Foundations of Timed Concurrent Constraint Programming

Vijay A. Saraswat — Sys. and Practices Lab., Xerox PARC, Palo Alto, Ca 94304

Radha Jagadeesan — Dept. of Math. Sci., Loyola University, Chicago, Il 60626

Vineet Gupta — Dept. of Comp. Sci., Stanford University, Stanford, Ca 94305

April 1994*

Abstract

We develop a model for timed, reactive computation by extending the asynchronous, untimed concurrent constraint programming model in a simple and uniform way. In the spirit of process algebras, we develop some combinators expressible in this model, and reconcile their operational, logical and denotational character. We show how programs may be compiled into finite-state machines with loop-free computations at each state, thus guaranteeing bounded response time.

1 Introduction and Motivation

Reactive systems [12,3,9] 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.

This paper is concerned with elaborating a simple model for determinate, timed, reactive systems, and providing a language to describe processes in this model. The intended application of such languages forces them to satisfy the following criteria:

*To Appear in Proceedings of the 1994 IEEE Symposium on Logic in Computer Science, July 1994, IEEE Press.

Declarative view: There must be a logical view of the language. For programming, the advantage of simple equational presentations is well known [15]. We recall the advantages of reasoning with executable specifications; *cf.* Berry's "What you prove is what you execute" principle [3], or executable intermediate representations in compilers [22].

Modularity: The language should support hierarchical and modular construction of programs/specifications. This is tantamount to demanding an *algebra* of programs/specifications that includes concurrency and pre-emption – the ability to stop a process in its tracks.

Determinacy: Determinate programs and specifications are easier to construct and analyze. So, the language should not impose indeterminacy.

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

1.1 Synchronous languages

Arguably, the most natural way of programming such systems is in terms of automata with simple loop-free transitions, to ensure bounded response. However, automata do not have hierarchical or parallel structure; in particular, small and succinct changes in the specification can lead to global changes in the automaton [20].

Synchronous languages such as [6], [10], [8], [11], [7] are based on the hypothesis of Perfect Synchrony: *Program combinators are determinate primitives that respond instantaneously to input signals. At any instant the presence* and *the absence of signals can be detected.*

In synchronous languages, physical time has the same status as any other external event, *i.e.* time is multiform. So, combination of programs with different notions of time is allowed. Programs that operate only on "signals" can be compiled into finite state automata with simple transitions. Thus, the single step execution time of the program is bounded and makes the synchrony assumption realizable in practice.

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. Approximate static analysis is used to eliminate such programs (exact analysis is not possible for standard computability reasons). This results, in some cases, in rejecting intuitively correct programs [9]. 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 [9, Page 93]. This is in direct contrast to the standard compilation procedures for traditional programming languages [2].

1.2 Our contributions

A re-analysis of the elegant ideas underlying synchronous programming, starting from the viewpoint of asynchronous computation leads us to *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.

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 pre-emption/abortion constructs supported by ESTEREL. Thus, tcc encapsulates the rudiments of a theory of pre-emption constructs. In addition, tcc inherits the ability to specify cyclic, dynamically-changing networks of processes from concurrent constraint programming (cf. "mobility" [19]).

Executability: tcc programs have bounded response times.

"Paradox-free": tcc resolves the tension between synchrony and causality. This is *concretely* reflected in the modular compilation algorithm that we describe.

In essence, this paper provides a reasonably general construction for lifting an untimed, asynchronous computation model to a "timed asynchronous" computation model, while providing finite-state compilability and bounded response time, and preserving determinacy of computation and the ability to specify parallel composition. For the reader interested in the general construction — and not necessarily sympathetic to the declarative programming point of view — we want to point out that the details of the concurrent constraint programming (CCP) paradigm are not crucial. From such a point of view, CCP may be seen as a simple development of a very general form of "blocking read" and asynchronous "write" data-flow that must almost inevitably underly any theory of determinate, asynchronous computation. The specific commitments made in CCP (namely, to an underlying notion of constraint system) are relatively bland and rarely come in the way — while providing to the initiated a powerful way to fix intuitions; the unsympathetic reader should feel free to substitute her favorite untimed asynchronous computation model instead.

1.3 Organization of the paper

The rest of this paper is organized as follows. We first describe the basic intuitions underlying tcc. Next, we present formally the underlying mathematical model of timed constraint processes; this encompasses operational, denotational and logical semantics, and their correspondence properties. We then show how to derive various forms of pre-emption constructs in the language. We describe a compilation of tcc programs to automata, and describe the restrictions on constraint systems to guarantee bounded response times.

2 Basic Intuitions

2.1 Concurrent Constraint Programming revisited

Our starting point is the paradigm of concurrent constraint programming [24,25]. 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. 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.

Formally, constraint systems $\mathcal{C}$ are essentially arbitrary systems of partial information [23,24]. They are taken to consist of a set D of tokens with minimal first-order structure (variables, substitutions, existential quantification), 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. Examples: (1) Herbrand underlying logic programming (variables range over finite trees, tokens specify equality of terms), (2) FD [13] for finite domains (variables range over finite domains, with tokens for equality and for specifying that a variable lies in a given range), etc.

In the rest of this paper 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 *constraint* is an element of $|D|$ generated from a 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.

2.2 Operational intuitions of tcc: Timed Asynchrony

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 interval 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 interval, 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 up to 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.

In contrast to synchronous programming, computation at each instant is regarded as having a notion of *internal temporality*: it happens *not instantaneously* but over a *very short* — bounded — period of time. This allows the treatment of causality within a time-instant which 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.

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.

2.3 Denotational intuitions

Earlier work [25] 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 glb of any non-empty subset of S above x is also in S. Such a model is rich enough to distinguish an "abort" process — that quiesces on no input (and has hence the empty set of quiescent points) — from the process that causes the store to become inconsistent. Technically, the set of quiescent points determinate a partial closure (extensive and idempotent) operator.[1]

The model is extended to the timed case by following the intuition that "Processes are relations extended over time" [1]. Thus, processes are taken to be subsets of finite sequences of finite constraints that are determinate at every time instant.

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

Definition 2.1 **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.

We let c, d range over finite sets of tokens, and a, b over constraints. 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 $\mathbf{after}$ 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 2.2 $P \subseteq \mathbf{Obs}$ is a *process* iff it satisfies the following conditions:

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

□

We will let **Proc** designate the set of all processes, and let P, Q range over **Proc**. From elementary considerations it follows that **Proc** is a complete lattice with least upper bounds (henceforth, lubs) given by intersection and greatest lower bounds (henceforth, glbs) given by the glb-closure of the union. The least element is **True** $=$ **Obs** and the greatest element is **False** $= \{\epsilon\}$.

3 Process algebra

We now identify basic processes and process combinators in the model. The combinators fall into two categories: (1) CCP constructs: Tell, Parallel composition and Timed Positive Ask are inherited from CCP. These do not, by themselves, cause "extension over time". (2) Timing constructs: Timed Negative Ask, Unit Delay, and Abortion that cause extension over time.

We present the denotational, operational and logical semantics simultaneously. The denotational and logical semantics are summarized in Appendix A and B. Unless otherwise noted, all the combinators introduced are well-defined (produce processes if

[1] [25] presents several variant models for determinate CCP; the construction we are describing here generates correspondingly variant timed models.

their arguments are processes) and monotone and continuous in their process arguments, allowing recursion to be handled using least fixed-points.

The operational semantics is defined via two binary transition relations $\longrightarrow, \leadsto$ over configurations. $\longrightarrow$ represents transitions *within* a time instant. $\leadsto$ 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.

Judgements in the proof system have the form:

$$A_1, \ldots, A_n \vdash A$$

where A, A_i are agents. Intuitively, such a judgement is valid iff the intersection of the denotations of the A_i is contained in the denotation of A; equivalently, if any observation that can be made of the parallel system of agents $A_1, \ldots, A_n$ can also be made of A.

3.1 Timing constructs

We first present the denotational and logical views of the constructs and then the operational semantics, in particular the $\leadsto$ relation.

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

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

Operationally, **skip** has no effect:

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

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.

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

Operationally, **abort** annihilates the environment:

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

Unit Delay. Unit Delay is a variant of the unit delay primitives in synchronous languages [18,4]; intuitively, $A = \textbf{next}\ B$ is the process that behaves like B in the next time instant. When is $o = a \cdot s$, an arbitrary (non-empty) element of $\mathbf{Obs}$, in $[\![A]\!]$?

There are no restrictions on o. Since A behaves like B from the next time instant, s must be an observation of B.

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

Logically, unit delay should be read as the temporal "next time" modality, following the identification in [1], and has the associated proof-rules.

Timed Negative Asks Timed Negative Ask is the only way to detect "negative information" in tcc. Intuitively, $A = \textbf{now } c \textbf{ else } B$ 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.[2]

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

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

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

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

Logically, **now** c **else** A behaves as the formula $c \vee \textbf{next } A$, with the proof rules those induced by this reading. In the following for an agent **now** c **else** A we say that c is the "antecedent" and A the "body".

Operational semantics: the $\rightsquigarrow$ 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, and all Negative Asks whose antecedents are entailed have been eliminated. 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, Unit Delay and Abort agents, we can write the transition rule as:

$$\frac{\Delta, \{\textbf{now } c_i \textbf{ else } A_i \mid i < n\} \not\longrightarrow}{\begin{array}{l}\Delta, \{\textbf{now } c_i \textbf{ else } A_i \mid i < n\}, \{\textbf{next } B_j \mid j < m\}\\ \quad \rightsquigarrow \{A_i \mid i < n\}, \{B_j \mid j < m\}\end{array}}$$

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

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

Guarded Recursion. A *program* is a pair of declarations D and an agent A; a declaration is of the form $g :: A$ (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 a function of 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". (In actuality, we make a slightly more liberal assumption that is detailed in Appendix D.) From a logic programming perspective, this restriction may seem severe. However, this is not the case since 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 using least fixed-points. The proof rules for recursion are the same as for the least fixed points in [16]. In the following, we shall sometimes use the syntax $\mu Q.A$ to denote an agent Q under the assumption that the program contains a (unique) procedure declaration of the form $Q :: A$.

3.2 Combinators from CCP

The semantics of these combinators follows traditional treatment [25] and we only sketch them here.

Tell. This process adds c in the current store. Denotationally, it is quiescent in any observation that contains at least as much information as c at the first instant. Since the store is represented in the configuration, there is no explicit operational transition. Logically, the relevant proof rule relates entailment in the underlying constraint system to entailment on agents.

Parallel composition. The quiescent points of the parallel composition of two agents agents A and B is the intersection of the quiescent points of A and B. Operationally, parallel composition is transformed into the "," of multiset union [5]. Logically, $A \mid\mid B$ behaves like the conjunction $A \wedge B$.

Timed Positive Ask. $A =$ **now** c **then** B is the process that checks if the current store is strong enough to entail c; and if so, behaves like B. The quiescent states of A follows this operational intuition. Logically, positive ask behaves like intuitionist implication.

Hiding. $X\hat{}A$ is a process where the variable X is hidden from the environment. For the operational semantics, we find a variable Y that does not occur free in the configuration, and substitute it for the bound variable. Logically, $X\hat{}A$ behaves like $\exists X.A$. The denotational semantics follows the logical intuitions [21].

3.3 Correspondence Results

The observations of a program G_1 for inputs $(c_1, \ldots, c_n)$ (in the free variables V) is denoted $O(G_1)(c_1, \ldots, c_n)$. For a finite set of variables V and a constraint d, let $\delta V.d$ designate the constraint obtained from d by existentially quantifying all its variables except those that appear in V.

$$O(G_1)(c_1, \ldots c_n) \stackrel{d}{=} (\delta V.d_1, \ldots, \delta V.d_n),$$
$$\text{if } G_i \parallel c_i \longrightarrow R_{i+1}, \sigma(R_i) = d_i, R_i \rightsquigarrow G_{i+1}, 1 \leq i < n$$

The full-abstraction proof follows analogous proofs for CCP languages [25,14].

Theorem 3.1 (Adequacy, Full abstraction) *Two programs are denotationally distinct iff there is a context which can operationally distinguish between them.*

For the logical theory, we have Cut Elimination, Soundness and Partial Completeness. The reason for the partial completeness is that as in CCP, certain non-logical laws hold for existential quantification [25]. These laws arise because we are using an impoverished enough fragment of first-order logic (no arbitrary universal quantification) that the identity of variables substituted for existentially quantified variables cannot be distinguished from the environment.

Theorem 3.2

Soundness: $A \vdash B \rightarrow [\![A]\!] \subseteq [\![B]\!]$

Partial Completeness: *Let A and B be two agents without procedure calls and existential quantifiers. If $[\![A]\!] \subseteq [\![B]\!]$ then $A \vdash B$ is derivable using the above rules and axioms.*

3.4 Expressiveness: Some derived combinators

We now show how to define some natural patterns of temporal activity in tcc.

Multiform time: time A on c. **time** A **on** c denotes a process whose notion of time is the occurrence of the tokens c — A evolves only at the time instants at which the store entails c. Formally, using the notation $s \downarrow a$ to denote the subsequence of s with all elements $\supseteq a$, we have:

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

Watchdogs: do A **watching** c. This is an interrupt primitive related to *weak abortion* in ESTEREL [4]. **do** A **watching** c behaves like A till a time instant when c is entailed; when c is entailed A is killed from the next time instant onwards. (We can similarly define the related *exception handler* primitive, **do** A **watching** c **timeout** B, that also activates a handler B when A is killed.) In the following, for $s \in \mathbf{Obs}$, we use the notation $s \not\geq a$ to mean $b \not\supseteq c$ for every element $b \in s$. Formally,

$$\begin{array}{l} [\![\textbf{do } A \textbf{ watching } c]\!] \stackrel{d}{=} \\ \qquad \{s \in \mathbf{Obs} \mid s \in [\![A]\!], s \not\geq [c]\ \} \\ \qquad \cup \{s \cdot a \cdot t \in \mathbf{Obs} \mid s \cdot a \in [\![A]\!], a \supseteq [c]\} \end{array}$$

Suspension-Activation primitive: $\mathbf{S}_c\mathbf{A}_e(A)$. This is a premeption primitive that is a variant of *weak suspension* in ESTEREL [4]. $\mathbf{S}_c\mathbf{A}_e(A)$ behaves like A till a time instant when c is entailed; when c is entailed A is suspended from the next time instant onwards (hence the S_c). A is reactivated in the time instant when e is entailed (hence the A_e). The familiar $(\mathtt{control} - \mathtt{Z}, \mathtt{fg})$ is a construct in this vein. Formally,

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

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:

$$\begin{array}{rcl} \epsilon_P & = & \epsilon \\ (s \cdot a)_P & = & \left\{ \begin{array}{l} (s_P) \cdot a, \text{ if } a \in (P \textbf{ after } (s_P)) \\ (s_P), \text{ otherwise} \end{array} \right. \end{array}$$

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

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

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 sufficient condition is that $[\![B]\!]$ $\mathbf{after}$ 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:

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

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

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

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

Table C in the Appendix shows how to compositionally reduce programs containing this construct into programs that do not. We use the following abbreviations. **always** A executes A repeatedly; it is the agent $\mu g.A \parallel$ **next** g. **whenever** c **do** A executes A at the first instant at which c is entailed; it is the agent $\mu g.($**now** c **then** $A) \parallel ($**now** c **else** $g)$. Note that $[\![$**always** $A]\!]$ and $[\![$**whenever** c **do next** $A]\!]$ are basic processes if $[\![A]\!]$ is.

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:

$$
\begin{array}{l}
\textbf{now}\ c\ \textbf{else}\ A = \\
\quad \textbf{clock}\ (\textbf{now}\ c\ \textbf{then next abort})\ \textbf{do next}\ A
\end{array}
$$

$$\textbf{whenever}\ c\ \textbf{do}\ A = \textbf{clock}\ c\ \textbf{do}\ A$$

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

$$
\begin{array}{l}
\textbf{do}\ A\ \textbf{watching}\ c = \\
\quad \textbf{clock}\ (\textbf{whenever}\ c\ \textbf{do next abort})\ \textbf{do}\ A
\end{array}
$$

$$\mathbf{S}_c\mathbf{A}_e(A) = \textbf{clock}\ (\textbf{whenever}\ c\ \textbf{do next}\ e)\ \textbf{do}\ A$$

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

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

4 Implementation

In this section, we sketch the compositional compilation of tcc programs into deterministic finite automata, with loop-free computation at each state. This is the *only* result in this paper that uses the assumption that procedures do not have parameters.

The automaton for a program is specified by the following data: (1) a set of states Z, with each state $z \in Z$ labeled with a finite, recursion-free determinate concurrent constraint program (2) a distinguished start state, and (3) a set of directed edges between pairs of states, labeled with constraints. The automata will satisfy the property that for every node the set of labels on outgoing edges are closed under least upper bounds (lubs).

Execution of automaton. At any given time instant, the automaton is in a given state; at the first instant it is in the start state. When the environment passes in an input, the program in the state is executed in conjunction with the input. The resulting store is the output at that time instant. The edge labeled with the greatest constraint less than the output of the current state is used to go to the next state. The execution process is repeated in this state at the next time instant.

Bounded response. We assume that the time taken by the underlying constraint system to answer a query is bounded in the size of the inputs. This assumption is satisfied by most constraint systems (Herbrand (which is linearly-bounded), FD, real arithmetic etc.).

Since the cc program labeling a state is finite and recursion free, there are compile-time determinable bounds on the number of constraints and queries at each state. Under the further assumption that at any time-step the environment supplies constraints whose total size is bounded, we get that the computation at each step is bounded.

tcc programs are finite state. This result crucially depends on the assumption that procedures do not have parameters. As in analogous results for synchronous programming languages, the finite state character of tcc programs is attested to by the fact that the set of all possible *derivatives* [9], the "state space" of the program, is finite.

Construction of automaton. As in synchronous languages, the finiteness of the set of derivatives of a program induces a non-compositional compilation algorithm for tcc programs. However, tcc admits a compositional compilation as well. We sketch below the automaton construction for parallel composition. This is the *key case* that causes non-compositionality in synchronous languages [9]; other cases are not difficult and omitted from this extended abstract.

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

The cc program in $\langle q_1, q_2 \rangle$ is the parallel composition of the programs in the q_i's. The correct handling of the causality *within* a time instant is captured by the parallel composition of the cc programs from q_1 and q_2.

There is a transition $\langle t_1, t_2 \rangle$ from $\langle q_1, q_2 \rangle$ to $\langle q_1', q_2' \rangle$, with label $c = c_1 \sqcup c_2$ if
1) there are a pair of transitions, say t_1 with label c_1 from q_1 to q_1' in A_1 and t_2 with label c_2 from q_2 to q_2' in A_2.
2) For $i = 1, 2$, c_i is the greatest constraint entailed by c in the set of labels in A_i.
An example of a tcc program and its compiled automaton is in the Appendix.

5 Future work

A theory of pre-emption. The unification of the pre-emption combinators in LUSTRE, SIGNAL and ESTEREL 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 [1], a general description of synchronous processes.

Verification. Recall that deterministic safety properties correspond precisely to safety automata [17] that have a single designated failure state. Also note that tcc allows programs and properties to be expressed in the same language. This leads to the promise of reducing general safety properties to checking the state of a single variable, say a boolean variable. We intend to tackle the verification process itself via extant model-checking techniques, for example [7], or theorem-proving techniques.

Acknowledgements. We gratefully acknowledge extended discussions with Danny Bobrow, Jerry Burch, Adam Farquhar, Lalita Jategaonkar, John Lamping, and Brian Smith.

References

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

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

[3] 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.

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

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

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

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

[8] 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.

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

[10] 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.

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

[12] 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.

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

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

[15] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proceeedings of IFIP Congress 74*, pages 471–475., August 1974.

[16] Dexter Kozen. Results on the propositional μ-calculus. *Theretical Computer Science*, 27:333 – 354, 1983.

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

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

[19] 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.

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

[21] P. Panangaden, V. Saraswat, P. Scott, and R. Seely. A hyperdoctrinal view of concurrent constraint programming. In J. deBakker, G. Roszenberg, and W. deRoever, editors, *Proceedings of the REX Workshop*, 1992. LNCS 666.

[22] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An algebraic approach to program dependencies. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

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

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

[25] 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.

A Summary of syntax and semantics

Syntax.

$$
\begin{array}{llcll}
(Agents) & A & ::= & c & \text{(Tell)} \\
 & & \mid & \textbf{now } c \textbf{ then } A & \text{(Timed Positive Ask)} \\
 & & \mid & \textbf{now } c \textbf{ else } A & \text{(Timed Negative Ask)} \\
 & & \mid & \textbf{next } A & \text{(Unit Delay)} \\
 & & \mid & \textbf{abort} & \text{(Abort)} \\
 & & \mid & \textbf{skip} & \text{(Skip)} \\
 & & \mid & A \mid\mid A & \text{(Parallel composition)} \\
 & & \mid & X\hat{\ }A & \text{(Hiding)} \\
 & & \mid & P & \text{(Nested program)} \\
 & & \mid & g & \text{(Procedure Call)}
\end{array}
\qquad (1)
$$

$$
\begin{array}{llcll}
(Proc) & g & ::= & p(t_1, \ldots, t_n) & \\
(Decl) & D & ::= & g :: A & \text{(Definition)} \\
 & & \mid & D.D & \text{(Conjunction)} \\
(Prog) & P & ::= & \{D.A\} &
\end{array}
$$

Semantic Domain. We take as fixed a constraint system $\mathcal{C}$, with underlying set of tokens (or primitive constraints) D and let the finitary inference relation $\vdash$ (that relates finite set of tokens to tokens) record which (primitive constraint) follows from which ollection of primitive constraints. 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 primitive constraints. 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 A.1 **Obs**, the set of observations, is the set of finite sequences of constraints. $\square$

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 A.2 A process $P \in \mathbf{Proc}$ is a non-empty, prefix-closed subset of $\mathbf{Obs}$ such that P $\mathbf{after}$ s is determinate for all $s \in P$. $\square$

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

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

B Logic underlying tcc

The syntax of formulas in the logic is exactly the same as that of agents.

$$
\begin{array}{lll}
(Agents) \quad A & ::= & c \mid \textbf{now } c \textbf{ then } A \mid \textbf{now } c \textbf{ else } A \mid \textbf{next } A \\
& & \mid A \parallel A \mid \textbf{skip} \mid \textbf{abort} \mid X\hat{\ }A \mid g
\end{array}
\qquad (2)
$$

Sequents are of the form $A_1, \ldots, A_n \vdash A$, where A_i, A are agents. Intuitively, such a sequent is valid if every observation that can be made of system consisting of the A_i running in parallel can be made of A.

The rules of inference for the logic include the structural rules of Exchange, Weakening and Contraction, and the Identity and Cut rules. The entailment relation of the underlying constraint system is tied to that for agents by:

$$\frac{d_1, \ldots, d_n \vdash_{\mathcal{C}} c}{d_1, \ldots, d_n \ \vdash\ c} \text{(Constraint)}$$

The rules for the combinators are obtained from those of intuitionistic logic by writing **now** c **then** A as $c \rightarrow A$, **now** c **else** A as $c \vee$ **next** A, $A \mid\mid B$ as $A \wedge B$, $X\hat{}A$ as $\exists X.A$, **skip** as *true* and **abort** as $\perp$ (false). In addition, the rules for **next** A are:

$$\frac{\Gamma \ \vdash\ B}{\textbf{next}\,\Gamma \ \vdash\ \textbf{next}\ B} \text{(Step)}$$

C Clock algebra

The following laws hold for the `clock` combinator:

clock P **do abort** $=$ **abort**
clock P **do skip** $=$ **skip**
clock P **do** $(A_1 \mid\mid A_2) = ($**clock** P **do** $A_1) \mid\mid ($**clock** P **do** $A_2)$
clock P **do** $X\hat{}A = Z\hat{}$**clock** P **do** $A[Z/X]$[3]
clock P **do** $\mu Q.A = \mu Q.$**clock** P **do** A

clock c **do** $A =$ **whenever** c **do** A

clock abort do $A =$ **skip**

clock skip do $A = A$

clock $(B_1 \mid\mid B_2)$ **do** $A =$ **clock** B_1 **do** $($**clock** B_2 **do** $A)$

clock $\mu G.B$ **do** $A = \mu G.$**clock** B **do** A

For $P =$ **next** B:

clock P **do** $d = d$
clock P **do** $($**now** d **then** $A) =$ **now** d **then clock** P **do** A
clock P **do** $($**now** d **else** $A) =$ **now** d **else clock** B **do** A
clock P **do** $($**next** $A) =$ **next clock** B **do** A

For $P =$ **now** c **then next** B:

$$
\begin{array}{l}
\textbf{clock } P \textbf{ do } d = d \\
\textbf{clock } P \textbf{ do } (\textbf{now } d \textbf{ then } A) = \textbf{now } d \textbf{ then clock } P \textbf{ do } A \\
\textbf{clock } P \textbf{ do } (\textbf{now } d \textbf{ else } A) \\
\quad = \quad \textbf{now } c \textbf{ then clock next } B \textbf{ do } (\textbf{now } d \textbf{ else } A) \\
\qquad\quad \| \textbf{ now } (c, d) \textbf{ else } A \\
\textbf{clock } P \textbf{ do } (\textbf{next } A) \\
\quad = \quad (\textbf{now } c \textbf{ then } (\textbf{next clock } B \textbf{ do } A)) \\
\qquad\quad \| (\textbf{now } c \textbf{ else } A)
\end{array}
$$

For $P =$ **now** c **else** B:

$$
\begin{array}{l}
\textbf{clock } P \textbf{ do } d = d \\
\textbf{clock } P \textbf{ do } (\textbf{now } d \textbf{ then } A) = \textbf{now } d \textbf{ then clock } P \textbf{ do } A \\
\textbf{clock } P \textbf{ do } (\textbf{now } d \textbf{ else } A) \\
\quad = \quad \textbf{now } c \textbf{ then now } d \textbf{ else } A \\
\qquad\quad \| \textbf{ now } (c, d) \textbf{ else clock } B \textbf{ do } A \\
\textbf{clock } P \textbf{ do } (\textbf{next } A) \\
\quad = \quad \textbf{now } c \textbf{ then next } A \\
\qquad\quad \| \textbf{ now } c \textbf{ else clock } B \textbf{ do } A
\end{array}
$$

D Example of compilation

We will allow ourselves the liberty of writing programs with recursive procedures that take parameters. However, we shall require that any procedure call (for a recursive procedure) 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 $\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{n})$, $\mathtt{q}(\mathtt{Y}_1, \ldots, \mathtt{Y}_\mathtt{n}), \ldots$.

It is not difficult to show that this liberalization does not affect finite-state compilability. (In essence, 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.)

The following program is a controller for a mouse. It waits for the signal `start`, then starts counting the number of clicks of the mouse upto the time it receives `stop`.

It then reports whether it received zero, one or many clicks. Here the variables `start`, `stop`, `zero`, `one`, `many` are assumed to be bound in the lexical scope. We use the syntax

$$\textbf{now } c \textbf{ then } A \textbf{ else } B$$

for the agent **now** c **then** A $\|$ **now** c **else** B, and use "," for parallel composition.

```
controller ::
   X^(do whenever start do (X = 0, mouse(X))
        watching stop,
      whenever stop do (now X = 0 then zero,
                        now X = 1 then one,
                        now X = 2 then many)).

mouse(Z) :: mouse_zero(Z), mouse_one(Z), mouse_many(Z).

mouse_zero(Z) :: now Z = 0 then
   now click then next Z = 1 else (Z = 0, mouse_zero(Z)).

mouse_one(Z) :: now Z = 1 then
   now click then next Z = 2 else (Z = 1, mouse_one(Z)).

mouse_many(Z) :: now Z = 2 then always Z = 2.
```

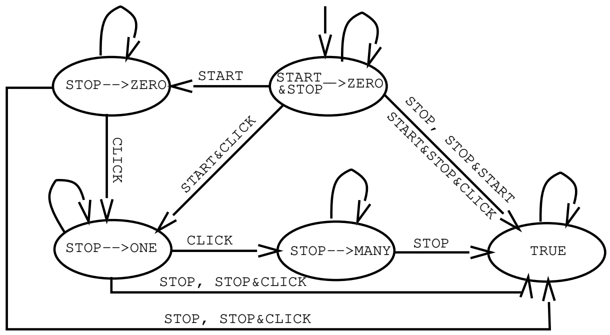

Figure 1: Automaton for the mouse program