Concurrency, Time and Constraints

Frank D. Valencia ⋆

Dept. of Information Technology, Uppsala University
Box 337 SE-751 05 Uppsala, Sweden
`Email: frankv@it.uu.se   Fax: +46 18 511 925`

Abstract *Concurrent constraint programming* (ccp) is a model of concurrency for systems in which agents (also called processes) interact with one another by telling and asking information in a shared medium. *Timed* (or *temporal*) ccp extends ccp by allowing agents to be constrained by time requirements. The novelty of timed ccp is that it combines in one framework an *operational and algebraic* view based upon process calculi with a *declarative* view based upon temporal logic. This allows the model to benefit from two well-established theories used in the study of concurrency.
This essay offers an overview of timed ccp covering its basic background and central developments. The essay also includes an introduction to a temporal ccp formalism called the `ntcc` calculus.

1 Introduction

Concurrency theory has made progress by extending well-established models of computation to capture new and wider phenomena. These extensions should not come as a surprise since the field is indeed large and subject to the advents of new technology. One particular phenomenon, for which extensions of the very first theories of concurrency were needed, is the notion of *time*.

Time is not only a fundamental concept in concurrency but also in science at large. Just like modal extensions of logic for temporal progression study time in logic reasoning, mature models of concurrency were extended to study time in concurrent activity. For instance, neither Milner's CCS [23], Hoare's CSP [18], nor Petri Nets [31], in their original form, were concerned with temporal behavior but they all have been extended to incorporate an explicit notion of time. Namely, Timed CCS [52], Timed CSP [34], and Timed Petri Nets [53].

The notion of *constraint* is certainly not rare in concurrency. After all, concurrency is about the interaction of agents and such an interaction often involves constraints of some sort (e.g., synchronization constraints, access-control, actions that must eventually happen, actions that cannot happen, etc).

Saraswat's *concurrent constraint programming* (ccp) [44] is a well-established formalism for concurrency based upon the shared-variables communication model where interaction arises via constraint-imposition over shared-variables. In ccp, agents can interact by *adding* (or *telling*) partial information in a medium, a so-called *store*. Partial

⋆ This work was supported by the **PROFUNDIS** Project.

information is represented by *constraints* (i.e., first-order formulae such as $x > 42$) on the shared variables of the system. The other way in which agents can interact is by *asking* partial information to the store. This provides the synchronization mechanism of the model; asking agents are suspended until there is enough information in the store to answer their query.

As other models of concurrency, ccp has been extended to capture aspects such as mobility [8, 36, 12], stochastic behavior [13], and most prominently time [39, 5, 41, 14]. *Timed* ccp extends ccp by allowing agents to be constrained by time requirements.

A distinctive feature of timed ccp is that it combines in one framework an *operational and algebraic* view based upon process calculi with a *declarative* view based upon temporal logic. So, processes can be treated as computing agents, algebraic terms and temporal formulae. At this point it is convenient to quote Robin Milner:

> *I make no claim that everything can be done by algebra ... It is perhaps equally true that not everything can be done by logic; thus one of the outstanding challenges in concurrency is to find the right marriage between logic and behavioral approaches*
> — Robin Milner, [23]

In fact, the combination in one framework of the alternative views of processes mentioned above allows timed ccp to benefit from the large body of techniques of well established theories used in the study of concurrency and, in particular, *timed systems*.

Furthermore, timed ccp allows processes to be (1) expressed using a vocabulary and concepts appropriate to the *specific domain* (of some application under consideration), and (2) read and understood as temporal logic *specifications*. This feature is suitable for timed systems as they often involve *specific domains* (e.g., controllers, databases, reservation systems) and have time-constraints *specifying* their behavior (e.g., the lights must be switched on within the next three seconds). Indeed, several timed extensions of ccp have been developed in order to provide settings for the modeling, programming and specification of timed systems [39, 42, 5, 14].

This paper offers an overview of timed ccp with its basic background and various approaches explored by researchers in this field. Furthermore, it offers an introduction to a timed ccp process calculus called `ntcc`. The paper is organized as follows. In Section 2 we discuss briefly those issues from models of concurrency, in particular process calculi, relevant to temporal ccp. In Sections 3 and 4 we give an overview of ccp and timed ccp. Section 5 is devoted to address, in the context of timed ccp and by using `ntcc`, those issues previously discussed in Section 2. Finally in Section 6 we discuss central developments in timed ccp.

2 Background: Models of Concurrency

Concurrency is concerned with the fundamental aspects of systems consisting of multiple computing agents, usually called *processes*, that interact among each other. This covers a vast variety of systems, so-called *concurrent systems*, which nowadays most people can easily relate to due to technological advances such as the Internet, programmable robotic devices and mobile computing . For instance, *timed* systems, in which agents are constrained by temporal requirements. Some example of timed systems are: Browser applications which are constrained by timer-based exit conditions (i.e., *time-outs*) for the case in which a sever cannot be contacted; E-mailer applications can be required to check for messages every k time units. Also, robots can be programmed with time-outs (e.g., to wait for some signal) and with timed instructions (e.g., to go forward for 42 time-units).

Because of the practical relevance, complexity and ubiquity of concurrent systems, it is crucial to be able to describe, analyze and, in general, reason about concurrent behavior. This reasoning must be precise and reliable. Consequently, it ought to be founded upon mathematical principles in the same way as the reasoning about the behavior of sequential programs is founded upon logic, domain theory and other mathematical disciplines.

Nevertheless, giving mathematical foundations to concurrent computation has become a serious challenge for computer science. Traditional mathematical models of (sequential) computation based on functions from inputs to outputs no longer apply. The crux is that concurrent computation, e.g., in a reactive system, is seldom expected to terminate, it involves constant interaction with the environment, and it is *nondeterministic* owing to unpredictable interactions among agents.

Models of Concurrency: Process Calculi. Computer science has therefore taken up the task of developing *models*, conceptually different from those of sequential computation, for the precise understanding of the behavior of concurrent systems. Such models, as other scientific models of reality, are expected to comply with the following criteria: They must be *simple* (i.e., based upon few basic principles), *expressive* (i.e., capable of capturing interesting real-world situations), *formal* (i.e., founded upon mathematical principles), and they must provide *techniques* to allow reasoning about their particular focus.

Process calculi are one of the most common frameworks for modeling concurrent activity. These calculi treat processes much like the λ-calculus treats computable functions. They provide a language in which the structure of *terms* represents the structure of processes together with an *operational semantics* to represent computational steps. For example, the term $P \parallel Q$, which is built from P and Q with the *constructor* $\parallel$, represents the process that results from the parallel execution of those represented by P and Q. An operational semantics may dictate that if P can evolve into P' in a computational step P' then $P \parallel Q$ can also evolve into $P' \parallel Q$ in a computational step.

An appealing feature of process calculi is their *algebraic* treatment of processes. The constructors are viewed as the *operators* of an algebraic theory whose equations and inequalities among terms relate process behavior. For instance, the construct $\parallel$ can be viewed as a commutative operator, hence the equation $P \parallel Q \equiv Q \parallel P$ states that the behavior of the two parallel compositions are the same. Because of this algebraic emphasis, these calculi are often referred to as *process algebras*.

There are many different process calculi in the literature mainly agreeing in their emphasis upon algebra. The main representatives are CCS [23], CSP [18], and the process algebra ACP [2]. The distinctions among these calculi arise from issues such as the process constructions considered (i.e., the language of processes), the methods used for giving meaning to process terms (i.e. the semantics), and the methods to reason about process behavior (e.g., process equivalences or process logics).

Semantics. The methods by which process terms are endowed with meaning may involve at least three approaches: *operational*, *denotational* and *algebraic* semantics. The *operational method* was pioneered by Plotkin [32]. An operational semantics interprets a given process term by using transitions (labeled or not) specifying its computational steps. A labeled transition $P \xrightarrow{\mathbf{a}} Q$ specifies that P performs $\mathbf{a}$ and then behaves as Q. The *denotational method* was pioneered by Strachey and provided with a mathematical foundation by Scott. A denotational semantics interprets processes by using a function $[\![\cdot]\!]$ which maps them into a more abstract mathematical object (typically, a structured set or a category). The map $[\![\cdot]\!]$ is *compositional* in that the meaning of processes is determined from the meaning of its sub-processes. The *algebraic method* has been advocated by Bergstra and Klop [2]. An algebraic semantics attempts to give meaning by stating a set of laws (or axioms) equating process terms. The processes and their operations are then interpreted as structures that obey these laws.

Behavioral Analysis. Much work in the theory of process calculi, and concurrency in general, involves the analysis of process equivalences. Let us say that our equivalence under consideration is denoted by $\sim$. Two typical questions that arise are: (1) Is the equivalence decidable ? (2) Is the equivalence a congruence ? The first question refers to the issue as to whether there can be an algorithm that fully determines (or decides) for every P and Q if $P \sim Q$ or $P \not\sim Q$. Since most process calculi can model Turing machines most natural equivalences are therefore undecidable. So, the interesting question is rather for what subclasses of processes is the equivalence decidable. The second question refers to the issue as to whether the fact that P and Q are equivalent implies that they are still equivalent in any process context. A process context C can be viewed as a term with a hole $[\cdot]$ such that placing a process in the hole yields a process term. Hence, the equivalence is a congruence if $P \sim Q$ implies $C[P] \sim C[Q]$ for every process context C. The congruence issue is fundamental for algebraic as well as practical reasons; one may not be content with having $P \sim Q$ equivalent but $R \parallel P \not\sim R \parallel Q$ (here the context is $C = R \parallel [\cdot]$).

Specification and Logic. One often is interested in verifying whether a given process satisfies a specification; the so-called *verification problem*. But process terms themselves specify behavior, so they can also be used to express specifications. Then this verification problem can be reduced to establishing whether the process and the specification process are related under some behavioral equivalence (or pre-order). Another way of expressing process specifications, however, is by using temporal logics. These logics were introduced into computer science by Pnueli [33] and thereafter proven to be a good basis for specification as well as for (automatic and machine-assisted) reasoning about concurrent systems. Temporal logics can be classified into linear and branching time logics. In the *linear* case at each moment there is only one possible future whilst in the *branching* case at may split into alternative futures.

3 Concurrent Constraint Programming

In his seminal PhD thesis [38], Saraswat proposed concurrent constraint programming as a model of concurrency based on the shared-variables communication model and a few primitive ideas taking root in logic. As informally described in this section, the ccp model elegantly combines logic concepts and concurrency mechanisms.

Concurrent constraint programming traces its origins back to Montanari's pioneering work [26] leading to constraint programming and Shapiro's concurrent logic programming [45]. The ccp model has received a significant theoretical and implementational attention: Saraswat, Rinard and Panangaden [44] as well as De Boer, Di Pierro and Palamidessi [6] gave fixed-point denotational semantics to ccp whilst Montanari and Rossi [35] gave it a (true-concurrent) Petri-Net semantics; De Boer, Gabrielli et al [7] developed an inference system for proving properties of ccp processes; Smolka's Oz [47] as well as Haridi and Janson's AKL [17] programming languages are built upon ccp ideas.

Description of the model. A fundamental issue of the ccp model is the *specification of concurrent systems* in terms of constraints. A constraint is a first-order formula representing *partial information* about the shared variables of the system. For example, the constraint $x + y > 42$ specifies possible values for x and y (those satisfying the inequation). The ccp model is parameterized in a *constraint system* which specifies the constraints of relevance for the kind of system under consideration and an *entailment relation* $\models$ between constraints (e.g, $x + y > 42 \models x + y > 0$).

During computation, the state of the system is specified by an entity called the *store* where items of information about the variables of the system reside. The store is represented as a constraint and thus it may provide only partial information about the variables. This differs fundamentally from the traditional view of a store based on the Von Neumann memory model, in which each variable is assigned a uniquely determined value (e.g., $x = 42$ and $y = 7$) rather than a set of possible values.

Some readers may feel uneasy as the notion of store in ccp suggests a model of concurrency with central memory. This is, however, an abstraction that simplifies the presentation of the model. The store can be distributed in several sites according to the agents that share the same variables (see [38] for further discussions about this matter). Conceptually, the store in ccp is the *medium* through which agents interact with each other.

The ccp processes can update the state of the system only by *adding* (or *telling*) information to the store. This is represented as the (logical) conjunction of the constraint being added and the store representing the previous state. Hence, the update is not about changing the values of the variables but rather about ruling out some of the previously possible values. In other words, the store is *monotonically refined*.

Furthermore, processes can synchronize by *asking* information to the store. Asking is blocked until there is enough information in the store to *entail* (i.e., answer positively) their query. The ask operation is seen as determining whether the constraint representing the store entails the query.

A ccp computation terminates whenever it reaches a point, called *resting* or *quiescent* point, in which no more new information is added to the store. The final store, also called *quiescent store* (i.e,. the store at the quiescent point), is the output of the computation.

Example 1. To make the description of the ccp model clearer, consider the simple ccp scenario illustrated in Figure 1. We have four agents (or processes) wishing to interact through an initially empty medium. Let us name them, starting from the upper rightmost agent in a clockwise fashion, A_1, A_2, A_3 and A_4, respectively. Suppose that they are scheduled for execution in the same order they were named.

This way A_1 moves first and tells the others through the medium that the temperature value is greater than 42 degrees but without specifying the exact value. In other words A_1 gives the others partial information about the temperature. This causes the addition of the item "temperature>42" to the previously empty store.

Now A_2 asks whether the temperature is exactly 50 degrees, and if so it wishes to execute a process P. From the current information in the store, however, it cannot be determined what the exact value of the temperature is. The agent A_2 is then blocked and so is the agent A_3 since from the store it cannot be determined either whether the temperature is between 0 and 100 degrees.

The turn is for A_4 which tells that the temperature is less than 70 degrees. The store becomes "temperature $>$ 42 $\wedge$ temperature $<$ 70". Now A_3 can execute Q as its query is entailed by the information in the store . The other ask agent A_2 is doomed to be blocked forever unless Q adds enough information to the store to entail its query. $\square$

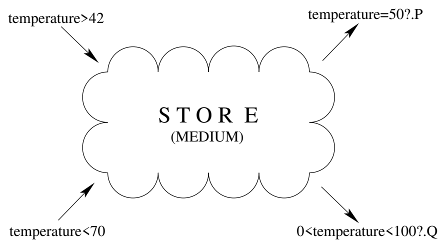

Figure1. A simple ccp scenario

The CCP Process Language. In the spirit of process calculi, the language of processes in the ccp model is given with a reduced number of primitive operators or combinators. Rather than giving the actual syntax of the language, we content ourselves with describing the basic intuition that each construct embodies. So, in ccp we have:

- *The tell action*, for expressing tell operations. E.g., agent A_1 above.
- *The ask action (or prefix action)*, for expressing an ask operation that prefixes another process; its continuation. E.g., the agent A_2 above.
- *Parallel composition*, which combines processes concurrently. E.g., the scenario in Figure 1 can be specified as the parallel composition of A_1, A_2, A_3 and A_4.

– *Hiding* (or *locality*), for expressing local variables that delimit the interface through which a process can interact with others.
– *Summation*, which expresses a disjunctive combination of agents to allow alternate courses of action.
– *Recursion*, for defining infinite behavior.

It is worth pointing out that without summation, the ccp model is deterministic in the sense that the quiescent or final store is always the same, independently from the execution order (scheduling) of the parallel components [44].

4 Timed Concurrent Constraint Programming

The first timed ccp model was introduced by Saraswat et al [39] as an extension of ccp aimed at programming and modeling timed, reactive systems. This model, which has attracted growing attention during the last five years or so, elegantly combines ccp with ideas from the paradigms of Synchronous Languages [3, 15].

As any other model of computation, the tcc model makes an ontological commitment about computation. It emphasizes the view of reactive computation as proceeding *deterministically* in discrete time units (or time *intervals*). More precisely, time is conceptually divided into discrete intervals. In each time interval, a deterministic ccp process receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the *initial store*, and when it reaches its resting point, it responds to the environment with the *final store*. Also, the resting point determines a residual process, which is then executed in the next time interval.

This view of reactive computation is particularly appropriate for programming reactive systems such as robotic devices, micro-controllers, databases and reservation systems. These systems typically operate in a cyclic fashion; in each cycle they receive and input from the environment, compute on this input, and then return the corresponding output to the environment.

The fundamental move in the tcc model is to extend the standard ccp with *delay* and *time-out* operations. These operations are fundamental for programming reactive systems. The delay operation forces the execution of a process to be postponed to the next time interval. The time-out (or weak *pre-emption*) operation waits during the current time interval for a given piece of information to be present and if it is not, triggers a process in the *next time interval*.

Pre-emption and multi-form time. In spite of its simplicity, the tcc extension to ccp is far-reaching. Many interesting temporal constructs can be expressed, in particular:

– $\mathbf{do}\; P\; \mathbf{watching}\; c$. This interrupt process executes P continuously until the item of information (e.g, a signal) c is present (i.e., entailed by the information in the store); when c is present P is *killed* from the next time unit onwards. This corresponds to the familiar `kill` command in Unix or clicking on the stop bottom of your favorite web browser.
– $\mathbf{S}_c\mathbf{A}_d(P)$. This pre-emption process executes P continuously until c is present; when c is present P is *suspended* from the next time unit onwards. The process P

is *reactivated* when d is present. This corresponds to the familiar (`ctrl -Z, fg`) mechanism in Unix.
- $\mathbf{time}\, P\, \mathbf{on}\, c$. This denotes a process whose *notion of time* is the occurrence of the item of information c. That is, P evolves only in those time intervals where c holds.

In general, tcc allows processes to be "clocked" by other processes. This provides meaningful pre-emption constructs and the ability of defining *multiple forms of time* instead of only having a unique global clock.

4.1 More Timed CCP Models

The `ntcc` calculus is generalization of the tcc model originated in [28] by Palamidessi, Nielsen and the present author. The calculus is built upon few basic ideas but it captures several aspects of timed systems. As tcc, `ntcc` can model unit delays, time-outs, pre-emption and synchrony. Additionally, it can model *unbounded but finite delays, bounded eventuality, asynchrony* and *nondeterminism*. The applicability of the calculus has been illustrated with several examples of discrete-time systems involving , mutable data structures, robotic devices, multi-agent systems and music applications [37].

Another interesting extension of tcc, which does not consider nondeterminism or unbounded finite-delay, has been proposed in [42]. This extension adds strong pre-emption: the time-out operations can trigger activity in the current time interval. In contrast, `ntcc` can only express weak pre-emption. Other extensions of tcc have been proposed in [14]. In [14] processes can evolve continuously as well as discretely. None of these extensions consider nondeterminism or unbounded finite-delay.

The tccp framework, introduced in [5] by Gabrielli et al, is a fundamental representative model of (nondeterministic) timed ccp. The authors in [5] also advocate the need of nondeterminism in the context of timed ccp. In fact, they use tccp to model interesting applications involving nondeterministic timed systems (see [5]). The major difference between tccp and `ntcc` is that the former extends the original ccp while the latter extends the tcc model (so, except for allowing nondeterminism, it makes the same commitments about computation). In tccp the information about the store is carried through the time units, thus the semantic setting is completely different. The notion of time is also different; in tccp each time unit is identified with the time needed to ask and tell information to the store. As for the constructs, unlike `ntcc`, tccp provides for arbitrary recursion and does not have an operator for specifying unbounded but finite delays.

As briefly described in this section, there are several models of timed ccp, and it would be hard to introduce all of them in detail. I shall introduce in detail the generalization of Saraswat's tcc, the `ntcc` calculus, and then indicate as related work the developments in other models which appear to me to be central.

5 The `ntcc` process calculus

This section gives an introduction to the `ntcc` model. We shall give evidence of the compliance of `ntcc` with the criteria for models of concurrency previously mentioned (i.e., it is simple, expressive, formal and it provides reasoning techniques). First, we shall see that it captures fundamental aspects of concurrency (i.e., discrete-time reactive computations, nondeterminism, synchrony and asynchrony) whilst keeping a pleasant degree of *simplicity*. Second, the expressiveness of `ntcc` will be illustrated by modeling robotic devices. Furthermore, we shall see that `ntcc` is founded upon *formal* theories such as process calculi and first-order logic. Finally, we shall present some of the techniques that `ntcc` provides to reason about concurrent activity. Namely,

1. Several *equivalences*, characterized operationally, to compare the behavior of processes much like the behavioral equivalences for existing process calculi (e.g., bisimilarity and trace-equivalence).
2. A *denotational semantics* which interprets a given process as the set of sequences of actions it can potentially exhibit while interacting with arbitrary environments.
3. A *process logic* with an associated *inference system* than can be used much like the Hoare's program logic for sequential computation. The logic can be used to express required timed behaviors of processes, i.e., *temporal specifications*. The inference system can be used to prove that a process fulfills the specification.

We shall begin with an informal description of the process calculus with examples. These examples are also meant to give a flavor of the range of application of timed ccp.

5.1 Intuitive Description of Constraint Systems

In this section we introduce the basic ideas underlying the `ntcc` calculus in an informal way. We shall begin by introducing the notion of a constraint system, which is central to concurrent constraint programming. We then describe the basic process constructs by means of examples. Finally, we shall describe some convenient derived constructs.

The `ntcc` processes are parametric in a *constraint system.* A constraint system provides a *signature* from which syntactically denotable objects called *constraints* can be constructed and an *entailment relation* $\models$ specifying inter-dependencies between these constraints.

A constraint represents a piece of information (or *partial information*) upon which processes may act. For instance, processes modeling temperature controllers may have to deal with partial information such as $42 < \mathtt{tsensor} < 100$ expressing that the sensor registers an unknown (or not precisely determined) temperature value between 42 and 100. The inter-dependency $c \models d$ expresses that the information specified by d follows from that by c, e.g., $(42 < \mathtt{tsensor} < 100) \models (0 < \mathtt{tsensor} < 120)$.

We can set up the notion of constraint system by using first-order logic. Let us suppose that Σ is a signature (i.e., a set of constants, functions and predicate symbols) and that Δ is a consistent first-order theory over Σ (i.e., a set of sentences over Σ having at least one model). Constraints can be thought of as first-order formulae over Σ. We can then decree that $c \models d$ if the implication $c \Rightarrow d$ is valid in Δ. This gives us a simple and general formalization of the notion of constraint system as a pair (Σ, Δ).

In the examples below we shall assume that, in the underlying constraint system, Σ is the set $\{=, <, 0, 1 \ldots\}$ and Δ is the set of sentences over Σ valid on the natural numbers.

Intuitive Description of Processes

We now proceed to describe with examples the basic ideas underlying the behavior of ntcc processes. For this purpose we shall model simple behavior of controllers such as Programmable Logic Controllers (PLC's) and RCX bricks.

PLC's are often used in timed systems of industrial applications [9], whilst RCX bricks are mainly used to construct autonomous robotic devices [20]. These controllers have external input and output ports. One can attach, for example, sensors of light, touch or temperature to the input ports, and motors, lights or alarms to the output ports. Typically PLC's and RCX bricks operate in a cyclic fashion. Each cycle consist of receiving an input from the environment, computing on this input, and returning the corresponding output to the environment.

Our processes will operate similarly. Time is conceptually divided into *discrete intervals (or time units)*. In a particular time interval, a process P_i receives a *stimulus* c_i from the environment (see Equation 1 below). The stimulus is some piece of information, i.e., a constraint. The process P_i executes with this stimulus as the initial store, and when it reaches its resting point (i.e., a point in which no further computation is possible), it *responds* to the environment with a resulting store d_i. Also the resting point determines a residual process P_{i+1}, which is then executed in the next time interval.

The following sequence illustrates the stimulus-response interactions between an environment that inputs $c_1, c_2, \ldots$ and a process that outputs $d_1, d_2, \ldots$ on such inputs as described above.

$$P_1 \xRightarrow{(c_1,d_1)} P_2 \xRightarrow{(c_2,d_2)} \ldots P_i \xRightarrow{(c_i,d_i)} P_{i+1} \xRightarrow{(c_{i+1},d_{i+1})} \ldots \qquad (1)$$

Communication: Telling and Asking Information. The ntcc processes communicate with each other by posting and reading partial information about the variables of system they model. The basic actions for communication provide the *telling* and *asking* of information. A tell action adds a piece of information to the common store. An ask action queries the store to decide whether a given piece of information is present in it. The store as a constraint itself. In this way addition of information corresponds to logic conjunction and determining presence of information corresponds to logic implication.

The tell and ask processes have respectively the form

$$\mathbf{tell}(c) \text{ and } \mathbf{when}\ c\ \mathbf{do}\ P. \qquad (2)$$

The only action of a tell process $\mathbf{tell}(c)$ is to add, within a time unit, c to the current store d. The store then becomes $d \wedge c$. The addition of c is carried out even if the store becomes inconsistent, i.e., $(d \wedge c) = \mathtt{false}$, in which case we can think of such an addition as generating a *failure*.

Example 2. Suppose that $d = (\mathtt{motor_1\_speed} > \mathtt{motor_2\_speed})$. Intuitively, d tells us that the speed of motor one is greater than that of motor two. It does not tell us what the specific speed values are. The execution in store d of process

$$\mathbf{tell}(\mathtt{motor_2\_speed} > 10)$$

causes the store to become $(\texttt{motor}_1\texttt{\_speed} > \texttt{motor}_2\texttt{\_speed} > 10)$ in the current time interval, thus increasing the information we know about the system – we now know that both speed values are greater than 10.

Notice that in the underlying constraint system $d \models \texttt{motor}_1\texttt{\_speed} > 0$, therefore the process

$$\mathbf{tell}(\texttt{motor}_1\texttt{\_speed} = 0)$$

in store d causes a failure. □

The process $\mathbf{when}\ c\ \mathbf{do}\ P$ performs the action of asking c. If during the current time interval c can eventually be inferred from the store d (i.e., $d \models c$) then P is executed within the same time interval. Otherwise, $\mathbf{when}\ c\ \mathbf{do}\ P$ is precluded from execution in any future time interval (i.e., it becomes constantly inactive).

Example 3. Suppose that $d = (\texttt{motor}_1\texttt{\_speed} > \texttt{motor}_2\texttt{\_speed})$ is the store. The process

$$P = \mathbf{when}\ \texttt{motor}_1\texttt{\_speed} > 0\ \mathbf{do}\ Q$$

will execute Q in the current time interval since $d \models \texttt{motor}_1\texttt{\_speed} > 0$, by contrast the process

$$P' = \mathbf{when}\ \texttt{motor}_1\texttt{\_speed} > 10\ \mathbf{do}\ Q$$

will not execute Q unless more information is added to the store, during the current time interval, to entail $\texttt{motor}_1\texttt{\_speed} > 10$.

The intuition is that any process in $d = (\texttt{motor}_1\texttt{\_speed} > \texttt{motor}_2\texttt{\_speed})$ can execute a given action if and only if it can do so whenever $\texttt{motor}_1\texttt{\_speed}$ and $\texttt{motor}_2\texttt{\_speed}$ are set to arbitrary values satisfying d. So, P above executes Q if $\texttt{motor}_1\texttt{\_speed}$ and $\texttt{motor}_2\texttt{\_speed}$ take on any value satisfying d. □

The above example illustrates the partial information allows us to model the actions that a system can perform, regardless of the alternative values a variable may assume, as long they comply with the constraint representing the store.

Nondeterminism. As argued above, partial information allows us to model behavior for alternative values that variables may take on. In concurrent systems it is often convenient to model behavior for *alternative courses* of action, i.e., nondeterministic behavior.

We generalize the processes of the form $\mathbf{when}\ c\ \mathbf{do}\ P$ described above to guarded-choice summation processes of the form

$$\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \tag{3}$$

where I is a finite set of indices. The expression $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ represents a process that, in the current time interval, *must nondeterministically* choose a process P_j ($j \in I$) whose corresponding constraint c_j is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible during the current time unit, all the alternatives are precluded from execution.

In the following example we shall use "+" for binary summations.

Example 4. Often RCX programs operate in a set of simple stimulus-response rules of the form $\mathbf{IF}\ E\ \mathbf{THEN}\ C$. The expression E is a condition typically depending on the sensor variables, and C is a command, typically an assignment. In [11] these programs respond to the environment by choosing a rule whose condition is met and executing its command.

If we wish to abstract from the particular implementation of the mechanism that chooses the rule, we can model the execution of these programs by using the summation process. For example, the program operating in the set

$$\left\{\begin{array}{l}(\mathbf{IF}\ \mathtt{sensor}_1 > 0\ \ \mathbf{THEN}\ \mathtt{motor}_1\mathtt{\_speed} := 2),\\ (\mathbf{IF}\ \mathtt{sensor}_2 > 99\ \mathbf{THEN}\ \mathtt{motor}_1\mathtt{\_speed} := 0)\end{array}\right\}$$

corresponds to the summation process

$$P = \begin{array}{l}\mathbf{when}\ \mathtt{sensor}_1 > 0\ \ \mathbf{do}\ \mathbf{tell}(\mathtt{motor}_1\mathtt{\_speed} = 2)\\ +\\ \mathbf{when}\ \mathtt{sensor}_2 > 99\ \mathbf{do}\ \mathbf{tell}(\mathtt{motor}_1\mathtt{\_speed} = 0).\end{array}$$

In the store $d = (\mathtt{sensor}_1 > 10)$, the process P causes the store to become $d \wedge (\mathtt{motor}_1\mathtt{\_speed} = 2)$ since $\mathbf{tell}(\mathtt{motor}_1\mathtt{\_speed} = 2)$ is chosen for execution and the other alternative is precluded. In the store $\mathtt{true}$, P cannot add any information. In the store $e = (\mathtt{sensor}_1 = 10 \wedge \mathtt{sensor}_2 = 100)$, P causes the store to become either $e \wedge (\mathtt{motor}_1\mathtt{\_speed} = 2)$ or $e \wedge (\mathtt{motor}_1\mathtt{\_speed} = 0)$. ☐

Parallel Composition. We need a construct to represent processes acting *concurrently*. Given P and Q we denote their parallel composition by the process

$$P \parallel Q \tag{4}$$

In one time unit (or interval) processes P and Q operate concurrently, "communicating" via the common store by telling and asking information.

Example 5. Let P be defined as in Example 4 and

$$Q = \begin{array}{l}\mathbf{when}\ \mathtt{motor}_1\mathtt{\_speed} = 0\ \mathbf{do}\ \mathbf{tell}(\mathtt{motor}_2\mathtt{\_speed} = 0)\\ +\\ \mathbf{when}\ \mathtt{motor}_2\mathtt{\_speed} = 0\ \mathbf{do}\ \mathbf{tell}(\mathtt{motor}_1\mathtt{\_speed} = 0).\end{array}$$

Intuitively Q turns off one motor if the other is detected to be off. The parallel composition $P \parallel Q$ in the store $d = (\mathtt{sensor}_2 > 100)$ will, in one time unit, cause the store to become $d \wedge (\mathtt{motor}_1\mathtt{\_speed} = \mathtt{motor}_2\mathtt{\_speed} = 0)$. ☐

Local Behavior. Most process calculi have a construct to restrict the interface through which processes can interact with each other, thus providing for the modeling of *local (or hidden)* behavior. We introduce processes of the form

$$(\mathbf{local}\, x)\, P \tag{5}$$

The process $(\mathbf{local}\, x)\, P$ declares a variable x, private to P. This process behaves like P, except that all the information about x produced by P is hidden from external processes and the information about x produced by other external processes is hidden from P.

Example 6. In modeling RCX or PLC's one uses "global" variables to represent ports (e.g., sensor and motors). One often, however, uses variables which do not represent ports, and thus we may find it convenient to declare such variables as local (or private).

Suppose that R is a given process modeling some controller task. Furthermore, suppose that R uses a variable z, which is set at random, with some unknown distribution, to a value $v \in \{0, 1\}$. Let us define the process

$$P = \; (\sum_{v \in \{0,1\}} \mathbf{when}\ \mathtt{true}\ \mathbf{do}\, \mathbf{tell}(z = v)) \;\; \| \, R$$

to represent the behavior of R under z's random assignment.

We may want to declare z in P to be local since it does not represent an input or output port. Moreover, notice that if we need to run two copies of P, i.e., process $P \parallel P$, a failure may arise as each copy can assign a different value to z. Therefore, the behavior of R under the random assignment to z can be best represented as $P' = (\mathbf{local}\, z)\, P$. In fact, if we run two copies of P', no failure can arise from the random assignment to the z's as they are private to each P'. □

The processes hitherto described generate activity within the current time interval only. We now turn to constructs that can generate activity in future time intervals.

Unit Delays and Time-Outs. As in the Synchronous Languages [3] we have constructs whose actions can delay the execution of processes. These constructs are needed to model time dependency between actions, e.g., actions depending on the absence or presence of preceding actions. Time dependency is an important aspect in the modeling of timed systems.

The unit-delay operators have the form

$$\mathbf{next}\, P \;\; \text{and} \;\; \mathbf{unless}\;\; c \;\; \mathbf{next}\, P \tag{6}$$

The process $\mathbf{next}\, P$ represents the activation of P in the next time interval. The process $\mathbf{unless}\; c\; \mathbf{next}\, P$ is similar, but P will be activated only if c cannot be inferred from the resulting (or final) store d in the current time interval, i.e., $d \not\models c$. The "unless" processes add time-outs to the calculus, i.e., they wait during the current time interval for a piece of information c to be present and if it is not, they trigger activity in the next time interval.

Notice that $\mathbf{unless}\; c\; \mathbf{next}\; P$ is not equivalent to $\mathbf{when}\; \neg c\; \mathbf{do}\; \mathbf{next}\; P$ since $d \not\models c$ does not necessarily imply $d \models \neg c$. Notice that $Q = \mathbf{unless}\;\; \mathtt{false}\;\; \mathbf{next}\, P$ is not the same as $R = \mathbf{next}\, P$ since unlike Q, even if the store contains `false`, R will still activate P in the next time interval (and the store in the next time interval may not contain `false`).

Example 7. Let us consider the following process:

$$P = \mathbf{when}\;\; \mathtt{false}\;\; \mathbf{do}\; \mathbf{next}\; \mathbf{tell}(\mathtt{motor_1\_speed} = \mathtt{motor_2\_speed} = 0).$$

P turns the motors off by decreeing that $\mathtt{motor_1\_speed} = \mathtt{motor_2\_speed} = 0$ in the next time interval if a failure takes place in the current time interval. Similarly, the process

$$\textbf{unless}\ \texttt{false}\ \textbf{next}\,(\textbf{tell}(\texttt{motor}_1\texttt{\_speed} > 0)\ \|\ \textbf{tell}(\texttt{motor}_2\texttt{\_speed} > 0))$$

makes the motors move at some speed in the next time unit, unless a failure takes place in the current time interval. □

Asynchrony. We now introduce a construct that, unlike the previous ones, can describe arbitrary (finite) delays. The importance of this construct is that it allows us to model asynchronous behavior across the time intervals.

We use the operator "$\star$" which corresponds to the unbounded but finite delay operator for synchronous CCS [24]. The process

$$\star\, P \tag{7}$$

represents an arbitrary long but finite delay for the activation of P. Thus, $\star\,\textbf{tell}(c)$ can be viewed as a message c that is eventually delivered but there is no upper bound on the delivery time.

Example 8. Let $S = \star\,\textbf{tell}(\texttt{malfunction}(\texttt{motor}_1\texttt{\_status}))$. The process S can be used to specify that $\texttt{motor}_1$, at some unpredictable point in time, is doomed to malfunction □

Infinite Behavior. Finally, we need a construct to define infinite behavior. We shall use the operator "!" as a delayed version of the replication operator for the $\pi-$calculus [25]. Given a process P, the process

$$!\,P \tag{8}$$

represents $P \parallel (\textbf{next}\, P) \parallel (\textbf{next}\,\textbf{next}\, P) \parallel \cdots \parallel !\, P$, i.e., unboundedly many copies of P, but one at a time. The process $!\,P$ executes P in one time unit and persists in the next time unit.

Example 9. The process R below repeatedly checks the state of $\texttt{motor}_1$. If a malfunction is reported, R tells that $\texttt{motor}_1$ must be turned off.

$$R = !\,\textbf{when}\ \texttt{malfunction}(\texttt{motor}_1\texttt{\_status})\ \textbf{do}\ \textbf{tell}(\texttt{motor}_1\texttt{\_speed} = 0)$$

Thus, $R \parallel S$ with $S = \star\,\textbf{tell}(\texttt{malfunction}(\texttt{motor}_1\texttt{\_status}))$ (Example 8) eventually tells that $\texttt{motor}_1$ is turned off. □

Some Derived Forms

We have informally introduced the basic process constructs of ntcc and illustrated how they can be used to model or specify system behavior. In this section we shall illustrate how they can be used to obtain some convenient derived constructs.

In the following we shall omit "$\textbf{when}\ \texttt{true}\ \textbf{do}$" if no confusion arises. The "blind-choice" process $\sum_{i\in I} \textbf{when}\ \texttt{true}\ \textbf{do}\ P_i$, for example, can be written as $\sum_{i\in I}\ P_i$. We shall use $\prod_{i\in I} P_i$, where I is finite, to denote the parallel composition of all the P_i's. We use $\textbf{next}^n(P)$ as an abbreviation for $\textbf{next}(\textbf{next}(\ldots(\textbf{next}\, P)\ldots))$, where $\textbf{next}$ is repeated n times.

Inactivity. The process doing nothing whatsoever, $\mathbf{skip}$ can be defined as an abbreviation of the empty summation $\sum_{i\in\emptyset} P_i$. This process corresponds to the inactive processes $\mathbf{0}$ of CCS and $STOP$ of CSP. We should expect the behavior of $P \parallel \mathbf{skip}$ to be the same as that of P under any reasonable notion of behavioral equivalence.

Abortion. Another useful construct is the process $\mathbf{abort}$ which is somehow to the opposite extreme of $\mathbf{skip}$. Whilst having $\mathbf{skip}$ in a system causes no change whatsoever, having $\mathbf{abort}$ can make the whole system fail. Hence $\mathbf{abort}$ corresponds to the $CHAOS$ operator in CSP. In Section 5.1 we mentioned that a tell process causes a failure, at the current time interval, if it leaves the store inconsistent. Therefore, we can define $\mathbf{abort}$ as $!\,\mathbf{tell}(\mathtt{false})$, i.e., the process that once activated causes a constant failure. Therefore, any reasonable notion of behavioral equivalence should not distinguish between $P \parallel \mathbf{abort}$ and $\mathbf{abort}$.

Asynchronous Parallel Composition. Notice that in $P \parallel Q$ both P and Q are forced to move in the current time unit, thus our parallel composition can be regarded as being a synchronous operator. There are situations where an asynchronous version of "$\parallel$" is desirable. For example, modeling the interaction of several controllers operating concurrently where some of them could be faster or slower than the others at responding to their environment.

By using the star operator we can define a *(fair) asynchronous* parallel composition $P \mid Q$ as

$$(P \parallel \star\, Q) \;+\; (\star\, P \parallel Q)$$

A move of $P \mid Q$ is either one of P or one of Q (or both). Moreover, both P and Q are eventually executed (i.e. a fair execution of $P \mid Q$). This process corresponds to the asynchronous parallel operator described in [24].

We should expect operator "$|$" to enjoy properties of parallel composition. Namely, we should expect $P \mid Q$ to be the same as $Q \mid P$ and $P \mid (Q \mid R)$ to be the same as $(P \mid Q) \mid R$. Unlike in $P \parallel \mathbf{skip}$, however, in $P \mid \mathbf{skip}$ the execution of P may be arbitrary postponed, therefore we may want to distinguish between $P \mid \mathbf{skip}$ and P. Similarly, unlike in $P \parallel \mathbf{abort}$, in $P \mid \mathbf{abort}$ the execution of $\mathbf{abort}$ may be arbitrarily postponed. In a timed setting we may want to distinguish between a process that aborts right now and one that may do so sometime later after having done some work.

Bounded Eventuality and Invariance. We may want to specify that a certain behavior is exhibited within a certain number of time units, i.e., *bounded eventuality*, or during a certain number of time units, i.e., *bounded invariance*. An example of bounded eventuality is "the light must be switched off within the next ten time units" and an example of bounded invariance is "the motor should not be turned on during the next sixty time units".

The kind of behavior described above can be specified by using the bounded versions of $!\,P$ and $\star\, P$, which can be derived using summation and parallel composition in the obvious way. We define $!_I P$ and $\star_I P$, where I is a closed interval of the natural numbers, as an abbreviation for

$$\prod_{i \in I} \mathbf{next}^i P \text{ and } \sum_{i \in I} \mathbf{next}^i P$$

respectively. Intuitively, $\star_{[m,n]}P$ means that P is eventually active between the next m and $m+n$ time units, while $!_{[m,n]}P$ means that P is always active between the next m and $m+n$ time units.

Nondeterministic Time-Outs. The ntcc calculus generalizes processes of the form $\mathbf{when}\ c\ \mathbf{do}\ P$ by allowing nondeterministic choice over them. It would therefore be natural to do the same with processes of the form $\mathbf{unless}\ c\ \mathbf{next}\ P$. In other words, one may want to have a *nondeterministic time-out operator*

$$\sum_{i \in I} \mathbf{unless}\ c_i\ \mathbf{next}\ P_i$$

which chooses one P_i such that c_i cannot be eventually inferred from the store within the current time unit (if no choice is possible then the summation is precluded from future execution). Notice that this is not the same as having a blind-choice summation of the $\mathbf{unless}\ c_i\ \mathbf{next}\ P_i$ operators. It is not difficult to see, however, that the behavior of such a nondeterministic time-out operator can be described by the ntcc process:.

$$(\mathbf{local}\, I')\,(\prod_{i \in I} (\mathbf{unless}\ c_i\ \mathbf{next}\ \mathbf{tell}(i \in I')) \parallel \mathbf{next} \sum_{i \in I} \mathbf{when}\ i \in I'\ \mathbf{do}\ P_i)$$

where $i \in I'$ holds iff i is in the set I'.

5.2 The Operational Semantics of ntcc

In the previous section we gave an intuitive description of ntcc . In this section we shall make precise such a description. We shall begin by defining the notion of constraint system and the formal syntax of ntcc . We shall then give meaning to the ntcc processes by means of an operational semantics. The semantics, which resembles the reduction semantics of the π-calculus [25], provides *internal* and *external* transitions describing process evolutions. The internal transitions describe evolutions within a time unit and thus they are regarded as being unobservable. In contrast, the external transitions are regarded as being observable as they describe evolution across the time units.

Constraint Systems. For our purposes it will suffice to consider the notion of constraint system based on first-order logic, as was done in [46].

Definition 1 (Constraint System). *A* constraint system (cs) *is a pair (Σ, Δ) where Σ is a signature of function and predicate symbols, and Δ is a decidable theory over Σ (i.e., a decidable set of sentences over Σ with a least one model).*

Given a constraint system (Σ, Δ), let $(\Sigma, \mathcal{V}, \mathcal{S})$ be its underlying first-order language, where $\mathcal{V}$ is a countable set of variables $x, y, \ldots$, and $\mathcal{S}$ is the set of logic symbols $\neg, \wedge, \vee, \Rightarrow, \exists, \forall, \mathtt{true}$ and $\mathtt{false}$. *Constraints* $c, d, \ldots$ are formulae over this first-order language. We say that c *entails* d in Δ, written $c \models d$, iff $c \Rightarrow d$ is true in all models of Δ. The relation $\models$, which is decidable by the definition of Δ, induces an equivalence $\approx$ given by $c \approx d$ iff $c \models d$ and $d \models c$. Henceforth, $\mathcal{C}$ denotes *the set of constraints under consideration* modulo $\approx$ in the underlying cs. Thus, we simply write $c = d$ iff $c \approx d$.

Definition (Processes, *Proc***).** *Processes P, Q, $\ldots \in$ Proc are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by:*

$$\begin{array}{rl} P, Q, \ldots ::= & \mathbf{tell}(c) \ \mid \sum\limits_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \mid P \parallel Q \ \mid (\mathbf{local}\, x)\, P \\ \mid & \mathbf{next}\, P \mid \mathbf{unless}\ c\ \mathbf{next}\, P \ \mid \star\, P \quad \mid !\, P \end{array}$$

Intuitively, $\mathbf{tell}(c)$ adds an item of information c to the store in the current time interval. The *guarded-choice summation* $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$, where I is a finite set of indexes, chooses in the current time interval one of the P_i's whose c_i is entailed by the store. If no choice is possible, the summation is precluded from execution. We write $\mathbf{when}\ c_{i_1}\ \mathbf{do}\ P_{i_1} + \ldots + \mathbf{when}\ c_{i_n}\ \mathbf{do}\ P_{i_n}$ if $I = \{i_1, \ldots, i_n\}$. We omit the "$\sum_{i \in I}$" if $|I| = 1$ and use $\mathbf{skip}$ for $\sum_{i \in \emptyset} P_i$.

The process $P \parallel Q$ represents the *parallel execution* of P and Q. In one time unit P and Q operate concurrently, communicating through the store.

The process $(\mathbf{local}\, x)\, P$ behaves like P, except that all the information about x produced by P can only be seen by P and the information about x produced by other processes cannot be seen by P. In other words $(\mathbf{local}\, x)\, P$ declares an x *local* to P, and thus we say that it *binds* x in P. The *bound variables* $bv(Q)$ (*free variables* $fv(Q)$) are those with a bound (a not bound) occurrence in Q.

The *unit-delay* process $\mathbf{next}\, P$ executes P in the next time interval. The *time-out* $\mathbf{unless}\ c\ \mathbf{next}\, P$ is also a unit-delay, but P will be executed only if c cannot eventually be entailed by the store during the current time interval. Note that $\mathbf{next}\, P$ is not the same as $\mathbf{unless}\ \mathtt{false}\ \mathbf{next}\, P$ since an inconsistent store entails $\mathtt{false}$. We use $\mathbf{next}^n P$ for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\, P)\ldots))$, where $\mathbf{next}$ is repeated n times.

The operator "$\star$" represents an *arbitrary (or unknown) but finite delay* (as "ϵ" in SCCS [24]) and allows asynchronous behavior across the time intervals. Intuitively, $\star\, P$ means $P + \mathbf{next}\, P + \mathbf{next}^2 P + \ldots$, i.e., an unbounded finite delay of P.

The *replication* operator "!" is a delayed version of that of the π-calculus [25]: $!\, P$ means $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^2 P \parallel \ldots \parallel !\, P$, i.e., unboundedly many copies of P but one at a time.

A Transition Semantics.

The structural operational semantics (SOS) of ntcc considers *transitions* between process-store *configurations* of the form $\langle P, c\rangle$ with stores represented as constraints and processes quotiented by $\equiv$ below. Intuitively $\equiv$ describes irrelevant syntactic aspects of processes.

Definition 2 (Structural Congruence). *Let $\equiv$ be the smallest congruence satisfying: (1) $P \parallel \mathbf{skip} \equiv P$, (2) $P \parallel Q \equiv Q \parallel P$, and (3) $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$. Extend $\equiv$ to configurations by decreeing that $\langle P, c\rangle \equiv \langle Q, c\rangle$ iff $P \equiv Q$.*

Following standard lines, we extend the syntax with a construct $\mathbf{local}\,(x, d)\,\mathbf{in}\,P$, to represent the evolution of a process of the form $\mathbf{local}\,x\,\mathbf{in}\,Q$, where d is the local information (or store) produced during this evolution. Initially d is "empty", so we regard $\mathbf{local}\,x\,\mathbf{in}\,P$ as $\mathbf{local}\,(x, \mathtt{true})\,\mathbf{in}\,P$.

The transitions of the SOS are given by the relations $\longrightarrow$ and $\Longrightarrow$ defined in Table 1. The *internal* transition $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ should be read as "P with store d reduces, in one internal step, to P' with store d' ". The *observable transition* $P \xRightarrow{(c,d)} R$ should be read as "P on input c, reduces in one *time unit* to R and outputs d". The observable transitions are obtained from terminating sequences of internal transitions.

$$\text{TELL}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle} \qquad \text{SUM}\ \frac{d \models c_j \;\; j \in I}{\langle \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i, d\rangle \longrightarrow \langle P_j, d\rangle}$$

$$\text{PAR}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle} \qquad \text{LOC}\ \frac{\langle P, c \wedge \exists_x d\rangle \longrightarrow \langle P', c'\rangle}{\langle (\mathbf{local}\,x, c)\,P, d\rangle \longrightarrow \langle (\mathbf{local}\,x, c')\,P', d \wedge \exists_x c'\rangle}$$

$$\text{UNL}\ \frac{}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}\ \text{if } d \models c$$

$$\text{REP}\ \frac{}{\langle !\,P, d\rangle \longrightarrow \langle P \parallel \mathbf{next}\,!\,P, d\rangle} \qquad \text{STAR}\ \frac{}{\langle \star\,P, d\rangle \longrightarrow \langle \mathbf{next}^{\,n}\,P, d\rangle}\ \text{if } n \geq 0$$

$$\text{STR}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'}\ \text{if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\text{OBS}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} R}\ \text{if } R \equiv F(Q)$$

Table1. Rules for internal reduction $\longrightarrow$ (upper part) and observable reduction $\Longrightarrow$ (lower part). $\gamma \not\longrightarrow$ in OBS holds iff for no γ', $\gamma \longrightarrow \gamma'$. $\equiv$ and F are given in Definitions 2 and 3.

We shall only describe some of the rules of in Table 1 due to space restrictions (see [28] for further details). As clarified below, the seemingly missing cases for "next" and "unless" processes are given by OBS. The rule STAR specifies an arbitrary delay of P. REP says that $!P$ creates a copy of P and then persists in the next time unit. We shall dwell a little upon the description of Rule LOC as it may seem somewhat complex. Let us consider the process

$$Q = (\mathbf{local}\,x, c)\,P$$

in Rule LOC. The global store is d and the local store is c. We distinguish between the *external* (corresponding to Q) and the *internal* point of view (corresponding to P). From the internal point of view, the information about x, possibly appearing in the "global" store d, cannot be observed. Thus, before reducing P we should first hide the information about x that Q may have in d. We can do this by existentially quantifying x in d. Similarly, from the external point of view, the observable information about x that the reduction of internal agent P may produce (i.e., c') cannot be observed. Thus we hide it by existentially quantifying x in c' before adding it to the global store corresponding to the evolution of Q. Additionally, we should make c' the new private store of the evolution of the internal process for its future reductions.

Rule OBS says that an observable transition from P labeled with (c, d) is obtained from a terminating sequence of internal transitions from $\langle P, c\rangle$ to a $\langle Q, d\rangle$. The process R to be executed in the next time interval is equivalent to $F(Q)$ (the "future" of Q). $F(Q)$ is obtained by removing from Q summations that did not trigger activity and any local information which has been stored in Q, and by "unfolding" the sub-terms within "next" and "unless" expressions.

Definition 3 (Future Function). *Let $F : Proc \rightharpoonup Proc$ be defined by*

$$F(Q) = \begin{cases} \mathbf{skip} & \textit{if } Q = \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ Q_i \\ F(Q_1) \parallel F(Q_2) & \textit{if } Q = Q_1 \parallel Q_2 \\ (\mathbf{local}\, x)\, F(R) & \textit{if } Q = (\mathbf{local}\, x, c)\, R \\ R & \textit{if } Q = \mathbf{next}\, R \textit{ or } Q = \mathbf{unless}\ c\ \mathbf{next}\, R \end{cases}$$

*Remark 1. F need no to be total since whenever we need to apply F to a Q (*OBS *in Table 1), every $\mathbf{tell}(c)$, $\mathbf{abort}$, $\star R$ and $!\, R$ in Q will occur within a "next" or "unless" expression.*

5.3 Observable Behavior

In this section we recall some notions introduced in [29] of what an observer can see from a process behavior . We shall refer to such notions as *process observations*. We assume that what happens within a time unit cannot be directly observed, and thus we abstract from internal transitions. The ntcc calculus makes it easy to focus on the observation of input-output events in which a given process engages and the order in which they occur.

Notation 1 *Throughout this paper $\mathcal{C}^\omega$ denotes the set of infinite (or ω) sequences of constraints in the underlying set of constraints $\mathcal{C}$. We use $\alpha, \alpha', \ldots$ to range over $\mathcal{C}^\omega$.*

Let $\alpha = c_1.c_2.\ldots$ and $\alpha' = c'_1.c'_2.\ldots$. Suppose that P exhibits the following infinite sequence of observable transitions (or *run*): $P = P_1 \xRightarrow{(c_1, c'_1)} P_2 \xRightarrow{(c_2, c'_2)} \ldots$. Given this run of P, we shall use the notation $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$.

IO and Output Behavior. Observe the above run of P. At the time unit i, the environment *inputs* c_i to P_i which then responds with an output c'_i. As observers, we can see that on α, P responds with α'. We refer to the set of all (α, α') such that $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$ as the *input-output (io) behavior* of P. Alternatively, if $\alpha = \mathtt{true}^\omega$, we interpret the run as an interaction among the parallel components in P *without the influence of any (external) environment*; as observers what we see is that P produces α on its own. We refer to the set of all α' such that $P \xRightarrow{(\mathtt{true}^\omega,\alpha')}{}^{\omega}$ as the *output* behavior of P.

Quiescent Sequences and SP. Another observation we can make of a process is its quiescent input sequences. These are sequences on input of which P can run without adding any information; we observe whether $\alpha = \alpha'$ whenever $P \xrightarrow{(\alpha,\alpha')}{}^{\omega}$.

In [28] it is shown that the set of quiescent sequences of a given P can be alternatively characterized as *the set of infinite sequences that P can possibly output under arbitrary environments*; the strongest postcondition (sp) of P.

The following definition states the various notions of observable behavior mentioned above.

Definition 4 (Observable Behavior). *The behavioral observations that can be made of a process are:*

1. *The* input-output (or stimulus-response) behavior *of P, written, $io(P)$, defined as*
$$io(P) = \{(\alpha, \alpha') \mid P \xRightarrow{(\alpha,\alpha')}{}^{\omega}\}.$$
2. *The* (default) output behavior *of P, written $o(P)$, defined as*
$$o(P) = \{\alpha' \mid P \xRightarrow{(\mathtt{true}^{\omega},\alpha')}{}^{\omega}\}.$$
3. *The* strongest postcondition *behavior of P, written $sp(P)$, defined as*
$$sp(P) = \{\alpha \mid P \xRightarrow{(\alpha',\alpha)}{}^{\omega} \textit{ for some } \alpha'\}.$$

The following are the obvious equivalences and congruences induced by our behavioral observations. (Recall the notion of congruence given in Section 2.)

Definition 5 (Behavioral Equivalences). *Let $l \in \{io, o, sp\}$. Define $P \sim_l Q$ iff $l(P) = l(Q)$. Furthermore, let $\approx_l$ the congruence induced by $\sim_l$, i.e., $P \approx_l Q$ iff $C[P] \sim_l C[Q]$ for every process context C.*

We shall refer to equivalences defined above as observational equivalences as they identify processes whose internal behavior may differ widely (e.g. in the number of internal actions). Such an abstraction from internal behavior is essential in the theory of several process calculi; most notably in weak bisimilarity for CCS [23].

Example 10. Let a, b, c, d and e be mutually exclusive constraints. Consider the processes P and Q below:

$$\underbrace{\mathbf{when}\ a\ \mathbf{do\ next}\ \begin{array}{l}\mathbf{when}\ b\ \mathbf{do\ next\ tell}(d)\\ +\\ \mathbf{when}\ c\ \mathbf{do\ next\ tell}(e)\end{array}}_{P},\ \underbrace{\begin{array}{l}\mathbf{when}\ a\ \mathbf{do\ next\ when}\ b\ \mathbf{do\ next\ tell}(d)\\ +\\ \mathbf{when}\ a\ \mathbf{do\ next\ when}\ c\ \mathbf{do\ next\ tell}(e)\end{array}}_{Q}$$

The reader may care to verify that $P \sim_o Q$ since $o(P) = o(Q) = \{\mathtt{true}^{\omega}\}$. However, $P \not\sim_{io} Q$ nor $P \not\sim_{sp} Q$ since if $\alpha = a.c.\mathtt{true}^{\omega}$ then $(\alpha, \alpha) \in io(Q)$ and $\alpha \in sp(Q)$ but $(\alpha, \alpha) \notin io(P)$ and $\alpha \notin sp(P)$. □

Congruence and Decidability Issues. Several typical questions about these equivalence may then arise. For example, one may wonder which of them coincides with their corresponding induced congruences and whether there are interesting relationships between them.

In [28] it is proven that none of the equivalences is a congruence. However, $\sim_{sp}$ is a congruence in a restricted sense; Namely, if we confine our attention to the so-called *locally-independent* fragment of the calculus. This fragment only forbids non-unary summations (and "unless" processes) whose guards depend on local variables.

Definition 6 (Locally-Independent Processes). *P is* locally-independent *iff for every* $\mathbf{unless}\ c\ \mathbf{next}\ Q$ *and* $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ Q_i$ *($|I| \geq 2$) in P, neither c nor the c_i's contain variables in $bv(P)$ (i.e., the bound variables of P).*

The locally-independent fragment is indeed very expressive. Every summation process whose guards are either all equivalent or mutually exclusive can be encoded in this fragment [50]. Moreover, the applicability of this fragment is witnessed by the fact all the ntcc application the author is aware of [28, 29, 50] can be model as locally-independent processes. Also, the (parameterless-recursion) tcc model can be expressed in this fragment as, from the expressiveness point of view, the local operator is redundant in tcc with parameterless-recursion [27]. Furthermore, the fragment allows us to express infinite-state processes [51](i.e., processes that can evolve into infinitely many other processes). Hence, it is rather surprising that $\sim_{sp}$ is decidable for the local-independent fragment as recently proved in [51].

As for the input-output and output equivalences, in [50] it is shown how to characterize their induced congruence in a satisfactory way. Namely, $P \approx_o Q$ iff $U(P,Q)[P] \sim_o U(P,Q)[Q]$ where $U(P,Q)$ is a context which, given P and Q, can be effectively constructed. Also [50] shows that although $\sim_{io}$ is stronger than $\sim_o$, their induced congruences match. Perhaps the most significant theoretical value of these two results is its computational consequence:Both input-output and output congruence are decidable if output equivalence is decidable.

In fact, output equivalence is decidable for processes with a restricted form of nondeterminism [29]. Namely, $\star$-free processes in which local operators do not exhibit nondeterminism. This also represent a significant fragment of the calculus including all the application examples in [28, 29, 50] and the parameterless-recursion fragment of the tcc model. It then follows, from the previously mentioned results, that $\approx_{io}$ and $\approx_o$ are also decidable if we restrict our attention to these restricted nondeterministic processes.

5.4 Denotational Semantics

In the previous section we introduced the notion of strongest-postcondition of ntcc processes in operational terms. Let us now show the abstract denotational model of this notion first presented in [30]. Such a model is of great help when arguing about the strongest-postcondition of ntcc processes.

The denotational semantics is defined as a function $[\![\cdot]\!]$ which associates to each process a set of infinite constraint sequences, namely $[\![\cdot]\!] : \mathit{Proc} \to \mathcal{P}(\mathcal{C}^\omega)$. The definition of this function is given in Table 2. Intuitively, $[\![P]\!]$ is meant to capture the set of all

DTELL:	$[\![\mathbf{tell}(c)]\!] \;=\; \{d.\alpha \mid d \models c\}$
DSUM:	$[\![\ \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i\ ]\!] \;=\; \bigcup\limits_{i\in I} \{d.\alpha \mid d \models c_i \text{ and } d.\alpha \in [\![P_i]\!]\} \cup \bigcap_{i\in I}\{d.\alpha \mid d \not\models c_i\}$
DPAR:	$[\![P \parallel Q]\!] \;=\; [\![P]\!] \cap [\![Q]\!]$
DLOC:	$[\![(\mathbf{local}\, x)\ P]\!] \;=\; \{\alpha \mid \text{there exists } \alpha' \in [\![P]\!] \text{ s.t. } \exists_x \alpha' = \exists_x \alpha\}$
DNEXT:	$[\![\mathbf{next}\, P]\!] \;=\; \{d.\alpha \mid \alpha \in [\![P]\!]\}$
DUNL:	$[\![\mathbf{unless}\ c\ \mathbf{next}\, P]\!] \;=\; \{d.\alpha \mid d \models c\} \cup \{d.\alpha \mid d \not\models c \text{ and } \alpha \in [\![P]\!]\}$
DREP:	$[\![!\, P]\!] \;=\; \{\alpha \mid \text{ for all } \beta, \alpha' \text{ s.t. } \alpha = \beta.\alpha', \text{ we have } \alpha' \in [\![P]\!]\}$
DSTAR:	$[\![\star\, P]\!] \;=\; \{\beta.\alpha \mid \alpha \in [\![P]\!]\}$

Table2. Denotational semantics of ntcc . Symbols α and α' range over the set of infinite sequences of constraints $\mathcal{C}^\omega$; β ranges over the set of finite sequences of constraints $\mathcal{C}^*$. Notation $\exists_x \alpha$ denotes the sequence resulting by applying $\exists_x$ to each constraint in α.

sequences P can possibly output. For instance, the sequences that $\mathbf{tell}(c)$ can output are those whose first element is stronger than c (see DTELL, Table 2). Process $\mathbf{next}\ P$ has not influence in the first element of a sequence, thus $d.\alpha$ can be output by it iff α is can be output by P (see DNEXT, Table 2). A sequence can be output by $!\,P$ iff every suffix of it can be output by P (see DREP, Table 2). The other cases can be explained analogously.

From [7], however, we know that there cannot be a $f : \mathit{Proc} \to \mathcal{P}(\mathcal{C}^\omega)$, compositionally defined, such that $f(P) = \mathit{sp}(P)$ for all P. Nevertheless, as stated in the theorem below, Palamidessi et al [30] showed that that sp denotational semantics matches its operational counter-part for the local independent-fragment, which as argued before is very expressive.

Theorem 1 (Full Abstraction). *For every* ntcc *process P, $\mathit{sp}(P) \subseteq [\![P]\!]$ and if P is locally-independent then $[\![P]\!] \subseteq \mathit{sp}(P)$.*

The full-abstraction result has an important theoretical value; i.e., for a significant fragment of the calculus we can abstract away from operational details by working with $[\![P]\!]$ rather than $\mathit{sp}(P)$. In fact, the congruence result for $\sim_{sp}$ mentioned in the previous section is a corollary of the above theorem.

5.5 LTL Specification and Verification

Processes in ntcc can be used to specify properties of timed systems, e.g., that an action must happen within some finite but not fixed amount of time. It is often convenient, however, to express specifications in another formalism, in particular a logical one. In this section we present the ntcc logic first introduced in [30]. We start by defining a linear-time temporal logic (LTL) to expresses temporal properties over infinite sequences of constraints. We then define what it means for a process to satisfy a specification given as a formula in this logic. We shall then say that P satisfies a specification F iff every infinite sequence P can possibly output (on inputs from arbitrary environments) satisfies F, i.e., iff the strongest-postcondition of P implies F. Finally, we present an inference system aimed at proving whether a process fulfills a given specification.

A Temporal Logic. The `ntcc` LTL expresses properties over sequences of constraints and we shall refer to it as **CLTL**. We begin by giving the syntax of LTL formulae and then interpret them with the **CLTL** semantics.

Definition 7 (LTL Syntax). *The formulae $F, G, \ldots \in \mathcal{F}$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by:*

$$F, G, \ldots := c \mid \dot{\mathtt{true}} \mid \dot{\mathtt{false}} \mid F \dot{\wedge} G \mid F \dot{\vee} G \mid \dot{\neg} F \mid \dot{\exists}_x F \mid \circ F \mid \Box F \mid \Diamond F$$

The constraint c (i.e., a first-order formula in the cs) represents a *state formula*. The dotted symbols represent the usual (temporal) boolean and existential operators. As clarified later, the dotted notation is needed as in **CLTL** these operators do not always coincide with those in the cs. The symbols $\circ$, $\Box$, and $\Diamond$ denote the LTL modalities *next*, *always* and *eventually*. We use $F \dot{\Rightarrow} G$ for $\dot{\neg} F \dot{\vee} G$. Below we give the formulae a **CLTL** semantics. First, we need some notation and the notion of *x-variant*. Intuitively, d is an x-variant of c iff they are the same except for the information about x.

Notation 2 *Given a sequence $\alpha = c_1.c_2 \ldots$, we use $\exists_x \alpha$ to denote the sequence $\exists_x c_1 \exists_x c_2 \ldots$. We shall use $\alpha(i)$ to denote the i-th element of α.*

Definition 8 (x-variant). *A constraint d is an x-*variant *of c iff $\exists_x c = \exists_x d$. Similarly α' is an x-*variant *of α iff $\exists_x \alpha = \exists_x \alpha'$.*

Definition 9 (CLTL Semantics). *We say that the infinite sequence α satisfies (or that it is a model of) F in* **CLTL**, *written $\alpha \models_{\mathrm{CLTL}} F$, iff $\langle \alpha, 1\rangle \models_{\mathrm{CLTL}} F$, where:*

$$\begin{array}{lll}
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} \dot{\mathtt{true}} & & \langle \alpha, i\rangle \not\models_{\mathrm{CLTL}} \dot{\mathtt{false}} \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} c & \textit{iff} & \alpha(i) \models c \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} \dot{\neg} F & \textit{iff} & \langle \alpha, i\rangle \not\models_{\mathrm{CLTL}} F \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} F \dot{\wedge} G & \textit{iff} & \langle \alpha, i\rangle \models_{\mathrm{CLTL}} F \textit{ and } \langle \alpha, i\rangle \models_{\mathrm{CLTL}} G \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} F \dot{\vee} G & \textit{iff} & \langle \alpha, i\rangle \models_{\mathrm{CLTL}} F \textit{ or } \langle \alpha, i\rangle \models_{\mathrm{CLTL}} G \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} \circ F & \textit{iff} & \langle \alpha, i+1\rangle \models_{\mathrm{CLTL}} F \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} \Box F & \textit{iff} & \textit{for all } j \geq i \;\; \langle \alpha, j\rangle \models_{\mathrm{CLTL}} F \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} \Diamond F & \textit{iff} & \textit{there is a } j \geq i \textit{ such that } \langle \alpha, j\rangle \models_{\mathrm{CLTL}} F \\
\langle \alpha, i\rangle \models_{\mathrm{CLTL}} \dot{\exists}_x F & \textit{iff} & \textit{there is an } x\textit{-variant } \alpha' \textit{ of } \alpha \textit{ such that } \langle \alpha', i\rangle \models_{\mathrm{CLTL}} F.
\end{array}$$

Define $[\![F]\!]=\{\alpha \mid \alpha \models_{\mathrm{CLTL}} F\}$. *$F$ is* **CLTL** valid *iff $[\![F]\!] = \mathcal{C}^\omega$, and* **CLTL** satisfiable *iff $[\![F]\!] \neq \emptyset$.*

Let us discuss a little about the difference between the boolean operators in the constraint system and the temporal ones to justify our dotted notation. A state formula c is satisfied only by those $e.\alpha'$ such that $e \models c$. So, the state formula `false` has at least one sequence that satisfies it; e.g. $\mathtt{false}^\omega$. On the contrary the temporal formula $\dot{\mathtt{false}}$ has no models whatsoever. Similarly, $c \dot{\vee} d$ is satisfied by those $e.\alpha'$ such that either $e \models c$ or $e \models d$ holds. Thus, in general $[\![c \dot{\vee} d]\!] \neq [\![c \vee d]\!]$. The same holds true for $\neg c$ and $\dot{\neg} c$.

$$\text{LTELL:}\quad \mathbf{tell}(c) \vdash c \qquad\qquad \text{LPAR:}\ \frac{P \vdash F \quad Q \vdash G}{P \parallel Q \vdash F \dot{\wedge} G}$$

$$\text{LSUM:}\ \frac{\forall i \in I \quad P_i \vdash F_i}{\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \vdash \dot{\bigvee}_{i \in I} (c_i \dot{\wedge} F_i) \dot{\vee} \dot{\bigwedge}_{i \in I} \dot{\neg} c_i} \qquad \text{LLOC:}\ \frac{P \vdash F}{(\mathbf{local}\ x)\ P \vdash \dot{\exists}_x F}$$

$$\text{LNEXT:}\ \frac{P \vdash F}{\mathbf{next}\ P \vdash \circ F} \qquad\qquad \text{LUNL:}\ \frac{P \vdash F}{\mathbf{unless}\ c\ \mathbf{next}\ P \vdash c \dot{\vee} \circ F}$$

$$\text{LREP:}\ \frac{P \vdash F}{!\,P \vdash \square F} \qquad\qquad \text{LSTAR:}\ \frac{P \vdash F}{\star\, P \vdash \diamond F}$$

$$\text{LCONS:}\ \frac{P \vdash F}{P \vdash G} \quad \text{if } F \Rightarrow G$$

Table3. A proof system for linear-temporal properties of `ntcc` processes

Example 11. Let $e = c \vee d$ with $c = (x = 42)$ and $d = (x \neq 42)$. One can verify that $\mathcal{C}^\omega = [\![c \vee d]\!] \ni e^\omega \notin [\![c \dot{\vee} d]\!]$ and also that $[\![\neg c]\!] \ni \mathtt{false}^\omega \notin [\![\dot{\neg} c]\!]$. $\square$

From the above example, one may be tempted to think of **CLTL** as being intuitionistic. Notice, however, that statements like $\dot{\neg} F \dot{\vee} F$ and $\dot{\neg}\dot{\neg} F \Rightarrow F$ are **CLTL** valid.

Process Verification. Intuitively, $P \models_{\mathrm{CLTL}} F$ iff every sequence that P can possibly output, on inputs from arbitrary environments, satisfies F. In other words if every sequence in the strongest-postcondition of P is a model of A..

Definition 10 (Verification). P satisfies F, *written* $P \models_{\mathrm{CLTL}} F$, *iff* $sp(P) \subseteq [\![F]\!]$.

So, for instance, $\star\,\mathbf{tell}(c) \models_{\mathrm{CLTL}} \diamond c$ as in every sequence output by $\star\,\mathbf{tell}(c)$ there must be an e entailing c. Also $P = \mathbf{tell}(c) + \mathbf{tell}(d) \models_{\mathrm{CLTL}} c \vee d$ and $P \models_{\mathrm{CLTL}} c \dot{\vee} d$ as every e output by P entails either c or d. Notice, however, that $Q = \mathbf{tell}(c \vee d) \models_{\mathrm{CLTL}} c \vee d$ but $Q \not\models_{\mathrm{CLTL}} (c \dot{\vee} d)$ in general, since Q can output an e which certainly entails $c \vee d$ and still entails neither c nor d - take e, c and d as in Example 11. Therefore, $c \dot{\vee} d$ distinguishes P from Q. The reader may now see why we wish to distinguish $c \dot{\vee} d$ from $c \vee d$.

Proof System for Verification. In order to reason about statements of the form $P \models_{\mathrm{CLTL}} F$, [30] proposes a *proof (or inference) system* for assertions of the form $P \vdash F$. Intuitively, we want $P \vdash F$ to be the "counterpart" of $P \models F$ in the inference system, namely $P \vdash F$ should approximate $P \models_{\mathrm{CLTL}} F$ as closely as possible (ideally, they should be equivalent). The system is presented in Table 3.

Definition 11 **($P \vdash F$).** *We say that $P \vdash F$ iff the assertion $P \vdash F$ has a proof in the system in Table 3.*

Inference Rules. Let us now describe some of the inference rules of the proof system. The inference rule for the tell operator is given by

$$\text{LTELL:}\ \ \mathbf{tell}(c)\ \vdash\ c$$

Rule LTELL gives a proof saying that every output of $\mathbf{tell}(c)$ on inputs of arbitrary environments should definitely satisfy the atomic proposition c, i.e., $\mathbf{tell}(c) \models_{\mathrm{CLTL}} c$.

Consider now the rule for the choice operator:

$$\text{LSUM:}\ \ \frac{\forall i \in I \quad P_i\ \vdash\ F_i}{\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i\ \vdash\ \dot{\bigvee}_{i\in I}(c_i \mathbin{\dot\wedge} F_i) \mathbin{\dot\vee} \dot{\bigwedge}_{i\in I}\ \dot\neg\, c_i}$$

Rule LSUM can be explained as follows. Suppose that for $P = \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ we are given a proof that each P_i satisfies F_i. Then we certainly have a proof saying that every output of P on arbitrary inputs should satisfy either: (a) some of the guards c_i and their corresponding F_i (i.e., $\dot{\bigvee}_{i\in I}(c_i \mathbin{\dot\wedge} F_i)$), or (b) none of the guards (i.e., $\dot{\bigwedge}_{i\in I}\ \dot\neg\, c_i$).

The inference rule for parallel composition is defined as

$$\text{LPAR:}\ \ \frac{P\ \vdash\ F \qquad Q\ \vdash\ G}{P \parallel Q\ \vdash\ F \mathbin{\dot\wedge} G}$$

The soundness of this rule can be justified as follows. Assume that each output of P, under the influence of arbitrary environments, satisfies F. Assume the same about Q and G. In $P \parallel Q$, the process Q can be thought as one of those arbitrary environment under which P satisfies F. Then $P \parallel Q$ must satisfy F. Similarly, P can be one of those arbitrary environment under which Q satisfies G. Hence, $P \parallel Q$ must satisfy G as well. We therefore have grounds to conclude that $P \parallel Q$ satisfies $F \mathbin{\dot\wedge} G$.

The inference rule for the local operator is

$$\text{LLOC:}\ \ \frac{P\ \vdash\ F}{(\mathbf{local}\, x)\, P\ \vdash\ \dot\exists_x F}$$

The intuition is that since the outputs of $(\mathbf{local}\, x)\, P$ are outputs of P with x hidden then if P satisfies F, $(\mathbf{local}\, x)\, P$ should satisfy F with x hidden, i.e., $\dot\exists_x F$.

The following are the inference rules for the temporal `ntcc` constructs:

$$\text{LNEXT:}\ \ \frac{P\ \vdash\ F}{\mathbf{next}\, P\ \vdash\ \circ F} \qquad \text{LUNL:}\ \ \frac{P\ \vdash\ F}{\mathbf{unless}\ c\ \mathbf{next}\ P\ \vdash\ c \mathbin{\dot\vee} \circ F}$$

$$\text{LREP:}\ \ \frac{P\ \vdash\ F}{!\, P\ \vdash\ \Box F} \qquad \text{LSTAR:}\ \ \frac{P\ \vdash\ F}{\star\, P\ \vdash\ \Diamond F}$$

Assume that P satisfies F. Rule LNEXT says that if P is executed next, then in the next time unit it will also satisfy F. Hence, $\mathbf{next}\, P$ satisfies $\circ F$. Rule LUNL is similar, except that P can also be precluded from execution if some environment provides c. Thus $\mathbf{unless}\ c\ \mathbf{next}\ P$ satisfies either c or $\circ F$. Rule LREP says that if P is executed in each time interval, then F is always satisfied by P. Therefore, $!\,P$ satisfies $\Box F$. Rule LSTAR says that if P is executed in some time interval, then in that time interval P satisfies F. Therefore, $\star\, P$ satisfies $\Diamond F$.

Finally, we have a rule that allows reasoning about temporal formulae to be incorporated in proofs about processes satisfying specifications:

$$\text{LCONS:}\ \frac{P\ \vdash\ F}{P\ \vdash\ G}\qquad \text{if } F \Rightarrow G$$

Rule LCONS simply says that if P satisfies a specification F then it also satisfies any weaker specification G.

Notice that the inference rules reveal a pleasant correspondence between `ntcc` operators and the logic operators. For example, parallel composition and locality corresponds to conjunction and existential quantification. The choice operator corresponds to some special kind of conjunction. The next, replication and star operators correspond to the next, always, and eventuality temporal operator.

The Proof System at Work. Let us now give a simple example illustrating a proof in inference system.

Example 12. Recall Example 9. We have a process R which was repeatedly checking the state of $\texttt{motor}_1$. If a malfunction is reported, R would tell that $\texttt{motor}_1$ must be turned off. We also have a process S stating that motor $\texttt{motor}_1$ is doomed to malfunction. Let $R\ =!\,\mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e)$ and $S\ =\ \star\,\mathbf{tell}(c)$ with the constraints $c = \texttt{malfunction}(\texttt{motor}_1\texttt{\_status})$ and $e = (\texttt{motor}_1\texttt{\_speed} = 0)$. We want to provide a proof of the assertion: $R \parallel S\ \vdash\ \Diamond\, e$. Intuitively, this means that the parallel execution of R and S satisfies the specification stating that $\texttt{motor}_1$ is eventually turned off. The following is a derivation of the above assertion.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) \vdash (c \mathbin{\dot\wedge} e) \mathbin{\dot\vee} \dot\neg\, c}\ \text{LSUM}}{\mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) \vdash c \Rightarrow e}\ \text{LCONS}}{R \vdash \Box\,(c \Rightarrow e)}\ \text{LREP} \qquad \dfrac{\dfrac{}{\mathbf{tell}(c) \vdash c}\ \text{LTELL}}{S \vdash \Diamond\, c}\ \text{LSTAR}}{\dfrac{R \parallel S \vdash \Box\,(c \Rightarrow e) \mathbin{\dot\wedge} \Diamond c}{R \parallel S \vdash \Diamond\, e}\ \text{LCONS}}\ \text{LPAR}$$

More complex examples of the use of the proof system for proving the satisfaction of processes specification can be found in [28]—in particular for proving properties of mutable data structures. □

Let us now state how close the relation $\vdash$ to the verification relation $\models_{\text{CLTL}}$.

Theorem 2 (Relative Completeness). *Suppose that P is a locally-independent process. Then $P \vdash F$ iff $P \models_{\text{CLTL}} F$.*

The reason why the above result is called "relative completeness" is because we need to determine the validity of the temporal implication in the rule LCONS. This means that our proof system is complete, if we are equipped with an oracle that is guaranteed to provide a proof or a confirmation of each valid temporal implication. Because of the validity issues above mentioned, one may wonder about decidability of the validity problem for our temporal logic. We look at these issues next.

Decidability Results. In [51] it is shown that the verification problem (i.e., given P and F whether $P \models_{\mathrm{CLTL}} F$) is decidable for the locally independent fragment and negation-free **CLTL** formulae. A noteworthy aspect of this result is that, as mentioned before, the ntcc fragment above admits infinite-state processes. Another interesting aspect is that **CLTL** is first-order. Most first-order LTL's in computer science are not recursively axiomatizable let alone decidable [1].

Furthermore, [51] proves the decidability of the validity problem for implication of negation-free **CLTL** formulae. This is done by appealing to the close connection between ntcc processes and LTL formulae to reduce the validity of implication to the verification problem. More precisely, it is shown that given two negation-free formulae F and G one can construct a process P_F such that $sp(P_F) = [\![F]\!]$ and then it follows that $P_F \models_{\mathrm{CLTL}} G$ iff $F \Rightarrow G$. As a corollary of this result, we obtain the decidability of *satisfiability* for the negation-free first-order fragment of **CLTL** —recall G is satisfiable iff $G \Rightarrow \mathtt{false}$ is not valid.

A theoretical application of the theory of ntcc is presented in [51] by stating a new positive decidability result for a first-order fragment of Pnueli's first-order **LTL** [21]. The result is obtained from a reduction to **CLTL** satisfiability and thus it also contributes to the understanding of the relationship between (timed) ccp and (temporal) classic logic.

6 Concluding Remarks and Related Work

There are several developments of timed ccp and, due to space restriction, it would be difficult to do justice to them all. I shall indicate a few which appear to me to be central.

Related Work. Saraswat el al were the first proposing a denotational semantics and proof system for timed ccp in the context of tcc [39]. The denotational semantics is fully abstract and it was later generalized by Palemidessi et al for the ntcc case. The proof system of [39] is based on an intuitionistic logic enriched with a next operator—the logic for the ntcc case is classic. The system is complete for hiding-free and finite processes.

Gabrielli et al also provided a fully-abstract denotational semantics [5] and proof system [4] for the tccp model (see Section 4). The underlying second-order linear temporal logic in [4] can be used for describing input-output behavior. In contrast, the ntcc logic can only be used for the strongest-postcondition, but also it is semantically simpler and defined as the standard first-order linear-temporal logic of [21].

The decidability results for the ntcc equivalences here presented are based on reductions from ntcc processes into finite-state automata [29, 27, 51]. The work in [42] also shows how to compile tcc into finite-state machines. Rather than a direct way of verifying process equivalences, such machines provide an execution model of tcc.

Nielsen et al [27] compared, relatively to the notion of input-output behavior, the expressive power of various tcc variants differing in their way of expressing infinite behavior. It is shown that: (1) recursive procedures with parameters can be encoded into parameterless recursive procedures with dynamic scoping, and vice-versa. (2) replication can be encoded into parameterless recursive procedures with static scoping, and vice-versa. (3) the languages from (1) are strictly more expressive than the languages from (2). Furthermore, it is shown that behavioral equivalence is undecidable for the languages from (1), but decidable for the languages from (2). The undecidability result holds even if the process variables take values from a fixed finite domain.

Also Tini [48] explores the expressiveness of tcc languages, but focusing on the capability of tcc to encode synchronous languages. In particular, Tini shows that Argos [22] and a version of Lustre restricted to finite domains [16] can be encoded in tcc.

In the context of tcc, Tini [49] introduced a notion of bisimilarity, an elemental process equivalence in concurrency, with a complete and elegant axiomatization for the hiding-free fragment of tcc. The notion of bisimilarity has also been introduced for `ntcc` by the present author in his PhD thesis [50].

On the practical side, Saraswat el al introduced Timed Gentzen [40], a particular tcc-based programming language for reactive-systems implemented in PROLOG. More recently, Saraswat el al released jcc [43], an integration of timed (default) ccp into the popular JAVA programming language. Rueda et al [37] demonstrated that essential ideas of computer generated music composition can be elegantly represented in `ntcc`. Hurtado and Muñoz [19] in joint work with Fernández and Quintero [10] gave a design and efficient implementation of an `ntcc`-based reactive programming language for LEGO RCX robots [20]—the robotic devices chosen in Section 5.1 as motivating examples.

Future Work. Timed ccp is still under development and certainly much remain to be explored. I shall indicate briefly a few research directions that I believe are necessary for the development of timed ccp as a well-established model of concurrency. Future research in timed ccp should address those issues that are central to other matured models of concurrency. In particular, the development of solid theories and tools for behavioral equivalences, which at present time is still very immature. For instance, currently there are neither axiomatizations nor automatic tools for reasoning about process equivalences. Furthermore, the decision algorithms for the verification problem, are very costly as the initial interest in them was purely theoretical. For practical purposes, it is then fundamental to conduct studies on the design and implementation of efficient algorithms for this problem.

Acknowledgments. I am specially grateful to the people with whom I have worked on temporal ccp with great pleasure: Mogens Nielsen, Catuscia Palamidesi, and Camilo Rueda.

References

1. M. Abadi. The power of temporal proofs. *Theoretical Computer Science*, 65:35–84, 1989.
2. J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
3. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
4. F. de Boer, M. Gabbrielli, and M. Chiara. A temporal logic for reasoning about timed concurrent constraint programs. In *TIME 01*. IEEE Press, 2001.

5. F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 161:45–83, 2000.
6. F. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1):37–78, 1995.
7. F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5), 1997.
8. J.F. Diaz, C. Rueda, and F. Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), 1998.
9. H. Dierks. A process algebra for real-time programs. In *FASE*, volume 1783 of *LNCS*, pages 66–81. Springer Verlag, 2000.
10. D. Fernádez and L. Quintero. *VIN: An ntcc visual language for LEGO Robots*. BSc Thesis, Universidad Javeriana-Cali, Colombia, 2003. http://www.brics.dk/∼fvalenci/ntcc-tools.
11. J. Fredslund. The assumption architecture. Progress Report, Department of Computer Science, University of Aarhus, 1999.
12. D. Gilbert and C. Palamidessi. Concurrent constraint programming with process mobility. In *Proc. of the CL 2000*, LNAI, pages 463–477. Springer-Verlag, 2000.
13. V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *Symposium on Principles of Programming Languages*, pages 189–202, 1999.
14. V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2):3–49, 1998.
15. N. Halbwachs. Synchronous programming of systems. *LNCS*, 1427:1–16, 1998.
16. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.
17. S. Haridi and S. Janson. Kernel andorra prolog and its computational model. In *Proc. of the International Conference on Logic Programming*, pages 301–309. MIT Press, 1990.
18. C. A. R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.
19. R. Hurtado and M. Muñoz. *LMAN: An ntcc Abstract Machine for LEGO Robots*. BSc Thesis, Universidad Javeriana-Cali, Colombia, 2003. http://www.brics.dk/∼fvalenci/ntcc-tools.
20. H. H. Lund and L. Pagliarini. Robot soccer with LEGO mindstorms. *LNCS*, 1604:141–151, 1999.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer, 1991.
22. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR '92*, volume 630 of *LNCS*, pages 550–564. Springer-Verlag, 1992.
23. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.
24. R. Milner. A finite delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1992.
25. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
26. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7, 1974.
27. M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of concurrent constraint programming languages. In *Proc. of PPDP'02*, pages 156–167. ACM Press, 2002.
28. M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(2):145–188, 2002.
29. M. Nielsen and F. Valencia. *Temporal Concurrent Constraint Programming: Applications and Behavior*, chapter 4, pages 298–324. Springer-Verlag, LNCS 2300, 2002.
30. C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proc. of CP'01*. Springer-Verlag, LNCS 2239, 2001.

31. C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress '62*, 1962.
32. G. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, University of Aarhus, 1981.
33. A. Pnueli. The temporal logic of programs. In *Proc. of FOCS-77*, pages 46–57. IEEE, IEEE Computer Society Press, 1977.
34. G.M. Reed and A.W. Roscoe. A timed model for communication sequential processes. *Theoretical Computer Science*, 8:249–261, 1988.
35. F. Rossi and U. Montanari. Concurrent semantics for concurrent constraint programming. In *Constraint Programming: Proc. 1993 NATO ASI*, pages 181–220, 1994.
36. J.H. Réty. Distributed concurrent constraint programming. *Fundamenta Informaticae*, 34(3), 1998.
37. C. Rueda and F. Valencia. Proving musical properties using a temporal concurrent constraint calculus. In *Proc. of the 28th International Computer Music Conference (ICMC2002)*, 2002.
38. V. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.
39. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*, pages 71–80, 1994.
40. V. Saraswat, R. Jagadeesan, and V. Gupta. Programming in timed concurrent constraint languages. In *Constraint Programming*, NATO Advanced Science Institute Series, pages 361–410. Springer-Verlag, 1994.
41. V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proc. of POPL'95*, pages 272–285, 1995.
42. V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, 1996.
43. V. Saraswat, R. Jagadeesan, and V. Gupta. jcc: Integrating timed default concurrent constraint programming into java. http://www.cse.psu.edu/∼saraswat/jcc.html, 2003.
44. V. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91*, pages 333–352, 1991.
45. E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510, 1990.
46. G. Smolka. A Foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *LNCS*, 1994. Invited Talk.
47. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.
48. S. Tini. On the expressiveness of timed concurrent constraint programming. *Electronics Notes in Theoretical Computer Science*, 1999.
49. S. Tini. An axiomatic semantics for the synchronous language gentzen. In *FOSSACS'01*, volume 2030 of *LNCS*. Springer-Verlag, 2001.
50. F. Valencia. *Temporal Concurrent Constraint Programming*. PhD thesis, BRICS, University of Aarhus, 2003.
51. F. Valencia. Timed concurrent constraint programming: Decidability results and their application to LTL. In *Proc. of ICLP'03*. Springer-Verlag, LNCS, 2003.
52. W. Yi. *A Calculus for Real Time Systems*. PhD thesis, Chalmers Institute of Technology, Sweden, 1991.
53. W. M. Zuberek. Timed petri nets and preliminary performance evaluation. In *Proc. of the 7th Annual Symposium on Computer Architecture*, pages 88–96. ACM and IEEE, 1980.