modeling transistors) and are supported in languages such as ESTEREL (through instantaneous watchdogs) and LUSTRE (through the "current" operator). The fundamental conceptual difficulty posed by these operators is that they are non-monotonic. We provide a simple compositional semantics to the non-monotonic version of concurrent constraint programming (CCP) obtained by changing the underlying logic from intuitionistic logic to Reiter's default logic [Rei80]. This allows us to use the same construction (uniform extension through time) to develop Default Timed CCP (Default tcc) as we had used to develop Timed CCP (tcc) from CCP [SJG94b]. Indeed the smooth embedding of CCP processes into Default cc processes lifts to a smooth embedding of tcc processes into Default tcc processes. Interesting tcc properties such as determinacy, multi-form time, a uniform pre-emption construct ("clock"), full-abstraction, and compositional compilation into automata are preserved.

Default tcc thus provides a simple and natural (denotational) model capable of representing the full range of pre-emption constructs supported in ESTEREL, LUSTRE and other synchronous programming languages.

Keywords: Programming paradigms — constraint programming, reactive systems, synchronous programming; Formal approaches — denotational semantics, semantics of concurrency

[1]To appear in the Proceedings of the 22^d Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, January 1995.

1 Introduction and Motivation

We elaborate a framework for the design of programming languages that permit instantaneous detection of *negative information*, that is, detection of the absence of information. In such systems the fact that the environment has failed to respond in an expected way (i.e., an interrupt signaling a jam has not been received; a response to a password query has not been received even though the time-period allowed has elapsed) is a piece of information of the same status as information received in an explicit message from the environment. In particular it should be possible to act instantaneously in response to this implicit information (e.g., power should continue to be supplied to motors in the first case; the connection should time-out in the second).

While the problem of representing and reasoning about negative information is present in all reactive programming languages, it shows up in a particularly pure form in frameworks based on a computational interpretation of logic, such as concurrent constraint programming (CCP) [Sar93,SRP91]. This framework is based on the idea that concurrently executing systems of agents interact by posting (telling) and checking (asking) constraints in a shared pool of (positive) information. Constraints are expressions of the form $\mathtt{X} \geq \mathtt{Y}$, or "the sum of the weights of the vehicles on the bridge must not exceed a given limit". They come equipped with their own entailment relation, which determines what pieces of information (e.g., $\mathtt{X} \geq \mathtt{Z}$) follow from which collections of other pieces (e.g. $\mathtt{X} \geq \mathtt{Y}, \mathtt{Y} \geq \mathtt{Z}$). Synchronization is achieved by suspending ask agents until enough information is available to conclusively answer the query; the query is answered affirmatively if it is entailed by the constraints accumulated hitherto.

Such a framework for concurrent computation is proving fruitful in several investigations [SKL90,HSD92,JH91,SHW94,Kac93], with applications in areas ranging from modeling physical systems, to combinatorial exploration and natural language analysis.

There are however some fundamental limitations to this "monotonic accumulation" approach to concurrent computation.

The Quiescence Detection Problem. Within the framework, *quiescence* of computation cannot be detected and triggered on.[2] Two examples should make matters clearer.

Example 1.1 (Histogram, due to K.Pingali) Assume given an array $A[1 \ldots n]$ taking on values in $1 \ldots m$. It is desired to obtain an array $B[1 \ldots m]$ such that for all k, $B[k]$ contains exactly the indices i such that $A[i] = k$. (The histogram of A can then be obtained by associating with each $k \in 1 \ldots m$ the cardinality of $B[k]$.) The computation of B should be done in parallel.

In a language based on monotonic accumulation it is possible to simultaneously assert, for every $j \in 1 \ldots n$ that $j \in B[A[j]]$. This is however, not good enough to force the sets $B[k]$ to contain *exactly* the required indices — all that is being forced is that $B[k]$ contains *at least* the given indices. $\square$

[2] In many cases, quiescence detection can be explicitly programmed. However, this can become quite cumbersome to achieve.

Example 1.2 (Composition of model fragments) Similar examples arise when using such languages for compositional modeling of physical systems (see, e.g.[For88]). In such an application computation progresses via repeated iteration of two phases: a model-construction phase and a model execution phase. In the construction phase, pieces of information ("model fragments") about the variables and constraints relevant in the physical situation being modeled are generated. For example, it may be determined that some real-valued variable, e.g.`current`, is monotonically dependent on `voltage_drop`, and also on `conductance`. On termination of this phase, it is desired to collect together all the variables that `current` is now *known* to depend on (say, its just `voltage_drop` and `conductance`) and then postulate that these are the *only* variables that it depends on. That is, it is desired to postulate the existence of a function f and assert the relationship $\texttt{current} = \texttt{f}(\texttt{voltage\_drop}, \texttt{conductance})$. □

Such detection of quiescence is inherently non-monotonic: if *more* information is provided in the input, *different* (rather than just more) information may be produced at the output.

The Instantaneous Interrupts Problem. Another fundamental source of examples is real-time systems, where the detection of absence of information is necessary to handle interrupts. To get at these examples, however, we first take a short detour to explain Timed Concurrent Constraint (tcc) languages [SJG94b].

tcc arises from combining CCP with work on the synchronous languages such as [BG92], [HCP91], [GBGM91], [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 CCP with synchronous languages yields tcc: at each time step the computation executed is a concurrent constraint 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.

To the combinators of CCP (namely, tell (c), ask ($c \rightarrow A$) and parallel composition ($A_1 \parallel A_2$)), tcc adds unit delay (**next**), and *delayed negative ask* ($c \leadsto$ **next** A). [3] $c \leadsto$ **next** A allows A to be executed at the *next* time instant if the store on quiescence is

[3] In addition, CCP allows for a form of hiding, corresponding to existential quantification. The proper treatment of hiding, however, introduces several complications that we have chosen to ignore. See Section 4 for more comments.

not strong enough to entail c. This allows the programming of *weak* time-outs — if an event A does not happen by time t, cause event B to happen by time $t+1$ — while still allowing the computation at each time step to be monotone and determinate. We showed that the mathematical framework of such an integration is obtained in a simple way — by uniformly extending the mathematical framework of CCP over (discrete) time. Indeed, many complex patterns of temporal behavior — such as the "**do** `A` **watching** `c`" construct of ESTEREL which allows the agent `A` to execute, aborting it at the time instant after a `c` is detected — could be programmed as defined combinators in tcc. In general, it was possible to capture the idea of having processes "clocked" by other (recursive) processes, thus getting very powerful "user-programmable" pre-emption control constructs. The denotational model is very simple and in full accord with an intuitive operational semantics and an underlying logic — discrete time, intuitionistic linear temporal logic.

More generally, tcc provides a powerful setting in which to program systems of reactive, embedded agents — perhaps modeling aspects of the real, physical world — which autonomously maintain internal beliefs in the face of change induced by interaction with the environment. At each step, the agent has an "internal theory" that describes its computational state, its assumptions about its environment, and rules for inferring new information from old. On the basis of these, and the input information, the agent decides to act (send messages to the outside world) and revises its internal state. In particular, it is useful for agents to consider their beliefs to be *interruptible*, subject to abandonment in the face of new information communicated by the environment.

The main drawback of the tcc model, however, is its inability to express *strong* time outs [Ber93]: if an event A does not happen by time t, cause event B to happen at time t. This is the behavior, for example, of the "**do** `A` **watching immediately** `c`" construct of ESTEREL: the execution of `A` is interrupted *as soon as* `c` is detected (rather than one step later). Weak time outs cause the action to be taken to be queued up for the *next* interaction with the environment. While this unit delay is unproblematic in many cases, it is intolerable in cases where these delays can cascade, thereby causing these queued actions to become arbitrarily out of sync with the time when they were actually supposed to happen. If there is a feedback loop, then such a model of pre-emption may simply fail to work.

Example 1.3 (Modeling a transistor) More concretely, consider a transistor whose emitter is grounded, and whose collector is connected to high voltage by a resistor. Unless there is current flowing into the base, the collector is not shorted to ground, and remains pulled high. Because the user may desire to cascade several such transistors (and introduce feedbacks), it is not possible to tolerate a unit delay between detection of absence of current in the base, and determination of the status of the collector – such unit delays can build up unboundedly wrecking the timing information in the circuit being modeled. □

Examples of the need for instantaneous detection of negative information abound in the literature on default reasoning (e.g. [Rei80]).

Example 1.4 (Constraint-based User Interfaces) Consider a setting like THINGLAB [Bor79], in which it is possible for users to draw diagrams, e.g. a parallelogram, that must obey certain constraints. If the user moves a vertex of the parallelogram, then the system moves other vertices in response so as to maintain the constraints. Here it would not do to queue up the computed location of a vertex X for the next interaction, because the user may move X in that interaction. Rather the location of the vertex should be computed and displayed instantaneously, even if no constraint on the location of the vertex arrives from the environment. □

As an example of the use of tcc to model aspects of time-varying, real world situations, consider the following problem.

Example 1.5 (Yale Shooting Problem, [Sho88]) The scenario to be modeled is this: a gun is loaded at time $T = 2$. It is fired at Fred at time $T = 4$. Inbetween, it is possible that the gun may have been subject to various other acts: for example, it may have become unloaded. Various other "common-sense" facts are known: for instance, guns once loaded do not spontaneously become unloaded, if a loaded gun is fired at a live person, and the gun is functioning normally, then the person may cease to be live, etc.

In a setting such as this, it is crucial that a gun be deemed to be loaded at present only if it was loaded at some time in the past, and not unloaded at any time since then *including* the present. Similarly, for success, the gun should be fired in the direction of the perceived *current* position of the target, not the known past position of the target. Even one-step delays introduced due to the modeling framework can invalidate the representation. □

1.1 Defaults

The fundamental conceptual difficulty with the instantaneous detection of negative information is that it is not monotonic. On receipt of further information a conclusion arrived at earlier may have to be withdrawn. This is just not expressible in the CCP framework, which is monotone. Furthermore, no principled analyses of non-monotonic processes have hitherto been available which would allow us integrate them into a reactive real-time programming framework.

The fundamental move we now make is to allow the expression of *defaults*, after [Rei80]. We allow agents of the form $c \leadsto A$ (to be read as "c else A"), which intuitively mean that *in the absence of* information c, 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 c be absent amounts to making an *assumption* about the future evolution of the system: not only does it not entail c now, but also it will not entail c 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.

How expressive is the resulting system? All the ESTEREL-style combinators, including "**do** `A` **watching immediately** `c`" (which we write as **do** `A` **watching** `c`) are now expressible (see Section 3.5). All the examples considered above can be represented here. (In the following **always** A is the agent that executes A at every time instant.)

Example 1.6 (Histogram, revisited) The program is:

```
histogram(A, N, B, M) ::
   B : array(1..M),
   ∀I in 1..N : (I in B[A[I]]),
   ∀I in 1..M :
     ∀S ⊆ B[I] :  S ≠ B[I] ⇝ S = B[I].
```

Intuitively, for every subset S of $B[I]$ other than the largest subset, it will be possible to establish that $S \neq B[I]$. Hence, for each `I`, the default will fire just once — for the largest subset, and will assert then that S is equal to the largest subset. For example, if the assertions `3 in B[2]`, `6 in B[2]`, `5 in B[2]` had been made, then it can be established that $\mathtt{B}[2] \neq \{\mathtt{3}, \mathtt{6}\}$. However, it cannot be established that $\mathtt{B}[2] \neq \{\mathtt{3}, \mathtt{5}, \mathtt{6}\}$. □

The compositional modeling example is similar in flavor to the Histogram problem. Assertions about the dependence of a variable V on other variables can be stated as positive pieces of information, e.g. as constraints imposing membership in the set of dependent variables of V. The associated set can then be "completed" by using defaults as above, and then decomposed as a fully-formed set to build the term (e.g. $\mathtt{f}(\mathtt{V}_1, \ldots, \mathtt{V}_n)$ to be equated to V).

Example 1.7 (Default values for variables) Consider
the program:

```
default(X, V) :: X ≠ V ⇝ X = V.
```

It establishes the value of `X` as `V` unless it can be established that the value of `X` is something other than `V`. □

Example 1.8 (Transistor model) Using defaults, we can express the transistor model as:

```
transistor(Base, Emitter, Collector) ::
   Emitter = 0v,
   Base = on→Emitter = Collector,
   default(Base, off),
   default(Collector, 5v).
```

In the absence of any information, the least reachable solution is `Collector=5v, Base=off`; however in the presence of `Base=on`, we get `Collector=0v, Base=on`. □

Example 1.9 (Default setting for vertices) In this setting it may be desirable to impose the default that the location of a vertex `V` remains unchanged, unless there is a reason to change it. This can be expressed by:

```
always ∀P.location(V) = P
          → next default(location(V), P).
```

Note that always the last value of the location will be tracked. Also note that every agent can be wrapped in a "do/watching" construct — even an **always** assertion. Thus, if it was desired to be able to "retract" the above default, all that needs to be done is to "wrap" it in a do/watching that awaits the retraction command (`let_float(V)`):

```
   do always ∀P.location(V) = P
               → next default(location(V), P)
   watching let_float(V).
```

□

Example 1.10 (Yale Shooting Problem) Various elements of this scenario can be modeled directly. Variables are introduced to correspond to objects in the situation to be modeled (possibly with time-varying state). Constraints are placed on the values that the variables may take over time. Typically, one states (using a do/watching loop) that the value of a variable is to be kept the same, unless some actions of interest take place. Actions are represented as base atomic formulas whose applicability may be contingent on the presence of some information in past stores, and whose effect is stated in terms of changes in the values of affected variables from the present moment onwards.

Thus, for example, the occurrence of a `load` action causes the gun to maintain the state of being loaded until such time as an occurrence of a `shoot` or an `unload` action:

---

always $(\mathtt{occurs}(\mathtt{load})$
$\quad\rightarrow$**do always** $\mathtt{loaded}$
$\quad\quad$**watching**$(\mathtt{occurs}(\mathtt{shoot}) \vee \mathtt{occurs}(\mathtt{unload}))))$.

The default persistence of life, and other facts, are formulated as:

---

do always $\mathtt{alive}$ **watching** $\mathtt{death}$.
always $\mathtt{occurs}(\mathtt{shoot}) \rightarrow \mathtt{loaded} \rightarrow \mathtt{death}$.
always $\mathtt{occurs}(\mathtt{shoot}) \rightarrow \neg\mathtt{loaded}$.
always $\mathtt{occurs}(\mathtt{unload}) \rightarrow \neg\mathtt{loaded}$.
always $\mathtt{death} \rightarrow$**always** $\mathtt{dead}$.

Note that the `death` event causes the fluent `dead` to be unequivocally asserted for all time to come — the state of being dead cannot be "interrupted".

Executing a program like this, in the presence of no additional information from the environment, will ensure that Fred is dead when shot at time $T = 4$. $\square$

Thus the addition of the construct $c \leadsto A$ to the language gives us a very powerful programming system. However, three central issues arise immediately.

Model. First, what should a model for such a programming language look like? The basic intuition behind our approach is as follows. An agent in CCP denotes a closure operator (a function on constraints that is idempotent, monotone and extensive) which can be represented by its range. In the presence of defaults, an agent A is taken to denote a *set of closure operators*, a different operator for each "assumption" with respect to which the defaults in A are to be resolved. That is, the denotation is a set of pairs (f, c) where f is a closure operator on the sub-lattice of constraints below c. On this space of denotations we define the combinators for conjunction (parallel composition), tell, positive ask and negative ask. Furthermore, we provide a simple operational semantics and show that the denotational semantics is natural by providing a full-abstraction theorem.

Checking for Determinacy. Second, a program may now easily have zero or more distinct evolution paths (terminating in different answers), as opposed to CCP in which there is exactly one distinct evolution path terminating in a single answer. For example, the program $X = 1 \leadsto X = 1$ (or more generally $c \leadsto c$) allows the addition of $X = 1$ in the empty store ... only to have that violate the assumption underlying its addition, namely that $X = 1$ not be entailed by the store. So this program has no evolution path. Similarly the program $(X = 1 \leadsto Y = 1) \parallel (Y = 1 \leadsto X = 1)$ has multiple evolution paths — one in which the first assumption is made, resulting in the addition of $Y = 1$ to the store (which blocks the assumption of the second default), and one in which the second assumption is made, resulting in the addition of $X = 1$ (which blocks the assumption of the first default). In reactive systems intended for embedded control, preserving system determinacy is crucial, and thus identifying determinate programs (those with exactly one distinct evolution path, on every input) is a central problem.

In Section 2.3 we present an algorithm — uniform over constraint systems — to check at compile-time whether a program is determinate or not. The key idea here is to recognize that the effect of running a program P on any input d can be simulated by running the program on one of only finitely many projections (onto the space of constraints the program can discriminate on). This, in essence, allows a finite representation of the effects of P on any input, and hence provides an algorithm for checking that every input is mapped to a single output. Unfortunately, because of the free composition of defaults allowed, such a determinacy checking algorithm cannot be compositional: determinacy is a global property of the entire program, and cannot be established by examining pieces in isolation.

Compiling programs. Third, how are programs to be implemented efficiently? A naive implementation may involve performing the actual guessing at run-time, and backtracking if the assumption about the future evolution of the system is violated dynamically. In Section 2.3 we show (extending [SJG94b]) that in fact it is possible to (compositionally) compile determinate (recursion-free) programs into finite constraint automata so that there is no guessing or backtracking involved at run-time. We are able to achieve compositionality — unlike compilers for ESTEREL and LUSTRE — by labeling the nodes of the automata with Default cc programs (for which a notion of parallel composition is already defined).

Rest of this paper. The rest of this paper contains the detailed technical development of these ideas. After a discussion of related work, we develop and explore the mathematical foundation, operational semantics, determinacy checking and compilation algorithms for Default cc, and then repeat this for Default tcc. In particular, we develop a sound and complete axiomatization for the (monotonic) logic of default cc programs. This logic can be used to establish the equivalence of two agents A and B.

1.2 Related work.

More broadly our contributions can be cast in the following general light. The integration of defaults with constraint programming is a long-standing problem for which there has been no clean mathematical or practical solution. We believe that this paper makes a basic contribution to this problem, with ramifications in non-monotonic reasoning and knowledge representation. Furthermore, from the viewpoint of the theory of (synchronous) reactive systems, the basic model we present can be adapted, with minor adjustments to provide a model for ESTEREL and LUSTRE as well — indeed Default tcc provides a setting in which ESTEREL and LUSTRE can be combined smoothly.

Non-monotonic reasoning. Our work builds on [Rei80] directly, and is related to the stable semantics model of [GL88]. To our knowledge this is the first paper that provides a compositional semantics for default logic and that mathematically connects default logic with reasoning about time-outs in reactive, synchronous programming.

There is a very large literature on non-monotonic reasoning ([GHR94] is a recent handbook on this subject), doing justice to which is not possible in the space available to us. So a few remarks will have to suffice. Our analysis seems to bring the following novel ideas to the research around non-monotonic reasoning. First, we explicitly introduce the notion of a *two-level* logical system: the program combinators provide a logical scaffolding on top of an underlying logical language of constraints. Questions of entailment and disentailment have to be decided *purely with respect to constraints*. This makes the languages far more practical than non-monotonic formalisms based directly on reasoning about entailment/disentailment in full first-order logic; since the constraint language can be chosen so that its expressiveness, and hence complexity, is appropriate for the needs of the application at hand.

Second, we explore these ideas in the context of agents embedded in an autonomous world with which they cannot control the rate of interaction. This necessarily implies that the computations that an agent can afford to perform between interactions with the external world must be limited, indeed bounded by some a priori constant. In Default tcc this means that recursion in a time instant is not allowed; consequently there is hope for compiling away default programs into a finite state machine, so that only some very simple tests have to be done at run-time.

Third, the notion of reactive computation forces us to view a default theory as a *transducer*: it must be open to the receipt of unknown new information at run-time, and must produce then an "extension" beyond that input. This emphasis on the relational nature of default deduction — also to be found in [MNR90,MNR92] — is a key idea behind our development of a denotational semantics. It forces us not to look at just what is deducible from the given theory, but ask what is deducible from the theory in the presence of new information. And in particular, it causes is to develop conditions for the determinacy of default programs.

Similarly, the desire to get a *denotational* semantics for such transducers forced us to ask the question: what aspects of the internal construction of a default theory need to be preserved in order for us to construct the denotation of a conjunctive composition from the denotation of its constituents? It forced us to develop the *internal* logic of default theories: $A \vdash B$ if any observation that can be made of B can also be made of A. This logic can be used to establish the equivalence of two default agents. To our knowledge, the development of such an inference relation between default programs is original to this paper.

The model presented in this paper smoothly enriches the model in [SJG94b] by allowing the instantaneous detection of negative information. All the results of [SJG94b] continue to hold in this richer setting; there is a straightforward embedding of (the denotations of) tcc programs into Default tcc.

Concurrent Constraint Programming. A non-monotonic framework for concurrent constraint programming has been presented in [dBKPR93]. The paper focuses on providing for general constructions for *retracting* constraints once they have been established, and for checking for disentailment. A version of existentials are worked out. The connection between this work and default logic (and its notions of extensions) and reactive programming is however, not clear. This will be the subject of future investigations.

Synchronous languages. The synchronous languages mentioned above implicitly adopt specialized forms of default reasoning for handling absence of signals: A signal is absent at a time instant if and only if it is not emitted by some process. This paper extends this view to generic constraint systems, and provides a "formal recipe" to design such languages. Our analysis breaks down the design of synchronous languages into three inter-related components: (1) details of actual synchronization mechanisms are suppressed through the entailment relation of a constraint system (2) the notion of defaults is analyzed at the level of the basic (untimed) concurrent logic language (3) the synchronous language is obtained by extending the untimed language uniformly over time. In addition, this analysis clarifies the hitherto not well understood relationship between the combinators that are present in the above languages. We show that Timed Default CCP supports the *derivation* of a "**clock**" construct that allows a process to be clocked by another (recursive) process; it generalizes the *undersampling* constructs of SIGNAL and LUSTRE, and the pre-emption/abortion constructs supported by ESTEREL[4].

2 Default Concurrent Constraint Programming

2.1 Basic model

Constraint systems. A constraint system [Sar92] is essentially a first-order system of partial information. Briefly, we may understand a constraint system $\mathcal{C}$ as coming equipped with the following data: (1) a set D of first-order formulas (closed under variable renamings, where an infinite underlying set of variables is assumed) called *primitive constraints* or *tokens*, and (2) an inference relation $\vdash_{\mathcal{C}}$ that is finitary (in that it relates finite sets of tokens to tokens), decidable and records which tokens follow from which collection of tokens.

[4] tcc exhibited this relationship for monotone/weak pre-emption/abortion constructs. Timed Default CCP generalizes this to non-monotone/instantaneous pre-emption and abortion

Let the $\vdash$-closed subsets of D be denoted by $|D|$. $(|D|, \subseteq)$ is a complete algebraic lattice; we will use the notation $\sqcup$ and $\sqcap$ for the joins and meets of this lattice, and the name `true` for the element D. A (finite) *constraint* is an element of $|D|$ generated from a (finite) set of tokens; for a set of tokens u, we will use $\overline{u}$ to denote the closure of u under $\vdash$. We usually denote finite sets of tokens by the letters a, b, and constraints by the letters c, d, e. We write $a \approx b$ if $\overline{a} = \overline{b}$. Sometimes, we will say d entails e to mean that $d \supseteq e$.

For real-time computation we have found the simple constraint system Gentzen ($\mathcal{G}$) to be very useful [SJG94a]. The tokens of Gentzen are atomic formulas drawn from a pre-specified logical vocabulary; the entailment relation is trivial, i.e. $c_1, \ldots, c_n \vdash_{\mathcal{G}} c$ iff $c = c_i$ for some i. Gentzen provides the very simple level of functionality that is needed to represent signals, e.g. as in ESTEREL and LUSTRE.

2.1.1 The model.

What should a model for defaults look like?

Consider first the definition of the model for CCP [SRP91]. The crucial insight there was to develop a very simple notion of observation: observe for each agent A those stores d in which they are quiescent, that is those stores d in which executing A does not result in the generation of any more information. Formally, define the predicate $A{\downarrow}^d$ (read: "A converges on d"). The intended interpretation is: A when executed in d does not produce any information that is not entailed by d. We then have the evident axioms for the primitive combinators:

abort There are no rules for **abort**: it does not converge on any input.

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

$$\frac{d \supset a}{a \downarrow^d}$$

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

$$\frac{d \not\supseteq a}{(a \to A) \downarrow^d} \qquad \frac{A \downarrow^d}{(a \to A) \downarrow^d}$$

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

$$\frac{A_1 \downarrow^d \quad A_2 \downarrow^d}{(A_1 \mid\mid A_2) \downarrow^d}$$

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

We can now use the denotational semantics of an agent to reason about the actual input/output behavior (the "operational semantics"): the output of an agent A on an input c is exactly the least d above c (if any) for which A converges.

Conversely, one can ask which sets of observations can be viewed as determining the denotation of a process. The answer is quite straightforward: the key idea is that from the set it should be possible to determine a unique output above every input (if the process converges). That is, the set S should have the property that above every (input) constraint c, there is a unique minimal element in S (the output). We can say this generally by requiring that S be closed under glbs of arbitrary non-empty subsets. In particular, $\emptyset$ is a process – the process which diverges on every input, i.e. $\emptyset$ is the denotation of **abort**.

We thus have an independent notion of processes, on which all the combinators of interest to us are definable. Two further questions arise: (1) Expressive completeness: are all processes definable by agents, and (2) Full abstraction: if the denotations of two agents A and B are distinct, then is there in fact a context, i.e. a third agent P with a "hole" in it, such that plugging the hole with A and B separately would produce agents with observably different behaviors? If the given language is expressively complete, then we know that every process is the denotation of some agent. If the model is fully abstract for the given language and notion of observation, then we know that the model does not make distinctions that are too fine: if the denotations of two agents are different, then there is a reason, namely, there is another agent which can be used to distinguish between the two. Together, these two nice properties imply that a logic for reasoning about processes (semantic entities) can be used to reason safely about agents and their operational behavior.

The model for CCP we motivated above is both expressively complete and fully abstract.

Adding defaults. How does the situation change in the presence of defaults?

The critical question is: how should the notion of observation be extended? Intuitively, the answer seems obvious: observe for each agent A those stores d in which they are quiescent, *given the guess e about the final result*. Note that the guess e must always be stronger than d — it must contain at least the information on which A is being tested for quiescence. Formally, we define a predicate $A \downarrow^d_e$ (read as: "A converges on d under the guess e"). The intended interpretation is: if the guess e is used to resolve defaults, then executing A in d does not produce any information not entailed by d, and executing A in e does not produce any information not entailed by e.

We then have the evident axioms for the primitive combinators:

abort There are no rules for **abort**: it does not converge on any input.

Tell The information about the guess e is not needed:

$$\frac{d \supseteq a}{a \downarrow^d_e}$$

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:

$$\frac{e \not\supseteq a}{(a \rightarrow A) \downarrow_e^d} \quad \frac{d \not\supseteq a, A \downarrow_e^e}{(a \rightarrow A) \downarrow_e^d} \quad \frac{A \downarrow_e^d}{(a \rightarrow A) \downarrow_e^d}$$

Parallel Composition Note that a guess e for $A_1 \parallel A_2$ is propagated down as the guess for A_1 and A_2:

$$\frac{A_1 \downarrow_e^d \quad A_2 \downarrow_e^d}{(A_1 \parallel A_2) \downarrow_e^d}$$

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

$$\frac{e \supseteq a}{(a \rightsquigarrow A) \downarrow_e^d} \quad \frac{A \downarrow_e^d}{(a \rightsquigarrow A) \downarrow_e^d}$$

Again, note that these axioms for the relation are "compositional": whether an agent converges on (d, e) is determined by some conditions involving whether its sub-agents converge on (d, e). This suggests taking the denotation of an agent A to be the set of all (d, e) such that $A \downarrow_e^d$; because of the axioms above, the denotation is compositional. Furthermore, we can recover the "input/output" relation exhibited by A from its denotation: the output of A on input c are exactly those d's above c such that $A \downarrow_d^d$ and there is no constraint $e \supseteq c$ distinct from d such that $A \downarrow_d^e$. That is, the result of running A on input c should be just those constraints d such that A can produce no more information than d (under the guess that d is the output), and such that there is "no place to stop" above c and strictly below d (so that d, can, in fact, be generated by A on input c).

Again, conversely, one can ask which sets S of observations can be viewed as determining the denotation of a process. The two intuitive conditions we wish to capture are the following: *Local determinacy* — the idea is that once a guess is made, every process behaves like a determinate CCP agent. This is expressed by saying that under every guess d (that is, for every d such that $(d, d) \in S$) the set of constraints on which the process is claimed to be convergent under the guess d (i.e., the set $\{c \mid (c, d) \in S\}$) should be closed under glbs. The second idea to capture is the *anti-monotonicity* of defaults—the only affect a stronger guess can have is to cause *fewer* defaults to fire, resulting in even more convergent points. This is the requirement that for every agent A, if it converges on input d under the guess e, then it is going to also converge on input d under any guess $e' \supseteq e$.

We now have the basic ideas in hand to proceed somewhat more formally. We establish the basic notion of a process, provide an operational semantics for processes, explore some properties, and show that the model is fully abstract.

The basic model.

Definition 2.1 (Observations) **SObs**, the set of simple observations is the set $\{(d, e) \in |D| \times |D| \mid e \supseteq d\}$. □

For a set S and element e, we use the notation (S, e) to stand for the set $\{(d, e) \mid d \in S\}$, and $\sqcap S$ to stand for the greatest lower bound of S.

Definition 2.2 (Process) A process P is a set of simple observations satisfying:

Guess-convergence $(e, e) \in P$ if $(d, e) \in P$

Local Determinacy $(\sqcap S, e) \in P$ if $S \neq \emptyset$ and $(S, e) \subseteq P$.

Anti-monotonicity $(c, e) \in P$ if $(c, d) \in P, d \subseteq e, (e, e) \in P$.

Processes are naturally ordered by inverse set inclusion: define $P \leq Q$ if $P \supseteq Q$. Under this ordering, **SObs** is a process, the "least" process which converges on every constraint; it is the denotation of the agent `true`. $\emptyset$ is the unique maximal process. Furthermore, the limit of every chain of processes is itself a process: that is, given a set of processes $S_1 \supseteq S_2 \ldots$, the set $\bigcap_i S_i$ denotes a process. This means that recursion can be handled in the model: the denotation of a recursive agent A is simply the intersection of the denotation of the all the agents obtained by replacing recursive calls by i-fold expansions.

We can now provide explicit semantic definition for various combinators:

$$
\begin{aligned}
\mathcal{D}[\![\textbf{abort}]\!] &\stackrel{d}{=} \emptyset \\
\mathcal{D}[\![a]\!] &\stackrel{d}{=} \{(d, e) \in \mathbf{SObs} \mid d \supseteq a\} \\
\mathcal{D}[\![a \rightarrow A]\!] &\stackrel{d}{=} \{(d, e) \in \mathbf{SObs} \mid \ e \supseteq a \Rightarrow (e, e) \in \mathcal{D}[\![A]\!], \\
& \qquad\qquad\qquad\qquad\quad d \supseteq a \Rightarrow (d, e) \in \mathcal{D}[\![A]\!]\} \\
\mathcal{D}[\![a \leadsto A]\!] &\stackrel{d}{=} \{(d, e) \in \mathbf{SObs} \mid e \not\supseteq a \Rightarrow (d, e) \in \mathcal{D}[\![A]\!]\} \\
\mathcal{D}[\![A \parallel B]\!] &\stackrel{d}{=} \mathcal{D}[\![A]\!] \cap \mathcal{D}[\![B]\!]
\end{aligned}
$$

Each of these combinators is seen to yield a process when applied to a process, and to be continuous and monotone in its process argument.

Recursion. Default cc programs are given as a set of declarations $g :: A$ along with an agent. (Here g names a procedure.) The names of the agents g can now occur in the program. We will denote a recursively defined process as $\mu X.A[X]$. As indicated above, the meaning of recursive processes is obtained in the standard way by taking least fixed points in the given complete partial order of processes.

Obtaining the operational semantics. How do we obtain the "result" of executing an agent A on an input constraint i from the denotation $\mathcal{D}[\![A]\!]$? The output is going to be all those constraints $o \supseteq i$ such that there is no place for the process to stop strictly below o:

Definition 2.3 (I/O mapping) The input-output relation $r(P)$ induced by a process P is defined by:

$$r(P) = \{(i, o) \in \mathbf{SObs} \mid (o, o) \in P, \forall (j, o) \in P.\, j \supseteq i \Rightarrow j = o\}$$

□

Note that $r(P)$ may be non-monotone, e.g. $r(\mathcal{D}[\![a \leadsto b]\!])$ is non-monotone — it maps $\overline{\emptyset}$ to $\overline{b}$ and $\overline{a}$ to $\overline{a}$. (A relation R is non-monotone iff it is not monotone. It is monotone iff $a\ R\ b$ and $a' \supseteq a$ implies there is a $b' \supseteq b$ such that $a'\ R\ b'$.)

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

$$\mathcal{D}[\![a \leadsto a]\!] = \{(d, e) \in \mathbf{SObs} \mid e \supseteq a\}$$

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 $b \leadsto b$, whereas in default logic and synchronous languages both these agents are considered the same, i.e. meaningless, and are thrown away.

$$\begin{aligned}&\mathcal{D}[\![a \rightarrow b \mid\mid a \leadsto b]\!] = \\ &\quad \{(d, e) \in \mathbf{SObs} \mid e \supseteq b, ((e \not\supseteq a) \vee (d \supseteq a)) \Rightarrow d \supseteq b\}\end{aligned}$$

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, $a \leadsto b$ is not the same as $\neg a \rightarrow b$, in the second case some agent must explicitly write $\neg a$ in the store, if $\neg a$ is a constraint, 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 $a \rightarrow b \mid\mid a \leadsto b$ in parallel with $b \rightarrow a$ — b produces $a \sqcup b$ on $true$, while $a \rightarrow b \mid\mid a \leadsto b$ produces no output.

These two examples show that designing a logic for this language is not entirely trivial. We come back to this a little later.

Expressive completeness. Given a process S, is there a finite (recursion-free) agent A such that $S = \mathcal{D}[\![A]\!]$?

We can characterize the class of processes for which this is possible — these are precisely the finite elements in the lattice of processes ($P \leq Q \Leftrightarrow P \supseteq Q$). Recall that a finite element P is one such that if $\sqcup_i P_i \geq P$, then there is a $P_i \geq P$.

Now given any process P, we can write down a (possibly infinitary) agent which has P as its denotation. For each $(e, e) \in P$, define $P_e = \{d \mid (d, e) \in P\}$. Then P_e is (the fixed-point set of) a closure operator, and can be written as an ask-tell agent. Now form the conjunction

$$\mid\mid_e (x_1^e \leadsto x_2^e \leadsto \ldots \leadsto P_e)$$

for all $e \in q(P)$, where x_i^e's are all the constraints greater than e. We also take all the constraints e which are mapped to nothing by the i/o relation, and add the agents $e \rightarrow$ **abort** to the agent. The antimonotonicity property assures us that the denotation of this agent is P.

Theorem 2.1 *If P is a finite process, then its agent can be expressed finitely. Conversely, every finite agent denotes a finite process.*

Operational Semantics A simple non-deterministic execution mechanism (operational semantics) can be provided for recursion-free Default cc by extending the operational semantics of cc computations.

We take a configuration to be simply a multiset of agents. For any configuration Γ, let $\sigma(\Gamma)$ be the subset of primitive constraints in Γ. We define binary transition relations $\longrightarrow_e$ on configurations indexed by "final" constraints e that will be used to evaluate defaults:

$$\frac{\sigma(\Gamma) \vdash a}{(\Gamma, a \rightarrow B) \longrightarrow_e (\Gamma, B)}$$

$$\frac{\sigma(\Gamma) \vdash a}{(\Gamma, a \leadsto B) \longrightarrow_e \Gamma}$$

$$\frac{e \not\vdash a}{(\Gamma, a \leadsto B) \longrightarrow_e (\Gamma, B)}$$

$$(\Gamma, A \parallel B) \longrightarrow_e (\Gamma, A, B)$$

From this family of transition relations, we may provide a definition of the operational semantics:

Definition 2.4

$$\mathcal{O}[\![A]\!] \stackrel{d}{=} \{(\overline{a}, e) \in \mathbf{SObs} \mid \exists B.(A, a) \stackrel{\star}{\rightarrow}_e B \not\rightarrow_e \text{ and } \sigma(B) \approx e\}$$

□

The "result" of running A on input a, $r_o[\![A]\!](\overline{a})$, may be extracted thus: $\{e \mid (\overline{a}, e) \in \mathcal{O}[\![A]\!]\}$.

The operational semantics described above can be used to compute the result of running the agent in a given store only if the "final store" is known beforehand. For finite agents P, we now show how this non-determinism can be bounded, and hence made effective (e.g., by backtracking).

Let $c(P)$ be the sublattice generated by the finite number of constraints that syntactically occur in P. Now, clearly, any input d can be mapped by P only to some element in $d \sqcup c(P)(\stackrel{d}{=} \{d \sqcup e \mid e \in c(P)\}$. Therefore $r_o(P)(d)$ can be computed by considering just the finitely many relations $\{\longrightarrow_e \mid e \in d \sqcup c(P)\}$.

Full abstraction. The following results establish the connections between these two characterizations:

Theorem 2.2 $\mathcal{O}[\![A]\!] = \mathcal{D}[\![A]\!]$ *and* $r_o([\![A]\!])(d) = r([\![A]\!])(d)$

Theorem 2.3 (Full abstraction for Default cc) *If* $\mathcal{D}[\![P]\!] \neq \mathcal{D}[\![Q]\!]$*, then there exists an agent* C *such that* $P \mid\mid C$ *is observationally distinct from* $Q \mid\mid C$*.*

Proof Sketch 2.3 Since the two denotations are different, suppose there is a $(d, e) \in \mathcal{D}[\![P]\!]$, but $(d, e) \notin \mathcal{D}[\![Q]\!]$. Now if $(e, e) \notin Q$, then since $(d, e) \in \mathcal{D}[\![P]\!] \Rightarrow (e, e) \in \mathcal{D}[\![P]\!]$, e is an agent separating P and Q.

Otherwise, we follow extant proofs in [SRP91,JPP91]. Consider the closure operators $P_e = \{d \mid (d, e) \in \mathcal{D}[\![P]\!]\}$ and $Q_e = \{d \mid (d, e) \in \mathcal{D}[\![Q]\!]\}$. These are unequal, so differ on some input, say a. If $P_e(a) \not\subseteq Q_e(a)$, then the agent $a \mid\mid (P_e(a) \rightarrow e)$ is the required agent: $a \mid\mid (P_e(a) \rightarrow e) \mid\mid P$ produces e, but $a \mid\mid (P_e(a) \rightarrow e) \mid\mid Q$ does not. (Note that if $P_e(a)$ and $Q_e(a)$ are not finite, there is some finite element where they differ, and this may be chosen instead.)

$\square$

Determinate processes.

Definition 2.5 (Determinate processes) A process P is said to be determinate if $r(P)$ is the graph of a total function. $\square$

How can the functions generated by determinate processes be characterized?

Closure operators are functions $f : L \rightarrow L$ that satisfy (for all $a, b \in L$):

$$a \leq fb \leftrightarrow fa \leq fb \tag{1}$$

and

$$a \leq b \rightarrow fa \leq fb \tag{2}$$

The functions generated by determinate processes will continue to satisfy Condition 1. Instead of being monotone, however, they are *locally* monotone:

$$a \leq b \leq fa \rightarrow fa \leq fb \tag{3}$$

Let us call functions that satisfy Conditions 1 and 3 local closure operators. We now show that the i/o functions associated with determinate processes are precisely local closure operators.

Definition 2.6 For $f : L \rightarrow L$ a local closure operator, define $p(f)$, the process associated with f, by:

$$p(f) \stackrel{d}{=} \{(d, e) \in \mathbf{SObs} \mid f(e) = e, (f(d) = e \rightarrow d = e)\}$$

$\square$

Roughly, $p(f)$ is the *complement* of the graph of f. It is easy to verify that $p(f)$ is a process. In fact, $p(f)$ is maximal among all processes whose i/o relation is given by f.

Theorem 2.4 *r(p(f)) = f*

In fact (r, p) form a Galois connection.

Determinate, Monotone processes How do CCP processes embed in the space of Default cc processes? It is easy to see that:

Proposition 2.5 *For any process P, $r(P)$ is the graph of a monotone function iff P satisfies the property:*

Monotonicity $(d, d) \in P$ *if* $(d, e) \in P$.

In such a case, the fixed point set of $r(P)$ is just $\{d \mid (d, d) \in P\}$. Conversely, given a closure operator f, the Default cc process corresponding to it is given by $\{(d, e) \in \mathbf{SObs} \mid d, e \in f\}$.

2.2 Logic for Default cc

In this section we consider a proof-system for recursion-free Default cc agents.

The denotational semantics for Default cc induces a natural logic, namely the logic for proving, for agents A and B that $[\![A]\!] \subseteq [\![B]\!]$. Note that this logic is necessarily a monotone logic.

The syntax of formulas in the logic is

$$(Agents) \quad A \quad ::= \quad a \mid Ma \mid a \rightarrow A \mid a \leadsto A \mid A \parallel A \tag{4}$$

Sequents are of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_k$, where the A_i, B_j are all agents, with the requirement that all except at most one of the B_j is of the form Ma, which is understood to stand for $a \leadsto a$. We say that the remaining B_j is the *non-trivial formula* of the RHS. Intuitively, a sequent is valid if every observation that can be made of system consisting of the A_i running in parallel can be made of (at least) one of the B_j. In the following, we will let Γ, Δ range over multisets of agents. $\sigma(\Gamma)$ will stand for the sub-multiset of constraints in Γ and $M^{-1}(\Gamma)$ for the multiset $\{a \mid Ma \in \Gamma\}$.

The Structural and Identity rules of inference for the logic are the rules of Exchange, Weakening and Contraction, and the Identity and Cut rules. Thus the logic is classical.

The other proof rules are:

$$\frac{\sigma(\Gamma) \vdash a}{\Gamma \vdash \Delta, a}\,(\mathrm{C}) \qquad \frac{M^{-1}(\Gamma), \sigma(\Gamma) \vdash a}{\Gamma \vdash \Delta, Ma}\,(\mathrm{M})$$

$$\frac{\Gamma \vdash a \qquad \Gamma, A \vdash \Delta}{\Gamma, a \rightarrow A \vdash \Delta}\,(L \rightarrow) \qquad \frac{\Gamma, a \vdash \Delta, A}{\Gamma \vdash \Delta, a \rightarrow A}\,(R \rightarrow)$$

$$\frac{\Gamma, Ma \vdash \Delta \qquad \Gamma, A \vdash \Delta}{\Gamma, a \leadsto A \vdash \Delta}\,(L \leadsto) \qquad \frac{\Gamma \vdash Ma, \Delta, A}{\Gamma \vdash \Delta, a \leadsto A}\,(R \leadsto)$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \parallel B \vdash \Delta}\,(L \parallel) \qquad \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \parallel B}\,(R \parallel)$$

Let $\cup[\![\Delta]\!]$ be defined as $\bigcup_{A \in \Delta}[\![A]\!]$. Note that while in general the union of two processes is not a process, the restriction that we place upon Δ — all but one of its processes is of the form Ma — ensures that it is a process. In fact, $[\![Ma]\!] \cup [\![A]\!] = [\![a \rightsquigarrow A]\!]$.

Theorem 2.6 (Soundness) $\Gamma \vdash \Delta$ *implies* $[\![\Gamma]\!] \subseteq \cup[\![\Delta]\!]$.

Proof Sketch 2.6 Straightforward. The only possibly non-trivial case is $R \rightarrow$ which is proved thus.

$$\begin{array}{ll} & (d,e) \in [\![\Gamma, a]\!] \Rightarrow (d,e) \in \cup[\![\Delta, A]\!] \\ \Longleftrightarrow & (d,e) \in [\![\Gamma]\!], d \supseteq a \Rightarrow (d,e) \in \cup[\![\Delta, A]\!] \\ \Longleftrightarrow & (d,e) \in [\![\Gamma]\!] \Rightarrow d \supseteq a \rightarrow (d,e) \in \cup[\![\Delta, A]\!] \\ \Rightarrow & (d,e) \in [\![\Gamma]\!] \Rightarrow (d,e) \in \cup[\![a \rightarrow A, \Delta]\!] \end{array}$$

The last step follows since Γ is a process, $[\![Ma]\!] \cup [\![A]\!] = [\![a \rightsquigarrow A]\!]$ and $b \rightarrow a \rightsquigarrow A = a \rightsquigarrow b \rightarrow A$. □

Completeness is proved by structural induction on the non-trivial formula B in Δ.

Theorem 2.7 (Completeness) $[\![\Gamma]\!] \subseteq \cup[\![\Delta]\!]$ *implies* $\Gamma \vdash \Delta$

Proof Sketch 2.7 Suppose the non-trivial formula $B = b$. Then the left rules are applied until they can be applied no further. Now the left side consists of $a, Ma_1, \ldots, Ma_k$, and some implications (which are ignored). Now if we cannot use (M) to prove

$$a, Ma_1, \ldots, Ma_k \vdash Mb_1, \ldots, Mb_n, b$$

then that means that for each Mb_i on the right, there must be a pair $(\overline{a}, \overline{b'_i})$ on the left such that $\overline{b'_i} \not\supseteq \overline{b_i}$. Then the pair $(\overline{a}, \sqcap\{\overline{b'_i}\})$ is also on the left, and so it must be in $[\![b]\!]$. So $\overline{a} \supseteq \overline{b}$, and we can use (C) to prove the result.

The other cases for B are straightforward. For $B = B_1 \wedge B_2$, apply $R\wedge$, and use the proof trees for B_1 and B_2. For $B = a \rightsquigarrow B'$, apply $R \rightarrow$ and use the proof tree for B'. For $B = a \rightarrow B'$, proceed as follows. By assumption every $(d,e) \in [\![\Gamma]\!]$ lies either in $[\![a \rightarrow B']\!]$ or in $[\![\Delta]\!]$. But this is means that $(d,e) \in [\![\Gamma \mid\mid a]\!]$ implies $(d,e) \in [\![B', D]\!]$, which can be established by the induction hypothesis. □

2.3 Default cc: Implementation issues

We consider now the implementation of recursion-free Default cc agents — these are the agents that will be generated by a compiler for Default tcc.

The two key issues to be resolved are: a determinacy detection algorithm, and an implementation of Default cc. Note that both these issues are resolved by the operational semantics. However, the operational semantics involves
"guessing" of defaults; thus, a priori, it is not clear that it induces a backtracking free

implementation of Default cc. Following synchronous languages, we will exploit the denotational semantics to yield an efficient implementation.

The key idea behind both aspects of the implementation is to construct a finite representation of $r(P)$ — the input-output behavior of the process P. This construction is developed formally below.

Definition 2.7 Let $\mathcal{C}$ be a constraint system. Then, $B(\mathcal{C})$ is the free Boolean Algebra over the generators (constraints occurring in $\mathcal{C}$) and relations ($c \rightarrow d = true$, if $c \vdash d$). □

The finite constraint system relevant to an agent P, denoted C_P, is the sub-Boolean algebra of $B(\mathcal{C})$ that is generated by the constraints occurring in P. Note that C_P is a finite poset. Also, since C_P has finite joins, it can be viewed as a constraint system. Furthermore, there is a natural projection function $p : \mathcal{C} \rightarrow C_P$ defined as

$$p(d) = \sqcup\{c \in C_P \mid d \Rightarrow c = true \text{ in } B(\mathcal{C})\}$$

Though we have described $p(d)$ abstractly, we note that for any $d \in \mathcal{C}$, $p(d)$ can be computed using the entailment relation of $\mathcal{C}$ (*i.e.* queries to the constraint solver) and the axioms of Boolean Algebras. In this extended abstract, we do not present these standard details.

Determine the denotation of the agent P with respect to the constraint system C_P. In turn, this denotation yields a *finite* input-output relation, denoted by $r^f(P)$. The following theorem relates $r^f(P)$ to $r(P)$ – the input-output relation of P with respect to the constraint system $\mathcal{C}$.

Theorem 2.8 (Representation theorem) $r(P)(d) = \{d \sqcup d' \mid d' \in r^f(P)(p(d))$, *for any agent* P.

The above theorem is exploited to yield a determinacy detection algorithm and a compilation algorithm for Default cc.

Determinacy detection: The determinacy of P is established by showing that $r^f(P)$ is the graph of a function.

Compilation: The relation $r^f(P)$ is computed at compile time. The execution proceeds as follows. On input d, first compute $p(d)$; next, use the relation $r^f(P)$ to determine the output on $p(d)$; next, use Theorem 2.8 to determine the output of P in d. This last step involves one more tell action on the constraint solver.

2.4 Definable combinators

We now show how to generalize the ask combinator ($c \rightarrow A$) to allow for simple process arguments. Intuitively, in $B \rightarrow A$, B evolves only if the store is a quiescent point of

A. Formally, $B \rightarrow A$ is defined as:

$$\begin{aligned}\mathcal{D}[\![B \rightarrow A]\!] \quad \stackrel{d}{=} \quad \{ \ (d,e) \in \mathbf{Obs} \mid \\ (d,e) \in \mathcal{D}[\![A]\!] \Rightarrow (d,e) \in \mathcal{D}[\![B]\!], \\ (e,e) \in \mathcal{D}[\![A]\!] \Rightarrow (e,e) \in \mathcal{D}[\![B]\!] \}\end{aligned}$$

However, $B \rightarrow A$ may not be a process for arbitrary $\mathcal{D}[\![B]\!]$. So, we will restrict the processes B to be generated by the grammar:

$$B \quad ::= \quad c \mid B \parallel B \mid c \rightarrow \textbf{abort}$$

For such processes B, $B \rightarrow A$ is indeed a process.

The extension to process arguments can be compiled away compositionally to the existing combinators of Default cc using the laws:

$$\begin{aligned}(c \rightarrow \textbf{abort}) \rightarrow A \quad &= \quad c \leadsto A \\ (B_1 \parallel B_2) \rightarrow A \quad &= \quad B_1 \rightarrow (B_2 \parallel A)\end{aligned}$$

3 Timed Default CCP

We consider now the extension of Default cc to the Timed setting.

3.1 Basic Model

First, we extend the set of observations over time:

Definition 3.1 **Obs**, the set of observations, is the set of finite sequences of simple observations. □

Intuitively, we shall observe the *quiescent sequences* of interactions for the system.

We let s, u, v range over sequences of simple observations, and let z be a simple observation. We use "ϵ" to denote the empty sequence. The concatenation of sequences is denoted by "$\cdot$"; for this purpose a simple observation z is regarded as the one-element sequence $\langle z \rangle$. Given $S \subseteq \mathbf{Obs}$ and $s \in \mathbf{Obs}$, we will write S $\mathbf{after}$ s for the set $\{z \in \mathbf{SObs} \mid s \cdot z \in S\}$ of simple observations of S in the instant after it has exhibited the observation s.

Definition 3.2 $P \subseteq \mathbf{Obs}$ is a *process* iff it satisfies the following conditions:

1. (*Non-emptiness*) $\epsilon \in P$,
2. (*Prefix-closure*) $s \in P$ whenever $s \cdot t \in P$, and
3. (*Determinacy*) P $\mathbf{after}$ s is a Default cc process whenever $s \in P$.

□

The new combinators introduced by the additional structure are:

$$\begin{array}{rcl} \mathcal{D}[\![\textbf{skip}]\!] & \stackrel{d}{=} & \mathbf{Obs} \\ \mathcal{D}[\![\textbf{abort}]\!] & \stackrel{d}{=} & \{\epsilon\} \\ \mathcal{D}[\![\textbf{next}\ B]\!] & \stackrel{d}{=} & \{\epsilon\} \cup \{z \cdot s \in \mathbf{Obs} \mid s \in \mathcal{D}[\![B]\!])\} \end{array}$$

The definitions of the other basic combinators of Default tcc are straightforward counterparts of their definitions for Default cc.

$$\begin{array}{rcl} \mathcal{D}[\![a]\!] & \stackrel{d}{=} & \{(d,e)\cdot s \in \mathbf{Obs} \mid d \supseteq a\} \\ \mathcal{D}[\![a \rightarrow A]\!] & \stackrel{d}{=} & \{(d,e)\cdot s \in \mathbf{Obs} \mid \\ & & \qquad e \supseteq a \Rightarrow (e,e)\cdot s \in \mathcal{D}[\![A]\!], \\ & & \qquad d \supseteq a \Rightarrow (d,e)\cdot s \in \mathcal{D}[\![A]\!]\} \\ \mathcal{D}[\![a \leadsto A]\!] & \stackrel{d}{=} & \{(d,e)\cdot s \in \mathbf{Obs} \mid e \not\supseteq a \Rightarrow (d,e)\cdot s \in \mathcal{D}[\![A]\!]\} \\ \mathcal{D}[\![A \mid\mid B]\!] & \stackrel{d}{=} & \mathcal{D}[\![A]\!] \cap \mathcal{D}[\![B]\!] \end{array}$$

Guarded Recursion. Default tcc programs are given as a set of declarations $g :: A$ along with an agent. (Here g names a parameterless procedure.) The names of the agents g can now occur in the program, the only restriction being that they occur within the scope of a **next** . This is necessary to make the computation in each step lexically bounded, and also gives us unique solutions for recursive equations. We will write a recursively defined agent as $\mu X.A[X]$.

3.2 Operational semantics

The operational semantics for Default tcc is just like the operational semantics for tcc except that a Default cc agent is executed at every time step rather than a cc agent. Further details are provided by the automata construction in Section 3.4.

The proof of full abstraction for Default tcc follows essentially immediately from the proof for Default cc:

Theorem 3.1 (Full abstraction for Default tcc**)** *If* $\mathcal{D}[\![P]\!] \neq \mathcal{D}[\![Q]\!]$, *then there exists a context* C *such that* $P \mid\mid C$ *is observationally distinct from* $Q \mid\mid C$.

3.3 Determinacy detection for Default tcc.

From the compilation algorithm described below, it suffices to describe an algorithm to check the determinacy of a finite recursion free Default cc agent —– a Default tcc agent is determinate iff the Default cc agents at each node of the compiled automaton are determinate.

3.4 Compilation

Default constraint automata The automata construction for Default tcc is similar to the construction for tcc provided in [SJG94b]. A Default cc automaton is specified by the following data (1) a set of states Q, with each state $q \in Q$ labeled with a Default cc agent (2) a distinguished start state, and (3) a set of directed edges between pairs of states, labeled with constraints. The set of labels will be drawn from the constraints of the finite sublattice of constraints occurring in the agent. The automaton will satisfy the property that for every node the set of labels on outgoing edges are closed under least upper bounds (lubs).

The execution is as follows — The automaton starts in the start state. Upon receiving an input i, it executes its Default cc agent P in conjunction with the input, and the output, $r(P)(i)$ is the output for this time instant. (The Default cc agent can be executed as described in Section 2.2.) The edge labeled with the greatest constraint less than $r(P)(i)$ is then taken to a new state, where this process is repeated.

In order to prove the finiteness of the number of states, we need the notion of a derivative of an agent. Given a process P, a derivative of P is a process $\{t \in \mathbf{Obs} \mid s \cdot t \in \mathcal{D}[\![P]\!]\}$ — this is the residual process after P has produced the sequence of observations s. The finiteness of the number of states of the automaton is then guaranteed by the following theorem.

Theorem 3.2 *Every* Default tcc *agent has a finite number of derivatives.*

Each state now corresponds with one derivative, which predicts the entire future of the process.

Compilation algorithm. Following synchronous languages, Theorem 3.2 induces a non-compositional compilation of Default tcc agents. However, Default tcc admits a compositional compilation as well. We sketch below the automaton construction for parallel composition and $a \leadsto P$, the other cases are simple and hence omitted.

Automaton for $P_1 \parallel P_2$. This is a variant of the classical product construction on automata. We are given the Default cc automaton for P_1 and P_2, say A_1 and A_2 respectively. The states of the automaton for $P_1 \parallel P_2$ are induced by pairs of states q_1, q_2 from A_1, A_2. We will call the induced state $\langle q_1, q_2 \rangle$. The start state corresponds to the pair of start states. The Default cc agent in $\langle q_1, q_2 \rangle$ is the parallel composition of the agents in the q_i's.

Now transitions are induced by the following rule — if on output a, there is a transition from q_1 to q_1' in A_1, and also on output a there is a transition from q_2 to q_2' in A_2, then we get a transition on a from $\langle q_1, q_2 \rangle$ to $\langle q_1', q_2' \rangle$. In order to determine all the possible transitions, we take all the a's in the finite sublattice of the constraint system generated by the constraints in agents in q_1 and q_2.

Automaton for $a \leadsto P$. The automaton for $a \leadsto P$ is derived from A, the automaton for P. We make a copy q_0' of the start state q_0 of A, and label it with the Default cc agent $a \leadsto q$, where q was the Default cc agent labeling q_0. For each transition from q_0 to q_1 labeled by d, we create a transition from q_0' to q_1, *if* $d \not\geq a$. We also create a transition from q_0' to a dead state and label it a. The rest of the automaton for $a \leadsto P$ is a copy of the automaton of P.

3.5 Definable Combinators

We present **clock** a powerful derived combinator, and show how to define a number of other common patterns of temporal activity in terms of it.

The clock combinator. **clock** B **do** A is a process that executes A only on those instants which are quiescent points of B. It is the extension of the Default cc construct $B \rightarrow A$ over time. Clearly, it is in the flavor of the `when` construct (undersampling) in LUSTRE and SIGNAL, generalizd to general processes B instead of boolean streams.

Let P be a process. We identify the maximal subsequence t_P of the sequence t that is an element of the process P. t_P is defined inductively as follows.

$$\begin{aligned} \epsilon_P &= \epsilon \\ (s \cdot (d,e))_P &= \begin{cases} (s_P) \cdot (d,e), \text{ if } (d,e) \in (P \textbf{ after } s_P) \\ (s_P), \text{ otherwise} \end{cases} \end{aligned}$$

Now, recognizing that A is executed only at the quiescent points of B we can state:

$$\textbf{clock } B \textbf{ do } A \quad \stackrel{d}{=} \quad \{t \in \mathbf{Obs} \mid t_{\mathcal{D}[\![B]\!]} \in \mathcal{D}[\![A]\!]\}$$

However, $B \rightarrow A$ may not be a process for arbitrary $\mathcal{D}[\![B]\!]$. So, we will restrict the processes B to be generated by the grammar:

$$\begin{aligned} B \quad ::= \quad & a \mid B \parallel B \mid a \rightarrow \textbf{abort} \\ & \mid a \rightarrow \textbf{next } B \mid a \leadsto \textbf{next } B \mid \mu X.B[X] \end{aligned}$$

For such processes B, $B \rightarrow A$ is indeed a process.

The laws that allow us to eliminate occurrences of the **clock** construct are given in Table 1.

Expressiveness. We now show how various primitive combinators in ESTEREL and other languages can be defined on top of **clock**. We use the following abbreviations. **always** A executes A repeatedly; it is the agent $\mu g.A \parallel \textbf{next } g$. **whenever** a **do** A executes A at the first instant at which a is entailed; it is the agent $\mu g.(a \rightarrow A) \parallel (a \leadsto \textbf{next } g)$. (Note that **whenever** a **do** A = **clock** a **do** A.)

The following laws hold for the **clock** combinator:

$$\begin{array}{rcl}
\textbf{clock}\ a\ \textbf{do}\ A & = & a \rightarrow A \parallel \\
 & & a \rightsquigarrow \textbf{next clock}\ a\ \textbf{do}\ A \\
\textbf{clock}\ a \rightarrow \textbf{abort do}\ A & = & a \rightsquigarrow A \\
\textbf{clock}\ (B_1 \parallel B_2)\ \textbf{do}\ A & = & \textbf{clock}\ B_1\ \textbf{do}\ (\textbf{clock}\ B_2\ \textbf{do}\ A) \\
\textbf{clock}\ a \rightarrow \textbf{next}\ B\ \textbf{do}\ A & = & a \rightarrow \textbf{clock next}\ B\ \textbf{do}\ A \parallel \\
 & & a \rightsquigarrow A \\
\textbf{clock}\ a \rightsquigarrow \textbf{next}\ B\ \textbf{do}\ A & = & a \rightsquigarrow \textbf{clock next}\ B\ \textbf{do}\ A \parallel \\
 & & a \rightarrow A
\end{array}$$

For $P =$ **next** B, we do a case analysis on A:

$$\begin{array}{rcl}
\textbf{clock next}\ B\ \textbf{do abort} & = & \textbf{abort} \\
\textbf{clock next}\ B\ \textbf{do skip} & = & \textbf{skip} \\
\textbf{clock next}\ B\ \textbf{do}\ b & = & b \\
\textbf{clock next}\ B\ \textbf{do}\ (A_1 \parallel A_2) & = & (\textbf{clock next}\ B\ \textbf{do}\ A_1) \parallel \\
 & & (\textbf{clock next}\ B\ \textbf{do}\ A_2) \\
\textbf{clock next}\ B\ \textbf{do}\ (b \rightarrow A) & = & b \rightarrow \textbf{clock next}\ B\ \textbf{do}\ A \\
\textbf{clock next}\ B\ \textbf{do}\ (b \rightsquigarrow A) & = & b \rightsquigarrow \textbf{clock next}\ B\ \textbf{do}\ A \\
\textbf{clock next}\ B\ \textbf{do}\ (\textbf{next}\ A) & = & \textbf{next clock}\ B\ \textbf{do}\ A
\end{array}$$

Recursion in either argument is now done by expanding the code — we expand $\mu X.A[X]$ to $A[\mu X.A[X]]$, and the same for B, and then apply the laws above.

Table 1: Laws for **clock**

Multiform time: time A on a. **time** A **on** c 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:

$$\textbf{time } A \textbf{ on } a = \textbf{clock } (\textbf{always } a) \textbf{ do } A$$

Watchdogs: do A watching a. This is an interrupt primitive related to *strong abortion* in ESTEREL [Ber93]. **do** A **watching** a behaves like A till a time instant when a is entailed; when a is entailed A is killed instantaneously. (We can similarly define the related *exception handler* primitive, **do** A **watching** a **timeout** B, that also activates a handler B when A is killed.) Using **clock** this is definable as:

$$\textbf{do } A \textbf{ watching } a = \textbf{clock } (\textbf{whenever } a \textbf{ do abort}) \textbf{ do } A$$

Suspension-Activation primitive: $\mathbf{S}_a\mathbf{A}_b(A)$. This is a preemption primitive that is a variant of *weak suspension* in ESTEREL [Ber93]. $\mathbf{S}_a\mathbf{A}_b(A)$ behaves like A till a time instant when a is entailed; when a is entailed A is suspended from the next time instant onwards (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:

$$\mathbf{S}_a\mathbf{A}_b(A) = \textbf{clock } (\textbf{whenever } a \textbf{ do next } b) \textbf{ do } A$$

Compiling clock. The laws given in Table 1 provide one way of removing the clock construct from the top level. However it is possible to directly compile an automaton for **clock** B **do** A given the automata for A and B. The construction is similar to the product construction described above. The states of the automaton are given by the Cartesian product of the states of the automata for A and B. If p_1 is the agent labeling a state q_1 in A, and p_2 labels q_2 in B, then the label for $\langle q_1, q_2\rangle$ is the Default cc agent $p_2 \rightarrow p_1$. The syntax for the processes B ensures that the Default cc process $p_2 \rightarrow p_1$ is well-defined; *i.e.* p_2 satisfies the conditions in Section 2.4.

Transitions from the state $\langle q_1, q_2\rangle$ are given as follows — consider all the a's in the finite sublattice of the constraints occurring in p_1, p_2. If (a, a) is not in the denotation of p_2, then there is a transition back to the state $\langle q_1, q_2\rangle$. If (a, a) is in the denotation of p_2, there is a transition on a from $\langle q_1, q_2\rangle$ to $\langle q_1', q_2'\rangle$ — where, on a, the automaton for A goes to q_1' and B goes to q_2',

4 Conclusion and acknowledgements

This paper has used ideas from non-monotonic reasoning to extend real-time languages with a coherent, mathematically tenable notion of interrupts. The topic has been developed using the methodology of concurrency theory and denotational semantics of programming languages: the construction of a model, and the definition of a language and a process algebra on the model, and the definition of a logic for reasoning about substitutability of programs in the language. From our perspective, this synthesis of ideas is long overdue. Fundamentally the fields of Qualitative Physics, Reasoning about action and state change, reactive real-time computing and hybrid systems, and concurrent programming languages are about the same subject matter: the representation, design and analysis of (at least partially computational) continuous and discrete dynamical systems. We look forward to further developments in this very rich area.

The very simple compositional semantics for default logic opens up several possibilities. It is now possible to develop coherent notions of timed default logic, and possibly hybrid default logic, for talking about action and change for systems involving continuous and discrete values.

Existentials. Another avenue for future work is to enrich the model to allow for the definition of first-order existentials (hiding). Somewhat surprisingly, hiding is not definable in the current model. Intuitively, the process $X\hat{}A$ is supposed to behave like the process $A[Y/X]$, where Y is some new variable distinct from any variable occurring in the environment.

The reason is simple. The union of two processes is not a process. Therefore, the "internal choice" (or "blind" choice) combinator $A \sqcap B$ of Hoare is not expressible in the model. Intuitively, $A \sqcap B$ is expected to behave like either A or B, and the choice cannot be influenced by the environment.

Hiding, can, however, mimic internal choice, in the presence of defaults. To illustrate, consider the process $A \stackrel{d}{=} (X = 1 \leadsto Y = 1, X = 2) \mid\mid (X = 2 \leadsto Z = 1, X = 1)$. 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 $X\hat{}A$. Consequently, one would expect $X\hat{}A$ to exhibit the observation $(Y = 1, (Y = 1, Z = 1))$ and $(Z = 1, (Y = 1, Z = 1))$. If $X\hat{}A$ is to be a process however, it be locally determinate: it must also exhibit the glb of these two observations, namely $(true, (Y = 1, Z = 1))$. However, it cannot do that, since it must either produce $Y = 1$ or produce $Z = 1$. Thus, $X\hat{}A$ cannot be a process.

A pathway for resolving this problem seems clear: one must move to a richer model where in fact local determinacy is not required and such hidden choices can be expressed. Similar ideas have been worked out in [SRP91] around the semantics of the indeterminate cc languages (which support blind choice). We expect to elaborate such a model in future work.

Future work. The use of concurrent constraint programming as a basis for a synchronous language provides a natural setting for the combination of the combinators of Esterel and Lustre. The development of **clock**, a general strong preemption construct, should lead to a theory of strong preemption, which could not have been developed in tcc.

The underlying logical basis for the model of the present paper needs to be explored further. This should lead to the development of an intuitionistic version of temporal default logic.

Acknowledgements. We gratefully acknowledge discussions with Gerard Berry, George Gonthier, Johan de Kleer, Danny Bobrow and Markus Fromherz. We thank Peter Bigot for comments on earlier version of the paper, and the POPL referees for surprisingly detailed reviews.

Work on this paper has been supported in part by ONR through grants to Vijay Saraswat and to Radha Jagadeesan, and by NASA.

References

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

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

[Bor79] Alan Borning. *THINGLAB– A constraint oriented simulation laboratory*. PhD thesis, Stanford, 1979. Also published as Xerox PARC Report SSL-79-3, July 1979.

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

[dBKPR93] Frank S. de Boer, Joost N. Kok, Catuscia Palamidessi, and Jan J.M.M. Rutten. *Logic Programming — Proceedings of the 1993 International Symposium*, chapter Non-monotonic Concurrent Constraint Programming. MIT Press, 1993.

[For88] Ken Forbus. *Exploring Artificial Intelligence*, chapter Qualitative Physics: Past, Present and Future, pages 239–296. AAAI and Morgan Kaufmann, 1988.

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

[GHR94] Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol 3: Nonmonotonic Reasoning and Uncertain Reasoning*. Oxford Science Publications, 1994.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, August 1988.

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

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

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

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

[JPP91] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully-abstract semantics for a functional language with logic variables. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[Kac93] Hassan Ait Kaci. An introduction to LIFE— Programming with Logic, Inheritance, Functions and Equations. In Dale Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*. MIT Press, 1993.

[MNR90] W. Marek, A. Nerode, and J. Remmel. A theory of non-monotonic rule systems – i. *Annals of Mathematics and Artificial Intelligence*, 1:241 – 273, 1990.

[MNR92] W. Marek, A. Nerode, and J. Remmel. A theory of non-monotonic rule systems – ii. *Annals of Mathematics and Artificial Intelligence*, 5:229 – 264, 1992.

[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.

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

[Sho88] Yoav Shoham. Chronological ignorance: Experiments in nonmonotonic temporal reasoning. *Artificial Intelligence*, 36:279 – 331, 1988.

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

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

[SJG94b] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Press, July 1994.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, October 1990.

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