J. Symbolic Computation (1996) **11**, 1–000

Timed Default Concurrent Constraint Programming

VIJAY SARASWAT[1], RADHA JAGADEESAN[2] AND VINEET GUPTA[1]

[1] *Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto Ca 94304*
[2] *Dept. of Mathematical Sciences, Loyola University-Lake Shore Campus, Chicago, Il 60626*

(Received 12 June 1996)

Abstract

Synchronous programming (Berry (1989)) is a powerful approach to programming reactive systems. Following the idea that "processes are relations extended over time" (Abramsky (1993)), we propose a simple but powerful model for timed, determinate computation, extending the closure-operator model for untimed concurrent constraint programming (CCP). In (Saraswat *et al.* 1994a) we had proposed a model for this called tcc— here we extend the model of tcc to express strong time-outs: 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 synchronous programming languages.

The fundamental conceptual difficulty posed by these operations is that they are nonmonotonic. We provide a compositional semantics to the non-monotonic version of concurrent constraint programming (Default cc) obtained by changing the underlying logic from intuitionistic logic to Reiter's default logic. This allows us to use the same construction (uniform extension through time) to develop Timed Default cc as we had used to develop tcc from cc. Indeed the smooth embedding of cc processes into Default cc processes lifts to a smooth embedding of tcc processes into Timed Default cc processes.

We identify a basic set of combinators (that constitute the Timed Default cc programming framework), and provide a constructive operational semantics (implemented by us as an interpreter) for which the model is fully abstract. We show that the model is expressive by defining combinators from the synchronous languages. We show that Timed Default cc is compositional and supports the properties of multiform time, orthogonal preemption and executable specifications. In addition, Timed Default cc programs can be read as logical formulas (in an intuitionistic temporal logic) — we show that this logic is sound and complete for reasoning about (in)equivalence of Timed Default cc programs.

Like the synchronous languages, Timed Default cc programs can be compiled into finite state automata. In addition, the translation can be specified compositionally. This enables separate compilation of Timed Default cc programs and run-time tradeoffs between partial compilation and interpretation.

A preliminary version of this paper was published as (Saraswat *et al.* 1995). Here we present a complete treatment of hiding, along with a detailed treatment of the model.

Keywords:

Programming paradigms — constraint programming, reactive systems, synchronous programming; Formal approaches — denotational semantics, semantics of concurrency, default logic.

0747–7171/90/000000 + 00 $03.00/0

1. Introduction and Motivation

Reactive systems (Harel & Pnueli (1985),Berry (1989),Halbwachs (1993)) are those that react continuously with their environment at a rate controlled by the environment. Execution in a reactive system proceeds in bursts of activity. In each phase, the environment stimulates the system with an input, obtains a response in bounded time, and may then be inactive (with respect to the system) for an arbitrary period of time before initiating the next burst. Examples of reactive systems are controllers and signal-processing systems.

This paper proposes a simple model for determinate reactive systems, and provides a language to describe processes in this model. The intended application of such languages forces them to satisfy the following criteria:

Declarative view: There must be a logical view of the language. We consider this essential in allowing (1) programs to be written using a vocabulary and concepts appropriate to the application domains of interest, (2) programs to be read and understood independently of the details of the implementation, and (3) tools to be developed for directly reasoning with programs. These advantages of a dual operational and logical view are by now well known, e.g. Kahn (1974) for concurrency, logic programming, Berry's "What you prove is what you execute" principle (Berry (1989)), executable intermediate representations for compilers (Pingali *et al.* 1991) etc.

Modularity: The language should support hierarchical and modular construction of programs/specifications. This is tantamount to demanding an *algebra* of programs/specifications that includes concurrency and preemption – the ability to stop a "black-box" process (of unknown internal construction) in its tracks.

Determinacy: Determinate programs/specifications are easier to construct and analyze. So the language should not impose indeterminacy; in the worst case it should be possible to perform compile-time checks to guarantee determinacy.

Executability: The language should be "real-time realizable", that is, the programs should have bounded response time.

Ability to detect negative information instantaneously: To detect negative information is to detect 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).

Our contribution. A re-analysis of the elegant ideas underlying synchronous programming, starting from the viewpoint of asynchronous computation leads us to *timed default concurrent constraint programming*, henceforth called Timed Default cc. This approach has the following salient features.

Declarative view: Timed Default cc has a fully-abstract semantics based on solutions of equations. Timed Default cc programs can be viewed as formulas in an intuitionist linear time temporal logic, extended with defaults, whose models are precisely executions of the program.

Modularity: In the spirit of process algebra, we identify a set of basic combinators, from which programs and reasoning principles are built compositionally.

Expressiveness : Timed Default cc supports the *derivation* of a number of preemption based control constructs, such as `time` A `on` c, which provides to A only those clock ticks which contain at least the information c. These constructs are related to the *undersampling* constructs of SIGNAL and LUSTRE, and the preemption/abortion constructs supported by ESTEREL. Thus, Timed Default cc encapsulates the rudiments of a theory of preemption constructs. In addition, Timed Default cc inherits the ability to specify cyclic, dynamically-changing networks of processes from concurrent constraint programming (cf. "mobility" (Milner *et al.* 1989)).

Executability : Timed Default cc programs have an operational semantics that is concretely realized in a working prototype interpreter-based implementation, which we have used to develop several programs for typical synchronous programming problems. Programs can be compiled into automata, and we can analyze these to guarantee the bounded response time property which is necessary for real time applications.

Defaults for negative information: Borrowing ideas from default logic (Reiter (1980)), the combinator **if** c **else** A is introduced — it allows the agent A to execute if the information in c is *not* known to be true on quiescence.

Of these the last addresses perhaps the most technically tricky concept in synchronous programming. In the next subsection we survey the problem in more depth. Subsequently we motivate and outline informally the nature of our solution for this problem — using the notion of defaults from Reiter (1980) — and compare related work. The bulk of the paper develops the formal model for timed default concurrent constraint programming, and studies its properties.

1.1. The problem of negative information

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 (cc) (Saraswat (1993),(Saraswat *et al.* 1991)). 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 — (Saraswat *et al.* 1990),(Hentenryck *et al.* 1992),Janson & Haridi (1991), (Smolka *et al.* 1994),Kaci (1993), 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.† 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. □

Example 1.2 (Composition of model fragments) Similar examples arise when using such languages for compositional modeling of physical systems (see, e.g. Forbus (1988)). 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, 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 `current` = `f`(`voltage_drop`, `conductance`). □

Such detection of quiescence is inherently nonmonotonic: 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 (Saraswat *et al.* 1994a).

`tcc` arises from combining `cc` with work on the synchronous languages (Berry & Gonthier (1992), (Halbwachs *et al.* 1991), (Guernic *et al.* 1991), Harel (1987), (Clarke *et al.* 1991)). 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 `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

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

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 cc (namely, tell (a), ask (**if** a **then** A), hiding (**new** X **in** A) and parallel composition (A_1, A_2)), tcc adds unit delay (**next**), and *delayed negative ask* (**if** a **else next** A). **if** a **else next** A allows A to be executed at the *next* time instant if the store on quiescence is not strong enough to entail a. 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 cc over (discrete) time. Indeed, many complex patterns of temporal behavior — such as the "**do** A **watching** a" construct of ESTEREL, which allows the agent A to execute, aborting it at the time instant after a 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" preemption 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 setting for programming 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, an 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 assumptions and the input information, the agent decides to act (send messages to the outside world) and revise 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 (Berry (1993)): 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** `a`" construct of ESTEREL: the execution of `A` is interrupted *as soon as* `a` is detected (rather than one step later). Weak timeouts 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 preemption may simply fail to work.

Example 1.3 (Modeling a transistor) 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. Reiter (1980)).

Example 1.4 (Constraint-based User Interfaces) Consider a system such as THINGLAB (Borning (1979)), 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, Shoham (1988)) The scenario to be modeled is this: a gun is loaded at time $T = 2$. It is fired at Fred at time $T = 4$. Meanwhile, 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. □

1.2. 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 not expressible in the cc framework, which is monotone. We now extend cc to allow for non-monotonic processes, and integrate them into a reactive real-time programming framework.

The fundamental move we now make is to allow the expression of *defaults*, after Reiter (1980). 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.

How expressive is the resulting system? All the ESTEREL-style combinators, including "**do** A **watching immediately** a" (which we write as **do** A **watching** a) are now expressible (see Section 4.2). All the examples considered above can be represented here.

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] : if S ≠ B[I] else 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 `B[2]` $\neq \{3,6\}$. However, it cannot be established that `B[2]` $\neq \{3,5,6\}$.

□

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. `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) :: if X ≠ V else 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 this as:

```
transistor(Base, Emitter, Collector) ::
  Emitter = 0v,
  if Base = on then 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 as follows. Here **always** A is the agent that executes A at every time instant.

```
always ∀P.if location(V) = P
           then 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.if location(V) = P
              then 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 (if occurs(load)
          then do always loaded
               watching(occurs(shoot) ∨ occurs(unload)))).
```

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

```
do always alive watching death.
always if (occurs(shoot), loaded) then death.
always if occurs(shoot) then ¬loaded.
always if occurs(unload) then ¬loaded.
always if death then always 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 $T = 4$. □

Thus the addition of the construct **if** c **else** 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 cc 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 cc in which there is exactly one distinct evolution path terminating in a single answer. For example, the program **if** $X = 1$ **else** $X = 1$ 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 (**if** $X = 1$ **else** $Y = 1$), (**if** $Y = 1$ **else** $X = 1$) 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 3.7 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 b 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 3.7 we show (extending (Saraswat *et al.* 1994a)) that in fact it is possible to (compositionally) compile 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 Timed Default cc. 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.

This paper is a fuller, more expanded version of (Saraswat *et al.* 1995).

1.3. 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 present one such solution, 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 Timed Default cc provides a setting in which Esterel and Lustre can be combined smoothly.

Other notations for reactive programming. This paper provides a model and notation for reactive systems. Traditionally, the most natural way of programming such systems is in terms of automata with simple transitions, to ensure bounded response. However, automata do not have hierarchical or parallel structure; in particular, small and succinct changes in the specification can lead to global changes in the automaton (Murakami & Sethi (1990)). *Process calculi* (Hoare (1985),Milner (1989), (Milner *et al.* 1989)) support parallel composition and communication/synchronization via rendezvous. However, these calculi do not specify the "occurrence time" of the rendezvous. Consequently, program execution is inherently indeterminate. Furthermore, this results in inadequate support for preemption, which is not integrated into the calculi. *Temporal logic programming languages* (Brzoska (1991),(Barringer *et al.* 1990), Baudinet (1989),Moszkowski (1986),Merz (1993)) achieve bounded response by imposing syntactic restrictions — for example, by identifying *a priori*, global and fixed notions of "system-variables" and "environment-variables" to ensure true reactivity. This paradigm is also nondeterministic. Futhermore, motivated by process algebra, we desire a more algebraic view of processes and combinators. We believe that our treatment of negative information through defaults is novel in this setting.

Nonmonotonic reasoning. Our work builds on Reiter (1980) directly, and is related to the stable semantics model of Gelfond & Lifschitz (1988). However, we provide a compositional semantics for default logic and mathematically connect default logic with reasoning about time-outs in reactive, synchronous programming.

There is a very large literature on nonmonotonic reasoning ((Gabbay *et al.* 1994), Marek & Truszczynski (1993) are recent books 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 nonmonotonic reasoning. First, we explicitly introduce the notion of a *two-level* logical system: the program combinators provide a scaffolding on top of an underlying logical language of constraints, similar to the synchronous languages. Questions of entailment and disentailment have to be decided *purely with respect to constraints*. This makes the languages far more practical than nonmonotonic formalisms based directly on reasoning about entail-

ment/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 Timed Default cc 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 (Marek *et al.* 1990),(Marek *et al.* 1992) — 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 us 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 A can also be made of B. 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 enriches the model in (Saraswat *et al.* 1994a) by allowing the instantaneous detection of negative information. All the results of (Saraswat *et al.* 1994a) continue to hold; there is a straightforward embedding of (the denotations of) tcc programs into Timed Default cc.

Concurrent Constraint Programming. A nonmonotonic framework for concurrent constraint programming has been presented in (de Boer *et al.* 1993). 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 The 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 (timed) synchronous language is obtained by extending the untimed language uniformly over time.

We show that Timed Default cc supports the *derivation* of the preemption constructs found in synchronous programming languages.

2. Concurrent Constraint Programming

Concurrent Constraint Programming[†](cc) replaces the traditional notion of a store as a valuation of variables with the notion of a store as a constraint on the possible values of variables. The store consists of pieces of information that restrict the possible values of the variables. A program consists of a set of agents running concurrently. Agents act in two basic ways — they can add information to the store (*tell*) or they can query the store about the validity of some information (*ask*). Computation is monotonic — information can only be added to the store. Tell actions take place immediately. Ask actions are used for synchronization — if a query is answered positively, then the agent can proceed, otherwise it waits (possibly forever) until there is enough information in the store to entail the information in the query.

2.1. Constraint systems.

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

Definition 2.1 A *constraint system* is a structure $\langle D, \vdash, Var, \{\exists_X \mid X \in Var\}\rangle$ 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 Var is an infinite set of *variables*, such that for each variable $X \in Var$, $\exists_X : D \to D$ 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 technical development of concurrent constraint programming can be found in (Saraswat *et al.* 1991).

□

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 $\texttt{true} = \{a \mid \forall b \in D.\ b \vdash a\}$ and greatest element $\texttt{false} = D$. Reverse inclusion is written $\supseteq$. $\exists, \vdash$ lift to operations on constraints. Examples of such systems are the system Herbrand (underlying logic programming), FD (Hentenryck *et al.* 1992), and Gentzen (Saraswat *et al.* 1994b).

Example 2.1 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 2.2 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 2.3 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 $b_i = \texttt{true}$ otherwise. □

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

Model for cc. The model for cc (Saraswat *et al.* 1991) 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. Formally, 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:

$$\frac{a \notin u}{(\textbf{if } a \textbf{ then } A) \downarrow^u} \qquad \frac{A \downarrow^u}{(\textbf{if } a \textbf{ then } A) \downarrow^u}$$

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

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

Hiding Information about the variable X is local to A.

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

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 $A \downarrow^u$; 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 a is exactly the least b above a (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 a, 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.

Program equivalence. We thus have an independent notion of processes, on which all the combinators of interest to us are definable. One further question arises — 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 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. This property implies that a logic for reasoning about processes (semantic entities) can be used to reason safely about agents and their operational behavior. The model for cc we presented above is fully abstract.

3. Default Concurrent Constraint Programming

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 u in which they are quiescent, *given the guess v about the final result.* Note that the guess v must always be stronger than u — it must contain at least the information on which A is being tested for quiescence. 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:

$$\frac{a \not\in v}{(\textbf{if } a \textbf{ then } A) \downarrow_v^u} \quad \frac{a \not\in u, A \downarrow_v^v}{(\textbf{if } a \textbf{ then } A) \downarrow_v^u} \quad \frac{A \downarrow_v^u}{(\textbf{if } a \textbf{ then } A) \downarrow_v^u}$$

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

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

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, and we shall come back to it later.

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. Furthermore, we can recover the "input/output" relation exhibited by A from its denotation: the outputs of A on input a are exactly those b's above a such that $A \downarrow_{\overline{b}}^{\overline{b}}$ and there is no constraint v, $a \in v$ distinct from $\overline{b}$ such that $A \downarrow_{\overline{b}}^{v}$. That is, the result of running A on input a should be just those tokens b such that A can produce no more information than b (under the guess that b is the output), and such that there is "no place to stop" above a and strictly below b (so that b, can, in fact, be generated by A on input a). Note that due to indeterminacy, there may be several such b's.

3.1. The basic model.

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.

Definition 3.1 (Observations) **DObs**, the set of simple observations is the set $\{(u, v) \in |D| \times |D| \mid v \supseteq u\}$. □

A process is a collection of observations that satisfy the following intuitive conditions:

1. Guess convergence — 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. This is expressed by saying that under every guess v (that is, for every v such that $(v, v) \in S$) the set of constraints on which the process is claimed to be convergent under the guess v (i.e., the set $\{u \mid (u, v) \in S\}$) should be closed under glbs.

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

Definition 3.2 (Process) A process $P \subseteq \mathbf{DObs}$ satisfying:

Guess-convergence $(v, v) \in P$ if $(u, v) \in P$.
Local Determinacy $(\sqcap S, v) \in P$ if $S \neq \emptyset$ and $(S, v) \subseteq P$.

□

We can now provide the denotational definitions for the combinators. The information about the guess is not needed for the tell or ask combinator. The definition for the parallel composition follows cc — note that a guess v for A, B is propagated down as the guess for A and B. However, note the crucial use of the guess/default in the definition for **if** a **else** A — the guess is used to determine if A is initiated.

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

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. Note that we do not have recursion — we will implement recursion in Timed Default cc across time steps.

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

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

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

† $A \sqcap B$ behaves like either A or B, and the choice cannot be influenced by the environment

1 Recall that the variable X is local to Z. Thus, default assumptions (guesses) about the variable X must be reasonable, *i.e.* there must be some evolution of Z that generates the default assumptions on X — restricting Z to such defaults gives us a subset of Z, call it Z_1.
2 Identify the "maximal determinate subprocess" of Z_1 — call it Z_2. This eliminates the possibility of locally indeterminate processes, as was the case above. We provide (see Appendix A) a sufficient condition on processes that are not affected by this step — this is the class of determinate processes and it includes all processes that we are interested in.
3 Finally, we follow intuitions from cc to obtain the definition. Consider the behavior of **new** X **in** A on an input a. a may constrain X; however this X is the "external" X which the process must not see. Hence, to obtain the behavior on a, we should observe the behavior on $\exists_X a$. However, the result, say b, may constrain X, and this X is the "internal" X. Therefore, the result seen by the environment must be $a \sqcup \exists_X b$.

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

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

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

Define a process Z to be X-determinate if in the above definition $Z_1 = Z_2$. We will later provide a sufficient criterion for showing that a process is X-determinate.

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

Example 3.1

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

This is an example of a default theory which does not have any extensions (Reiter (1980)). 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 3.2

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

This 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. □

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

3.2. Input-output behavior

How do we obtain the "result" of executing an agent A on an input token a from the denotation $\mathcal{P}[\![A]\!]$? The output is going to be all those tokens b where $b \vdash a$ and there is no place for the process to stop strictly below b.

Definition 3.3 (I/O mapping) The input-output relation $r(P)$ induced by a process P is defined by:

$$r(P) = \left\{(a,b) \in D \times D \mid b \vdash a, (\overline{b},\overline{b}) \in P, (\forall u)(u,\overline{b}) \in P.\ a \in u \Rightarrow u = \overline{b}\right\}$$

□

The relation $r(P)$ may be easily extended to constraints. Note that $r(P)$ may be non-monotone, e.g. $r(\mathcal{P}[\![\textbf{if } a \textbf{ else } b]\!])$ is non-monotone — it maps $\emptyset$ to b and a to a. (A relation R is non-monotone iff it is not monotone. It is monotone iff $a\ R\ b$ and $a' \vdash a$ implies there is a $b' \vdash b$ such that $a'\ R\ b'$.)

Example 3.3 In the program **if** a **else** a, there is no possible output above the input $true$. On the other hand, there can be multiple outputs — both a and b are possible outputs on input $true$ for the program **if** a **else** b, **if** b **else** a. □

This leads us to the definition of determinacy.

Definition 3.4 A process Z is called determinate if its input-output behavior is a total function with domain D. □

For controlling reactive systems, it is necessary that the programs be determinate. Later in section 3.7 we will provide an algorithm to characterize the indeterminate programs. This semantic notion of indeterminacy captures the essence of the "causality cycles" in synchronous programming languages. In the appendix, we also show that a determinate program is X–determinate for all variables X.

3.3. 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 a multiset of agents. For any configuration $?$, let $\sigma(?)$ be the subset of tell tokens in $?$. We define binary transition relations $\longrightarrow_b$ on configurations indexed by "final" guesses b that will be used to evaluate defaults:

$$\frac{\sigma(?) \vdash a}{?, \textbf{if } a \textbf{ then } B \longrightarrow_b ?, B} \qquad \frac{b \not\vdash a}{?, \textbf{if } a \textbf{ else } B \longrightarrow_b ?, B}$$

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

$$?, (A, B) \longrightarrow_b ?, A, B$$

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, this non-determinism can be bounded, and hence made effective (e.g., by backtracking). The output of executing A on input a can now be described as

$$r_o(A)(a) = \left\{ b \in D \mid \exists b' \in D \ (A, a) \longrightarrow_{b'}^{*} ? \not\longrightarrow_{b'}, \ b' = \sigma(?), \exists_{\vec{Y}} b' \approx b \right\}$$

Here $\vec{Y}$ are the new local variables in $\sigma(?')$ introduced during the derivation (In the rest of this paper we will not mention this again, whenever we write $\exists_{\vec{Y}}$, $\vec{Y}$ will always stand for these new variables). We will show later that it is the same as the i/o relation given by the denotation. Once again, the transition relation and the relation r_o can be generalized to constraints.

Implementation. The operational semantics can be implemented in a straightforward way using backtracking. For each default **if** a **else** A there are two possibilities — either a is going to be true at the end, or it will not be, in which case we execute A. We have written an interpreter for Default cc in Sicstus Prolog, using the Gentzen constraint system. The interpreter chooses one of the two possibilities for each default, and proceeds. When all defaults have been processed, it checks to make sure that all the assumptions were correct — if so, the answer is output, otherwise, it backtracks to choose an alternative assumption.

Here we present a trace of the transistor program described in the introduction. **forall** C **do** A is a shorthand for a conjunction of agents, those which are produced by substituting all possible values of C, though of course in any program we need to test only finitely many. We have shown the output for two scenarios — one with current flowing to the base, and the other with no current. For more details about the implementation, see the Hybrid cc implementation in (Gupta *et al.*), which is quite similar.

```
% Resolve a default: if X is not bound to something term-unequal
% to V, bind it to V.
default(X, V) :: forall X:Y do (if (Y\==V) else {X:V}).

transistor(Base, Emitter, Collector) ::
    [ {Emitter:0},
      if Base:on then forall  Emitter:Y do {Collector:Y}),
      default(Base, off),
      default(Collector, 5) ].

-----------------------------------------------------

dtg ?- transistor(b,e,c).
e:0 b:off c:5 b:off
```

```
 Store: [ b:off, c:5, e:0]

-----------------------------------------------------
dtg ?- {b:on},transistor(b,e,c).
b:on e:0 c:0
Termination at T=0.
 Store: [ c:0, e:0, b:on]
```

3.4. Correspondence Theorems

We show that the denotational semantics and the transition relation are equivalent. The following lemma identifies the key properties of the transition relation.

Lemma 3.1. *The relation* $\longrightarrow_v$ *satisfies:*

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

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

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

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

Proof. All properties are based on the observation that $\longrightarrow_v$ is (essentially) the reduction relation in cc languages. □

From the family of transition relations, we may provide a definition of the operational semantics (obtained by extending the transition relation to constraints:

Definition 3.5

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

□

The following results establish the connections between these two characterizations.

Lemma 3.2.

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

Proof. The proof follows extant proofs (Saraswat *et al.* 1991),(Jagadeesan *et al.* 1991) for languages in the cc paradigm, and is presented in Appendix A. □

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

THEOREM 3.3. *For any program A which satisfies the X–determinacy condition for all X,*

1 $\mathcal{P}[\![A]\!] = \mathcal{O}[\![A]\!]$.
2 $r(\mathcal{O}[\![A]\!]) = r_o(A)$

PROOF. A simple structural induction using the above lemma 3.2 yields (1). (2) follows as a corollary — a detailed proof is in Appendix A. □

THEOREM 3.4. (FULL ABSTRACTION FOR Default cc) *If $\mathcal{P}[\![P]\!] \neq \mathcal{P}[\![Q]\!]$ and P and Q satisfy the X–determinacy condition for all variables X, then there exists an agent C such that P, C is observationally distinct from Q, C.*

3.5. Default cc IS CONSERVATIVE OVER cc

Note that programs written in the cc syntax can be interpreted as Default cc programs. In this section, we clarify how cc processes embed in the space of Default cc processes. We show that Default cc is in fact conservative over cc.

This result exploits a characterization of a large class of "monotone" processes — intuitively, this class captures the processes that do not exploit the ability to detect negative information. Note that the following characterization is semantic.

LEMMA 3.5. *For any determinate process P, $r(P)$ 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 $(u, v') \in P$.

PROOF. The proof is a routine manipulation of the denotational semantics and is postponed to Appendix A. □

THEOREM 3.6. Default cc *is conservative over* cc.

PROOF. For a P satisfying the conditions of the above lemma, the fixed point set of $r(P)$ is just $\{u \mid (u, u) \in P\}$. Conversely, given a closure operator f, the Default cc process corresponding to it is given by $\{(u, v) \in \mathbf{DObs} \mid u, v \in f\}$. Note that this satisfies both the properties given above, and thus is monotone. □

The above conservativity result is further reinforced by the logic for Default cc that we discuss next.

3.6. LOGIC FOR Default cc

In this section we consider a proof-system for Default cc agents without hiding. We encourage the reader to compare this proof system with the proof system for cc.

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

The syntax of formulas in the logic is

$$A \quad ::= \quad a \mid Ma \mid \textbf{if } a \textbf{ then } A \mid \textbf{if } a \textbf{ else } A \mid A, A \tag{3.1}$$

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. Ma is understood to stand for **if** a **else** a — it corresponds to the Ma of Default logic (Reiter (1980)). 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 a 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 $?, \Delta$ range over multisets of agents. $\sigma(?)$ will stand for the sub-multiset of tell tokens in $?$ and $M^{-1}(?)$ for the multiset $\{a \mid Ma \in ?\}$.

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 as follows, they are obtained by interpreting A, B as $A \wedge B$, **if** a **then** A as $a \rightarrow A$ and **if** a **else** A as $Ma \vee A$.

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

$$\frac{? \vdash a \qquad ?, A \vdash \Delta}{?, \textbf{if } a \textbf{ then } A \vdash \Delta}\ (Lthen) \qquad \frac{?, a \vdash \Delta, A}{? \vdash \Delta, \textbf{if } a \textbf{ then } A}\ (Rthen)$$

$$\frac{?, Ma \vdash \Delta \qquad ?, A \vdash \Delta}{?, \textbf{if } a \textbf{ else } A \vdash \Delta}\ (Lelse) \qquad \frac{? \vdash Ma, \Delta, A}{? \vdash \Delta, \textbf{if } a \textbf{ else } A}\ (Relse)$$

$$\frac{?, A, B \vdash \Delta}{?, (A, B) \vdash \Delta}\ (Lpar) \qquad \frac{? \vdash \Delta, A \qquad ? \vdash \Delta, B}{? \vdash \Delta, (A, B)}\ (Rpar)$$

Let $\cup[\![\Delta]\!]$ be defined as $\bigcup_{A \in \Delta} \mathcal{P}[\![A]\!]$. 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 be of the form Ma — ensures that it is a process. In fact, $\mathcal{P}[\![Ma]\!] \cup \mathcal{P}[\![A]\!] = \mathcal{P}[\![\textbf{if } a \textbf{ else } A]\!]$.

THEOREM 3.7. (SOUNDNESS AND COMPLETENESS) *Let $?, \Delta$ be multisets of hiding free agents. Then,*

$$? \vdash \Delta \Leftrightarrow \mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta]\!]$$

PROOF. The proofs are routine structural inductions, and follow extant proofs for cc. The proof is presented in detail in Appendix A. The reason for excluding the hiding combinator is the slight mismatch between hiding and logical existentials — this is characteristic of cc languages (Saraswat *et al.* 1991). □

3.7. Default cc: COMPILATION AND DETERMINACY

We consider now the implementation of Default cc agents. 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 algorithm proceeds in the following two steps.

1 Move all the hiding constructs to the top level. This can be done via the following equations, in each case using renaming to avoid capture.

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

Let the program now be of the form $P = \textbf{new } \vec{Y} \textbf{ in } A$. In the appendix, we show that if A is determinate, then P is determinate; similarly, the input-output behavior of P is easily computed in terms of the input-output behavior of A.

2 The next step is to construct a finite representation of $r(A)$ — the input-output behavior of the process A. This construction is developed formally below. Intuitively, a program A refers to only finitely many constraints, any other piece of input is just added to the output without making any difference to the execution. So our algorithm consists of running the program on all such "relevant" inputs, and making sure it is determinate on them. This can be done at compile time. In fact the input-output values for these inputs can be stored at compile time in the form of a table, and at runtime, execution reduces to a table lookup, enabling fast execution.

We now formalize the second step of the above algorithm.

Definition 3.6 Let $\mathcal{C}$ be a constraint system, with C its set of tokens. Then, $B(C)$ is the free Complete Atomic Boolean Algebra (CABA) over the generators C and relations $a \rightarrow b = \texttt{true}$, if $a \vdash b$. □

The free CABA can be generated from the constraint system as follows — consider the set of tokens as a join semilattice with the information ordering ($a \geq b$ iff $a \vdash b$). Now the CABA is the powerset of the set of proper filters of this lattice. Each token in C is embedded in the CABA as the set of filters containing it. An entailment relation on the CABA can be defined as follows: $a \vdash_{B(\mathcal{C})} b$ iff $a \supseteq b$. Conjunction is set union, and disjunction is set intersection. Complements are given by set complement. We denote the embedding of the token $a \in C$ in $B(C)$ as $B(a)$. The empty set of filters corresponds to the element `true`. Clearly, if $a \vdash b$, then every filter that contains b contains a, so we have $a \vdash_{B(\mathcal{C})} b$. On the other hand, if $a \nvdash b$ then there must be a filter containing b that does not have a — such a filter could be the set of elements which are not below b. So $a \nvdash_{B(\mathcal{C})} b$. Thus the entailment relation induced by the CABA conservatively extends the entailment relation of the constraint system. All least upper bounds in $\mathcal{C}$ are also automatically preserved by $B(\mathcal{C})$.

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two constraint systems, such that $C_1 \subseteq C_2$ and the entailment relation of $\mathcal{C}_2$ is a conservative extension of the relation of $\mathcal{C}_1$. Given $u \in |C_2|$, we define its projection over $\mathcal{C}_1$ as $\pi(u) = \overline{u \cap C_1}$.

Definition 3.7 Given two constraint systems $\mathcal{C}_1$ and $\mathcal{C}_2$, we say that $\mathcal{C}_2$ *extends* $\mathcal{C}_1$ if for all $u \in |C_2|$, we have $u \vdash_2 \pi(u)$. □

Note that if C_1 is a sublattice of C_2, then $\mathcal{C}_2$ extends $\mathcal{C}_1$. In this case we have $\pi(u) = u \cap C_1$. $B(\mathcal{C})$ extends $\mathcal{C}$ — the condition $\forall u \in |C_2|.u \vdash_2 \pi(u)$ follows from the fact that least upper bounds of $\mathcal{C}$ are preserved by $B(\mathcal{C})$.

LEMMA 3.8. *If P is a hiding-free program over the constraint system $\mathcal{C}_1$ with denotation $\mathcal{P}[\![P]\!]_1$, $\mathcal{C}_2$ extends $\mathcal{C}_1$, and the denotation of P over $\mathcal{C}_2$ is $\mathcal{P}[\![P]\!]_2$, then for all $u, v \in C_2$ we have $(u, v) \in \mathcal{P}[\![P]\!]_2$ iff $(\pi(u), \pi(v)) \in \mathcal{P}[\![P]\!]_1$.*

PROOF. The proof proceeds by a routine structural induction and is presented in Appendix A. □

The set of finite constraints relevant to an agent P, denoted $B(\mathcal{C}_P)$, is the sub-Boolean algebra of $B(\mathcal{C})$ that is generated by the constraints occurring in P. Note that $B(C_P)$ is a finite poset. Since the entailment relation of $\mathcal{C}_P$ is derived from that of $\mathcal{C}$, $B(\mathcal{C})$ extends $B(\mathcal{C}_P)$. By the above lemma, the denotation of P over $B(\mathcal{C})$ can be computed from the (finite) denotation over $B(\mathcal{C}_P)$.

This finite 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 $\mathcal{C}$.

LEMMA 3.9. (REPRESENTATION THEOREM) *If P is a hiding-free program, then $r(P)(a) = \{a \sqcup a' \mid a' \in r^f(P)(\pi(B(a)))\}$, where $B(a)$ is the embedding of a in $B(C)$.*

PROOF. The detailed proof is presented in Appendix A. The proof exploits the following crucial fact:

$$\forall u \in |B(C_P)|, v \in |B(C)|.\pi(u \sqcup v) = u \sqcup \pi(v)$$

Indeed the "free" construction of the CABA and C_P above is set up precisely to achieve the above property. □

The determinacy of P is established by showing that $r^f(P)$ is the graph of a function. Note that this is a conservative test — this test may say that P is indeterminate even if P over $\mathcal{C}$ is determinate — this is because $\mathcal{C}$ may not have enough tokens to produce the indeterminate behavior of P. However, if the test says that a program is determinate, then it is determinate in any constraint system where it is definable.

While we have described $\pi : B(\mathcal{C}) \to C_P$ abstractly, we note that for any $a \in C$, $\pi(a)$ can be computed using the entailment relation of $\mathcal{C}$ (*i.e.* queries to the constraint solver) and the axioms of Boolean Algebras.

Compilation. The relation $r^f(P)$ is computed at compile time. The execution proceeds as follows. On input a, first compute $\pi(B(a))$; next, use the relation $r^f(P)$ to determine the output on $\pi(B(a))$; next, use Lemma 3.9 to determine the output of P in a. This last step involves one more tell action on the constraint solver. Finally the local variables are hidden from the output.

4. Timed Default cc — Timed Default concurrent constraint programming

Timed Default cc arises from Default cc by the integration of a notion of time. The motivation for this integration is the ability to model, describe and program behaviors of

reactive systems, which, as defined earlier, are those that react to inputs, but otherwise stay dormant.

Our modeling philosophy is based on the intuitions underlying synchronous programming (Berry (1993),Halbwachs (1993),Benveniste & Berry (1991b))— as captured in synchronous programming languages (Berry & Gonthier (1992),(Halbwachs *et al.* 1991), (Guernic *et al.* 1991),Harel (1987),(Clarke *et al.* 1991)). Thus, we expect our model to satisfy the features characteristic of the above languages. In particular, we expect the notion of time in Timed Default cc to be *multiform* — any signal can serve as the notion of time.

We describe Timed Default cc as an "extension" of Default cc over discrete time. This construction is roughly analagous to the definition of discrete linear time temporal logic from ordinary classical logic. Concretely, we add to the untimed Default cc a single temporal control construct: **hence** A. Declaratively, **hence** A imposes the constraints of A at every time instant after the current one. Operationally, if **hence** A is invoked at time t, a new copy of A is invoked at each instant in $t' > t$.

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

Intuitively, **hence** might appear to be a very specialized construct, since it requires repetition of the *same* program at every subsequent time instant. However, **hence** can combine with positive and negative ask operations to yield rich patterns of temporal evolution. Later in this section, we demonstrate the power of the language by exhibiting several defined combinators. The key idea that we exploit is that negative asks allow the instantaneous preemption of a program — for example, a program **hence if** a **else** A will in fact not execute A at all those time instants at which a is true.

4.1. Denotational Model

Notation. We will be working with sequences, *i.e.* partial functions on the natural numbers — their domains will be initial segments of the natural numbers of the form $0..n$. We let s, t and its variations, $s', s'', \ldots$ denote sequences, whereas z will always represent a sequence of length 1. We use "ϵ" to denote the empty sequence. The concatenation of sequences is denoted by "$\cdot$"; for this purpose a singleton k is regarded as the one-element sequence $\langle k \rangle$. Given a subset of sequences S, and a sequence s, we will write S **after** s for the set $\{t \in \mathbf{SObs} \mid s \cdot t \in S\}$. The length of s is denoted by $|s|$, while $s(n)$ denotes its nth element. We also define $S(0) = \{z \mid z \cdot s \in S\}$.

For any sequence s and $n \leq |s|$, we define the restricted sequence $s^n = \langle s(0), s(1), \ldots, s(n-1) \rangle$, the sequence consisting of the first n elements of s. s^n is a prefix of s — ϵ is a prefix of all sequences.

We will use π_1 and π_2 for the first and second projection on pairs.

In the rest of this section we will assume that we are working in some constraint system $\langle D, \vdash, Var, \{\exists_X \mid X \in Var\} \rangle$.

Observations. We are going to identify observations with "runs of the system" — a tracing of the reactive system trajectory over time. Thus intuitively, we are observing the *quiescent sequences* of interactions for the system. Our observations will satisfy the following criteria.

Since we are going to model the programs executing at any instant by Default cc programs, the observation at any given instant of time is going to be an observation of Default cc — *i.e.* a pair of constraints (u, v), such that $u \subseteq v$. A run of a system is then, as a first approximation, a sequence of Default cc observations.

Note however that this first approximation fails to capture a fundamental property of execution in Default cc — namely that the input-output behaviour of a Default cc process is a subset of observations of the form (v, v). To put it another way, the observations of the form $(u, v), u \neq v$ in the denotation of a Default cc process are essential for the compositional description of Default cc processes, but *do not* appear in the input-output behavior of the process, as defined in Definition 3.3. The second condition, *observability*, in the following definition accommodates this intuition — all but the last element of an observation must be of the form (v, v).

Definition 4.1 An observation s of Timed Default cc satisfies:

1 s is a sequence of Default cc observations.
2 $\forall i < |s| - 1.\pi_1(s(i)) = \pi_2(s(i))$.

TDccObs is the the set of all Timed Default cc observations. □

A process is a collection of observations that satisfies

1 Prefix closure property of computational systems — the future cannot undo the past.
2 Instantaneous execution at any time instant is modeled by a Default cc process.

Definition 4.2 $P \subseteq \mathbf{TDccObs}$ 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 (*Point execution*) $(P \textbf{ after } s)(0)$ is a Default cc process whenever $s \in P$.

□

Combinators of Timed Default cc**.** The agents c, **if** a **then** A, **if** a **else** A and (A, B) are inherited from Default cc and their denotations are induced by their Default cc definitions.

$$\mathcal{D}[\![a]\!] \stackrel{d}{=} \{\epsilon\} \cup \{(u,v) \cdot s \in \mathbf{TDccObs} \mid a \in u\}$$

$$\mathcal{D}[\![\textbf{if } a \textbf{ then } A]\!] \stackrel{d}{=} \{\epsilon\} \cup \{(u,v) \cdot s \in \mathbf{TDccObs} \mid a \in v \Rightarrow (v,v) \in \mathcal{D}[\![A]\!], \; a \in u \Rightarrow (u,v) \cdot s \in \mathcal{D}[\![A]\!]\}$$

$$\mathcal{D}[\![\textbf{if } a \textbf{ else } A]\!] \stackrel{d}{=} \{\epsilon\} \cup \{(u,v) \cdot s \in \mathbf{TDccObs} \mid a \notin v \Rightarrow (u,v) \cdot s \in \mathcal{D}[\![A]\!]\}$$

$$\mathcal{D}[\![A, B]\!] \stackrel{d}{=} \mathcal{D}[\![A]\!] \cap \mathcal{D}[\![B]\!]$$

new X **in** A imposes the constraints of A, but hides the variable X from the other programs. Every observation $s \in \mathcal{D}[\![\textbf{new } X \textbf{ in } A]\!]$ is induced by an observation $s' \in \mathcal{D}[\![A]\!]$, *i.e.* at every time instant t, $s(t)$ must equal the result of hiding X in the Default cc process given by A at time t after history s^{t-1}. Formally, let $\exists_X s = \exists_X s'$ denote $|s| = |s'|$, and $\forall i < |s|, \forall j \in \{1,2\}.\exists_X \pi_j(s(i)) = \exists_X \pi_j(s'(i))$. Then

$$\mathcal{D}[\![\textbf{new } X \textbf{ in } A]\!] \stackrel{d}{=} \left\{ s \in \textbf{TDccObs} \mid \exists s' \in \mathcal{D}[\![A]\!].\exists_X s = \exists_X s', \; \forall n < |s| \left[s(n) \in \textbf{new } X \textbf{ in } \mathcal{D}[\![(A \textbf{ after } (s')^n)(0)]\!] \right] \right\}$$

The new combinator introduced by the additional structure is **hence** . The definition for **hence** is as expected — observations have to "satisfy" A everywhere after the first instant.

$$\mathcal{D}[\![\textbf{hence } B]\!] \stackrel{d}{=} \left\{ z \cdot s \in \textbf{TDccObs} \mid (\forall s_1, s_2) s = s_1 \cdot s_2 \Rightarrow \; s_2 \in \mathcal{D}[\![B]\!] \right\}$$

Equational laws. The above combinators satisfy the following equational laws, which are provided here to give an intuition for their behavior.

The combinators commute with parallel composition.

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

The order of conditions does not matter.

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

Finally, variables do not carry information across time.

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

4.2. Examples of definable combinators

We now show how various primitive combinators in Esterel and other languages can be defined in Timed Default cc. We will provide a fairly general methodology for constructing a variety of combinators that can manipulate time. All the combinators given below can be defined — the equational laws that they satisfy can be used to remove the defined combinators. As an illustration of reasoning with denotations, the appendix B contains the proofs of some of these equational laws.

Example 4.1 We can define the **next** A combinator that we had introduced as a primitive in (Saraswat *et al.* 1995) in terms of **hence** A as follows —

$$\mathcal{D}[\![\textbf{next } B]\!] \stackrel{d}{=} \{\epsilon\} \cup \left\{ z \cdot s \in \textbf{TDccObs} \mid s \in \mathcal{D}[\![B]\!]) \right\}$$

Now by substituting the definitions, we will show in the appendix that

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

□

Example 4.2 A useful variant of **hence** A is **always** $A \stackrel{d}{=} A, \textbf{hence}\ A$, it simply starts a new copy of A every time, instead of from the next time instant. □

Example 4.3 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 4.4 Another useful combinator is **first** a **then** B, which starts the process B at the first time instant that a becomes true. It can be defined as

$$\mathcal{D}[\![\textbf{first}\ a\ \textbf{then}\ B]\!] \stackrel{d}{=} \{s \cdot s' \in \textbf{TDccObs} \mid \begin{array}{l} \forall i < |s|, a \notin \pi_1(s(i)), \\ |s'| > 0 \Rightarrow (a \in \pi_1(s'(0)), s' \in \mathcal{D}[\![B]\!])\} \end{array}$$

We can express it in terms of the basic combinators as

$$\textbf{first}\ a\ \textbf{then}\ B = \textbf{new}\ \texttt{stop}\ \textbf{in always}\ [\ \textbf{if}\ \texttt{stop}\ \textbf{else if}\ a\ \textbf{then}\ B \quad \textbf{if}\ a\ \textbf{then hence}\ \texttt{stop}]$$

This program keeps on executing **if** a **then** A, unless it receives the signal `stop`. The `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 (Gupta *et al.*), this will also be the case for the other definitions given below. □

Example 4.5 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 $s \in \textbf{TDccObs}$, 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:

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

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

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

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

□

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 `go`. Now another process is set up to generate `go` whenever one wants A to proceed. The next few examples illustrate this.

Example 4.6 **do** A **watching** a is an interrupt primitive related to *strong abortion* in ESTEREL (Berry (1993)). **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:

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

□

Example 4.7 There is a related weak abortion construct (Berry (1993)) —**do** A **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

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

Note that the signal `stop` is generated from the time instant *after* a is seen. □

Example 4.8 The Suspension-Activation primitive, $\mathbf{S}_a\mathbf{A}_b(A)$, is a preemption primitive that is a variant of *suspension* in ESTEREL (Berry (1993)). $\mathbf{S}_a\mathbf{A}_b(A)$ 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 (`control` $-$ `Z`, `fg`) is a construct in this vein. This can be expressed as:

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

□

4.3. INPUT-OUTPUT BEHAVIOR.

Given a process Z, we can define its input-output behavior after a history s as the input-output behaviour of the Default cc process $(Z \textbf{ after } s)(0)$. This leads us to the definition of determinacy of Timed Default cc processes.

Definition 4.3 A process Z is called determinate if

$$(\forall s \in Z)\ [(Z \textbf{ after } s)(0) \text{ is determinate}]$$

□

The determinacy detection algorithm for Default cc lifts to a determinacy detection algorithm for Timed Default cc — this will become clear in later sections. We extend the definition of an X–determinate process to Timed Default cc as usual — a process Z is X–determinate if for all s in Z, $(Z$ **after** $s)(0)$ is X–determinate.

We extend the Default cc input-ouput definition to get the input-output relation for a Timed Default cc process — given finite sequences of constraints s and s', and $(s,s) = ((s(1),s(1)),\ldots(s(n),s(n)))$, where $|s| = n$:

$$\begin{aligned} rt(Z)(\epsilon) &= \{\epsilon\} \\ rt(Z)(s'\cdot a) &= \left\{s\cdot b \mid s \in rt(Z)(s'), b \in r((Z \textbf{ after } (s,s))(0))(a)\right\} \end{aligned}$$

4.4. Operational semantics

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

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

A configuration consists of a pair — the agents currently active, and the "continuation" — the program to be executed at subsequent times. The rules are given as:

$$\frac{\sigma(?) \vdash a}{((?, \textbf{if } a \textbf{ then } B), \Delta \longrightarrow_b ((?, B), \Delta)} \qquad \frac{b \nvdash a}{((?, \textbf{if } a \textbf{ else } B), \Delta) \longrightarrow_b ((?, B), \Delta)}$$

$$((?, (A, B)), \Delta) \longrightarrow_b ((?, A, B), \Delta) \qquad ((?, \textbf{hence } A), \Delta) \longrightarrow_b (?, (A, \textbf{hence } A, \Delta))$$

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

Now the transition for making a time step, $\rightsquigarrow$, is derived from the rule for termination of Default cc—

$$\frac{\exists b \in D \quad (?, \phi) \longrightarrow_b^* (?', \Delta) \not\longrightarrow_b \quad \sigma(?') = b}{? \rightsquigarrow \textbf{new } \vec{Y} \textbf{ in } \Delta}$$

The output, as before, at each time step is $\exists_{\vec{Y}}\sigma(?')$.

The operational semantics gives an input-output relation — analogous to the definition of $\mathcal{O}[\![P]\!]$, we define $rt_o(P)(s)$, the observed input output relation —

$$rt_o(P)(s) = \{s' \mid |s'| = |s| = n, P \stackrel{d}{=} P_0, \forall i < n.(P_i, s(i)) \rightsquigarrow P_{i+1}, \text{ output at step } i \text{ is } s'(i)\}$$

Implementation The operational semantics is realizable, and has been implemented as an interpreter on top of Sictus Prolog. We present here a Timed Default cc program for the Yale shooting problem, described in the introduction, and its trace.

At the end of each time instant, the program outputs the constraints in the store. For each time instant we use the untimed Default cc interpreter described earlier. The scenario for which the trace is shown is when the gun is loaded at time 1, and is fired at time 3. Thus initially we get `alive`, and at time 3, the store contains `death`. The shooting also unloads the gun, so we get `notloaded` at time 4. After that nothing happens, and we get `dead` forever. Since Timed Default cc is a reactive language, successive time ticks can be caused by feeding an external clock signal — in this case, the interpreter simply starts computing for the next phase as soon as one is over.

```
yale :: [
    always (if occurs_load then
        do
           (do (always {loaded}) watching_delay occurs_shoot)
        watching occurs_unload),
     do (always {alive}) watching death,
     always (if occurs_shoot then if loaded then {death}),
     always (if occurs_shoot then next {notloaded}),
     always (if occurs_unload then next {notloaded}),
     always (if death then always {dead}) ].

problem :: [ next {occurs_load}, next next next {occurs_shoot}, yale].

-----------------------------------------------------------
dtg ?- problem.
---------Time = 0-------
alive
---------Time = 1-------
occurs_load loaded alive
---------Time = 2-------
loaded alive
---------Time = 3-------
loaded occurs_shoot death dead
---------Time = 4-------
notloaded dead
---------Time = 5-------
dead
---------Time = 6-------
dead
Prolog interruption (h for help)? a

{Execution aborted}
```

4.5. Correspondence theorems

We will now establish the equivalence between the operational and denotational semantics. For any Timed Default cc process A, define the set of observable sequences of A inductively as

1. ϵ is an observable sequence of A.
2. $(u, v) \cdot s$ is an observable sequence of A if
   (a) $(A, u) \longrightarrow^*_{v'} A' \not\longrightarrow_{v'}, \exists_{\vec{Y}} \sigma(A') = u, \exists_{\vec{Y}} v' = v$
   (b) $(A, v) \longrightarrow^*_{v'} A'' \not\longrightarrow_{v'}, \sigma(A'') = v'$
   (c) s is an observable sequence of **new** $\vec{Y}$ **in** A'.

Now we can define $\mathcal{O}[\![A]\!] \stackrel{d}{=} \{f \in \mathbf{TDccObs} \mid f \text{ is an observable sequence of } A\}$. Analogous to lemma 3.2, we have the following:

LEMMA 4.1.

$$\begin{array}{rcl}
\mathcal{O}[\![a]\!] & = & \mathcal{D}[\![a]\!] \\
\mathcal{O}[\![A,B]\!] & = & \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!] \\
\mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!] & = & \{\epsilon\} \cup \{(u,v)\cdot s \in \textbf{TDccObs} \mid \begin{array}{l} a \in v \Rightarrow (v,v) \in \mathcal{O}[\![A]\!], \\ a \in u \Rightarrow (u,v)\cdot s \in \mathcal{O}[\![A]\!]\} \end{array} \\
\mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] & = & \{\epsilon\} \cup \{(u,v)\cdot s \in \textbf{DObs} \mid a \notin v \Rightarrow (u,v)\cdot s \in \mathcal{O}[\![A]\!]\} \\
\mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] & = & \{s \in \textbf{TDccObs} \mid \exists s' \in \mathcal{O}[\![A]\!].\exists_X s = \exists_X s', \\
 & & \qquad \forall n < |s|[s(n) \in \textbf{new } X \textbf{ in } \mathcal{O}[\![(A \textbf{ after } s'^n)(0)]\!]]\}, \\
 & & \qquad\qquad \textit{if } \mathcal{O}[\![A]\!] \textit{ is } X\textit{-determinate} \\
\mathcal{O}[\![\textbf{hence } A]\!] & = & \{\epsilon\} \cup \{z\cdot s \in \textbf{TDccObs} \mid (\forall s_1,s_2) s = s_1\cdot s_2 \Rightarrow s_2 \in \mathcal{O}[\![A]\!]\}
\end{array}$$

PROOF. The proof follows the proof for Default cc for all cases except the last. For the case for **hence** , the proof proceeds by induction on the length of the sequence. This proof is described in Appendix B. □

THEOREM 4.2. (FULL ABSTRACTION FOR Timed Default cc) *The folloing are true for all programs P and Q which satisfy the X–determinacy condition for all variables X.*

1. $\mathcal{O}[\![P]\!] = \mathcal{D}[\![P]\!]$.
2. *If $\mathcal{D}[\![P]\!] \neq \mathcal{D}[\![Q]\!]$, then there exists a context C such that P, C is observationally distinct from Q, C.*
3. $rt(\mathcal{O}[\![P]\!]) = rt_o(P)$.

PROOF. The first part follows by a routine structural induction using the above lemma. Now full abstraction is proved by merely "lifting" the full abstraction proof for Default cc, the detailed proof is given in Appendix B. The input-output correspondence follows by induction on the length of input sequences. □

4.6. LOGIC FOR Timed Default cc.

The proof system for Timed Default cc can be derived from the proof system for Default cc. The logic is induced once again by the denotational semantics, so $A \vdash B$ iff $\mathcal{D}[\![A]\!] \subseteq \mathcal{D}[\![B]\!]$.

Since each of the Default cc combinators produce an effect at the current time instant only, all the Default cc rules given above are also valid for the Timed Default cc logic. We need two new rules for **hence** A. The first rule allows us to consider only observable sequences, while the second rule steps through time. In the following **hence** ? will denote the set of formulas $\{\textbf{hence } A \mid A \in ?\}$.

$$\frac{?, a \vdash \Delta, \textbf{hence } D}{?, Ma \vdash \Delta, \textbf{hence } D}\ (\textbf{obs}) \qquad \frac{?, \textbf{hence } ? \vdash D}{\textbf{hence } ? \vdash \textbf{hence } D}\ (\textbf{step})$$

The rules are sound, and together with the Default cc rules, are complete with respect to the Timed Default cc logic.

THEOREM 4.3. (SOUNDNESS AND COMPLETENESS) *If A and B are programs without hiding*

$$\mathcal{D}[\![A]\!] \subseteq \mathcal{D}[\![B]\!] \Leftrightarrow A \vdash B$$

PROOF. The proofs are routine structural inductions, and follow extant proofs for Default cc. The proof is presented in detail in Appendix B. As in Default cc, the reason for excluding the hiding combinator is the slight mismatch between hiding and logical existentials — this is characteristic of cc languages. □

4.7. Determinacy detection and compilation

Determinacy detection. The algorithm described below converts Timed Default cc programs to finite state automata, with each state containing a Default cc program. We now check that the Default cc program in each state is determinate by the algorithm described in Section 3.7 — this suffices to ensure that the entire Timed Default cc program is determinate.

Default constraint automata. The automata construction for Timed Default cc is similar to the construction for tcc provided in (Saraswat *et al.* 1994a). 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 constraint system generated by the agent — there will be an edge for every element of the constraint system that can be an output of the agent.

The execution is as follows — The automaton starts in the start state, with the program **new** $\vec{X}$ **in** P, where P is hiding-free. Upon receiving an input i, it executes its Default cc agent P in conjunction with $\exists_{\vec{X}} i$. As described in section 3.3, this is done by projecting $\exists_{\vec{X}} i$ onto C_P, the constraint system generated by P, and looking up the output, o. $i \sqcup \exists_{\vec{X}} o$ is the output for this time instant. The edge labeled with o is taken to reach 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. We follow standard techniques used in synchronous programming languages. We need the notion of a derivative of an agent. Given a process P, the set of derivatives of P is defined as

$$\mathtt{Der}(P) = \{P \textbf{ after } s \mid s \in P\}$$

As in synchronous programming languages, the set $\mathtt{Der}(P)$ is finite.

LEMMA 4.4. *For all* Timed Default cc *programs* P, $|\mathtt{Der}(P)| < \infty$.

PROOF. The key case is that for the **hence** combinator — the intuition is that **hence** is essentially a "powerset" operation. The proof in the appendix makes this intuition precise. □

Following synchronous languages, this theorem provides us a with a compilation algorithm for Timed Default cc. Roughly, this proceeds as follows. The states are induced by *distinct* derivatives. Label each state $\mathcal{D}[\![A]\!]$ **after** s with the Default cc program $(\mathcal{D}[\![A]\!]$ **after** $s)(0)$. Let $s \cdot (v, v) \in \mathcal{D}[\![A]\!]$. Then, there is an edge labeled v from the state $(\mathcal{D}[\![A]\!]$ **after** s to the state $(\mathcal{D}[\![A]\!]$ **after** $(s \cdot (v, v))$. Note that this construction may yield infinitely many such arcs — however one need take the only the arcs labeled by the constraints in the finite constraint system C_P generated by P. The start state is the state $\mathcal{D}[\![A]\!]$.

Compilation algorithm. The above algorithm is not compositional. Timed Default cc

admits a compositional compilation as well. We sketch below the automaton construction for the various cases. Here we use a dead state to mean a state with label `true` and transitions labeled `true` and `false` leading back into it, thus a system upon entering this state does nothing further.

1 **Automaton for** a**.** This is a two state automaton — the start state is labeled a, and has transitions labeled a and `false` to the second state, which is a dead state.
2 **Automaton for if** a **else** P**.** The automaton for **if** a **else** 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 **if** a **else** q, where q was the Default cc agent labeling q_0. Transitions are drawn from the finite constraint system — any output above a goes to a dead state, while any output not above a must have arisen from the constraints in q, and hence the transition labeled by that goes to the corresponding state in A. The rest of the automaton for **if** a **else** P is a copy of A, the automaton of P.
3 The automaton for **if** a **then** P is constructed similarly.
4 **Automaton for new** X **in** A**.** The automaton consists of all the states of the automaton for A with the Default cc programs in the states P_i replaced by the programs **new** X **in** P_i. Note that the constraints on the arcs may contain references to the internal X, the hiding of X extends around these. So the variable X is local to P_i and the labels on transitions leaving the state q_i.
5 **Automaton for** P_1, 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, 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. Let a be an element of the finite constraint system generated by the constraints in the agents of a state. Project it on the constraint systems generated by the component programs, and if all the projections occur as outputs, then a is an output. The arc labeled by it goes to the composite state formed from the target states of the arcs labeled by the projections in the component automata.
6 **Automaton for hence** A**.** The states of this automaton are those sets of states of the automaton for A that contain the start state of A. The Default cc agent in each state is the parallel composition of the Default cc agent of the component states. The transitions are determined by the same rule as above, except that we add the start state to the set of target states to get the new state (this signifies the starting of a new copy of A). Finally, we add a state labeled `true` with transitions labeled `true` and `false` going to the state generated by the start state of A.

From the automaton for P, it is possible to derive the denotation of P by taking any valid execution path of the automaton, and taking the outputs v_i along that path and pairing them (v_i, v_i) (except for the last state in the path, where any (u, v) in the denotation of the program of that state may be taken). Now by arguments similar to the one made in showing that the operational semantics was equivalent to the denotational semantics, we can show that this is also the same semantics, showing that the construction is correct. We omit the proof, as it is routine.

5. Conclusions

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. We have presented the construction of a model, the definition of a language as a process algebra on the model, and the definition of a logic for reasoning about substitutability of programs in the language. 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.

In this paper, we have exploited these intuitions to break down the design of synchronous languages into two distinct pieces:

1 The notion of defaults is analyzed at the level of the basic (untimed) concurrent logic language.
2 The discrete timed synchronous language obtained by extending the untimed language uniformly over discrete time.

In other work (Gupta *et al.*), we have exploited the framework of this paper for the integration of conceptual frameworks for continuous and discrete change, as exemplified by the theory of differential equations and real analysis on the one hand, and the theory of programming languages on the other. In that paper, we present a concrete mathematical model and language (the Hybrid concurrent contraint programming model, Hybrid cc) instantiating these ideas. The language is intended to be used for modeling and programming *hybrid systems*. The language is again built by extending Default cc, now over continuous time.

More concretely, the language is obtained by extending Default cc with a single temporal construct **hence** — **hence** A is read as asserting that A holds *continuously* beyond the current instant. As in this paper, various patterns of temporal activity can be generated from this single construct by use of the other combinators in Default cc, in particular instantaneous preemption (as in synchronous programming). We provide a precise operational semantics according to which execution alternates between points at which discontinuous change can occur, and open intervals, in which the state of the system changes continuously. Transitions from such a state of continuous evolution can be triggered either by the establishment of a condition or by the dis-establishment of an existing condition. We show that the denotational semantics is correct for reasoning about the operational semantics, through an adequacy theorem.

We have started our first major effort at modeling a real physical system (Gupta *et al.* 1995). In this paper we developed a compositional model of the simple photocopier paperpath. Each transportation element (belt, roller etc.) is modeled by an agent that describes the effects of the various external forces on this component. A sheet of paper is modeled by a separate agent. Each sheet is under the influence of several transportation elements, and is consequently partitioned into segments. These segments, which are dynamically created and destroyed, are modeled by agents, transmit forces from one end to another, and compute the state of the sheet — buckled, straight, etc. Interaction processes are set up to make the segments interact with the transportation elements.

6. Acknowledgements.

We gratefully acknowledge discussions with Gérard Berry, Georges Gonthier, Johan de Kleer, Danny Bobrow and Markus Fromherz. We thank Peter Bigot for comments on earlier version of the paper, and the POPL 95 referees for detailed reviews.

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

References

Abramsky, S. (1993). Interaction categories. Available by anonymous ftp from papers/Abramsky:theory.doc.ic.ac.uk.

Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R. (1990). Metatem: A framework for programming in temporal logic. In de Bakker, J. W., de Roevere, W. P., Rozenberg, G., editors, *Stepwise Refinement of Distributed Systems– Models, Formalisms, Correctness*. Springer- Verlag. LNCS 430.

Baudinet, M. (1989). Temporal logic programming is complete and expressive. In *Proc. of the Sixteenth ACM conference on Principles of Programming Languages*.

Benveniste, A., Berry, G., editors (1991). *Another Look at Real-time Systems*, Special issue of Proceedings of the IEEE, September 1991.

Benveniste, A., Berry, G. (1991). The synchronous approach to reactive and real-time systems. In Benveniste & Berry (1991a)..

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

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

Berry, G. (1993). Preemption in concurrent systems. In *Proc. of FSTTCS*. Springer-Verlag. LNCS 781.

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

Brzoska, C. (1991). Temporal logic programming and its relation to constraint logic programming. In Saraswat, V. A., Ueda, K., editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 661 – 677.

Clarke, E. M., Long, D. E., McMillan, K. L. (1991). A language for compositional specification and verification of finite state hardware controllers. In Benveniste & Berry (1991a).

de Boer, F. S., Kok, J. N., Palamidessi, C., Rutten, J. J. (1993). *Logic Programming — Proceedings of the 1993 International Symposium*, chapter Non-monotonic Concurrent Constraint Programming. MIT Press.

Forbus, K. (1988). *Exploring Artificial Intelligence*, chapter Qualitative Physics: Past, Present and Future, pages 239–296. AAAI and Morgan Kaufmann.

Gabbay, D. M., Hogger, C., Robinson, J., editors (1994). *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol 3: Nonmonotonic Reasoning and Uncertain Reasoning*. Oxford Science Publications.

Gelfond, M., Lifschitz, V. (1988). The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080.

Guernic, P. L., Borgne, M. L., Gauthier, T., Maire, C. L. (1991). Programming real time applications with SIGNAL. In Benveniste & Berry (1991a).

Gupta, V., Jagadeesan, R., Saraswat, V. Computing with continuous change. *Science of Computer Programming*. To appear.

Gupta, V., Saraswat, V., Struss, P. (1995). A model of a photocopier paper path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*.

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

Halbwachs, N., Caspi, P., Pilaud, D. (1991). The synchronous programming language LUSTRE. In Benveniste & Berry (1991a).

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

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

Hentenryck, P. V., Saraswat, V. A., Deville, Y. (1992). Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, London.

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

Janson, S., Haridi, S. (1991). Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press.

Kaci, H. A. (1993). An introduction to LIFE— Programming with Logic, Inheritance, Functions and Equations. In Miller, D., editor, *Logic Programming: Proceedings of the 1993 International Symposium*. MIT Press.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In Rosenfeld, J., editor, *Proceeedings of IFIP Congress 74*, pages 471–475.

Marek, V. W., Truszczynski, M. (1993). *Nonmonotonic Logic*. Springer-Verlag.

Marek, W., Nerode, A., Remmel, J. (1990). A theory of non-monotonic rule systems – I. *Annals of Mathematics and Artificial Intelligence*, 1:241 – 273.

Marek, W., Nerode, A., Remmel, J. (1992). A theory of non-monotonic rule systems – II. *Annals of Mathematics and Artificial Intelligence*, 5:229 – 264.

Merz, R. (1993). Efficiently executing temporal logic programs. In Fisher, M., Owens, R., editors, *Proc. of IJCAI*.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Milner, R., Parrow, J., Walker, D. (1989). Mobile processes. Technical report, University of Edinburgh.

Moszkowski, B. (1986). *Executing Temporal Logic Programs*. Cambridge Univ. Press.

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

Pingali, K., Beck, M., Johnson, R., Moudgill, M., Stodghill, P. (1991). Dependence Flow Graphs: An algebraic approach to program dependencies. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages*, pages 67–78.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13.

Saraswat, V. A. (1993). *Concurrent constraint programming*. Doctoral Dissertation Award and Logic Programming Series. MIT Press.

Saraswat, V. A., Kahn, K., Levy, J. (1990). Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*.

Saraswat, V. A., Rinard, M., Panangaden, P. (1991). Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*.

Saraswat, V. A., Jagadeesan, R., Gupta, V. (1994). Foundations of Timed Concurrent Constraint Programming. In Abramsky, S., editor, *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Press.

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

Saraswat, V. A., Jagadeesan, R., Gupta, V. (1995). Default Timed Concurrent Constraint Programming. In *Proceedings of Twenty Second ACM Symposium on Principles of Programming Languages, San Francisco*.

Shoham, Y. (1988). Chronological ignorance: Experiments in nonmonotonic temporal reasoning. *Artificial Intelligence*, 36:279 – 331.

Smolka, G., Henz, Werz, J. (1994). *Constraint Programming: The Newport Papers*, chapter Object-oriented programming in Oz. MIT Press.

A. Proofs of Default cc theorems

LEMMA A.1. *The relation* $\longrightarrow_v$ *satisfies:*

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

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

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

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

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

LEMMA A.2. *For all* Default cc *programs* A *and* B,

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

PROOF. As indicated by lemma A.1, the proof of the lemma follows extant proofs for languages in the cc paradigm.

1 Let $(u,v) \in \mathcal{P}[\![a]\!]$, which means $a \in u$. Thus $\overline{a} \sqcup u = u$, so $a, u \not\longrightarrow_v$, $(u,v) \in \mathcal{O}[\![a]\!]$. Conversely, as $(u,a) \not\longrightarrow_v$, $u = \overline{a} \sqcup u$, so $a \in u$. Thus $\mathcal{O}[\![a]\!] = \mathcal{P}[\![a]\!]$.

2 The case for parallel composition is similar to the proof for cc languages.
Let $(u,v) \in \mathcal{O}[\![A_1, A_2]\!]$. Then $u, A_1, A_2 \longrightarrow_{v'} B \not\longrightarrow_{v'}$, and $\sigma(B) = \exists_{\vec{Y}} u$. We show that $(u,v) \in \mathcal{O}[\![A_1]\!]$. Let $u, A_1 \longrightarrow_{v'} A_1' \not\longrightarrow_{v'}$. Using lemma A.1, $u \subseteq \sigma(A_1')$. Using lemma A.1 again, $u, A_1, A_2 \longrightarrow_{v'} A_1', A_2$. Using confluence of $\longrightarrow_{v'}$, $A_1', A_2 \longrightarrow_{v'} B$. Using lemma A.1, $\sigma(A_1') \subseteq \sigma(B)$. Thus $\exists_{\vec{Y}} \sigma(A_1') = u$ and we deduce $(u,v) \in \mathcal{O}[\![A_1]\!]$. A symmetric argument shows that $(u,v) \in \mathcal{O}[\![A_2]\!]$. Thus, $\mathcal{O}[\![A_1, A_2]\!] \subseteq \mathcal{O}[\![A_1]\!] \cap \mathcal{O}[\![A_2]\!]$.
Let $(u,v) \in \mathcal{O}[\![A_1]\!] \cap \mathcal{O}[\![A_2]\!]$. Then, $u, A_1 \longrightarrow_{v'} A_1' \not\longrightarrow_{v'}$, and $\exists_{\vec{Y}} \sigma(A_1') = u$; also, $u, A_2 \longrightarrow_{v'} A_2' \not\longrightarrow_{v'}$, and $\exists_{\vec{Y}} \sigma(A_2') = u$. Note that in general $\longrightarrow_v$ satisfies:

$$A_1' \not\longrightarrow_v, A_2' \not\longrightarrow_v, \sigma(A_1') = \sigma(A_2') \Rightarrow A_1', A_2' \not\longrightarrow_v$$

Thus, we deduce that $u, A_1, A_2 \longrightarrow_{v'} A_1', A_2' \not\longrightarrow_{v'}$. Since $\exists_{\vec{Y}} \sigma(A_1', A_2') = u$ (the new variables in the two derivations are disjoint), $(u,v) \in \mathcal{O}[\![A_1, A_2]\!]$. Thus, $\mathcal{O}[\![A_1, A_2]\!] \supseteq \mathcal{O}[\![A_1]\!] \cap \mathcal{O}[\![A_2]\!]$.

3 Let $(u,v) \in \mathcal{O}[\![A]\!]$. If $a \in u$, we get $u, \textbf{if } a \textbf{ then } A \longrightarrow_{v'} u, A \longrightarrow_{v'} A' \not\longrightarrow_{v'}$; where $\exists_{\vec{Y}} \sigma(A') = u, \exists_{\vec{Y}} v' = v$, $\vec{Y}$ are the variables introduced in the derivation $A \rightarrow_{v'} A'$ as $(u,v) \in \mathcal{O}[\![A]\!]$. Thus $(u,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!]$. If $a \notin u$, then $u, \textbf{if } a \textbf{ then } A \not\longrightarrow_v$. Now if $a \in v$, then $(v,v) \in \mathcal{O}[\![A]\!]$ and we can show $(u,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!]$. Otherwise $a \notin v$, so by a similar argument, $(u,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!]$.
Conversely, if $(u,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!]$, then if $a \in u$, then $u, \textbf{if } a \textbf{ then } A \longrightarrow_{v'} u, A$, so $(u,v) \in \mathcal{O}[\![A]\!]$. If $a \in v$, then as $(v,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ then } A]\!]$, we have $v, \textbf{if } a \textbf{ then } A \longrightarrow_{v''} v, A \longrightarrow_{v''} A'' \not\longrightarrow_{v''}$ with $\exists_{\vec{Y}} \sigma(A'') = \exists_{\vec{Y}} v'' = v$, so $(v,v) \in \mathcal{O}[\![A]\!]$.

4 Let $(u,v) \in \mathcal{O}[\![A]\!]$, and $a \notin v$. Then, since all the new variables in v' are not free in a, we get $a \notin v'$. Thus $u, \textbf{if } a \textbf{ else } A \longrightarrow_{v'} u, A \longrightarrow_{v'} A'$, $\exists_{\vec{Y}} \sigma(A') = u$, so $(u,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!]$.
Conversely, suppose $(u,v) \in \mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!]$, $a \notin v$. Then $a \notin v'$, so $u, \textbf{if } a \textbf{ else } A \longrightarrow_{v'} u, A$. Thus $(u,v) \in \mathcal{O}[\![A]\!]$.

5 Next, we handle the key elements of the proof for the case of new variables. We reproduce the definition of the denotation for new variables for convenience.

We are simplifying the definition by using the fact that $\mathcal{O}[\![A]\!]$ is X–determinate. $\mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] = \mathbf{new}_X \mathcal{O}[\![A]\!]$, where

Define $Z_1 \stackrel{d}{=} \bigcup\{(Z_v, v) \subseteq Z \mid \forall u \in Z_v, u \supseteq \exists_X v \Rightarrow u = v\}$.

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

Below, we sketch the proof that $\mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] \subseteq \mathbf{new}_X \mathcal{O}[\![A]\!]$. The key case of the proof is to show that $(v, v) \in \mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$ implies that $(v, v) \in \mathbf{new}_X \mathcal{O}[\![A]\!]$. Let $(v, v) \in \mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$. Then, $\exists v'$ such that:$(\mathbf{new}\ X\ \mathbf{in}\ A, v) \longrightarrow_{v'}^{*} A'' \not\longrightarrow_{v'}$, $v = \exists_{newX} \exists_{\vec{Y}} v'$, $v' = \sigma(A'')$, where $newX$ is the new variable introduced for X, and $\vec{Y}$ are the *other* new variables (apart from $newX$) introduced in the derivation. Let $v'' = (\exists_{\vec{Y}} v')[X/newX]$. Since $\exists_X v = \exists_X v''$, it suffices to show:

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

(b) $(\forall (w, v'') \in \mathcal{O}[\![A]\!]), w \supseteq \exists_X v'' \Rightarrow w = v''$. We note that from the monotonicity of the $\longrightarrow$. relation, it suffices to show:

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

This follows from

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

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

The converse follows by standard `cc` methods.

□

Lemma A.3. *For any program A which satisfies the X–determinacy condition for all X, $r(\mathcal{O}[\![A]\!]) = r_o(A)$.*

Proof. We wish to show that

$$r_o(P) = \{(i, o) \in \mathbf{DObs} \mid (o, o) \in \mathcal{O}[\![P]\!], \forall (j, o) \in \mathcal{O}[\![P]\!].\ j \supseteq i \Rightarrow j = o\}$$

Let $(i, o) \in r_o(A)$. Then, $(\exists o')\ \exists_{\vec{Y}} o' = o$ and

$$(A, i) \longrightarrow_{o'} A' \not\longrightarrow_{o'}, \sigma(A') = o'$$
$$(A, o) \longrightarrow_{o'} A' \not\longrightarrow_{o'}, \sigma(A') = o'$$

The second condition above ensures that $(o, o) \in \mathcal{O}[\![A]\!]$. Also, by monotonicity of the transition relation $\longrightarrow_{o'}$, we know that for all $i \subseteq j \subseteq o$, $(A, j) \longrightarrow_{o'} A' \not\longrightarrow_{o'}, \sigma(A') = o'$ yielding $\forall (j, o) \in \mathcal{O}[\![P]\!].\ j \supseteq i \Rightarrow j = o$.

Conversely let (i, o) be such that $(o, o) \in \mathcal{O}[\![P]\!], \forall (j, o) \in \mathcal{O}[\![P]\!].\ j \supseteq i \Rightarrow j = o$. From $(o, o) \in \mathcal{O}[\![P]\!]$ we deduce that there exists o' such that $\exists_{\vec{Y}} o' = o$ and

$$(A, o) \longrightarrow_{o'} A' \not\longrightarrow_{o'}, \sigma(A') = o'$$

Consider $(A, i) \longrightarrow_{o'} A'' \not\longrightarrow_{o'}$. Then $(\exists_{\vec{Y}} \sigma(A''), \exists_{\vec{Y}} o') \in \mathcal{O}[\![A]\!]$ Since, $i \subseteq \exists_{\vec{Y}} \sigma(A'') \subseteq o$, we deduce that $\exists_{\vec{Y}} \sigma(A'') = o$. □

THEOREM A.4. (FULL ABSTRACTION FOR Default cc) *Let P, Q be programs that satisfy the X–determinacy condition for all variables X. If $\mathcal{P}[\![P]\!] \neq \mathcal{P}[\![Q]\!]$, then there exists an agent C such that P, C is observationally distinct from Q, C.*

PROOF. We first show that if there is an v such that $(v, v) \in \mathcal{P}[\![P]\!], (v, v) \notin \mathcal{P}[\![Q]\!]$, then P, Q are observationally distinguishable. Consider the context $[\cdot], v$. Then, on no input v is a possible output of P, v; but v is not a possible output of Q, v.

Otherwise, we follow extant proofs in (Saraswat *et al.* 1991),(Jagadeesan *et al.* 1991). Consider the closure operators $P_v = \{u \mid (u, v) \in \mathcal{P}[\![P]\!]\}$ and $Q_v = \{u \mid (u, v) \in \mathcal{P}[\![Q]\!]\}$. These are unequal — without loss of generality assume that on a, $P_v(a) \not\subseteq Q_v(a)$. Consider the context $[\cdot], a, (\textbf{if } P_v(a) \textbf{ then } v)$. Then, $\mathcal{P}[\![P, a, \textbf{if } P_v(a) \textbf{ then } v]\!]_v = \{v\}$, but $(Q_v(a), v) \in Q, a, (\textbf{if } P_v(a) \textbf{ then } v)$. Thus on input `true`, $Q, a, (\textbf{if } P_v(a) \textbf{ then } v)$ cannot produce output v, whereas $P, a, (\textbf{if } P_v(a) \textbf{ then } v)$ can produce output v.

Note that if v or $P_v(a)$ are not finite, there are some finite elements where P and Q differ, and these may be chosen instead (see lemmas in Section 3.7). □

cc can be embedded in Default cc.

LEMMA A.5. *For any determinate process P, $r(P)$ 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 $(u, v') \in P$.*

PROOF. Suppose P satisfies the above two conditions. Then, define

$$\texttt{diagonal}(P) = \{u \mid (u, v) \in P\}$$

We note that P is completely determined by $\texttt{diagonal}(P)$.

We prove that $\texttt{diagonal}(P)$ is closed under non–empty glbs. Let $\emptyset \neq S \subseteq \texttt{diagonal}(P)$; let $v = r(P)(\sqcup S)$; then the set $\{u \mid (u, v) \in P\}$ is closed under glbs of arbitrary non–empty subsets. Thus, $(\bigcap S, v) \in P \Rightarrow \bigcap S \in \texttt{diagonal}(P)$. Thus, $\texttt{diagonal}(P)$ is the fixed point set of a closure operator. Furthermore, note that we can rewrite $r(P)$ as:

$$r(P) = \{(i, o) \mid o \vdash i, o \in \texttt{diagonal}(P), \forall (j, o) \in P.\ j \supseteq i \Rightarrow j = o\}$$

Thus, $r(P)$ coincides with the input–output relation of the closure operator $\texttt{diagonal}(P)$.

Result is now immediate. □

The following lemmas allow us to determine if a program is determinate.

LEMMA A.6. *If P is a hiding-free program over the constraint system $\mathcal{C}_1$ with denotation $\mathcal{P}[\![P]\!]_1$, $\mathcal{C}_2$ extends $\mathcal{C}_1$, and the denotation of P over $\mathcal{C}_2$ is $\mathcal{P}[\![P]\!]_2$, then for all $u, v \in |C_2|$ we have $(u, v) \in \mathcal{P}[\![P]\!]_2$ iff $(\pi(u), \pi(v)) \in \mathcal{P}[\![P]\!]_1$.*

PROOF. The proof is by induction over the structure of a program P.

Case $P = a$. For any $u \in |C_2|$, $a \in u \Leftrightarrow a \in \pi(u)$. From this and the definition of the semantics of a, we have the result.

Case $P = \mathbf{if}\ a\ \mathbf{else}\ A$. We have

$$\begin{array}{rcl}(u,v) \in \mathcal{P}[\![\mathbf{if}\ a\ \mathbf{else}\ A]\!]_2 & \Leftrightarrow & a \in v \vee (u,v) \in \mathcal{P}[\![A]\!]_2 \\ & \Leftrightarrow & a \in \pi(v) \vee (\pi(u),\pi(v)) \in \mathcal{P}[\![A]\!]_1 \\ & \Leftrightarrow & (\pi(u),\pi(v)) \in \mathcal{P}[\![\mathbf{if}\ a\ \mathbf{else}\ A]\!]_1\end{array}$$

The proof for the case $P = \mathbf{if}\ a\ \mathbf{then}\ A$ is similar. If $P = (A, B)$, the proof follows from simple properties of set intersection. □

We can now prove the representation theorem :

THEOREM A.7. (REPRESENTATION THEOREM) *If P is a hiding-free program, then* $r(P)(i) = \left\{ i \sqcup o \mid o \in r^f(P)(\pi(B(i))) \right\}$.

PROOF. This is proved using the following claims:

Claim 1. If $\mathcal{C}_1$ is extended by $\mathcal{C}_2$, then for any $i \in C_1$, $r_1(P)(i) = r_2(P)(i)$, where r_1 and r_2 are the respective input output relations.

Suppose $o \in r_1(P)(i)$. Then $(o,o) \in \mathcal{P}[\![P]\!]_1 \subseteq \mathcal{P}[\![P]\!]_2$. Also, if $(j,o) \in \mathcal{P}[\![P]\!]_2, j \supseteq i$, then $\pi(j) \supseteq \pi(i) = i$, and $(\pi(j), o) \in \mathcal{P}[\![P]\!]_1$, so $\pi(j) = o$, so $j = o$, thus $o \in r_2(P)(i)$.

Now if $o' \in r_2(P)(i)$, then $(o',o') \in \mathcal{P}[\![P]\!]_2$. Thus $(\pi(o'), \pi(o')) \in \mathcal{P}[\![P]\!]_1$. If $(j, \pi(o')) \in \mathcal{P}[\![P]\!]_1, j \supseteq i$, then $(j, o') \in \mathcal{P}[\![P]\!]_2$, so $j = o'$. Thus $\pi(o') \in r_1(P)(i)$. However $(\pi(o'), o') \in \mathcal{P}[\![P]\!]_2$, so $\pi(o') = o'$, thus $o' \in r_1(P)(i)$.

Claim 2. If $\mathcal{C}_1$ is extended by $\mathcal{C}_2$, then for any $i \in \mathcal{C}_2$, $r_2(P)(i) = \left\{ i \sqcup o \mid o \in r_1(P)(\pi(i)) \right\}$, provided $\forall u \in |C_1|, v \in |C_2|.\pi(u \sqcup v) = u \sqcup \pi(v)$.

Suppose $o \in r_1(P)(\pi(i))$. Then $(o,o) \in \mathcal{P}[\![P]\!]_1$, and as $\pi(i \sqcup o) = o$, $(i \sqcup o, i \sqcup o) \in \mathcal{P}[\![P]\!]_2$. Also, if $(j, i \sqcup o) \in \mathcal{P}[\![P]\!]_2, j \supseteq i$, then $(\pi(j), o) \in \mathcal{P}[\![P]\!]_1, \pi(j) \supseteq \pi(i)$, so $\pi(j) = o$, thus $j = i \sqcup o$.

Conversely, let $o' \in r_2(P)(i)$. Then $\pi(o')$ is an output of $r_1(P)(\pi(i))$ — $(\pi(o'), \pi(o')) \in \mathcal{P}[\![P]\!]_1$, and given $(j, \pi(o')) \in \mathcal{P}[\![P]\!]_1, j \supseteq \pi(i)$, let $j' = j \sqcup i$. Then $\pi(j') = j$, so $(j', o') \in \mathcal{P}[\![P]\!]_2$, so $j' = o'$, hence $j = \pi(o')$. Also, this shows that $o' = j' = i \sqcup \pi(o')$. establishing the claim.

Now we observe that $B(\mathcal{C})$, the free CABA on $\mathcal{C}$ extends it, so its input output relation is the same as $r(P)$. And the CABA also extends C_P, and the required condition is satisfied, so by the second claim we have the result. □

LEMMA A.8. *If A is a determinate program, then so is* **new** X **in** A.

PROOF. We show that for each output of **new** X **in** A there is a corresponding output of A. Without loss of generality, we consider only the input `true`, any other input can be treated by conjoining it with the program, and then considering the composite program.

Let o be an output of **new** X **in** A on `true`. Then $(o,o) \in \mathcal{P}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$, and $\exists_X o = o$. Thus there is a witness $(o',o') \in \mathcal{P}[\![A]\!]$, such that $\exists_X o' = \exists_X o = o$. Let $(j,o') \in \mathcal{P}[\![A]\!]$, then we need to show that $j = o'$. However, $(j,o') \in \mathcal{P}[\![A]\!]$ means that $(\exists_X j, \exists_X o') \in \mathcal{P}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!]$, and as $\exists_X o' = o$ is an output, we have $\exists_X j = \exists_X o'$. Thus $j \supseteq \exists_X o'$. Now from the first step for $\mathbf{new}_X A$ we have $j = o'$, thus o' is an output of A on `true`.

Now since A is determinate, this output is unique, so o is the unique output on **new** X **in** A on `true`. (Any other output o_1 would satisfy $\exists_X o_1 = o_1 \neq o$, thus would have a witness different from o'.) □

The following lemma, with the previous one, shows that all our results are valid unconditionally if we restrict ourselves to determinate programs.

LEMMA A.9. *If Z is a determinate process, then Z is X-determinate for all variables X. Thus it also satisfies the X-determinacy condition for all X.*

PROOF. Let $(v,v),(v',v') \in Z$, with $\exists_X v = \exists_X v'$. Then since $Z_v(\exists_X v) = v$ and $Z_{v'}(\exists_X v') = v'$, both v and v' are possible outputs of Z on input $\exists_X v$. By determinacy of Z, $v = v'$, thus $Z_2 = Z_1$. □

The soundness and completeness of the logic for recursion free Default ccprograms without hiding is shown as follows:

THEOREM A.10. (SOUNDNESS) $? \vdash \Delta$ *implies* $\mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta]\!]$.

PROOF. The proof is by induction, and follows directly from the definitions. The only (slightly) nontrivial case is $(Rthen)$ which we prove here.

$$\begin{array}{ll} & (u,v) \in \mathcal{P}[\![?\,, a]\!] \Rightarrow (u,v) \in \cup[\![\Delta, A]\!] \\ \iff & (u,v) \in \mathcal{P}[\![?]\!], a \in u \Rightarrow (u,v) \in \cup[\![\Delta, A]\!] \\ \iff & (u,v) \in \mathcal{P}[\![?]\!] \Rightarrow a \in u \rightarrow (u,v) \in \cup[\![\Delta, A]\!] \\ \Rightarrow & (u,v) \in \mathcal{P}[\![?]\!] \Rightarrow (u,v) \in \cup[\![\textbf{if } a \textbf{ then } A, \Delta]\!] \end{array}$$

The last step follows since $?$ is a process, $\mathcal{P}[\![Ma]\!] \cup [\![A]\!] = \mathcal{P}[\![\textbf{if } a \textbf{ else } A]\!]$ and $\textbf{if } b \textbf{ then if } a \textbf{ else } A = \textbf{if } a \textbf{ else if } b \textbf{ then } A$. □

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

THEOREM A.11. (COMPLETENESS) $\mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta]\!]$ *implies* $? \vdash \Delta$

PROOF. Suppose the non-trivial formula is $B = b$, let the rest of the formulas in Δ be denoted by Δ'. 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 $b'_i \not\vdash b_i$. Then the pair $(\overline{a}, \sqcap\{\overline{b'_i}\})$ is also on the left, and so it must be in $\mathcal{P}[\![b]\!]$. So $\overline{a} \supseteq \overline{b}$, and we can use (C) to prove the result.

The other cases for B are straightforward. For $B = (B_1, B_2)$, from $\mathcal{P}[\![B]\!] = \mathcal{P}[\![B_1]\!] \cap \mathcal{P}[\![B_2]\!]$ and $\mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta']\!] \cup \mathcal{P}[\![(B_1, B_2)]\!]$ we have $\mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta']\!] \cup \mathcal{P}[\![B_1]\!]$ and $\mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta']\!] \cup \mathcal{P}[\![B_2]\!]$. Thus by the induction hypothesis, $? \vdash \Delta', B_1$ and $? \vdash \Delta', B_2$. Apply $Rpar$ to obtain $? \vdash \Delta$.

For $B = \textbf{if } a \textbf{ else } B'$, since $\mathcal{P}[\![B]\!] = \mathcal{P}[\![Ma]\!] \cup \mathcal{P}[\![B']\!]$, we have $\mathcal{P}[\![?]\!] \subseteq \cup[\![\Delta']\!] \cup \mathcal{P}[\![Ma]\!] \cup B'$. Thus $? \vdash \Delta', Ma, B'$. Applying $(Relse)$, we have the result.

For $B = \textbf{if } a \textbf{ then } B'$, proceed as follows. By assumption every $(u,v) \in \mathcal{P}[\![?]\!]$ lies either in $\mathcal{P}[\![\textbf{if } a \textbf{ then } B']\!]$ or in $\cup[\![\Delta']\!]$. But this is means that $(u,v) \in \mathcal{P}[\![?\,, a]\!]$ implies $(u,v) \in \cup[\![\Delta']\!] \cup \mathcal{P}[\![B']\!]$, which can be established by the induction hypothesis and the rule $(Rthen)$. □

B. Proof of Timed Default cc theorems

The following lemma captures the essence of the evolution in each time instant of Timed Default cc — the first consequence says that the correct Default cc observation is captured in the execution at the point, and the next says that the correct "continuation" is passed to the succeeding time instants.

LEMMA B.1. *Let $\mathcal{D}[\![?]\!]$ satisfy the X-determinacy condition for all variables X. Let $(?, \emptyset) \longrightarrow_v (?', \Delta) \not\longrightarrow_v$, with $\vec{Y}$ being the new variables introduced by the transition system. Let s be the single element sequence $s = (\sigma_{\Gamma'}, v)$. Let $\exists_{\vec{Y}} s$ stand for the single element sequence such that $(\exists_{\vec{Y}} \sigma_{\Gamma'}, \exists_{\vec{Y}} v)$. Then,*

$\exists_{\vec{Y}} s \in \mathcal{D}[\![?]\!]$
If $v = \sigma(?')$, $\mathcal{D}[\![?]\!]$ **after** $\exists_{\vec{Y}} s = \mathcal{D}[\![\mathbf{new}\ \vec{Y}\ \mathbf{in}\ \Delta]\!]$ **after** $\exists_{\vec{Y}} s$

PROOF. The first proof is essentially a statement about Default cc — it follows by induction on the number of rules in $(?, \emptyset) \longrightarrow_v (?', \Delta)$.

The second proof follows from the following intermediate facts:

If $v = \sigma(?')$, then $\mathcal{D}[\![?']\!]$ **after** $s = \mathcal{D}[\![\Delta]\!]$ **after** s. Proof is a routine structural induction on $?'$.
If $v = \sigma(?')$, then $\mathcal{D}[\![?]\!]$ **after** $\exists_{\vec{Y}} s = \mathcal{D}[\![\mathbf{new}\ \vec{Y}\ \mathbf{in}\ (?', \mathbf{hence}\ \Delta)]\!]$ **after** $\exists_{\vec{Y}} s$. Proof follows by induction on the number of rules in $(?, \emptyset) \longrightarrow_v (?', \Delta)$.

□

We now give proofs of the various Timed Default cc theorems. First the equivalence lemma and full abstraction:

LEMMA B.2.

$$
\begin{array}{rcl}
\mathcal{O}[\![a]\!] & = & \mathcal{D}[\![a]\!] \\
\mathcal{O}[\![A, B]\!] & = & \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!] \\
\mathcal{O}[\![\mathbf{if}\ a\ \mathbf{then}\ A]\!] & = & \{\epsilon\} \cup \{(u,v) \cdot s \in \mathbf{TDccObs} \mid \quad a \in v \Rightarrow (v,v) \in \mathcal{O}[\![A]\!], \\
 & & \qquad\qquad a \in u \Rightarrow (u,v) \cdot s \in \mathcal{O}[\![A]\!]\} \\
\mathcal{O}[\![\mathbf{if}\ a\ \mathbf{else}\ A]\!] & = & \{\epsilon\} \cup \{(u,v) \cdot s \in \mathbf{DObs} \mid a \notin u \Rightarrow (u,v) \cdot s \in \mathcal{O}[\![A]\!]\} \\
\mathcal{O}[\![\mathbf{new}\ X\ \mathbf{in}\ A]\!] & = & \{s \in \mathbf{TDccObs} \mid \exists s' \in \mathcal{O}[\![A]\!]. \exists_X s = \exists_X s', \\
 & & \qquad \forall n \leq |s| [s_n \in \mathbf{new}\ X\ \mathbf{in}\ \mathcal{O}[\![A\ \mathbf{after}\ (s'_0, \ldots, s'_{n-1})]\!]]\}, \\
 & & \qquad\qquad \textit{if } \mathcal{O}[\![A]\!] \textit{ is } X\textit{-determinate} \\
\mathcal{O}[\![\mathbf{hence}\ A]\!] & = & \{\epsilon\} \cup \{z \cdot s \in \mathbf{TDccObs} \mid (\forall s_1, s_2) s = s_1 \cdot s_2 \Rightarrow s_2 \in \mathcal{O}[\![A]\!]\}
\end{array}
$$

PROOF. The result is immediate that all sequences of length 0. We prove by induction on the length of the sequence $s \in \mathbf{TDccObs}$ that forall programs A that satisfy the X-determinacy condition:

$$
\begin{array}{rcl}
s \in \mathcal{O}[\![a]\!] & \Leftrightarrow & s \in \mathcal{D}[\![a]\!] \\
s \in \mathcal{O}[\![A, B]\!] & \Leftrightarrow & s \in \mathcal{O}[\![A]\!] \cap \mathcal{O}[\![B]\!] \\
s \in \mathcal{O}[\![\mathbf{if}\ a\ \mathbf{then}\ A]\!] & \Leftrightarrow & s = (u,v) \cdot t, a \in u \Rightarrow s \in \mathcal{O}[\![A]\!], \\
 & & \qquad a \in v \Rightarrow (v,v) \in \mathcal{O}[\![A]\!]
\end{array}
$$

$$\begin{aligned} s \in \mathcal{O}[\![\textbf{if } a \textbf{ else } A]\!] &\Leftrightarrow s = (u,v)\cdot t, a \notin v \Rightarrow s \in \mathcal{O}[\![A]\!] \\ s \in \mathcal{O}[\![\textbf{new } X \textbf{ in } A]\!] &\Leftrightarrow s \in \textbf{new}_X \mathcal{O}[\![A]\!] \\ s \in \mathcal{O}[\![\textbf{hence } A]\!] &\Leftrightarrow s = z\cdot s', (\forall s_1, s_2) s' = s_1 \cdot s_2 \Rightarrow s_2 \in \mathcal{O}[\![A]\!] \end{aligned}$$

For the combinators from Default cc, the second part of lemma B.1, attesting to the correctness of the continuations, allows us to use the inductive hypothesis on the remainder of s.

For **hence** A, lemma B.1 allows us to conclude that the operational and denotational semantics agree on the first element of s. The second part of lemma B.1, attesting to the correctness of the the continuations after the first element of s, allows us to use the inductive hypothesis on the remainder of s.

□

Theorem B.3. (Full abstraction for Timed Default cc) *If $\mathcal{D}[\![P]\!] \neq \mathcal{D}[\![Q]\!]$ and P, Q satisfy the X–determinacy condition for all variables X, then there exists a context C such that P, C is observationally distinct from Q, C.*

Proof. If two programs are distinct, there must be some sequence $t \in \mathcal{D}[\![P]\!]$ which is not in $\mathcal{D}[\![Q]\!]$. Let $s \cdot (u,v)$ be the shortest prefix of t which is not in $\mathcal{D}[\![Q]\!]$. Thus P **after** $s \neq Q$ **after** s.

By the full abstraction theorem for Default cc, let A be a context distinguishing P **after** s and Q **after** s. Now if $s = (o_1, o_1)\cdot(o_2,o_2)\cdots(o_n,o_n)$, then the context $o_1, \textbf{next } o_2, \ldots, \textbf{next}^{\,n-1} o_n, \textbf{next}^{\,n} A$ distinguishes P and Q. □

Now we prove a couple of identities for combinators.

Lemma B.4.

$$\begin{aligned} &\textbf{next } A = \textbf{new stop in hence } [\textbf{if stop else } A, \textbf{hence stop}] \\ &\textbf{first } a \textbf{ then } A = \textbf{new stop in always } (\ \textbf{if stop else if } a \textbf{ then } A \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } a \textbf{ then hence stop}) \end{aligned}$$

Proof. We will compute the denotation of the right-hand side. Let $P = [\textbf{if stop else } A, \textbf{hence stop}]$, and $Q = \textbf{hence } P$.

$$\begin{aligned} \mathcal{D}[\![\texttt{stop}]\!] &= \{(u,v)\cdot s \mid \texttt{stop} \in u\} \\ \mathcal{D}[\![\textbf{hence stop}]\!] &= \{z\cdot s \mid \forall i.\texttt{stop} \in \pi_1(s(i))\} \\ \mathcal{D}[\![\textbf{if stop else } A]\!] &= \{(u,v)\cdot s \mid \texttt{stop} \in v \vee (u,v)\cdot s \in \mathcal{D}[\![A]\!]\} \\ \mathcal{D}[\![P]\!] &= \{(u,v)\cdot s \mid \texttt{stop} \in v \vee (u,v)\cdot s \in \mathcal{D}[\![A]\!], \forall i.\texttt{stop} \in \pi_1(s(i))\} \\ \mathcal{D}[\![Q]\!] &= \{z\cdot z'\cdot s \mid \texttt{stop} \in \pi_2(z') \vee z'\cdot s \in \mathcal{D}[\![A]\!], \forall i.\texttt{stop} \in \pi_1(s(i))\} \end{aligned}$$

The last equation follows as for all further i, the condition $z_2 \supseteq \texttt{stop} \vee z\cdot s \in \mathcal{D}[\![A]\!]$, the second condition automatically satisfies the first. Now $(\mathcal{D}[\![Q]\!]$ **after** $\epsilon)(0) = \textbf{DObs}$, so **new** X **in** $(\mathcal{D}[\![Q]\!]$ **after** $\epsilon)(0) = \textbf{DObs}$. Thus for any z, $\mathcal{D}[\![Q]\!]$ **after** $z = \{(u,v) \mid \texttt{stop} \in v \vee (u,v) \in \mathcal{D}[\![A]\!]\}$. If $\texttt{stop} \in v$, then $(v - \texttt{stop}, v) \in \mathcal{D}[\![Q]\!]$ **after** z, so all such v's are dropped by the first condition. Also, since there is no occurrence of stop in A, nothing is dropped by either the first or second steps for hiding, and the final step, taking $\exists_{\texttt{stop}}$ of the denotation of A allows those v's containing stop also. Thus we get **new stop in** $(\mathcal{D}[\![Q]\!]$ **after** $z)(0) = (\mathcal{D}[\![A]\!]$ **after** $\epsilon)(0)$. For any future $(\mathcal{D}[\![Q]\!]$**after** $(z\cdot s))0)$, we know that the first step for hiding is satisfied as for all (u,v), $\texttt{stop} \in u$. The second

step once again does not drop anything, since $\exists_{\texttt{stop}} v = \exists_{\texttt{stop}} v' \Rightarrow v = v'$. The final step allows any external occurrences of `stop`, giving us exactly $(\mathcal{D}[\![A]\!]\textbf{after } s)(0)$.

For the second identity, we first get the denotation of **always** (**if** `stop` **else if** a **then** A, **if** a **then hence** `stop`) by substituting the definitions, like for the first identity. We get

$$\left\{s \mid \forall i, \texttt{stop} \in \pi_2(s(i)_2) \vee s^i \in \mathcal{D}[\![\textbf{if } a \textbf{ then } A]\!], a \in \pi_1(s(i)) \Rightarrow \forall j > i, \texttt{stop} \in \pi_1(s(j))\right\}$$

This can be simplified by considering two cases — either $\forall i, a \notin \pi_1(s(i))$, which removes the second clause, and also simplifies $\mathcal{D}[\![\textbf{if } a \textbf{ then } A]\!]$. Otherwise let k be the least number such that $a \in \pi_1(s(k))$. Then `stop` is always entailed after k, and at k we have $\texttt{stop} \in \pi_2(s(k)) \vee s^k \in \mathcal{D}[\![A]\!]$. This is almost identical to the denotation we had in the previous case, and now similar reasoning about hiding the `stop` gives us the identity. □

Lemma B.5. *The rules* **obs** *and* **step** *are sound.*

Proof. Let $s \in \mathcal{D}[\![?, Ma]\!]$. Now if $|s| = 1$, then $s \in \mathcal{D}[\![\textbf{hence } D]\!]$, so $s \in \cup[\![\Delta]\!] \cup \mathcal{D}[\![\textbf{hence } D]\!]$. If $|s| > 1$, then $s(1) = (v, v)$, for some $v \in |D|$. Then $s \in \mathcal{D}[\![?, Ma]\!] \Leftrightarrow s \in \mathcal{D}[\![?, a]\!]$, which gives the result.

Assume that $\mathcal{D}[\![\textbf{hence } ?, ?]\!] \subseteq \mathcal{D}[\![A]\!]$. Let $z \cdot s$ be an observation in $\mathcal{D}[\![\textbf{hence } ?]\!]$. We want to show that it also belongs to $\mathcal{D}[\![\textbf{hence } A]\!]$. Suppose $s = s_1 \cdot s_2$. We already know, from the definition of **hence** ?, that $s_2 \in \mathcal{D}[\![?]\!]$. Also, it must be the case that $s_2 \in \mathcal{D}[\![\textbf{hence } ?]\!]$, since it satisfies all the conditions. Thus from the hypothesis we get $s_2 \in \mathcal{D}[\![A]\!]$. Thus we know that $z \cdot s \in \mathcal{D}[\![\textbf{hence } A]\!]$, which shows that the rule is sound. □

Theorem B.6. (Completeness) *If A and B are programs without hiding, and $\mathcal{D}[\![A]\!] \subseteq \mathcal{D}[\![B]\!]$ then $A \vdash B$ is derivable from the above rules.*

Proof. The proof goes exactly the same way as the proof for theorem A.11. We do induction on the non-trivial formula of the right side. The proofs for all the cases considered in theorem A.11 remain the same. The only new case is if $B = \textbf{hence } D$. Apply the left rules and **obs** until they cannot be applied any further, resulting in a sequent of the form $?', \textbf{hence } ? \vdash \Delta, \textbf{hence } D$. Note that $?'$ has formulas of the form $a, \textbf{if } a \textbf{ then } A$ only, while Δ can have formulas of the type Mb only. If the rule (M) can be used to prove this sequent, we are done.

Otherwise, suppose $z \cdot s \in \mathcal{D}[\![\textbf{hence } ?]\!]$. Let $y = (\sigma(?'), \sigma(?'))$ be a simple observation. Then $y \cdot s \in \mathcal{D}[\![?']\!] \cap \mathcal{D}[\![\textbf{hence } ?]\!]$, since it is in the denotations of all the agents (note the since no more left rules were applicable, $\sigma(?)$ cannot trigger any asks). Now by the hypothesis, $y \cdot s \in \cup[\![\Delta]\!] \cup \mathcal{D}[\![\textbf{hence } D]\!]$. But since (M) was inapplicable, it cannot be in $\cup[\![\Delta]\!]$, so it must be in $\mathcal{D}[\![\textbf{hence } D]\!]$. Then $z \cdot s \in \mathcal{D}[\![\textbf{hence } D]\!]$, so $\mathcal{D}[\![\textbf{hence } ?]\!] \subseteq \mathcal{D}[\![\textbf{hence } D]\!]$.

Now we apply the **step** rule to complete the proof, as we can drop $?'$ and Δ by weakening. Suppose $z \cdot s \in \mathcal{D}[\![?]\!] \cap \mathcal{D}[\![\textbf{hence } ?]\!]$ then we know that $z' \cdot z \cdot s \in \mathcal{D}[\![\textbf{hence } ?]\!]$ for any simple observation z'. Thus $z' \cdot z \cdot s \in \mathcal{D}[\![\textbf{hence } D]\!]$, so $z \cdot s \in \mathcal{D}[\![D]\!]$, so $\mathcal{D}[\![?]\!] \cap \mathcal{D}[\![\textbf{hence } ?]\!] \subseteq \mathcal{D}[\![D]\!]$. Now be our induction hypothesis, we can build a proof tree for $?, \textbf{hence } ? \vdash D$, completing our proof. □

Theorem B.7. *Every* Timed Default cc *agent A has a finite number of derivatives.*

PROOF. The proof is by induction on the structure of A. Let the number of derivatives of any agent A be $n(A)$.

$A = a$. $n(a) = 2$, the two derivatives are $\mathcal{D}[\![a]\!]$ and **TDccObs**. $\mathcal{D}[\![A]\!]$ **after** $\epsilon = \mathcal{D}[\![a]\!]$, while for any other sequence s, $\mathcal{D}[\![A]\!]$ **after** $s =$ **TDccObs**.

$A =$ **if** a **then** B. $n($**if** a **then** $B) \leq 2 + n(B)$. $\mathcal{D}[\![A]\!]$ is one derivative, and if $s_1 = (v, v), a \notin v$ then $\mathcal{D}[\![A]\!]$ **after** $s =$ **TDccObs**. If $a \in v$, then $\mathcal{D}[\![A]\!]$ **after** s must be a derivative of B, in fact $\mathcal{D}[\![A]\!]$ **after** $s = \mathcal{D}[\![B]\!]$ **after** s.

$A =$ **if** a **else** B. Like the above case, $n($**if** a **else** $B) \leq 2 + n(B)$.

$A =$ **new** X **in** B. The derivatives correspond to the derivatives of B, so $n(A) \leq n(B)$.

$A = B_1, B_2$. $n(A) \leq n(B_1) \times n(B_2)$. Since $\mathcal{D}[\![A]\!] = \mathcal{D}[\![B_1]\!] \cap \mathcal{D}[\![B_2]\!]$, for any sequence s we have $\mathcal{D}[\![A]\!]$ **after** $s = \mathcal{D}[\![B_1]\!]$ **after** $s \cap \mathcal{D}[\![B_2]\!]$ **after** s. Thus every derivative of A is formed from a derivative of B_1 and a derivative of B_2, so the maximum number of derivatives that can be formed is $n(B_1) \times n(B_2)$.

$A =$ **hence** B. $n(A) \leq 2^{n(B)} + 1$. $\mathcal{D}[\![A]\!]$ is clearly one such derivative. From the definition, if s is any sequence, then $\mathcal{D}[\![A]\!]$ **after** $s = \mathcal{D}[\![$**hence** $B]\!] \cap \bigcap_{i>0} \mathcal{D}[\![B]\!]$ **after** $s^{(i)}$, where $s^{(i)}$ is the suffix of s starting at the i'th position. Thus each derivative of **hence** B corresponds to a set of derivatives of B, giving us the result. □