Programming in Timed Concurrent Constraint Languages*

Vijay Saraswat
(saraswat@parc.xerox.com)
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto Ca 94304

Radha Jagadeesan
(radha@math.luc.edu)
Dept. of Mathematical Sciences
Loyola University-Lake Shore Campus
Chicago, Il 60626

Vineet Gupta
(vgupta@cs.stanford.edu)
Computer Science Department
Stanford University
Stanford Ca 94305

December 1993

*To appear in "Constraint Programming", Ed. B. Mayoh, E. Tyugu, J. Penjam, NATO Advanced Science Institute Series, Series F: Computer and System Sciences, 1994

Contents

List of Tables

1 Introduction

The areas of Qualitative Reasoning about physical systems [Wd89], reasoning about action and state change [Gin87], reactive, real-time computing [Rea91] and concurrent programming languages [Mil80,Hoa85b] are areas of inquiry that are fundamentally about the same subject matter — the representation, design and analysis of continuous and discrete dynamical systems.

Surprisingly, no robust, common analyses of this subject matter have hitherto emerged, though these areas have independently been the beneficiaries of vigorous investigation in the past decade(s). The area of reactive, real-time computing has put forth the elegant idea of synchronous programming — languages for specifying and implementing reactive systems that interact "instantaneously" with their environment. The well-developed languages in this family, such as ESTEREL, LUSTRE and SIGNAL, enjoy remarkable mathematical, computational and expressiveness properties. The analysis of concurrency — and, in general, the programme of research into the foundations of computing at the heart of theoretical computer science — has put forth several models for indeterminate, concurrent, communicating systems. And it has developed a methodology for research based on developing an "algebra" of processes, on identifying primitive processes and combinators that are definable in these models and that together (through "full abstraction" results) characterize these models.

The area of non-monotonic reasoning has identified the importance of *defeasible* reasoning — the possibly erroneous "jumping to conclusions" that is the inevitable consequence of the lack of omniscience about a dynamic environment characteristic of all finitary computational systems. (And it has highlighted the paucity of resources at our disposal in attacking these fundamental problems of of *efficient* reasoning in the face of (state) change. The more powerful theories of non-monotonic inference remain hamstrung by artificial model theories or impossibly hard computational problems.) The area of Qualitative Physics has highlighted the importance and centrality of explicit, compositional, declarative, models for simulation, monitoring, diagnosis, control and design of such dynamical systems.

The natural meeting ground for these hitherto non-confluent areas of inquiry is logic — the logic of time and change. Our thrust to this heartland is through the synthesis of (concurrent) constraint programming (CCP) — itself at the heart of the dual logicism and proceduralism of computational logic — with synchronous programming. From the viewpoint of programming language design, we do not claim that there is something profound about the choice of CCP as our starting point. Indeed we believe that the theory of timed computation we develop is robust enough to be applicable to other asynchronous models of computation. However, we have found the notion of an underlying constraint system to be a robust crutch to support our intuitions about the design of powerful computational systems.

CCP is based on the simple idea of a collection of (possibly distributed) agents communicating by imposing and checking pieces of partial information — constraints — on shared variables. Agents are not synchronized, the constraints posted accumulate monotonically — perhaps after arbitrary unbounded delay — in a (possibly mythical) shared store. The computational view is thus asynchronous, monotonic and untimed; the logical view is in terms of deduction in intuitionistic logic.

The fundamental move in the Timed Concurrent Constraint Programming (tcc) framework is to augment the ability of constraint programming to detect "positive information" ("some event is happening", some constraint is derivable) with the ability to detect *negative information* — "some event did not happen", some constraint was *not* derivable on quiescence. Such detection is at the heart of the notion of time-outs that are crucial to reactive, real-time computation and of the notion of "closed world" assumptions that are crucial to reasoning with incomplete information and in the face of change. It also causes the underlying logical view of computation to be in terms of deduction in (fixed-point extensions of) intuitionistic (linear-time) *temporal* logic.

In addition, tcc incorporates a simple idea with its roots in the work in Qualitative Reasoning on model-based simulation of physical systems: the current and future can have no causal impact on the past. Thus, once negative information is detected, it is too late to change the state of affairs that led to the event not happening; all that can be done is to set up for action now and in the future. This idea is instrumental in circumventing the "temporal paradoxes" [BG92] that arise from the confusion between causality and temporality that has hitherto seemed to be an integral part of the synchronous approach to computing. And it is at the heart of our identification of a powerful form of "safe" default reasoning that avoids the complicated mathematical and computational properties of more general forms of non-monotonic reasoning.

This paper explores the expressive power of the tcc paradigm. The origin of the work in the integration of synchronous and constraint programming is described. The basic conceptual and mathematical framework — developed in the spirit of the model-based approach characteristic of theoretical computer science — is reviewed. We show that a range of constructs for expressing timeouts, preemption and other complicated patterns of temporal activity are expressible in the basic model and language-framework. Indeed, we present a single construct on processes, definable in the language, that can simulate the effect of other preemption constructs.

We present Timed Gentzen, a concrete tcc language instantiated over the "generic" constraint system Gentzen that underlies "unification-free" computation in (classical, intuitionistic, linear) logic. Recall that the constraint system Herbrand [Sar93] originates in the use of "delayed instantiation" of existentially quantified variables on the "right hand side" of a sequent system: in the query $\vdash \exists X.(p(X) \wedge q(X))$ one may think of p and q as processes that dynamically impose constraints on X. The task of the Herbrand constraint system is to express and resolve equalities over uninterpreted terms, through variations of the Herbrand-Martelli-Montanari (first-order) unification algorithm. In contrast, the constraint system we name Gentzen is concerned with solving the core (incremental) inference problem that arises in computing with formulas on the *left* hand side of sequents (e.g., $p(X), \forall Y.p(Y) \rightarrow q(Y) \vdash a$). In brief, the "tokens" of the constraint system here are arbitrary (positive) atomic formulas (possibly with "eigenvariables"), and the inference relation is that forced by the non-logical structural rules (identity, cut, permutation, etc.). Thus the computational problem is essentially a database lookup problem, in the face of (1) arbitrary tree-structured data, (2) a monotonically increasing database, (3) universally quantified queries. Somewhat surprisingly, this constraint system has already been used heavily in implemented work in Qualitative Reasoning — it underlies the "active database" viewpoint for organizing problem-solving systems developed by Sussman and his colleagues in the 70's; see [Fd93] for a recent expository development.

Subsequently, we present solutions to several representative reactive and synchronous programming problems in Timed Gentzen; also we discuss how some common default inferencing techniques can be represented. In a paper of this form, it is possible only to select and treat a few representative examples.

No paper on programming techniques can be complete without providing the reader with the ability to test and develop new programs. We discuss and include a full working interpreter for Timed Gentzen, written in Prolog, that can be used for running these programs. It is short, under a hundred lines long, but fully functional.

Acknowledgements We gratefully acknowledge extended discussions on various aspects of this work with Danny Bobrow, Jerry Burch, Adam Farquhar, Markus Fromherz, Lalita Jategaonkar, John Lamping, Tim Lindholm, Olivier Raiman, Mark Shirley, Brian Smith and Brian Williams.

2 Synchronous programming

Reactive systems [HP85,Ber89,Hal93] 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. The primary issues that arise in programming reactive systems are time-criticality, reliability and maintainability in the face of change.

2.1 Automata

Arguably, the most natural way of programming such systems is in terms of automata with simple loop-free transitions, to ensure bounded response. The transitions of the automaton correspond to simple actions executed by the program at every time step.

Consider for example a simple protocol that implements the controller of a paper-tray of a photo-copier. The function of the controller is to switch the paper tray motor on and off, always trying to keep the position of the top of the stack of paper next to the feeder. There are two sensors, P and E, which are set to 1 when the height of the paper is OK. Whenever the paper level falls, i.e. one of the sensors becomes zero, the motor is activated to push up the paper stack. This protocol can be construed as a finite state machine with two states as in Figure 2.1.

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 [MS90]. For example, let us say that we want the controller to also be aware of the fact that the sensors may be broken, in which case, it should stop the motor after a certain delay, to prevent it from damaging the copier. An automaton for this protocol is as in Figure 2.1.

Note that there is no *structural* relationship between the two automata. This makes the maintenance of such code through changes in specification a very onerous task.

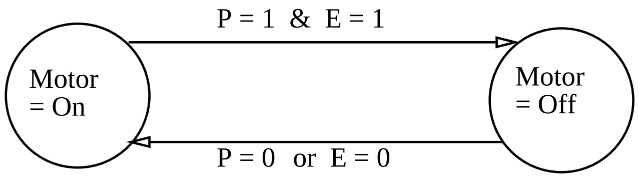

Figure 1: Automaton for a perfect paper-tray

2.2 Process calculi

Process calculi [Hoa85a,Mil89,MPW89] 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.

2.3 Temporal logic programming

Temporal logic programming languages [Brz91,BFG+90,Bau89,Mos86,Mer93] achieve bounded response by imposing syntactic restrictions. This paradigm is inherently nondeterministic. Furthermore, the languages are forced to identify *a priori*, global and fixed notions of "system-variables" and "environment-variables" to ensure true reactivity. This distinction has nebulous logical status, and goes against the algebraic view of parallel composition, where one process is part of the environment of the other process.

2.4 Synchronous languages

The problems with automata leads to the style of synchronous programming; the motivation behind these languages is to achieve the strong real time guarantees of automata in the context of a higher level of system description.

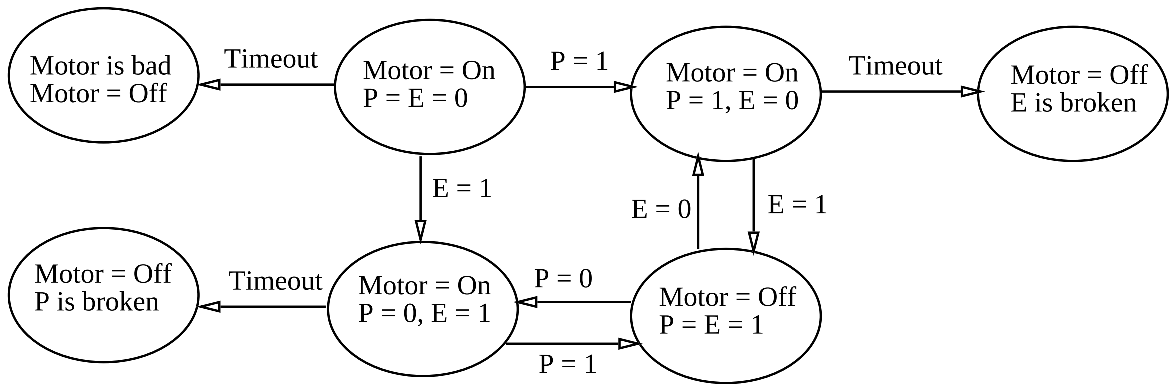

Figure 2: Automaton for a paper-tray with failure modes

The class of synchronous languages [BG92,HCP91,GBGM91,Har87] have the following features:

Perfect Synchrony: Program combinators are determinate primitives that respond instantaneously to input. Output at time t is a function of the input at time up to t; at any instant the presence *and* the absence of signals can be detected.

Bounded State: Programs that operate only on "signals" can be compiled into finite state automata with simple transitions.

Multiform Time: Physical time has the same status as any other external event.

Determinacy and synchrony ensure that programs are determinate in the input-output sense as well as the interactive temporal sense. The bounded state property bounds the single step execution time of the program and makes the synchrony assumption realizable in practice. The multiform nature of time allows the combination of programs with different notions of time. There are two classes of languages in this genré: Imperative real time synchronous languages like ESTEREL allow flexible programming constructs, and come with verification tools. The declarative synchronous real time languages, LUSTRE and SIGNAL, inherit elegant equational reasoning from the underlying dataflow model.

The incongruity between internal temporality (there is computation "in between" the stimulus from the environment and the response) and Perfect Synchrony (at each instant a signal is either present or not present, there is no "in between") affects programming language design and implementation. For example,

Temporal paradoxes: One can express programs that require a signal to be present at an instant only if it is not present at that instant. Indeed, the behavioral semantics of [BG92] requires (effectively) complex default reasoning to determine what must happen at each step — in a sense, "stable models" of a default theory must be computed by finding the fixed-points of a system of equations involving non-monotone operators. Approximate static analysis, a calculus of potentials in the case of ESTEREL, is used to eliminate programs with paradoxes (exact analysis is not possible for standard computability reasons). This results, in some cases, in rejecting intuitively correct programs [Hal93]. Furthermore, this poses problems in constructing process networks with cycles.

Compilation is not compositional: The compilation of a program fragment has to take into account the static structure of the entire program [Hal93, Page 93]. This is in direct contrast to the standard compilation procedures for traditional programming languages [ASU85].

2.5 Our contributions

A re-analysis of the elegant ideas underlying synchronous programming, starting from the viewpoint of asynchronous computation leads us to the framework of *timed concurrent constraint programming*, henceforth called tcc, with the following salient features.

Declarative view: tcc has a fully-abstract semantics based on solutions of equations. tcc programs can be viewed as formulas in an intuitionist linear time temporal logic, and computation can be viewed as generating a "least model" 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 : tcc supports the *derivation* of a "**clock**" construct that allows a process to be clocked by another (recursive) process; it generalizes the *undersampling* constructs of SIGNAL and LUSTRE, and the preemption/abortion constructs supported by ESTEREL. Thus, tcc encapsulates the rudiments of a theory of preemption constructs. In addition, by selecting an appropriate underlying constraint system (e.g., Herbrand underlying (concurrent) logic programming languages), tcc languages can be used to specify cyclic, dynamically-changing networks of processes (c.f. "mobility" [MPW89]).

Executability : Under simple syntactic conditions, tcc programs may be compiled into finite state "constraint" automata that have bounded response time.

"Paradox-free" : tcc resolves the tension between synchrony and causality.

2.6 Basic Intuitions

Our starting point is the paradigm of concurrent constraint programming, henceforth called CCP [Sar93,SRP91]. CCP is highly expressive; for example, it generalizes dataflow languages,

CCP is an asynchronous and determinate paradigm of computation. The basic move in this paradigm is to replace the notion of store-as-valuation central to von Neumann computing with the notion that the store is a constraint, that is, a collection of pieces of partial information about the values that variables can take. Computation progresses via the monotonic accumulation of information. Concurrency arises because any number of agents may simultaneously interact with the store. The usual notions of "read" and "write" are replaced by *ask* and *tell* actions. A tell operation takes a constraint and conjoins it with the constraints already in the store; tells are executed asynchronously: thus, it is guaranteed that the store eventually satisfies the constraint, but no assumptions are made on the amount of time taken to achieve this result. Synchronization is achieved via the ask operation: ask takes a constraint (say, c) and uses it to probe the structure of the store: it succeeds if the store contains enough information to entail c; the agent blocks if the store is not strong enough to entail the constraint it wishes to check.

The information accumulated by computation in the CCP paradigm is *positive* information: constraints on variables, or a piece of history recording that "some event happened". The fundamental move we now make is to consider the addition of *negative* information to the computation model: information of the form "an event did not happen". This is the essence of the "time out" notion in real-time programming: some sub-program may be aborted because of the absence of an event.

How can a coherent conceptual framework be fashioned around the detection of negative information? The first task is to identify states of the system in which no more positive information is being generated; the *quiescent* points of the computation. Only at such moments does it make sense to say that a certain piece of information has not been generated. Now *time* can be introduced by identifying quiescent points as the markers that distinguish one time step from the next. This allows the introduction of primitives that can trigger an action in the *next* time interval if some event did not happen through the extent of the *previous* time interval. Since actions of the system can only affect its behavior at current or future time step, the negative information detected is *stable*. Hence there is some hope that a semantic treatment as pleasant as that for the untimed case will be possible. Our basic ontological commitment then is to the following refinement of the Perfect Synchrony Hypothesis — the *Timed Asynchrony Hypothesis*:

Bounded Asynchrony Computation progresses asynchronously in bounded intervals of activity separated by arbitrary periods of quiescence. Program combinators are determinate primitives that respond in a bounded amount of time to the input.

Strict Causality Output at time t is a function of the positive information input upto and including time t and the negative information input at time upto t. The *absence* of a signal in a period can be detected only at the end of a period, and hence can trigger activity only in the next interval.

The Timed Asynchrony hypothesis addresses one practical and mathematical shortcoming of the Perfect Synchrony hypothesis. It is more pleasant and practical to regard the computation at each external clock tick as having a notion of *internal temporality*: it happens *not instantaneously* but over a *very short* — bounded — period of time. This is the reality in any case — sophisticated embedded real-time controllers may require that certain conditions be checked, and then values created for local variables, and then subsequently some other conditions be checked, etc. Indeed, even ESTEREL has such an internal notion of temporality — but it is achieved through a completely separate mechanism, the introduction of assignable variables and internal sequencing (";"). In contrast, `tcc` does away with the assignable variables of ESTEREL in favor of the same logical (denotational) variables that are used for representing signals. Computation progresses through the asynchronous accumulation and detection of positive information within an interval, and negative information at the end of an interval[1].

To summarize, computation in `tcc` proceeds in intervals. During the interval, positive information is accumulated and detected asynchronously, as in CCP. At the end of the interval, the absence of information can be detected, and the constraints accumulated in the interval are discarded. We do not provide for the implicit transfer of positive information across time boundaries to maintain the bounded size of the constraint store; this must be done explicitly by the programmer by using the basic combinators.

[1] Prima facie, this "monotonic approximation" of ESTEREL may seem to be less powerful than ESTEREL, which can react to negative information at the same clock tick. The variety of programming examples in the rest of the paper support our argument for the expressiveness of `tcc`.

3 The tcc model

This section is devoted to reviewing the formalization and technical development [SJG93] of the intuitions discussed in the previous section. Some readers may prefer to look at the more concrete examples in subsequent sections, and consult this section as needed for technical definitions. For this purpose, the syntax, denotational and operational semantics of tcc are summarized in Tables 1,2,3 respectively.

3.1 Constraint systems

Formally, constraint systems $\mathcal{C}$ are essentially arbitrary systems of partial information [Sar92,Sar93]. They are taken to consist of a set D of tokens with minimal first-order structure (variables, substitutions, existential quantification and conjunction), together with an entailment relation $\vdash_{\mathcal{C}}$ that specifies when a token x must follow given the tokens $y_1, \ldots, y_n$ are known to hold. Intuitionistic disjunction can be built in purely formally on top of the underlying constraint system. In particular, the constraint system needs to provide us only with an algorithm that will answer queries of the form "is c entailed by the store S". To answer the query " is $c \vee d$ entailed by the store S", send two different queries for c and d and return yes if either returns yes. Conversely, when a constraint $c \vee d$ is sent to the store, the store appears as if split into two, one receiving c the other d. All subsequent queries are answered successfully only if *both* splits answer successfully.

Examples of constraint systems: (1) Herbrand underlying logic programming (variables range over finite trees, tokens specify equality of terms), (2) FD [HSD92] for finite domains (variables range over finite domains, with tokens for equality and for specifying that a variable lies in a given range), etc.

A third example, Gentzen, will be the concrete constraint system used in the programming examples in this paper. For this constraint system, we assume that we are given a vocabulary of terms that includes variables, and a vocabulary of predicates with arities. The tokens of the constraint system are then atomic formulas, built out of the predicates and terms. The entailment relation is built out of only structural rules, i.e. is induced by the identity relation on atomic formulas.

In this section, we will take as fixed a constraint system $\mathcal{C}$, with underlying set of tokens D. Let the entailment closed subsets of D be denoted by $|D|$. $(|D|, \subseteq)$ is a complete algebraic lattice; we will use the notation $\sqcup$ and $\sqcap$ for the joins and meets of this lattice. We shall call (finite) sets of tokens *primitive constraints* and designate them by the letters c, d, e. A (finite) *constraint* is an element of $|D|$ generated from a (finite) set of tokens; they will be designated by the letters a, b. For any primitive constraint c, we let $[\![c]\!]$ stand for the constraint generated by c.

3.2 Denotational intuitions

Earlier work [SRP91] identified a cc program with the observations that can be made about the program. The observations of a program are the set of quiescent points – the set of elements of $|D|$ on which the program quiesces without adding any new information. It was further shown that the set S of quiescent points of a program is *determinate* that is, satisfies the property that for any point x, the greatest lower bound (glb) of any non-empty subset of S above x is also in S.[2] Such a model is rich enough to define an "abort" process that, in fact, quiesces on no input (and has hence the empty set of quiescent points).

What should the timed model be? Recall that the execution within a time step is captured by execution in cc and that discrete time is modeled by natural numbers. Combining these two ideas, we conclude that the timed model should be a sequence of models of cc programs. Formally, this is captured by the intuition that "Processes are relations extended over time" [Abr93].

The formal development follows. First, we extend the set of observations over time:

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

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

[2]Such a set is said to be determinate because it can be used to define a partial function: associate with every point x in the underlying partial set T, the least element of S above x, if any. These functions are partial *closure operators*, i.e., they are monotone, idempotent and extensive.

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

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

1. (*Non-emptiness*) $\epsilon \in P$,
2. (*Prefix-closure*) $s \in P$ whenever $s \cdot t \in P$, and
3. (*Determinacy*) P **after** s is determinate whenever $s \in P$.

□

We will let **Proc** designate the set of all processes, and let P, Q range over **Proc**.[3]

3.3 Process algebra

We now identify basic processes and process combinators in the model. The combinators fall into two categories:

CCP constructs: Tell, Parallel composition and Timed Positive Ask are inherited from CCP. These do not, by themselves, cause "extension over time". Parallel composition is the explicit representation of concurrency in the language. Timed Positive Ask and Tell combine to allow synchronization capabilities.

Timing constructs: Timed Negative Ask, Unit Delay, and Abortion that cause extension over time. Unit delay starts a process to be started in the next time instant. Timed Negative Ask is a conditional version of Unit Delay, based on detection of negative information. It causes

[3] An alternate way of thinking of processes is that they correspond exactly to the resting points of partial closure operators on **Obs** (in the lexicographic ordering on sequences) that are *stratified* ($f(s)$, if defined, has the same cardinality as s), and *causal* ($f(s)$ is a prefix of $f(s \cdot a)$, assuming both are defined). Space does not permit discussing this alternate viewpoint in detail.

Syntax.

(Agents)	A	$::=$	c	(Tell)
		$\mid$	**now** c **then** A	(Timed Positive Ask)
		$\mid$	**now** c **else** A	(Timed Negative Ask)
		$\mid$	**next** A	(Unit Delay)
		$\mid$	**abort**	(Abort)
		$\mid$	**skip**	(Skip)
		$\mid$	$A \parallel A$	(Parallel composition)
		$\mid$	$X\hat{\ }A$	(Hiding)
		$\mid$	g	(Procedure Call)
(Procedure Calls)	g	$::=$	$p(t_1, \ldots, t_n)$	
(Declarations)	D	$::=$	$g :: A$	(Definition)
		$\mid$	$D.D$	(Conjunction)
(Programs)	P	$::=$	$\{D.A\}$	

Table 1: Syntax for tcc languages

a process to be started in the the next time instant if on quiescence of the current time instant, the store was not strong enough to entail some information.

The formal description of these combinators follows. We present the denotational and operational semantics simultaneously.

The operational semantics is defined via two binary transition relations $\longrightarrow, \leadsto$ over configurations. $\longrightarrow$ represents transitions *within* a time instant. This relation is intimately related to the transition relation in CCP since it represents computation within a time step. $\leadsto$ represents a transition from one time instant to the next. A configuration will be simply a (possibly empty) multiset of *agents*, the syntactic representatives of processes. We implicitly represent the store in a configuration: for Γ a multiset of agents, we will let $\sigma(\Gamma)$ be the sub-multiset of tokens in Γ.

3.4 Timing constructs

Skip. **skip** is the process that does nothing at all at every time instant. Hence every sequence of constraints is quiescent for it. Thus:

Semantic Domain. We take as fixed a constraint system $\mathcal{C}$, with underlying set of tokens D and let the finitary inference relation $\vdash$ record which token follows from which ollection of tokens. Let the $\vdash$-closed subsets of D be denoted by $|D|$. $(|D|, \subseteq)$ is a complete algebraic lattice; we will use the notation $\sqcup$ and $\sqcap$ for the joins and meets of this lattice. A (finite) *constraint* is an element of $|D|$ generated from a (finite) set of tokens. We usually denote (finite) sets of tokens by the letters c, d, e, and constraints by the letters a, b. For any set of tokens c, we let $[\![c]\!]$ stand for the constraint generated by c.

Definition 3.3 **Obs**, the set of observations, is the set of finite sequences of constraints. □

Concatenation of sequences is denoted by "$\cdot$"; for this purpose a constraint a is regarded as the one-element sequence $\langle a \rangle$. For any $S \subseteq \mathbf{Obs}$, and sequence $s \in \mathbf{Obs}$, let S **after** s be the set $\{a \in |D| \mid s \cdot a \in S\}$. A subset S of a partially ordered set T is said to be *determinate* if for every $x \in T$, the subset of S above x is empty or contains a minimum.

Definition 3.4 A process $P \in \mathbf{Proc}$ is a non-empty, prefix-closed subset of **Obs** such that P **after** s is determinate for all $s \in P$. □

Semantic Equations. For a sequence of constraints s, by $\exists X.s$ we shall mean the element-wise existential quantification over X.

$$
\begin{array}{rcl}
[\![c]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \supseteq [\![c]\!]\} \\
[\![\mathbf{now}\ c\ \mathbf{then}\ A]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \supseteq [\![c]\!] \Rightarrow d \cdot s \in [\![A]\!]\} \\
[\![\mathbf{now}\ c\ \mathbf{else}\ A]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \not\supseteq [\![c]\!] \Rightarrow s \in [\![A]\!])\} \\
[\![\mathbf{next}\ A]\!] & = & \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid s \in [\![A]\!])\} \\
[\![\mathbf{abort}]\!] & = & \{\epsilon\} \\
[\![\mathbf{skip}]\!] & = & \mathbf{Obs} \\
[\![A \parallel B]\!] & = & \{s \in \mathbf{Obs} \mid s \in [\![A]\!] \wedge s \in [\![B]\!]\} \\
[\![X\hat{\ }A]\!] & = & \{s \in \mathbf{Obs} \mid \exists X.s = \exists X.t,\ \text{for some}\ t \in [\![A]\!]\}
\end{array}
$$

The semantics of recursion is handled in the usual way by taking least fixed points of the associated functional.

Table 2: Denotational Semantics for tcc languages

Configuration. A configuration is merely a multiset Γ of agents. Multiset union is viewed as parallel composition. We implicitly represent the store in a configuration: for Γ a configuration, $\sigma(\Gamma)$ denotes the sub-multiset of tokens in Γ.

Transition rules. Two binary transition relations $\longrightarrow, \leadsto$ are defined over configurations. $\longrightarrow$ represents transitions *within* a time instant. $\leadsto$ represents a transition from one time instant to the next.

Axioms for $\longrightarrow$. The binary relation $\longrightarrow$ on configurations is the least relation satisfying the rules:

$$
\begin{array}{rcl}
(\Gamma, \mathbf{skip}) & \longrightarrow & \Gamma \\
(\Gamma, \mathbf{abort}) & \longrightarrow & \mathbf{abort} \\
(\Gamma, A \parallel B) & \longrightarrow & (\Gamma, A, B) \\
(\Gamma, X\hat{\ }A) & \longrightarrow & (\Gamma, A[Y/X]) \quad (Y \text{not free in } \Gamma) \\
(\Gamma, p(t_1, \ldots, t_n)) & \longrightarrow & (\Gamma, A[X_1 \mapsto t_1, \ldots, X_n \mapsto t_n])
\end{array}
$$

and

$$
\frac{\sigma(\Gamma) \vdash c}{(\Gamma, \mathbf{now}\ c\ \mathbf{then}\ A) \longrightarrow (\Gamma, A)}
$$

$$
\frac{\sigma(\Gamma) \vdash c}{(\Gamma, \mathbf{now}\ c\ \mathbf{else}\ B) \longrightarrow \Gamma}
$$

In the case of procedure calls we assume that the underlying program contains a clause $p(X_1, \ldots, X_n) :: A$.

Axioms for $\leadsto$. The binary relation $\leadsto$ is the least relation satisfying the single rule:

$$
\frac{\Delta, \{\mathbf{now}\ c_i\ \mathbf{else}\ A_i \mid i < n\} \not\longrightarrow}{\Delta, \{\mathbf{now}\ c_i\ \mathbf{else}\ A_i \mid i < n\}, \{\mathbf{next}\ B_j \mid j < m\} \leadsto \{A_i \mid i < n\}, \{B_j \mid j < m\}}
$$

Above, Δ ranges over multisets of agents that do not contain Timed Negative Asks or Unit Delays.

Note that the empty configuration can always make a $\leadsto$ transition, and that a configuration containing **abort** can never quiesce (that is, if Γ contains **abort**, then there is always a Δ, namely, **abort**, such that $\Gamma \longrightarrow \Delta$). Therefore there are configurations Γ which are such that for no Δ_1, Δ_2 do we have:

$$
\Gamma \longrightarrow^* \Delta_1 \leadsto \Delta_2
$$

In that sense $\leadsto$ is "partial".

Table 3: Operational Semantics for tcc languages

$$[\![\mathbf{skip}]\!] \stackrel{d}{=} \mathbf{Obs}$$

Operationally, **skip** has no effect:

$$\Gamma, \mathbf{skip} \longrightarrow \Gamma$$

Abortion. **abort** is the process that instantly causes all interactions with the environment to cease. Hence the only observation that can be made of it is ϵ.

$$[\![\mathbf{abort}]\!] \stackrel{d}{=} \{\epsilon\}$$

Operationally, **abort** annihilates the environment:

$$\Gamma, \mathbf{abort} \longrightarrow \mathbf{abort}$$

Note that this implies that a configuration Γ containing **abort** cannot "$\longrightarrow$-quiesce", that is, reach, through $\longrightarrow$ transitions, a state which has no $\longrightarrow$ transition emanating from it.

Unit Delay. Unit Delay is a variant of the unit delay primitives in synchronous languages [Mil83,Ber93]; intuitively, $A = \mathbf{next}\ B$ is the process that behaves like B in the next time instant. It performs no computation in the present instant. When is $o = a \cdot s$, an arbitrary element of **Obs**, an observation of A? There are no restrictions on a. Since A behaves like B from the next time instant, s must be an observation of B.

$$[\![\mathbf{next}\ B]\!] \stackrel{d}{=} \{\epsilon\} \cup \{a \cdot s \in \mathbf{Obs} \mid s \in [\![B]\!]\}$$

Timed Negative Asks. Timed Negative Ask is the only way to detect "negative information" in tcc. Intuitively, $A = \mathbf{now}\ c\ \mathbf{else}\ B$ is the process that behaves like B in the next time instant if on quiescence of the current time instant the store was not strong enough to entail c.[4]

When is $o = a \cdot s$, an arbitrary element of **Obs**, an observation of A? There are two possibilities. Either $a \supseteq [\![c]\!]$ or not. If $a \supseteq [\![c]\!]$, A behaves like

[4] In reality, we allow the more general combinator $A = \mathbf{now}\ \{c_1; \ldots; c_n\}\ \mathbf{else}\ B$ — it behaves like B provided that the store on quiescence is not above each of the constraints c_i. The details are straightforward.

skip, so s can be any observation whatsoever. If not, A behaves like B from the next time instant; so s must be an observation of A. Thus:

$$[\![\mathbf{now}\ c\ \mathbf{else}\ A]\!] \stackrel{d}{=} \{\epsilon\} \cup \{a \cdot s \in \mathbf{Obs} \mid a \not\supseteq [\![c]\!] \Rightarrow s \in [\![A]\!]\}$$

Operationally, if the current store entails c, then we can eliminate A:

$$\frac{\sigma(\Gamma) \vdash c}{(\Gamma, \mathbf{now}\ c\ \mathbf{else}\ B) \longrightarrow \Gamma}$$

In the following for an agent **now** c **else** A we say that c is the "antecedent" and A the "body".

Operational semantics: the $\leadsto$ relation. Consider now the situation in which computation in the current time instant has quiesced. This means that all reductions that could have been caused because of the entailment of Positive Asks have been done, all Negative Asks whose antecedents are entailed have been eliminated, and computation has not aborted. Computation can now progress to the next time instant.

The active agents at the next time instant are the bodies of the remaining Negative Ask agents or the bodies of agents within a Unit Delay. All other agents, including the current store, will be discarded. Formally, if Δ ranges over multisets of agents (including constraints) other than Negative Ask or Unit Delay, we can write the transition rule as:

$$\frac{\Delta, \{\mathbf{now}\ c_i\ \mathbf{else}\ A_i \mid i < n\} \not\longrightarrow}{\Delta, \{\mathbf{now}\ c_i\ \mathbf{else}\ A_i \mid i < n\}, \{\mathbf{next}\ B_j \mid j < m\} \leadsto \{A_i \mid i < n\}, \{B_j \mid j < m\}}$$

Note that a configuration Γ can make a $\leadsto$-transition even if Γ is the empty set of agents; however, it cannot make any $\leadsto$-transition if it contains **abort**.

Guarded Recursion. A *program* is a pair of declarations D and an agent A; a declaration is of the form $g :: A$, where g is of the form $p(X_1, \ldots, X_n)$, for p a procedure name and X_i variables. (In the following, we let F, G range over programs.) We require that the recursion is guarded, *i.e.* the recursion variable occurs within the scope of an **else** or a **next**. The important consequence of this restriction is that the recursion equations have *unique* solutions, and that computation in each time-step is lexically bounded (i.e., bounded by the size of the program).

To ensure bounded "extension in time" we also need to ensure that recursions are *bounded*, that is, at run-time there are only boundedly many different procedure calls. There are several ways of achieving this. For instance, we could require that procedures take no parameters. This is a familiar restriction in the realm of synchronous programming; for example, the sole recursion construct in ESTEREL is "loop". We shall make the slightly more liberal assumption that any recursive procedure call take exactly the same parameters as the procedure declaration. That is, for any set of mutually recursive declarations:

```
p(X1, ..., Xn) :: A.
q(Y1, ..., Ym) :: B.
...
```

---

any call to a procedure p, q, ... in A, B, ... is exactly of the form $\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{n})$, $\mathtt{q}(\mathtt{Y}_1, \ldots, \mathtt{Y}_\mathtt{n})$, ... (This restriction is equivalent to the parameterless procedures restriction provided that procedure definitions are allowed to be nested within agents, so that bodies of procedure definitions can refer to variables in the lexical scope.)

From a logic programming perspective, this restriction is not as severe as it may seem because constraints are not carried over from one time instant to the next: hence the same variable name may be "reused" at subsequent time-instants. Note that procedures with parameters that take values in finite domains can be compiled away using the parameterless procedure and hiding constructs available in tcc.

The operational and denotational semantics of recursion is standard. In the following, we shall sometimes use the syntax $\mu p.A$ to denote an agent p under the assumption that the program contains a (unique) procedure declaration of the form $p :: A$.

3.5 Combinators from CCP

These combinators follows traditional treatment [SRP91].

Tell. This process adds c in the current store. It is quiescent in any observation that contains at least as much information as c at the first instant.

$$[\![c]\!] \quad = \quad \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \supseteq c\}$$

Since the store is represented in the configuration, there is no explicit operational transition.

Parallel composition. What are the quiescent points of the parallel composition of two agents agents A and B? Exactly those which are quiescent points of A and of B:

$$[\![A \parallel B]\!] \quad = \quad [\![A]\!] \cap [\![B]\!]$$

Operationally, the idea of parallel composition is already implicit in the "," of multiset union [BB90]:

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

Timed Positive Ask. $B = \mathbf{now}\ c\ \mathbf{then}\ A$ is the process that checks if the current store is strong enough to entail c; and if so, behaves like A. When is $o = d \cdot s$ an arbitrary element of $\mathbf{Obs}$ in $[\![B]\!]$? There are two possibilities. Either $d \supseteq [\![c]\!]$ or not. In the first case, B behaves like A; so o must be an observation of A. Otherwise o can be any observation whatsoever — since B will remain quiescent. Thus:

$$[\![\mathbf{now}\ c\ \mathbf{then}\ A]\!] \quad = \quad \{\epsilon\} \cup \{d \cdot s \in \mathbf{Obs} \mid d \supseteq [\![c]\!] \Rightarrow d \cdot s \in [\![A]\!]\}$$

Operationally, we have:

$$\frac{\sigma(\Gamma) \vdash c}{(\Gamma, \mathbf{now}\ c\ \mathbf{then}\ A) \longrightarrow (\Gamma, A)}$$

Hiding. $X\hat{\ }A$ is a process where the variable X is hidden from the environment. At any time step, the agent $\exists X.f$ transforms its input c into the value $c \sqcup \exists X.f(\exists X.c)$. No constraints cross time boundaries. This yields, in terms of fixed points:

$$[\![X\hat{\ }A]\!] \quad = \quad \{s \in \mathbf{Obs} \mid \exists X.s = \exists X.t, \text{for some } t \in [\![A]\!]\}$$

where $\exists X.(c_1.c_2 \ldots . c_n) \stackrel{d}{=} (\exists X c_1).(\exists X c_2) \ldots (\exists X.c_n)$.

For the operational semantics, we find a variable Y that does not occur free in the configuration, and substitute it for the bound variable:

$$(\Gamma, X\hat{\ }A) \quad \longrightarrow \quad (\Gamma, A[Y/X]) \quad (Y\text{not free in } \Gamma)$$

Review. This concludes our discussion of the basic mathematical and operational semantics of tcc constructs. The connections between operational and denotational semantics are straightforward. Indeed, it is not hard to determine that the denotational semantics captures precisely the quiescent sequences of the operational semantics. Full abstraction follows from standard considerations [SJG93].

We emphasize that the definitions given above work for *any* constraint system, and are hence very general. While we do not have space to discuss this further [SJG93] also provides a sound and essentially complete proof system for establishing inequivalence of programs. Because tcc programs have a logical reading, they may also be used to specify (safety) properties of other tcc programs. Hence this proof system can be used for establishing that programs meet their safety specifications.

[SJG93] also discusses a compositional compilation algorithm for tcc programs that takes as input programs with bounded recursion (as discussed above) and outputs a constraint-based FSA. The FSA accepts as input a constraint from the environment, computes (through a bounded number of calls to the underlying constraint system) the constraint that must be output at the current time step, and checks it against a bounded number of other constraints to determine the transition to the next state. The mere fact of such a compilability into FSA's makes possible the use of tcc to program reactive controllers, and the use of powerful analysis techniques for finite state systems.

4 General programming techniques

We now show how to define some natural patterns of temporal activity in tcc languages. We show that several interesting combinators are definable in the underlying model, and are hence definable for any tcc language.[5] We then present explicit definitions through simplification laws that show how to eliminate the combinators in favor of the basic combinators described in the previous section. (In the following when we write $A = B$, for two agents A and B, we mean that the denotations of these agents are identical processes.)

4.1 Some simple combinators

Unconditional persistence. Let **always** A be the agent that behaves like A at every time instant:

$$[\![\textbf{always}\ A]\!] = \{s \in \textbf{Obs} \mid s^i \in [\![A]\!], 0 \leq i \leq |s|\}$$

where $s^0 = s$ and $(d \cdot s)^{n+1} = s^n$. We have the law:

$$\textbf{always}\ A = A \parallel \textbf{next always}\ A$$

Thus the agent **always** c ensures that the constraint c will be posted at every time instant.

Conditionals. Let **now** c **then** A **else** B be the agent that behaves like A if c holds in the current time instant; otherwise it behaves like B from the *next* time instant onwards. Its observations are:

$$\begin{array}{lll}[\![\textbf{now}\ c\ \textbf{then}\ A\ \textbf{else}\ B]\!] = & \{\epsilon\} \cup \{d \cdot s \in \textbf{Obs} \mid & d \supseteq [\![c]\!] \Rightarrow d \cdot s \in [\![A]\!], \\ & & d \not\supseteq [\![c]\!] \Rightarrow s \in [\![B]\!]\}\end{array}$$

From this we can deduce:

$$\textbf{now}\ c\ \textbf{then}\ A\ \textbf{else}\ B \quad = \quad (\textbf{now}\ c\ \textbf{then}\ A \parallel \textbf{now}\ c\ \textbf{else}\ B)$$

[5]When we say that a combinator is definable in this model we mean that when it takes processes as arguments it returns a process. Furthermore, unless otherwise discussed all combinators will be monotone and continuous in their process arguments. This means that they can be used freely in recursive definitions.

Note that the conditional construct will take one branch or the other; there are no conditions under which it can "suspend". In contrast, assuming that the constraint $\neg c$ is definable in the underlying constraint system, the agent (**now** c **then** $A \parallel$ **now** $\neg c$ **then next** B) can suspend without executing either A in the current step or B in the next step — exactly in those stores which are not strong enough to entail either c or $\neg c$.[6] This is a reflection of the intuitionistic nature of the implication underlying the **now** c **then** A construct.

Multiple Prioritized Waits. Let A be the agent:

$$\begin{array}{ll}\textbf{now} & \\ & \textbf{case}\ \ c_1\ \textbf{do}\ A_1 \\ & \textbf{case}\ \ c_2\ \textbf{do}\ A_2 \\ & \ldots \\ & \textbf{case}\ \ c_n\ \textbf{do}\ A_n \\ & \textbf{default}\ \ B \\ \textbf{end} & \end{array} \tag{1}$$

defined to behave like A_1 if c_1 is now true, else behaves in the next time instant like A_2 if c_1 is not true, but c_2 was true, and so on, behaving like B if none of the c_i are true. Following the same denotational style reasoning as earlier, A is equivalent to the agent:

$$\begin{array}{ll} & \{\epsilon\} \\ \cup & \{d \cdot s \in \mathbf{Obs} \mid d \supseteq [\![c_1]\!], d \cdot s \in [\![A_1]\!]\} \\ \cup & \cup_{2 \leq i \leq n}\{d \cdot s \in \mathbf{Obs} \mid \forall j < i.\ d \not\supseteq [\![c_j]\!], d \supseteq [\![c_i]\!], s \in [\![A_i]\!]\} \\ \cup & \{d \cdot s \in \mathbf{Obs} \mid \ \mid \forall j \leq n.\ d \not\supseteq [\![c_j]\!], s \in [\![B]\!]\} \end{array}$$

From the structure of the denotation, it is easy to write down the following equivalent agent in the basic language:

$$\begin{array}{ll} & \textbf{now}\ c_1\ \textbf{then}\ A_1 \\ \parallel & \textbf{now}\ c_2\ \textbf{then}\ \textbf{now}\ c_1\ \textbf{else}\ A_2 \\ \parallel & \textbf{now}\ c_3\ \textbf{then}\ \textbf{now}\ (c_1; c_2)\ \textbf{else}\ A_3 \\ \ldots & \\ \parallel & \textbf{now}\ c_n\ \textbf{then}\ \textbf{now}\ (c_1; \ldots; c_{n-1})\ \textbf{else}\ A_n \\ \parallel & \textbf{now}\ (c_1; \ldots; c_n)\ \textbf{else}\ B \end{array}$$

[6]For example, if c is the constraint `X=1`, then every store which has no constraints on `X` will be unable to prove one of `X=1` or `X` $\neq$ `1`.

4.2 Temporal control constructs

Extended Wait. Let **whenever** c `do` A be the agent that suspends until the first time-instant at which c is true; it then behaves like A. Its denotation is:

$$[\![\textbf{whenever } c \texttt{ do } A]\!] \quad = \quad \begin{array}{ll} & \{\epsilon\} \\ \cup & \{s \cdot d \cdot t \in \textbf{Obs} \mid d \supseteq [\![c]\!], d \cdot t \in [\![A]\!]\} \\ \cup & \{s \cdot d \in \textbf{Obs} \mid d \not\supseteq [\![c]\!]\} \end{array}$$

with the law:

$$\textbf{whenever } c \textbf{ do } A = \textbf{now } c \textbf{ then } A \textbf{ else } (\textbf{whenever } c \textbf{ do } A)$$

Watchdogs: do P watching c. The Esterel interrupt construct **do** A **watching** S [Ber93] cannot be translated precisely into `tcc`. In Esterel the computation of A is aborted at the instant that S is emitted. This leads to semantic complexities — for example, the agent **do** S **watching immediately** S is required to produce S but if it were to produce S it must terminate before producing S! In `tcc` it is not possible to specify that a computation be aborted in the current instant on receipt of positive information from the environment. Instead, we describe a construct that aborts the computation at the next *stable* instant, that is, at the next moment of quiescence. This is achieved through polling as shown below.

do P **watching** c behaves like P till a time instant when c is entailed; when c is entailed P is killed from the next time instant onwards. (We can similarly define the related *exception handler* primitive, **do** P **watching** c **timeout** B, that also activates a handler B when P is killed.)

In the following, for $s \in \textbf{Obs}$, we use the notation $s \not\geq a$ to mean $b \not\supseteq a$ for every element $b \in s$. Formally,

$$[\![\textbf{do } P \textbf{ watching } c]\!] \quad \stackrel{d}{=} \quad \begin{array}{ll} & \{s \in \textbf{Obs} \mid s \in [\![P]\!], s \not\geq [\![c]\!]\ \} \\ \cup & \{s \cdot a \cdot t \in \textbf{Obs} \mid s \cdot a \in [\![P]\!], a \supseteq [\![c]\!]\} \end{array}$$

It satisfies the following laws:

$$\textbf{do } d \textbf{ watching } c \quad = \quad d$$

$$\begin{aligned}
\textbf{do } (A_1 \parallel A_2) \textbf{ watching } c &= \textbf{do } A_1 \textbf{ watching } c \parallel \textbf{do } A_2 \textbf{ watching } c \\
\textbf{do } (\textbf{now } d \textbf{ then } A) \textbf{ watching } c &= \textbf{now } d \textbf{ then } (\textbf{do } A \textbf{ watching } c) \\
\textbf{do } (\textbf{now } d \textbf{ else } A) \textbf{ watching } c &= \textbf{now } (c; d) \textbf{ else } (\textbf{do } A \textbf{ watching } c) \\
\textbf{do } (X\hat{}A) \textbf{ watching } c &= X\hat{}(\textbf{do } A \textbf{ watching } c)
\end{aligned}$$

In the last rule, we assume that X is not free in c (this can always be achieved by renaming the bound variable).

We believe that the "monotonic" approximation of asynchronous interrupts afforded by `tcc` is appropriate for all practical purposes. Consider the example of an Automated Teller Machine (presented in more detail in Section 6.1.2). In the case in which the customer presses a `start` button, the action desired (say, A) might be different from that (say B) in which the customer simultaneously presses a `start` and `cancel` button. In `tcc` the negative information that `cancel` was not pressed must be converted into positive information — and this can only be done at the next time instant. So the action associated with only the `start` button being pressed would be taken for one time step before being aborted.

If the exact functionality of ESTEREL is desired, this can be obtained by sampling the button state at a higher frequency than the frequency driving the main program. (For example, this can be achieved by downsampling from the basic clock — indeed there needs to be a clock more basic than the one associated with button-presses in order to time out an extremely slow customer.) The button sampler would use an adaptation of the mouse protocol (see below) to determine exactly how many buttons were pressed in a given interval and communicate that positive information at the end of the interval to the main program. The main program can then do A on receipt of the information ($\texttt{start}, \neg\texttt{cancel}$) and B on receipt of ($\texttt{start}, \texttt{cancel}$).

Suspension-Activation primitive: $\mathbf{S}_c\mathbf{A}_e(P)$. This is a preemption primitive that is a variant of *weak suspension* in ESTEREL [Ber93]. $\mathbf{S}_c\mathbf{A}_e(P)$ behaves like P till a time instant when c is entailed; when c is entailed P is suspended from the next time instant onwards (hence the $\mathbf{S}_c$). P is re-activated in the time instant when e is entailed (hence the $\mathbf{A}_e$). This, for instance, is the behavior of the ($\texttt{control} - \texttt{Z}, \texttt{fg}$) mechanism in Unix.

$$\begin{array}{rcl}[\![\mathbf{S}_c\mathbf{A}_e(P)]\!] & \stackrel{d}{=} & \{s \in \mathbf{Obs} \mid s \in [\![P]\!], s \not\supseteq [\![c]\!]\} \\ & \cup & \{s \cdot a \cdot t \in \mathbf{Obs} \mid s \cdot a \in [\![P]\!], s \not\supseteq [\![c]\!], a \supseteq [\![c]\!], t \not\supseteq [\![e]\!]\} \\ & \cup & \{s \cdot a \cdot t \cdot a' \cdot u \in \mathbf{Obs} \mid \; s \cdot a \cdot a' \cdot u \in [\![P]\!], s \not\supseteq [\![c]\!], \\ & & \qquad d \supseteq [\![c]\!], t \not\supseteq [\![e]\!], d' \supseteq [\![e]\!]\}\end{array}$$

Multiform time: time A on c. One of the most attractive aspects of the synchronous programming languages is that no particular notion of time is built in. There is no basic signal, say a milli-second clock, with control constructs that are defined explicitly for that signal. (This is done, for instance by Timed CSP [RR86].) Rather, the basic temporal sequencing constructs can be used uniformly with respect to *any* signal that extends over time.

On first glance, it might seem that the introduction of Unit Delay and Negative Ask constructs refer explicitly to an underlying clock and hence do not support a multi-form notion of time. This is however not the case. We shall see shortly that in fact it is possible to uniformly clock arbitrary processes using a general class of processes. For the moment, we concentrate on showing how an arbitrary process A may be exposed to some "sub-sampling" of the basic clock.

Intuitively, **time** A **on** c should denote a process whose notion of time is the occurrence of the primitive constraint c — A evolves only at the time instants at which the store entails c. Those instants at which c is not entailed by the store should be regarded as completely invisible to A. In particular this implies that the notion of the "next time instant" in A will in fact be the "next time instant at which c holds".

Formally, using the notation $s \downarrow a$ to denote the subsequence of s with all elements $\supseteq a$, we have:

$$[\![\mathbf{time}\ A\ \mathbf{on}\ c]\!] \stackrel{d}{=} \{s \in \mathbf{Obs} \mid s \downarrow [\![c]\!] \in [\![A]\!]\}$$

The laws for **time** A **on** c are easy to write down. We defer giving an explicit presentation because we now show how to define this construct in terms of the more general **clock** construct.

4.3 A general clocking construct: the clock combinator

The combinators discussed above have a common basic idea — they allow for the introduction of time steps that are ignored by the underlying agent A. This suggests the introduction of a combinator that directly captures this intuition: **clock** B **do** A is a process that executes A only on those instants which are quiescent points of B.

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

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

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

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

It is easy to see that **clock** B **do** A is non-empty and prefix-closed if $[\![B]\!]$ and $[\![A]\!]$ are. However, **clock** B **do** A may not be determinate for arbitrary $[\![B]\!]$. A necessary and sufficient condition is that $[\![B]\!]$ **after** s is upwards-closed, for all $s \in [\![B]\!]$. (Such processes can be thought of as arising by "extending over time" the basic processes of the form **tell** c.) Accordingly, we identify the syntax for "basic processes" by:

$$\begin{array}{llll} \text{(Basic Agents)} & B & ::= & c \\ & & \mid & \textbf{now } c \textbf{ then next } B \\ & & \mid & \textbf{now } c \textbf{ else } B \\ & & \mid & \textbf{next } B \\ & & \mid & \textbf{skip} \\ & & \mid & \textbf{abort} \\ & & \mid & B \parallel B \\ & & \mid & X\hat{\ }B \\ & & \mid & g \\ \\ \text{(Basic Procedures)} & D & ::= & g :: B \end{array}$$

Note that $[\![\mathbf{always}\ A]\!]$ and $[\![\mathbf{whenever}\ c\ \mathbf{do\ next}\ A]\!]$ are basic processes if $[\![A]\!]$ is.

The **clock** combinator is not monotone or anti-monotone in its first argument. Nevertheless, it is possible to establish

$$\mathbf{clock}\ (\mu g.B)\ \mathbf{do}\ A \quad = \quad \mu g.\mathbf{clock}\ B\ \mathbf{do}\ A$$

using the fact that recursion is guarded and fixed-points are unique.

Table 6 in Section 7 shows how to compositionally reduce programs containing this construct into programs that do not. (The table is given in the syntax of Timed Gentzen, explained in Section 5.)

We illustrate how some standard temporal constructs can be expressed using **clocks**. Clearly, this construct is in the flavor of the `when` construct (undersampling) in LUSTRE and SIGNAL, generalizd to general processes B instead of boolean streams. The combinators introduced above can be expressed thus:

$$\begin{aligned}
\mathbf{now}\ c\ \mathbf{else}\ A &= \mathbf{clock}\ (\mathbf{now}\ c\ \mathbf{then\ next\ abort})\ \mathbf{do\ next}\ A \\
\mathbf{whenever}\ c\ \mathbf{do}\ A &= \mathbf{clock}\ c\ \mathbf{do}\ A \\
\mathbf{time}\ A\ \mathbf{on}\ c &= \mathbf{clock}\ (\mathbf{always}\ c)\ \mathbf{do}\ A \\
\mathbf{do}\ A\ \mathbf{watching}\ c &= \mathbf{clock}\ (\mathbf{whenever}\ c\ \mathbf{do\ next\ abort})\ \mathbf{do}\ A \\
\mathbf{S}_c\mathbf{A}_e(A) &= \mathbf{clock}\ (\mathbf{whenever}\ c\ \mathbf{do\ next}\ e)\ \mathbf{do}\ A
\end{aligned}$$

Repeated pause/resumptions of a process can be expressed by:

$$\mathbf{clock}\ (\mu g.\mathbf{whenever}\ c\ \mathbf{do\ next}\ (e \parallel \mathbf{next}\ g))\ \mathbf{do}\ A$$

5 The language Timed Gentzen

We now turn to a discussion of Timed Gentzen, a concrete tcc programming language. Subsequently, we shall examine several programming idioms in Timed Gentzen, and discuss a simple implementation.

Syntax. The syntax of the language is summarized in Table 4. In general, Prolog lexical and syntactic conventions are followed. For technical reasons, Timed Gentzen source programs are prevented from using the functor `'_'/2`; this is reserved for internal use.

The syntax `{C}` is used to say that the constraints (atomic formulas) in `C` are to be added to the store. The syntax `C` $\rightarrow$ `A` is used for **now** C **then** A, `C` $\leadsto$ `A` for **now** C **else** A.

Parametric asks are used to communicate "data" with signals: they allow an agent to output in the store an atom with argument structure (e.g. `b(2,s)`), and another agent to explicitly recover the values: e.g., the agent `X$Y$(b(X,Y)` $\rightarrow$ `A)` will cause the generation of the agent $\mathtt{A}[\mathtt{X} \mapsto 2, \mathtt{Y} \mapsto \mathtt{s}]$. For simplicity of implementation, we require that all the existentially and universally quantified Timed Gentzen variables occurring in a clause are distinct Prolog variables, and distinct from variables occurring in the head of a clause.[7]

The syntax `[A1, ..., An]` is used to signify the parallel composition of the agents `A1`, ..., `An`, for $\mathtt{n} \geq 0$. In particular, `[]` stands for **skip**.

Using ";", an agent may check whether the current store entails one of a set of constraints (Positive Ask), or does not entail each of a set of constraints (Negative Ask).

Semantics. Timed Gentzen allows a somewhat more liberal syntax for procedure declarations than we have discussed in Section 3 for the general tcc framework. The syntax is motivated by programming considerations — in particular, the conventions we adopt below allow for a simple and succinct meta-interpreter for the language.

Timed Gentzen allows multiple clauses for a procedure, and allows a clause head to contain (non-variable) terms. In the operational semantics, the first

[7]Thus, for example, the body `X^A, X^B` is considered syntactically ill-formed, as is the clause `p(a, X) :: f(X), X^A`.

clause whose head can be instantiated to match the call is chosen for procedure reduction.

It is easy to transform any such program into a program that satisfies the procedure declaration syntax for general `tcc` languages. Assume the program has `k` clauses `p(ti1, ..., tin)::Ai`, for $i \in 1 \ldots k$. Assume further that all the variables appearing in clause head `i` are distinct from those appearing in clause head `j`, and from the variables `X1, ..., Xn`. The new program contains a single clause for `p/n`:

```
p(X1, X2, ..., Xn) ::
   [V1$(C1 → A1),
    V2$(C2 → A2),
    ...
    Vk$(Ck → Ak)].
```

---

where

- $\mathtt{C1} = \mathtt{D1}$,
- $\mathtt{C2} = (\mathtt{D2}, \neg\exists \mathtt{V1.D1})$,
- $\mathtt{C3} = (\mathtt{D3}, \neg\exists \mathtt{V2.D2}, \neg\exists \mathtt{V1.D1})$
- ...
- $\mathtt{Ck} = (\mathtt{Dk}, \neg\exists \mathtt{V1.D1}, \ldots, \neg\exists \mathtt{V(k-1).D(k-1)})$,

and $\mathtt{Di} = (\mathtt{X1} \doteq \mathtt{ti1}, \mathtt{X2} \doteq \mathtt{ti2}, \ldots \mathtt{Xn} \doteq \mathtt{tin})$. Here, the usual axioms for equality (Clark's equality theory) are understood to be in force for $\doteq$. However the store is not allowed contain any $\doteq$ assertions. Hence for a particular call `p(s1, ..., sn)`, each equality $\mathtt{sj} \doteq \mathtt{tij}$ can be evaluated "instantaneously". There are no other agents running simultaneously who can cause more equality assertions to be generated, hence there is no need to "suspend". Either the conditions for a particular clause are applicable or not.

Thus, given the program:

```
p(a(Y)) :: q(Y).
p(a(Y)) :: s(Y).
p(X) :: r(X).
```

---

the goal `p(b)` will reduce to `r(b)`, the goal `p(a(1))` to `q(1)` (not `s(1)`, and not `[s(1),q(1)]`).

Other aspects. Three other aspects of Timed Gentzen relevant to (meta-) programming should be pointed out.

First, some rudimentary arithmetic and communication abilities are provided through Ask and Tell operations. Conceptually, one imagines that the store contains the infinite collection of assertions about the extension of the relevant arithmetic relations (e.g, `3 > 2`, `2 = 2`, etc.). Thus, the program fragment `X$(a(X), X > 3` $\rightarrow$ `proc(X))` will invoke `proc` only if the store contains an atom `a(N)` where `N` is a number great than `3`. An ersatz ask-predicate `read(X)` is provided to input (synchronously) a term from an input channel. Conceptually, one imagines that an external agent places the atom `read(t)` in the store, if `t` is the term returned by the read operation. Conversely, some output operations are provided as Tell operations — conceptually, these are thought as being added to the store, and being read by and executed on by some external agent.

Second, each of the combinators:

```
({}/1, ->/2, ~>/2,next/1,$/2,^/2,[]/0,'.'/2,::/2)
```

are also data-constructors. Hence a Timed Gentzen agent is also directly representable as a Timed Gentzen data-structure.

Third, it is assumed that the user program is available in the store, as a collection of assertions of the form ($\forall$)`Head :: Body`. Asks are allowed to query this database, with the restriction that in all queries of the form `t::s` in the body of a clause K, all variables occurring in `t` must occur in the head of K. This ensures that by the time `t::s` has to be executed in the body of the clause, `t` will contain no uninstantiated universally quantified variables.

Such queries can be implemented by merely doing a lookup of the program stored in a Prolog database, and returning the first answer.[8]

Thus, the following is a legal `tcc` clause:

$$\texttt{tcc(Proc)} :: \texttt{Body}\$(\texttt{Proc} :: \texttt{Body} \rightarrow \texttt{tcc(Body)}).$$

(and indeed is a crucial clause in the definition of a meta-interpreter for Timed Gentzen, Section 6.4). Given a procedure call, say `tcc(f(3))`, the effect of this clause would be to determine the unique body, say `B` for the procedure call `f(3)`, and invoke `tcc(B)`.

The approach we have chosen to make the program available at run-time is very conservative. Clearly, it is possible to think of the program changing arbitrarily from one time-step to the next, just as the store does. Indeed, conceptually the program and the content of the store are identical — a program happens to have more logical structure in its formulas than the store does. We leave the investigation of this line of approach for future work.

[8] Without this restriction, the implementation could be forced to return all possible instantiations of the query. This can become complicated, given the semantics for mutually exclusive clauses above.

$$
\begin{array}{llcll}
\text{(Agents)} & \mathtt{A} & ::= & \{\mathtt{C}\} & \text{— Constraint Tells} \\
 & & | & \mathtt{E} \ \rightarrow \ \mathtt{A} & \text{— Positive Ask} \\
 & & | & (\mathtt{X}_1, \ldots, \mathtt{X}_n)\$\mathtt{E} \ \rightarrow \ \mathtt{A} & \text{— Parametric (positive) Ask} \\
 & & | & \mathtt{E} \leadsto \mathtt{A} & \text{— Negative Ask} \\
 & & | & \mathtt{next\ A} & \text{— Next} \\
 & & | & (\mathtt{X}_1, \ldots, \mathtt{X}_n)\hat{\ }\mathtt{A} & \text{— Hiding} \\
 & & | & \mathtt{abort} & \text{— Process abortion} \\
 & & | & [\mathtt{A}_1, \ldots, \mathtt{A}_n] & \text{— Parallel composition} \\
 & & | & \mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) & \text{— Procedure call} \\
 & & & & \\
\text{(Tell-Constraints)} & \mathtt{C} & ::= & \mathtt{c} & \text{— Primitive constraint} \\
 & & | & \mathtt{C}, \mathtt{C} & \text{— Conjunction} \\
 & & & & \\
\text{(Ask-Constraints)} & \mathtt{E} & ::= & \mathtt{c} & \text{— Primitive constraint} \\
 & & | & \mathtt{E}, \mathtt{E} & \text{— Conjunction} \\
 & & | & \mathtt{E}; \mathtt{E} & \text{— Disjunction} \\
 & & & & \\
\text{(Declarations)} & \mathtt{D} & ::= & \mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) :: \mathtt{A} & \text{— Clause} \\
 & & | & \mathtt{D.D} &
\end{array}
\tag{2}
$$

Multiple declarations are allowed per procedure. The first to match is selected.
The following defined agents are also accepted:

$$
\begin{array}{llcl}
\text{(Agents)} & \mathtt{A} & ::= & \mathtt{clock}(\mathtt{A}, \mathtt{B}) \\
 & & | & \mathtt{always}(\mathtt{A}) \\
 & & | & \mathtt{whenever}(\mathtt{C}, \mathtt{A}) \\
 & & | & \mathtt{watching}(\mathtt{C}, \mathtt{A})
\end{array}
\tag{3}
$$

Table 4: Syntax for Timed Gentzen.

6 Specific programming paradigms

We turn to the examination of various concrete programming problems. Because the origins of tcc were in synchronous programming, we first examine how some typical reactive and synchronous computations may be expressed in Timed Gentzen. Next we consider the representation of some common patterns of default inference.

6.1 Reactive programming

A typical reactive programming task is the programming of controllers. For such tasks, it is crucial that the controller program be compilable into a system with bounded run-time state, e.g. an FSA. The programs presented below for two reactive computation tasks satisfy the conditions for bounded compilability discused in Section 3. (In particular they do not use embedded parametric asks which also contain the seeds of arbitrary recursion.)

6.1.1 A simple mouse controller

The task is to program a clock-controller for a mouse [BdS91]. The program may assume that it is controlled by the environment using `start` and `stop` signals. The task of the program is to count the number of mouse-clicks that occur during each such interval, and output `none`, `single` or `many` depending on whether there are zero, one or more clicks.

Our solution is similar to the one in [BdS91]. A process (`mouse_zero(X)`) is set up whose task is to accumulate in `X` the number of clicks that occur. (The process ceases looking at the `click` signal once the count reaches two; instead it maintains forever the count two.) The process is aborted by the `stop` signal, which also triggers off a process which generates the appropriate output signal based on the accumulated count.

The program illustrates two common idioms in Timed Gentzen programming. First, note the tandem of `watching(stop,...)` and `whenever(stop, ...)` agents. They are used to get the effect of a `do ...watching ...trigger ...` construct in which an agent is activated when a `do ...watching ...` agent is aborted. Second note that the `controller` uses a newly created private channel (`X`) for communication with its `mouse_n` agents. This allows multiple controllers (perhaps on different mice) to be simultaneously active, without interference.

```
controller(Click, Start, Stop) ::
   X^[watching(Stop, whenever(Start, mouse_zero(X, Click))),
     whenever(Stop,
              [X : 0 → {zero},
               X : 1 → {one},
               X : 2 → {many}])].

mouse_zero(X, Click) ::
   watching(Click, always {X : 0}),
   whenever(Click, next mouse_one(X, Click)).

mouse_one(X, Click) ::
   watching(Click, always {X : 1}),
   whenever(Click, next mouse_many(X, Click)).

mouse_many(X, _Click) :: always ({X : 2}).
```

A single `mouse(Count, X, Click)` predicate could have been defined, but the recursion in its definition would not satisfy the syntactic conditions for boundedness.

6.1.2 An automated teller machine

We present a program for the bank teller machine (ATM) of [BRS93]. This machine accepts an ATM card and verifies the identification number. On receiving an incorrect number thrice, the machine swallows the card; else it asks for an amount, requests the bank's authorization, and releases the money if authorization is received; keeping the card otherwise (!). The customer may cancel the operation at any time.

Our solution is presented below. The structure of the program should be fairly clear. The basic technique of *persistent tells* is used to communicate information between losely coupled subsystems. Consider for example the interaction between the ATM controller and the bank. The ATM has access to the communication channels `card` and `code`. It must communicate these channels to the bank, thus allowing the bank to read other messages on the channel. (For example, the display would be expected to add (also using persistent tells) tokens of the form `card:N, code:P` to the store, for `N` the bank account number read from the card, and `P` the password read from the display.) Since there can be no guarantee that the bank will process the information sent to it in the current time instant — for example, the bank may be busy in this time instant servicing some other request — the sender must maintain the item in the store indefinitely. This is achieved by the idiom:

```
watching(received, always {message}).
```

Here, `message` is the signal being sent to some other agent, and `received` is the signal that it uses to acknowledge receipt.

The program also illustrates a simple convention. Typically an agent adds to the store a pure "signal" (that is, a 0-ary constant, e.g. `auth`), together with some messages on the signal (e.g. `auth:false`). Another agent may detect the signal by suspending on the constant; subsequently the messages associated with the signal may be examined in detail. This technique is particularly useful in situations (such as with the `card` signal) in which an agent in the environment may be passing a complex (unbounded) data-structure (e.g., card number) *through* the finite state controller to another agent in the environment (e.g., bank). The code for the controller never has to handle the complex data-structure, it merely senses information on a (pure) signal and passes that on. In this way, the mobility of the underlying communication mechanism can be used to maintain boundedness of the controller code.

```
ready ::
   [whenever(card,
     [watching((keep_card ; cancel),
        [verify_code(0),
         whenever(code_ok,
```

```
          [{display : enter_amount},
           whenever(amount, verify_funds)])]),
      whenever((keep_card ; cancel), keep_card ⤳ {emit_card})]),
   whenever((keep_card ; emit_card), next ready)].

verify_code(Num) ::
   [Num < 3
    → [{display : enter_code},
         whenever(code,
           [watching(verif,
                     always {bank : verify, bank : verify(card, code)}),
            whenever(verif,
              [verif : true → {code_ok},
               verif : false
                → N1$(N1 is Num + 1 → next verify_code(N1))])
           ])],
    Num = := 3 → {keep_card}].

verify_funds ::
   [watching(auth,
            always {bank : request_auth, bank : request_auth(card, amount)}),
    whenever(auth,
              [auth : true → {delivermoney, emit_card},
               auth : false → {keep_card}])].
```

6.1.3 Synchronous programming

An obvious application of synchronous programming ideas is in the specification of synchronous circuits, such as those used to describe the operation of systolic algorithms [Kun79]. Computation is performed by a network of cells connected together by "wires", possibly with finite buffering. Data moves rhythmically through the system, pulsed by an external clock.

It should be clear that an arbitrarily-connected network of cells can be described using an (untimed) Gentzen program. The synchronous execution of such a system is then described by merely "extending over time" their operation at a particular time instant. At every instant, the environment may supply input and accept output from interface cells. More powerfully, using Timed Negative Asks, a cell may sense if data was *not* present on a wire in the past cycle, and take subsequent action.

A simple concrete example is provided by the obvious program for computing the Fibonacci sequence. This consists of a single cell containing an adder and a two-place buffer, with the output wired into the input:

```
fib(Val) ::
   [{Val : 0}, next {Val : 1},
    always X$(Val : X
               → (next (Y, Z)$
                       ((Val : Y, Z is X + Y) → (next {Val : Z}))))].
```

A harness for running the program is provided by:

```
main(N) ::
   Index^[watching(Index : N,
                   Val^[fib(Val), counter(Index),
                        output(Val, Index, fib)]),
          whenever(Index : N, abort)].

counter(Init, I) ::
```

```
   [{I : Init}, always (X, X1)$((I : X, X1 is X + 1) → next {I : X1})].

output(In1, In2, Op) ::
   always (X, I)$((In1 : X, In2 : I) → {format('~p(~p) = ~p.~n', [Op, I, X])}).
```

Clearly, other synchronous algorithms can be implemented in a similar way.

6.2 Programming with defaults

We now turn to some techniques that make explicit use of Timed Negative Asks for defeasible default reasoning.

6.2.1 Sieve of Eratosthenes

Perhaps the simplest example is provided by the old algorithm for computing the sequence of primes.

Assume that the timed sequence of natural numbers, starting with `2` is produced on `Val`. The program asserts that each such number `N` is prime by default — unless there is reason to believe that it is not. If the default is not defeated, an agent `sieve` is set up that declares every subsequent number `X` `not_prime` if it is a multiple of `N`.

```
primes(Val, Output) ::
   always N$(Val : N → not_prime ⤳ [{Output : N}, always sieve(N, Val)].

sieve(N, Val) ::
   X$(Val : X → X mod N = :=0 → {not_prime}).
```

The overall program is obtained by coupling with the `counter` from the previous section:

```
primes(Primes) ::
   Val^[primes(Val, Primes), counter(2, Val)].
```

The program may be run in a harness (using `watching`) in the same way as `fib`.

6.2.2 Persistent assignments

The next few examples use a variation of the basic `tcc` technique of persistent tells to maintain defaults in the face of change.

The basic idea can be explained schematically by a simple example. Suppose we would like to implement the requirement that `X` should be constrained to be `2` unless stated otherwise. Under the assumption that agents wanting to alter the value of `X` can declare their intention to do so at least one step in advance, the following technique will work.

The agent that wishes to change the value of `X` must declare its intention by, say, asserting the token `changing(X)`. The persistence of the default value is then maintained by the agent `default(X,M)` defined by:

```
default(X, M) :: watching(changing(X), always {X = M}).
```

Once `changing(X)` is asserted, the asserting agent is guaranteed that the default will not be in effect at the next time instant. It may now establish its own constraints on `X`, including, perhaps, establishing a default for it:

```
change(X, N) :: [{changing(X)}, next default(X, N)].
```

This basic technique may be extended in all sorts of application-dependent ways using the power of `tcc` combinators. For example, using `clock`, defaults may be specified to hold exactly during the clock ticks supplied by some arbitrary basic process.

6.2.3 Safe defaults

The next example is from the domain of Qualitative Reasoning about physical systems. It arises from the goal of constructing compositionally symbolic models of complex, real-time dependent, computational electro-mechanical systems, for the purpose of simulation, control and diagnosis.

The task is to model a portion of the control system for the paper-path of a reprographics machine, such as a photo-copier. In general, the paper-path consists of a sequence of baffles, drives, rollers, gates, and electronic sensors. An electronic module takes input from the sensors and controls the flow of power to various motors. Rollers are driven (through gear-trains) by motors; in normal operation they acquire incoming sheets and move them down-stream. However, rollers may malfunction under a variety of conditions — they may be worn and may slip, or the paper may be greasy and difficult to acquire. Therefore, sensors are installed at various locations to detect whether a sheet dallies unduly at that location. When a sensor signals a jam, the power module immediately cuts power supply to drives, thus halting the paper-path.

We are concerned with building libraries of components models that may be used to assemble a system model given knowledge of system components and structure. In particular, we focus on modeling rollers, sensors and (the control logic for) the power-supply. We would like to use these models to construct a model of a simple system consisting of a pair of rollers in series, with a sensor in the middle:

```
system(R0, S0, R1, M0, M1, M2) ::
   [roller(R0, l0, l1, M0),
    sensor(S0, l1, M1),
    roller(R1, l1, l2, M2),
    power_module].
```

A typical call to `system/5` would be: `system(r35,s41,r36,ok,ok,ok)`.

What should such component models look like? The models for `roller` and `sensor` are straightforward (uniform extensions over time of point-wise, time-independent, monotone models):

```
roller(_R, Before, After, ok) ::
   always (power → at(Before) : paper → next {at(After) : paper}).

roller(_R, Before, _After, slipping) ::
   always (power → at(Before) : paper → next {at(Before) : paper}).

sensor(_S, Loc, ok) ::
   always (power → at(Loc) : paper → next (at(Loc) : paper → jam)).
```

The `power_module` model is however faced with the problem of lack of omniscience about its environment. A `jam` may be signaled in any of various complicated ways, by any of a collection of subsystems whose identity is not known at model-construction time. The simplest way to model the subsystem, then, is as a mechanism that initiates and maintains the supply of power, subject to *interrupts* by its environment:

```
power_module :: watching(jam, always {power}).
```

The example illustrates the basic idea behind what may be termed *safe defaults*: through its very organization, the program guarantees that any conclusions that are contingent on a default inference cannot in themselves bring about (causally determine) a state of affairs that undermines the applicability of the inference rule. No global reasoning needs to be done, no "default extensions" need to be created. On detection of quiescence, all applicable default rules are fired immediately and irrevocably.

It should be clear, however, that the use of `tcc` languages for system modeling, simulation and diagnosis is a topic in its own right. Of several remarks possible here, we have space for only three:

- The above model may be used not just for simulation but also diagnosis. Here the problem is: we are given observations of the system (e.g. `jam`) and have to determine the mode of operation of devices consistent with these observations. This is a natural motivation for examining the complexity of the *abduction* problem for tcc programs.

- The above model is stated in the fragment of tcc that can be compiled into FSAs. This suggests that the compositional tcc compilation algorithm can already be used to obtain an "envisionment" of the physical system (that is, a finite, succinct description of all its future behaviors). Standard techniques for model-checking finite state systems may then be used to analyze both component as well as system models constructed in this way.

- It should be clear how some standard problems about reasoning with action and change, such as the Yale shooting problem [HM86] have simple solutions in tcc. In essence, tcc makes it possible to say that the effects of certain actions should persist indefinitely *unless* interrupted by the environment. By computing the least solution above the input supplied by the environment, tcc ensures that there is no *deus ex machina* (no phantom actions such as mysterious "unloadings" of guns) — any deviation from defaults arises solely because of input from the environment.

6.3 Explicit Minimization

A common mode for organizing computation is this. In one phase, all agents post whatever constraints they want on the variables of interest. On quiescence it is desired to determine *minimal* values for the particular variable — perhaps *the* minimal one (if it is unique), perhaps *some* arbitrary one (if there are many minimal ones; this is sometimes called a "credulous" choice), perhaps the logical disjunction of all the minimal ones (this is sometimes called "skeptical" choice), while keeping the other variables in the store constant or varying. Various kinds of minimization procedures proposed for logic programming and for default reasoning (e.g., varieties of circumscription) fall into this general paradigm.

The `tcc` model makes possible the specification of "safe" versions of such primitive processes: it makes sense to perform the minimization only on quiescence of the current phase (the minimization operation is not monotone in its argument, hence cannot be run while its argument is being accumulated); hence the new constraint may be added to the store only at the *next* instant; hence structurally in a way that cannot interact with the actual process of minimization itself.

This leads to the addition of a class of minimization agents to the language. Because of the variability mentioned above, many such agents are definable and interesting. Assume $min(X, c)$ is such a minimization operation — it returns a constraint minimal in X given the information c per some chosen definition of minimization. Then we may define the basic process by:

$$[\![\texttt{minimize(X)}]\!] \quad = \quad \{\epsilon\} \cup \{c \in \mathbf{Obs}\} \cup \{c \cdot d \cdot s \in \mathbf{Obs} \mid d \geq min(X, c)\}$$

Of course once the basic agent is introduced into the language, it may be combined with all the other combinators in the language in completely general ways.

Two concrete examples should illustrate the matter further. Consider the *histogram* problem studied by researchers in working on determinate dataflow languages. The problem is to efficiently determine the histogram associated with a given input function f. That is, determine, for each point x in the range of f, the number of points in the domain mapped to x.

The following simple algorithm is expressible in Timed Gentzen augmented with a suitable minimization operation: examine the tuples `(Index, Value)` in f in parallel, asserting that `Index` belongs to a set associated with `Value`.

On quiescence, minimize the extension of these sets, and take their cardinality. Concretely:

```
histogram(In, Hist) :: (Temp)^
   [(Index, Value)$((Index, Value) in In → {Index in Temp.Value}),
    minimize(Temp),
    next sizeof(Temp, Hist)].
```

`minimize(X)` is an example of an "indexical" constraint [HSD92] — a functions from constraints to constraints that is not necessarily monotone. Obviously, several other constraint-system specific indexicals are possible in this vein.

The second concrete example is taken from work on model composition in Qualitative Reasoning and is representative of the kind of default inferencing employed therein. While building component models one may only be able to specify very incomplete information about the relationship between some "global" variable X and some variable V corresponding to the component. For example, one may merely know that X varies monotonically with V (assuming that both vary over some ordered set such as the reals). This can, of course, be achieved by asserting the constraint that the partial derivative of X with respect to V is positive. However, once system structure is completely determined, and the constraints between X and the ensemble of component variables $V_1, \ldots, V_n$ established, another piece of information can be deduced, namely, that X is a function of *just* the inputs $V_1, \ldots, V_n$. Structurally, this is a safe default inference because no inference licensed by this deduction should affect the applicability of the default: once the structure of the system has been determined, it has been determined. That is, the default depends only on predications that are going to remain stable for the rest of the deduction, that are causally independent of the consequences of the conclusion of the default.

Such a safe default rule can clearly be directly expressed in Timed Gentzen, augmented with minimization predicates. Indeed the basic notion of *staged computation* underlying tcc is in completely harmony with the ideas of staged model composition underlying model composition work in Qualitative Reasoning.

6.4 Meta-programming: A Timed Gentzen meta-interpreter

The design of Timed Gentzen makes the description of a meta-interpreter easy:

```
tcc({C}) :: {C}.
tcc(C → A) :: C → tcc(A).
tcc(C ⇝ A) :: C ⇝ tcc(A).
tcc(next A) :: next tcc(A).
tcc(Vars$A) :: Vars$tcc(A).
tcc(Vars^A) :: Vars^tcc(A).
tcc([]) :: [].
tcc([A | B]) :: [tcc(A) | tcc(B)].
tcc(Proc) :: Body$((Proc :: Body) → tcc(Body).
```

Only the last clause is interesting. Note that because of the way sequences of clauses are interpreted in Timed Gentzen, the last clause will be activated only on a agent that does not match any of the other clauses. This can happen only for calls to user-defined procedures. In such a case, the body of the procedure is looked up, using a "clause" meta-call, and passed on to a recursive invocation of the meta-interpreter.

7 Implementation

We now discuss a simple Prolog interpreter for Timed Gentzen. The implementation is intended to enable users to quickly develop, run, test and debug Timed Gentzen programs, thus getting a feel for the tcc computation model. The emphasis in this implementation is on simplicity rather than efficiency — several well-known techniques that have been developed in the literature on implementations of logic programming languages and problem-solving systems can be used to substantially improve performance.[9]

Separately, we are implementing the FSA compilation algorithm outlined in [SJG93] for bounded tcc programs. Clearly that compiler, once it exists, will be the implementation of choice to use in fielding embedded, reactive Timed Gentzen programs.

7.1 The basic interpreter

First we begin with basic operation declarations for symbols that will stand for tcc combinators.

```
← op(1100, xfy, '::').
← op(975, fy, [next, always]).
← op(950, xfy, [ → , ⇝ ]).
← op(150, xfy, ['^', '$']).
← use_module(library(ordsets)).
← use_module(library(lists)).
```

The interpreter keeps track of the following pieces of information:

1. The current agents not yet examined (Pool).

2. The Positive Asks that have already been discharged (Susp).

[9] For instance, one standard technique is to use partial evaluation to generate from this interpreter a compiler that inputs a source tcc program and outputs Prolog code. This techniques was used to generate very similar compiler-based implementations of Concurrent Prolog, GHC and Herbrand.

3. The Tells that have already been executed (Store).
4. The Nexts that have already been identified (Nexts).
5. The Elses that have already been identified (Elses).
6. The current cycle number (Count).
7. The counter for existential quantifiers (V).

At each step, the next agent from the pool is examined. Obvious things are done in simple cases: a `[]` agent is discarded, an **abort** agent is Prolog-failed (causing Prolog-failed termination of the entire computation), a parallel composition is repackaged, a `next` agent is pushed onto the Next list, an $\leadsto$ agent is pushed onto the Else list, a recursive procedure call is unfolded by examining the program database.

The interesting cases are Hiding, (parametric) Asks and Tells. Hiding is implemented by using the counters (and the functor `'_'/2`) to generate a unique identifier; this is then bound to the Prolog variable representing the quantified Timed Gentzen variable. In the case of a Tell, the argument of the tell is added to the store. At the same time, all possible instances of the Positive Asks (that have already been discharged) that match this tell are added to the pool. Symmetrically, when a new Positive Ask is being executed, all possible instantiations that reduce against tokens in the store are added to Pool. (This is discussed in more detail below.)

Computation in the cycle terminates when Pool is empty. All the accumulated Elses are now examined to see if they should be rejected or retained for the next cycle. The current Suspended agents and store are thrown away. The next cycle commences.

```
tcc([]) ←!.
tcc([A | Pool]) ←!, tcc(A, Pool, [[], a([], []), [], [], 0, 0]).
tcc(A) ← tcc(A, [], [[], a([], []), [], [], 0, 0]).

tcc([], [Store, _SuspA, Elses, Next, C, _V]) ←!,
   next(Elses, Store, Next, New),
   (New = = []
```

```
    → format('~n Termination at T = ~p.', [C])
    ; (C1 is C + 1,
   tcc(New, [[], a([], []), [], [], C1, 0]))).

tcc([A | Pool], Rest) ← tcc(A, Pool, Rest).

tcc({(A, B)}, Pool, State) ←!, tcc({A}, [{B} | Pool], State).
tcc({C}, Pool, [Store, SuspA | Rest]) ←!,
   install_tell(C, Pool, Store, SuspA, NewPool, NewStore),
   tcc(NewPool, [NewStore, SuspA | Rest]).

tcc((E1, E2) → A, Pool, State) ←!, tcc(E1 → E2 → A, Pool, State).
tcc((E1 ; E2) → A, Pool, State) ←!, tcc(E1 → A, [E2 → A | Pool], State).
tcc(C → A, Pool, [Store, SuspA | Rest]) ←!,
   install_ask(C → A, Pool, Store, SuspA, NewPool, NewSuspA),
   tcc(NewPool, [Store, NewSuspA | Rest]).

tcc(E ⇝ A, Pool, [Store, SuspA, Elses | Rest]) ←!,
   tcc(Pool, [Store, SuspA, [E ⇝ A | Elses] | Rest]).

tcc(next A, Pool, [Store, SuspA, Elses, Nexts | Rest]) ←!,
   tcc(Pool, [Store, SuspA, Elses, [A | Nexts] | Rest]).

tcc(_X$A, Pool, State) ←!, tcc(A, Pool, State).

tcc(Vars^A, Pool, [Store, SuspA, Elses, Nexts, C, V]) ←!,
   inst_ex_vars(Vars, C, V, NV),
   tcc(A, Pool, [Store, SuspA, Elses, Nexts, C, NV]).

tcc(abort, _Pool, _State) ←!, fail.
tcc([], Pool, State) ←!, tcc(Pool, State).
tcc([A | B], Pool, State) ←!, tcc(A, [B | Pool], State).

tcc(Goal, Pool, State) ←!, (Goal :: Body), !, tcc(Body, Pool, State).
```

7.2 The Gentzen constraint system.

Consider now the implementation of Ask and Tell operations.

Ask operations A check is made for whether the Ask operation is basic (e.g., an arithmetic routine). If it is, the operation is treated as a Prolog call and fails or succeeds as appropriate.

There are two remaining cases. If the Ask is not parametric, then the store is checked for the match; if there is a match, the Ask is reduced. As an optimization, note that there is no need to record the Ask in Susp since multiple firings will result only in the very same body being fired multiple times. If the Ask is parametric, it is recorded in Susp (multiple firings may potentially be different because of the values bound to variables). A check is made for instantiations in the current store.

The heart of the constraint system operation is this check. The Prolog `bagof/3` predicate is used to generate all possible matches of the Ask, and, for each match, to generate the corresponding instantiated body of the Ask. Of importance here is the ability to specify that the query in `bagof` must be existentially instantiated for all the variables that occur in the Ask. (It is for this purpose that Susp actually contains the list T of suspended Asks, together with the list of free variables in T. The call to `vars/6` is used to update this list.) This allows the correct treatment of Timed Gentzen universal variables in Ask-queries (and any Timed Gentzen existential variables in the body of the query). Indeed, the Prolog variables representing Timed Gentzen universal variables will never be instantiated in the course of the computation; `bagof` will use those variables to create copies exactly as needed. (Note that atoms added to the store will always be Prolog-ground, since existential variables are represented as Prolog-ground terms.)

The resulting set of agents is then added to Pool.

```
install_ask(C → A, Pool, Store, a(Var, SuspA), NewPool, NewSusp) ←
   ask_basic(C)
    → (NewSusp = a(Var, SuspA),
        (call(C)
          → NewPool = [A | Pool]
          ; NewPool = Pool))
```

```
    ; (vars(A, C, Var, VC, Vs, NewVar),
         (VC = = []
           → (member(C, Store)
                → (NewPool = [A | Pool], NewSusp = a(Var, SuspA))
                ; (NewPool = Pool, NewSusp = a(NewVar, [C → A | SuspA])))
           ; (NewSusp = a(NewVar, [C → A | SuspA]),
              (bagof(A, Vs^member(C, Store), Agents)
                → NewPool = [Agents | Pool]
                ; NewPool = Pool)))).

ask_basic((_X :: _Y)).
ask_basic(_X is _Y).
ask_basic(_X = : = _Y).
ask_basic(_X<_Y).
ask_basic(_X = <_Y).
ask_basic(_X = _Y).
ask_basic(_X> = _Y).
ask_basic(_X>_Y).
ask_basic(read(_X)).

vars(A, C, Var, VC, Vs, NewVar) ←
   vars(A, VA),
   vars(C, VC), ord_union(VA, VC, Vars1),
   ord_subtract(VC, VA, Vs),
   ord_union(Vars1, Var, NewVar).

vars(Term, Vars) ← vars(Term, [], Vars).
vars(X, V, Vars) ← var(X), !, ord_add_element(V, X, Vars).
vars(Term, V, Vars) ←
   functor(Term, _F, A),
   vars_each(0, A, Term, V, Vars).
vars_each(M, N, Term, V, Vars) ←
   M = N
    → V = Vars
    ; (M1 is M + 1,
   arg(M1, Term, Term_M1),
```

```
   vars(Term_M1, Var1),
   ord_union(Var1, V, Var2),
   vars_each(M1, N, Term, Var2, Vars)).
```

Tells. Tells are implemented dually to Asks. A basic Tell is simply executed; note that basic Tells allowed are such that they cannot fail. For non-basic Tells, a check is made to see if the atom being told is already in the store, in which case it can be discarded. Otherwise `bagof/3` is used to generate the list of instantiated Ask agents (from Susp) that can now fire; this list is added to Pool.

```
install_tell(C, Pool, Store, a(Vars, SuspA), NewPool, NewStore) ←
   tell_basic(C)
    → (call(C), NewStore = Store, NewPool = Pool)
    ; (member(C, Store)
       → (NewStore = Store, NewPool = Pool)
       ; (NewStore = [C | Store],
          (bagof(A, Vars^member(C → A, SuspA), Agents)
            → NewPool = [Agents | Pool]
            ; NewPool = Pool))).

tell_basic(format(_X, _Y)).
tell_basic(ttyflush).
```

Negative Asks. When moving from one time instant to the next, the list of Negative Ask agents is examined. Those agents whose condition is *not* entailed are carried over; others are discarded.

```
next([], _Store, Next, Next).
next([C ⇝ A | E], Store, Next, Out) ←!,
   (succeeds(C, Store)
```

```
always(A) :: [A, (next always(A))].

if(Cond, Then, Else) :: [Cond → Then, Cond ⇝ Else].

whenever(C, A) :: [C → A, C ⇝ whenever(C, A)].

watching(_C, D) :: D.
watching(C, D → A) :: D → watching(C, A).
watching(C, D ⇝ A) :: (C ; D) ⇝ watching(C, A).
watching(C, (next A)) :: C ⇝ watching(C, A).
watching(C, Vars^A) :: Vars^watching(C, A).
watching(C, _Vars$A) :: watching(C, A).
watching(_C, abort) :: abort.
watching(_C, []) :: [].
watching(C, [A | R]) :: [watching(C, A) | watching(C, R)].
watching(C, Proc) :: Body$((Proc :: Body) → watching(C, Body)).
time_on(C, A) :: clock(always(C), A).
do_watching(C, A) :: clock(whenever(C, (next abort)), A).
susp_act(C, E, A) :: clock(whenever(C, (next E)), A).
```

Table 5: Timed Gentzen definition of assorted combinators.

```
    → next(E, Store, Next, Out)
    ; (Out = [A | Out1], next(E, Store, Next, Out1))).

succeeds((C1, C2), Store) ←!, succeeds(C1, Store), succeeds(C2, Store).
succeeds((C1 ; C2), Store) ←!, (succeeds(C1, Store) ; succeeds(C2, Store)).
succeeds(C, Store) ← ask_basic(C) → call(C) ; member(C, Store).
```

```
clock(C, A) :: whenever(C, A).
clock(abort, _A) :: [].
clock(_A, abort) :: abort.
clock([], A) :: A.
clock(_A, []) :: skip.
clock([B | R], A) :: clock(B, clock(R, A)).
clock(B, [A | R]) :: [clock(B, A) | clock(B, R)].
clock(next _B, D) :: D.
clock(next B, D → A) :: D → clock(next B, A).
clock(next B, D ⇝ A) :: D ⇝ clock(B, A).
clock(next B, next A) :: next clock(B, A).
clock(next B, Vars^A) :: Vars^clock(next B, A).
clock(next B, _Vars$A) :: clock(next B, A).
clock(next B, Proc) :: Body$(Proc :: Body → clock(next B, Body)).
clock((_C → (next _B)), D) :: D.
clock((C → (next B)), (D → A)) :: D → clock((C → (next B)), A).
clock((C → (next B)), (D ⇝ A)) :: [(C → clock((next B), (D ⇝ A))), ((C, D) ⇝ A)].
clock((C → (next B)), next A) :: [(C → (next clock(B, A))), (C ⇝ A)].
clock((C → (next B)), Vars^A) :: Vars^clock((C → (next B)), A).
clock((C → (next B)), _Vars$A) :: clock((C → (next B)), A).
clock((C → (next B)), Proc) :: Body$(Proc :: Body → clock((C → (next B)), Body)).
clock((_C ⇝ _B), D) :: D.
clock((C ⇝ B), (D → A)) :: D → clock((C ⇝ B), A).
clock((C ⇝ B), (D ⇝ A)) :: [(C → (D ⇝ A)), ((C, D) ⇝ clock(B, A))].
clock((C ⇝ B), next A) :: [(C → (next A)), (C ⇝ clock(B, A))].
clock((C ⇝ B), Vars^A) :: Vars^clock((C ⇝ B), A).
clock((C ⇝ B), _Vars$A) :: clock((C ⇝ B), A).
clock((C ⇝ B), Proc) :: Body$(Proc :: Body → clock((C ⇝ B), Body)).
clock(Proc, A) :: Body$(Proc :: Body → clock(Body, A)).
```

Table 6: Timed Gentzen definition of clock/2.

That's it! This completes the description of the interpreter. The general combinators defined in previous sections (e.g. `whenever`, `watching`) can be defined as user programs (Tables 5, 6).

8 Future work

The ideas above open up a new line of attack on the problem of systematic, model-based design of embedded, reactive, real-time systems, e.g., those used to control complex electro-mechanical systems such as airplanes, photocopiers etc. Considerable work lies ahead on several fronts.

8.1 Developing methods for generating real-time schedulers from tcc models

An important use of the tcc is in specifying the behavior of simple, computationally-controlled physical mechanisms, such as the paper-path and video-path in a digital photo-copier. This leads to the problem of synthesizing optimal controllers for these physical mechanisms. For example, we can (and, in a slightly different way, have) specify the components of the paper-path in a photocopier and the Xerographic subsystem in tcc. The problem of synthesizing a scheduler is now the problem of synthesizing the time-sequenced control signals that must be input into the system such that the desired output (the properly constituted output job, e.g. 4 double-sided copies, stapled) is produced as early as possible. This can be solved by extracting from the tcc program a constraint problem and solving it using optimization techniques. For example, in the case of some of the scheduling problems we have looked at, the optimization problems generated are a subcase of the integer linear programming problem, and can be solved by dynamic programming. In other cases, we expect considerably harder optimization problems to arise.

Another interesting set of problems arise in re-planning. The schedule, once generated, are pressed into action. Various exceptions can arise at run-time. Sheets may get delayed because a roll is slipping, a video-image may be so complicated as to cause the decompression software to be late in delivering video bits to the output laser, etc. Thus execution has to be monitored, and then, in case of exceptions, subsidiary scheduling problems have to be generated. These are modified versions of the initial problem (with some of the initial commitments in place, and some others contradicted). This naturally motivates examining algorithms for incrementally (re-)solving these optimization problems.

8.2 Program Verification

The advantages of reasoning with executable specifications is well known; *c.f.* Berry's WYPIWYE principle or executable intermediate representations for compilers. `tcc` can be viewed as an executable temporal logic, thus permitting programs and properties to be expressed in the same language.

For example, let `p` be the property "any occurrence of event `a` must cause `b` to be true until the next occurrence of event `c`". The property can be directly expressed as the program:

`p` :: **whenever** `a` **do** (**do** **always** `b` **watching** `c`).

An agent `A` satisfies p iff:

$$\mathtt{A} \vdash \mathtt{p}$$

Hence, one may use the logic for reasoning about inequivalence of `tcc` programs for establishing that programs posses such properties.

We plan to investigate the use of automated tools(theorem provers and model checkers) to solve this verification problem. In particular, since parallel composition is modeled by the logical operation of conjunction, there is hope of performing compositional model checking, thus mitigating the state explosion problem.

For an alternate approach, recall that deterministic safety properties correspond precisely to safety automata [MP90] that have a single designated failure state. Combined with the fact that `tcc` allows programs and properties to be expressed in the same language, this leads to the promise of reducing general safety properties of `tcc` programs to checking the reachability of a designated failure state.

From the programming point of view, this suggests a further integration of the logic and programs. At present, `tcc` allows access only to the *current* value of signal variables. We will investigate the addition of *global* constraints on signal variables, say in the spirit of LUSTRE and SIGNAL. For example, let X, Y, Z be channels that receive one input at each time instant. Then, $\uparrow Z = \uparrow X + \uparrow Y$ would indicate that at all time instants, the value of Z is the sum of the values of X and Y. The utility of this construct for programming and verification needs to be investigated.

8.3 Towards a theory of preemption

In the traditional theory of sequential programming, preemption and abortion constructs have not been investigated separately. The primary reason for this apparent oversight is that control constructs of this form are "compilable" back to traditional combinators, via the continuation passing transform. For example, exceptions in ML are handled this way.

Unfortunately, this oversight has continued in much of the traditional work on process theory, where there is no obvious compilation of preemption constructs back into basic combinators. Indeed, synchronous languages arose in part as an attempt to remedy this flaw. However, the problem is now the plethora of preemption constructs. For example, what is the relationship, if any, between the constructs supported by ESTERELand those supported by SIGNAL? The unification of the pre-emption combinators in LUSTRE, SIGNAL and ESTEREL achieved by tcc suggests that tcc encapsulates the rudiments of a theory of pre-emption constructs. A fully developed theory would identify the properties of an ambient category of processes that would allow the "orthogonal" addition of pre-emption constructs. A first step would be to explore the connections of tcc with Interaction categories [Abr93], a general description of synchronous processes.

8.4 Integration of asynchronous and synchronous computation

This paper has focused on synchronous systems; the reason for focusing initially on synchronous systems is that they can be construed as the building blocks for asynchronous systems. Following the route similar to [BRS93], an asynchronous model of computation can be constructed by message passing between synchronous nodes; message delivery is guaranteed but there are no bounds on message delivery time. In our setting, this translates to a bunch of synchronous nodes running tcc programs, communicating with each other via constraints, where the time taken for stores to incorporate constraints can be unbounded. Thus, indeterminacy arises naturally as the result of abstracting away from timing considerations. From a logical point of view, this idea is captured by the idea of *eventual* tells, modeled by the "Eventual" modality in (linear-time) temporal logic. As in tcc, this program will involve development of foundational and pragmatic aspects.

- From a foundational point of view, this language to satisfy the same strong properties as tcc; namely, the logical and model theoretic readings will enable declarative programming styles and direct specification of complex process descriptions, and their algebraic manipulation.

- From a programming point of view, the language should recover the expressiveness associated with traditional process calculi, for example Hoare's CSP, Occam and Milner's CCS. In fact, since the tcc paradigm already incorporates the ability to "pass channel names" more powerful calculi, say Milner's Pi-calculus, should be mimicked. The significant difference from programming in traditional process calculi will be twofold: firstly, the ability to program declaratively that is made possible by the logical foundations of the language; and secondly, the integration of real time into programming and reasoning.

References

[Abr93] S. Abramsky. Interaction categories. Available by anonymous ftp from papers/Abramsky:theory.doc.ic.ac.uk, 1993.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, techniques and tools.* Addison Wesley series in Computer Science. Addison Wesley, 1985.

[Bau89] M. Baudinet. Temporal logic programming is complete and expressive. In *Proc. of ACM conference on Principles of Programming languages*, 1989.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, 1990.

[BdS91] Frederic Boussinot and Robert de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293 – 1304, 1991.

[Ber89] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11 – 17. Elsevier Science Publishers B.V. (North Holland), 1989.

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

[BFG+90] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. Metatem: A framework for programming in temporal logic. In J. W. de Bakker, W. P. de Roevere, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems– Models, Formalisms, Correctness.* Springer- Verlag, 1990. LNCS 430.

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

[BRS93] G. Berry, S. Ramesh, and R.K. Shyamsunder. Communicating reactive processes. In *Proceedings of Twentieth ACM Symposium on Principles of Programming Languages*, pages 85 – 98, 1993.

[Brz91] Christoph Brzoska. Temporal logic programming and its relation to constraint logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 661 – 677, 1991.

[Fd93] Ken Forbus and Johan deKleer, editors. *Building Problem Solvers.* MIT Press, 1993.

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

[Gin87] Matthew Ginsberg, editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, 1987.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

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

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

[HM86] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *Proceedings of Fifth National Conference on Artificial Intelligence*, pages 328–333, 1986.

[Hoa85a] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, London, 1985.

[Hoa85b] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.

[HP85] David Harel and Amir Pnueli. *Logics and Models of Concurrent Systems*, volume 13, chapter On the development of reactive systems, pages 471–498. NATO Advanced Study Institute, 1985.

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

[Kun79] H.T. Kung. Let's design algorithms for vlsi systems. Technical Report 151, CMU-CS, Jan 1979.

[Mer93] R. Merz. Efficiently executing temporal logic programs. In M. Fisher and R. Owens, editors, *Proc. of IJCAI*, 1993.

[Mil80] R. Milner. *A calculus of communicating systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge Univ. Press, 1986.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the ACM SIGPLAN conference on Principles of Programming langauges*, 1990.

[MPW89] R. Milner, J. G. Parrow, and D. J. Walker. A calculus for mobile processes, part i and ii. LFCS Report ECS-LFCS-89-85, University of Edinburgh, 1989.

[MS90] G. Murakami and R. Sethi. Terminal call processing in esterel. Technical Report 150, AT& TBell labs, 1990.

[Rea91] Special issue on real-time systems, 1991.

[RR86] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of ICALP '86*, volume 226 of *Springer LNCS*, 1986.

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

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* Logic Programming and Doctoral Dissertation Award Series. MIT Press, March 1993. Yes, it is actually out!

[SJG93] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. Technical report, Xerox Palo Alto Research Center, December 1993.

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

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