Notes on Timed CCP

Mogens Nielsen * and Frank D. Valencia **

[1] **BRICS** University of Aarhus, Denmark
[2] Dept. of Information Technology, Uppsala University
Email:mn,fvalenci@brics.dk

Abstract A *constraint* is a piece of (partial) information on the values of the variables of a system. *Concurrent constraint programming* (ccp) is a model of concurrency in which agents (also called processes) interact by telling and asking information (constraints) to and from a shared store (a constraint). *Timed* (or *temporal*) ccp (tccp) extends ccp by agents evolving over time. A distinguishing feature of tccp, is that it combines in one framework an *operational and algebraic* view from process algebra with a *declarative* view based upon temporal logic. Tccp has been widely used to specify, analyze and program reactive systems.

This note provides a comprehensive introduction to the background for and central notions from the theory of tccp. Furthermore, it surveys recent results on a particular tccp calculus, ntcc , and it provides a classification of the expressive power of various tccp languages.

1 Introduction

Saraswat's *concurrent constraint programming* (ccp) [45] 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 to an from a medium, a so-called *store*. Partial 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, 37, 12], stochastic behavior [13], and most prominently time [40, 5, 42, 14]. *Timed* ccp extends ccp by allowing agents to be constrained by time requirements.

Modal extensions of logic study time in logic reasoning, and in the same way mature models of concurrency have been extended with explicit notions of time. For instance, neither Milner's CCS [25], Hoare's CSP [19], nor Petri Nets [33], in their original form, were concerned explicitly with temporal behavior, but they all have been extended to

* The contribution of M. Nielsen to this work was suported by Basic Research in Computer Science, Centre of the Danish National Research Foundation.
** The contributation of F. Valencia to this work was supported by the **PROFUNDIS** Project.

incorporate an explicit notion of time, e.g. Timed CCS [53], Timed CSP [35], and Timed Petri Nets [54].

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, and the combination in one framework of the alternative views of processes, allows timed ccp to benefit from the large body of techniques of well established theories.

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 concurrent systems, since they often involve *specific domains* (e.g., controllers, databases, reservation systems) and have time-constraints *specifying* their behavior. Several timed extensions of ccp have been developed as settings for the modeling, programming and specification of timed systems [40, 43, 5, 14].

Organization. This note provides an overview of timed ccp with its basic background and various approaches explored in the literature. Furthermore, the note offers an introduction to a particular timed ccp process calculus called `ntcc` . In Sections 2 and 3 we give a basic background on ccp and timed ccp. Section 4 is devoted to present the developments of the timed ccp calculus `ntcc` [30]. In Section 5 we describe in detail several timed ccp languages and provide a classification of their expressive power. Finally, in Section 6 we discuss briefly some related and future work on timed ccp.

2 Background: Concurrent Constraint Programming

In his seminal PhD thesis [39], 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 the next section, the ccp model elegantly combines logic concepts and concurrency mechanisms.

Concurrent constraint programming traces its origins back to Montanari's pioneering work [28] leading to constraint programming and Shapiro's concurrent logic programming [46]. The ccp model has received a significant theoretical and implementational attention: Saraswat, Rinard and Panangaden [45] as well as De Boer, Di Pierro and Palamidessi [6] gave fixed-point denotational semantics to ccp, whilst Montanari and Rossi [36] gave a (true-concurrent) Petri-Net semantics (using the formalism of contextual nets); De Boer, Gabrielli et al [7] developed an inference system for proving properties of ccp processes; Smolka's Oz [48] as well as Haridi and Janson's AKL [17] programming languages are built upon ccp ideas.

The ccp model. A concurrent system is specified in the ccp model in terms of *constraints* over the variables of the system. A constraint is a first-order formula representing *partial information* about the values of variables. As an example, for a system with variables x and y taking natural numbers as values, the constraint $x + y > 16$ specifies possible values for x and y (those satisfying the inequation). The ccp model is parameterized by 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, $x + y > 16 \models x + y > 0$).

During a ccp computation, the state of the system is specified by an entity called the *store* in which information about the variables of the system resides. 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., $x = 16$ and $y = 7$), rather than a set of possible values.

The notion of store in ccp suggests a model of concurrency with a central memory. This is, however, only an abstraction which simplifies the presentation of the model. The store may be distributed in several sites according to the sharing of variables (see [39] for further discussions about this matter). Conceptually, the store in ccp is the *medium* through which agents interact with each other.

A ccp process 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 store representing the previous state and the constraint being added. Hence, updating does not change the values of the variables as such, but constrains further some of the previously possible values.

Furthermore, ccp processes can synchronize by querying (or *asking*) information from the store. Asking is blocked until there is enough information in the store to *entail* (i.e., answer positively) the query, i.e. the ask operation determines whether the constraint representing the store entails the query.

A ccp computation terminates whenever it reaches a point, called a *resting* or a *quiescent* point, in which no more information can be added to the store. The output of the computation is defined to be the final store, also called the *quiescent store*.

Example 1. Consider the simple ccp scenario illustrated in Figure 1. We have four agents (or processes) wishing to interact through an initially empty store. Let us name them, starting from the upper leftmost agent in a clockwise fashion, A_1, A_2, A_3 and A_4, respectively.

In this scenario, A_1 may move first and tell the others through the store the (partial) information that the temperature value is greater than 42 degrees. This causes the addition of the item "temperature>42" to the previously empty store.

Now A_2 may ask 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, the exact value of the temperature can not be entailed. Hence, the agent A_2 is 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.

However, A_4 may tell the information that the temperature is less than 70 degrees. The store becomes "temperature $> 42 \wedge$ temperature < 70", and now process A_3 can execute Q, since its query is entailed by the information in the store . The 2 agent A_2 is doomed to be blocked forever unless Q adds enough information to the store to entail its query. $\square$

In the spirit of process calculi, the language of processes in the ccp model is given by a small number of primitive operators or combinators. A typical ccp process language contains the following operators:

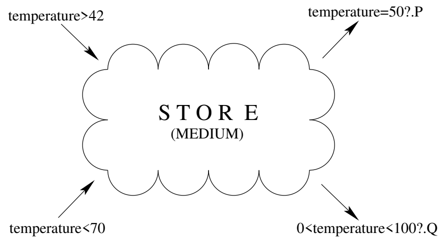

Figure1. A simple ccp scenario

- *A tell operator*, telling constraints (e.g., agent A_1 above).
- *An ask operator*, prefixing another process, its continuation (e.g. the agent A_2 above).
- *Parallel composition*, combining processes concurrently. For example the scenario in Figure 1 can be specified as the parallel composition of A_1, A_2, A_3 and A_4.
- *Hiding* (also called *restriction* or *locality*), introducing local variables, thus restricting the interface through which a process can interact with others.
- *Summation*, expressing a nondeterministic combination of agents to allow alternate courses of action.
- *Recursion*, defining infinite behavior.

It is worth pointing out that without summation, the ccp model is deterministic, in the sense that the final store is always the same, independently of the execution order (scheduling) of the parallel components [45].

3 Timed Concurrent Constraint Programming

The first timed ccp model was introduced by Saraswat et al [40] as an extension of ccp aimed at programming and modeling timed, reactive systems. This tcc model elegantly combines ccp with ideas from the paradigms of Synchronous Languages [2, 15].

The tcc model takes the view of reactive computation as proceeding *deterministically* in discrete time units (or time *intervals*). In other words, 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. Furthermore, 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 tcc model extends the standard ccp with fundamental operations for programming reactive systems, e.g. *delay* and *time-out* operations. 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*.

In spite of its simplicity, the tcc extension to ccp is far-reaching. Many interesting temporal constructs can be expressed, see [40] for details, As an example, 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.

The tcc model has attracted a lot of attention recently. Several extensions have been introduced and studied in the literature. One example can be found in [43], adding a notion of strong pre-emption: the time-out operations can trigger activity in the current time interval. Other extensions of tcc have been proposed in [14], in which processes can evolve continuously as well as discretely.

The tccp framework, introduced in [5] by Gabrielli et al, is a fundamental representative model of nondeterministic timed ccp. In [5] the authors 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]).

It would be hard to introduce all the tcc extensions in detail, and hence we focus in the following on the `ntcc` calculus, which is a generalization of the tcc model introduced in [30] by Palamidessi and the present authors. 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 [38].

The major difference between tccp model from [5] and `ntcc` is that the former extends the original ccp while the latter extends the tcc model. More precisely, 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.

4 The `ntcc` process calculus

This section gives a formal introduction to the `ntcc` model. We introduce the syntax and the semantics of the `ntcc` process language, and illustrate the expressiveness by modeling robotic devices. Furthermore, we shall present some of the reasoning techniques provided by `ntcc` focusing on

1. *Behavioural equivalences*, which are characterized operationally, relating process behavior much like the behavioral equivalences for traditional process calculi (e.g., bisimilarity and trace-equivalence).

2. A *denotational semantics* which interprets a given process as the set of sequences of input/output behaviours it can potentially exhibit while interacting with arbitrary environments.
3. A *process logic* expressing specifications of behaviors of processes, and an associated *inference system* providing proofs of processes fulfilling specifications.

Informal Description of ntcc Processes

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

As for the tcc model, the ntcc model is parameterized by 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 interdependencies between these constraints.

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 for the natural numbers.

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 [21]. These controllers have external input and output ports. One can attach, for example, sensors of light, touch or temperature to the input ports, and actuators like motors, lights or alarms to the output ports. Typically PLC's and RCX bricks operate in a cyclic fashion. Each cycle consists 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. 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.

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

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. The store is a constraint itself. In this way, addition of information corresponds to logical conjunction, and determining the presence of information corresponds to logical entailment.

The tell and ask processes have the syntactic forms respectively

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

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 the specific speed values. The execution in store d of process

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

causes the store to become $(\mathtt{motor_1\_speed} > \mathtt{motor_2\_speed} > 10)$ in the current time interval, thus increasing the information we know about the system.

Notice that in the underlying constraint system $d \models \mathtt{motor_1\_speed} > 0$, therefore the process

$$\mathbf{tell}(\mathtt{motor_1\_speed} = 0)$$

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 (i.e., it becomes permanently inactive).

Example 3. Suppose that $d = (\mathtt{motor_1\_speed} > \mathtt{motor_2\_speed})$ is the store. The process

$$P = \mathbf{when}\ \mathtt{motor_1\_speed} > 0\ \mathbf{do}\ Q$$

will execute Q in the current time interval since $d \models \mathtt{motor_1\_speed} > 0$, by contrast the process

$$P' = \mathbf{when}\ \mathtt{motor_1\_speed} > 10\ \mathbf{do}\ Q$$

will not execute Q, unless more information is added to the store during the current time interval entailing $\mathtt{motor_1\_speed} > 10$.

□

Nondeterminism. As argued above, partial information allows us to model behavior for alternative values of variables. 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, *nondeterministically* chooses 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. Given P and Q we denote their parallel composition by the process

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

In one time unit 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

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

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

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

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

The process $(\textbf{local}\, x)\, P$ 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). However, one often also uses variables, which represent some local (or private) computational data.

Suppose that R is a given process modeling some controller task. Furthermore, suppose that R uses a variable z, which is set at random to a value $v \in \{0, 1\}$ in the process P, i.e.

$$P = \Big(\sum_{v \in \{0,1\}} \textbf{when}\ \texttt{true}\ \textbf{do}\ \textbf{tell}(z = v)\Big) \parallel R$$

representing the behavior of R under P's random assignment of z.

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 by $P' = (\textbf{local}\, z)\, P$. 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 [2] 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 preceding actions.

The unit-delay operators have the form

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

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 also that $Q = \mathbf{unless}\ \mathtt{false}\ \mathbf{next}\, P$ is not the same as $R = \mathbf{next}\, P$, since R (unlike Q) always activates P in the next time interval, even if the store entails $\mathtt{false}$.

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

$$\mathbf{unless}\ \mathtt{false}\ \mathbf{next}\ (\mathbf{tell}(\mathtt{motor_1\_speed} > 0)\ \parallel\ \mathbf{tell}(\mathtt{motor_2\_speed} > 0))$$

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 [26]. The process

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

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

Example 8. Let $S = \star\, \mathbf{tell}(\mathtt{malfunction}(\mathtt{motor_1\_status}))$. The process S can be used to specify that $\mathtt{motor_1}$, 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 [27]. Given a process P, the process

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

represents $P \parallel (\mathbf{next}\, P) \parallel (\mathbf{next}\ \mathbf{next}\, P) \parallel \cdots \parallel !\, P$, 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 $\mathtt{motor}_1$. If a malfunction is reported, R tells that $\mathtt{motor}_1$ must be turned off.

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

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

Some Derived Forms

We have informally introduced the basic process constructs of $\mathtt{ntcc}$ 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 "$\mathbf{when}\ \mathtt{true}\ \mathbf{do}$" if no confusion arises. The "blind-choice" process $\sum_{i\in I} \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ P_i$, 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 $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\,P)\ldots))$, where $\mathbf{next}$ 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 4 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 [26].

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.

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

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

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.

4.1 The Operational Semantics of `ntcc`

Following the informal description of `ntcc` above, we now proceed with a formal definition. We shall begin by formalizing the notion of constraint system and the 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 [27], provides *internal* and *external* transitions describing process evolutions. The internal transitions describe evolutions within a time unit, and they are considered to be unobservable. The external transitions describe evolution across the time units, and they are considered to be observable.

Constraint Systems. For our purposes it will suffice to consider the notion of constraint system based on first-order logic, following e.g. [47].

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 $d \models c$.

Convention 1 *Henceforth, $\mathcal{C}$ denotes the set of constraints modulo $\approx$ under consideration in the underlying constraint system.*

Let us now give some examples of constraint systems. The classical example is the Herbrand constraint system [39].

Definition 2 (Herbrand Constraint System). *The Herbrand constraint system is such that:*

- *Σ is a set with infinitely many function symbols of each arity and equality $=$.*
- *Δ is given by Clark's Equality Theory with the schemas*
  $f(x_1,\ldots,x_n)=f(y_1,\ldots,y_n) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$
  $f(x_1,\ldots,x_n)=g(y_1,\ldots,y_n) \Rightarrow \mathtt{false}$, *if f,g are distinct symbols*
  $x = f(\ldots x \ldots) \Rightarrow \mathtt{false}$.

The importance of the Herbrand constraint system is that it underlies conventional logic programming and many first-order theorem provers. Its value lies in the Herbrand Theorem, which reduces the problem of checking unsatisfiability of a first-order formula to the unsatisfiability of a quantifier-free formula interpreted over finite trees.

Another widely used constraint system is the finite-domain constraint system $\mathbf{FD}$ defined in [18]. In $\mathbf{FD}$ variables are assumed to range over finite domains and, in addition to equality, we may have predicates that restrict the range of a variable to some finite set. The following is a simplified finite-domain constraint system.

Definition 3 (A Finite-Domain Constraint System). *Let $n > 0$. Define $\mathbf{FD}[n]$ as the constraint system such that:*

- *Σ is given by the constants symbols $0, 1, ..., n-1$ and the equality $=$.*
- *Δ is given by the axioms of equational theory $x = x$, $x = y \Rightarrow y = x$, $x = y \wedge y = z \Rightarrow x = z$, and $v = w \Rightarrow \mathtt{false}$ for each two different constants v, w in Σ.*

Intuitively $\mathbf{FD}[n]$ provides a theory of variables ranging over a finite domain of values $\{0, \ldots, n-1\}$ with syntactic equality over these values.

The following is a somewhat more complex finite-domain constraint system.

Definition 4 (Modular Arithmetic Constraint System). *Let $n > 0$. Define $\mathbf{A}[n]$ as the constraint system such that:*

- *Σ is given by $\{0, 1, ..., n-1, \mathrm{succ}, \mathrm{pred}, +, \times, =, >\}$.*
- *Δ is the set of sentences valid in arithmetic modulo n.*

The intended meaning of $\mathbf{A}[n]$ is the natural numbers interpreted as in arithmetic modulo n. Due to the familiar operations it provides, we shall often assume that $\mathbf{A}[n]$ is the underlying constraint system in our examples and applications.

Other examples of constraint systems include: Rational intervals, Enumerated type, the Kahn constraint system and the Gentzen constraint system (see [45] and [39] for details).

Process Syntax and Semantics.

Following the informal description above, the process constructions in the ntcc calculus are given by the following syntax:

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:*

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

The informal semantic meaning provided above of the constructs is formalized in terms of the following structural operational semantics (SOS) of ntcc . This semantics defines *transitions* between process-store *configurations* of the form $\langle P, c\rangle$, with stores represented as constraints and processes quotiented by the congruence $\equiv$ below.

Let us define precisely what we mean by the term "congruence" of processes, a key concept in the theory of process algebra. First, we need to introduce the standard notion of *process context*. Informally speaking, a process context is a process expression with a single hole, represented by $[\cdot]$, such that placing a process in the hole yields a well-formed process. More precisely,

Definition 5 (Process Context). Process contexts *C are given by the syntax*

$$\begin{array}{rl} C ::= & [\cdot] \mid \mathbf{when}\ c\ \mathbf{do}\ C + M \mid C \parallel P \mid P \parallel C \mid (\mathbf{local}\, x)\, C \\ \mid & \mathbf{next}\, C \mid \mathbf{unless}\ c\ \mathbf{next}\, C \mid \star\, C \mid !\, C \end{array}$$

where M stands for summations. The process $C[Q]$ results from the textual substitution of the hole $[\cdot]$ in C with Q.

An equivalence relation is a congruence if it respects all contexts:

Definition 6 (Process Congruence). *An equivalence relation $\cong$ on processes is said to be a process* congruence *iff for all contexts C, $P \cong Q$ implies $C[P] \cong C[Q]$.*

We can now introduce the structural congruence $\equiv$. Intuitively, the relation $\equiv$ describes irrelevant syntactic aspects of processes. It states that $(Proc/\equiv, \parallel, \mathbf{skip})$ is a commutative monoid.

Definition 7 (Structural Congruence). *Let $\equiv$ be the smallest congruence over processes satisfying the following axioms:*

1. $P \parallel \mathbf{skip} \equiv P$
2. $P \parallel Q \equiv Q \parallel P$
3. $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$.

We extend $\equiv$ to configurations by decreeing that $\langle P, c\rangle \equiv \langle Q, c\rangle$ iff $P \equiv Q$.

Convention 2 *Following standard notation, 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".

Intuitively, the observable reduction is obtained from a sequence of internal reductions starting in P with initial store c and terminating in a process Q with final store d. The process R, which is the one to be executed in the next *time interval* (or time unit), is obtained by removing from Q what was meant to be executed only during the current time interval. Notice that the store d is not transferred to the next time interval, i.e. information in d can only be transfered to the next time unit by P itself.

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

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

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

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

$$\text{STR}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'}\ \text{if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2' \qquad \text{ABORT}\ \frac{}{\langle \mathbf{abort}, d\rangle \longrightarrow \langle \mathbf{abort}, d\rangle}$$

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

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 7 and 8.

Most of the rules in Table 1 should be straightforward from the informal description of the intended semantics given above. For detailed comments we refer to [30], and here we only comment on two of the rules: the rule for local variables LOC and OBS (covering the seemingly missing rules for "next" and "unless" processes).

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

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

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.*

4.2 Observable Behavior

In this section we recall some notions introduced in [31] of *process observations*. We assume that what happens within a time unit cannot be directly observed, and thus we abstract from internal transitions, and focus on observations in terms of external transitions.

Notation 1 *Throughout this paper $\mathcal{C}^\omega$ denotes the set of infinite 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 $\alpha' = c'_1.c'_2.\ldots$. We use the notation $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$ to denote the existence of an infinite sequence of observable transitions (or *run*): $P = P_1 \xRightarrow{(c_1,c'_1)} P_2 \xRightarrow{(c_2,c'_2)} \ldots$.

IO and Output Behavior. Consider a run of P as above. 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. As a third alternative, we may observe the quiescent input sequences of a process. These are sequences of input on which P can run without adding any information; we observe whether $\alpha = \alpha'$ whenever $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$.

In [30] it is shown that the set of quiescent sequences of a given P can be characterized as *the set of infinite sequences that P can possibly output under arbitrary environments*; the strongest postcondition (sp) of P.

Summing up, we have the following notions of observable behavior.

Definition 9 (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*

$$sp(P) = \{\alpha \mid P \xRightarrow{(\alpha',\alpha)}{}^{\omega} \textit{ for some } \alpha'\}.$$

Given these notions of observable behaviors, we have the following naturally induced equivalences and congruences (recall the notion of congruence given in Definition 6.)

Definition 10 (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*. Notice, that they identify processes whose internal behavior may differ widely. Such an abstraction from internal behavior is essential in the theory of several process calculi; most notably in weak bisimilarity for CCS [25].

Example 10. Let a, b, c, d and e 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 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 $\alpha \notin sp(P)$. $\square$

Congruence and Decidability Issues. In [30] it is proven that none of the three observational equivalences introduced in Definition 10 are congruences. However, $\sim_{sp}$ is a congruence if we confine our attention to the so-called *locally-independent* fragment of the calculus, i.e. the fragment without non-unary summations and "unless" operations, whose guards depend on local variables.

Definition 11 (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 [51]. Moreover, the applicability of this fragment is witnessed by the fact all the ntcc applications we are aware of [30, 31, 51] 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 [29]. Furthermore, it allows us to express infinite-state processes (i.e., there are 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 [52]. In 5 below we shall present a number of other seemingly surprising decidability results for other fragments of ntcc .

⋆ ⋆ ⋆

4.3 Denotational Semantics

In the previous section we introduced the notion of strongest-postcondition of ntcc processes in operational terms. In the following we show the abstract denotational model of this notion, first presented in [32].

The denotational semantics is defined as a function $[\![\cdot]\!]$ associating with each process a set of infinite constraint sequences, $[\![\cdot]\!] : Proc \to \mathcal{P}(\mathcal{C}^\omega)$. The definition of this function is given in Table 2. Intuitively, $[\![P]\!]$ is meant to capture the set of all sequences P can possibly output. For instance, the sequences associated with $\mathbf{tell}(c)$ are those for which the first element is stronger than c (see DTELL, Table 2). Process $\mathbf{next}\,P$ has not influence on the first element of a sequence, thus $d.\alpha$ is a possible output if α is a possible output of P(see DNEXT, Table 2). The other cases can be explained analogously.

From [7] we know that there cannot be a $f : Proc \to \mathcal{P}(\mathcal{C}^\omega)$, compositionally defined, such that $f(P) = sp(P)$ for all P. Nevertheless, as stated in the theorem below, Palamidessi et al [32] showed that the sp denotational semantics matches its operational counter-part for the locally-independent fragment 11.

Theorem 1 (Full Abstraction, [32]). *For every* ntcc *process P, $sp(P) \subseteq [\![P]\!]$ and if P is locally-independent then $[\![P]\!] \subseteq sp(P)$.*

The full-abstraction result above has an important theoretical value; i.e., for a significant fragment of the calculus we can abstract away from operational details by working

DTELL:	$[\![\mathbf{tell}(c)]\!] = \{d.\alpha \mid d \models c\}$
DSUM:	$[\![\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i]\!] = \bigcup_{i\in I} \{d.\alpha \mid d \models c_i \text{ and } d.\alpha \in [\![P_i]\!]\} \cup \bigcap_{i\in I} \{d.\alpha \mid d \not\models c_i\}$
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.

with $[\![P]\!]$ rather than $sp(P)$. Furthermore, an interesting corollary of the full-abstraction result is that $\sim_{sp}$ is a congruence, if we confine ourselves to locally-independent processes.

4.4 LTL Specification and Verification

Processes in `ntcc` denote observable behavior of timed systems. As with other such formalisms, it is often convenient to express specifications of such behaviors in logical formalisms. In this section we present the `ntcc` logic first introduced in [32]. We start by defining a linear-time temporal logic (LTL) expressing 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. Finally, we present an inference system aimed at proving processes satisfying specifications.

A Temporal Logic. The `ntcc` LTL expresses properties of infinite sequences of constraints, and we shall refer to it as **CLTL**.

Definition 12 (CLTL 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:*

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

Here c is a constraint (i.e., a first-order formula in the underlying constraint system) representing a *state formula* c. The symbols $\dot{\mathtt{true}}, \dot{\mathtt{false}}, \dot{\wedge}, \dot{\vee}, \dot{\neg}, \dot{\exists}$ represent linear-temporal logic true, false, conjunction, disjunction, negation and existential quantification. As clarified later, the dotted notation is introduced since in **CLTL** these operators may have different interpretations from the symbols $\mathtt{true}, \mathtt{false}, \wedge, \vee, \neg, \exists$ in the underlying constraint system. The symbols $\circ$, $\Box$, and $\Diamond$ denote the temporal operators *next*, *always* and *sometime*.

The standard interpretation structures of linear temporal logic are infinite sequences of states [22]. In the case of `ntcc` , it is natural to replace states by constraints, and hence our interpretations are elements of $\mathcal{C}^\omega$.

The **CLTL** semantics is given in Definition 14. Following [22] we introduce the notion of x-*variant*.

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 13 (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* $\exists_x \alpha = \exists_x \alpha'$.

Intuitively, d and α' are x-variants of c and α, respectively, if they are logically the same except for information about x. For example, $y = 0$ is an x-variant of $x = y = 0$.

Definition 14 (CLTL Semantics). *We say that* $\alpha \in \mathcal{C}^\omega$ *satisfies (or that it is a model of) the* **CLTL** *formula* F, *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 $[\![F]\!]=\{\alpha \mid \alpha \models_{\mathrm{CLTL}} F\}$. *We say that* F *is* **CLTL** valid *iff* $[\![F]\!] = \mathcal{C}^\omega$, *and that* F *is* **CLTL** satisfiable *iff* $[\![F]\!] \neq \emptyset$.

State formulae as Constraints. Let us comment briefly on the role of constraints as state formulae in our logic. A temporal formula F expresses a property of sequences of constraints. As a state formula, c expresses a property, which is satisfied by those $e.\alpha'$ such that $e \models c$. Hence, the state formula `false` (and consequently $\Box$ `false`) is satisfied by $\mathtt{false}^\omega$. On the other hand, the temporal formula $\dot{\mathtt{false}}$ has no model whatsoever.

Similarly, the models of the temporal formula $c \dot{\vee} d$ are those $e.\alpha'$ such that either $e \models c$ or $e \models d$ holds. Therefore, the formula $c \dot{\vee} d$ and the atomic proposition $c \vee d$ may have different models since, in general, one can verify that $e \models c \vee d$ may hold while neither $e \models c$ nor $e \models d$ hold – e.g. take $e = (x = 1 \vee x = 2)$, $c = (x = 1)$ and $d = (x = 2)$.

In contrast, the formula $c \dot{\wedge} d$ and the atomic proposition $c \wedge d$ have the same models since $e \models (c \wedge d)$ holds if and only if both $e \models c$ and $e \models d$ hold.

The above discussion tells us that the operators of the constraint system should not be confused with those of the temporal logic. In particular, the operators $\vee$ and $\dot{\vee}$. This distinction does not make our logic intuitionistic. In fact, classically (but not intuitionistically) valid statements such as $\dot{\neg} A \dot{\vee} A$ and $\dot{\neg}\dot{\neg} A \dot{\Rightarrow} A$ are also valid in our logic.

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

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

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

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

LNEXT: $$\frac{P \ \vdash \ F}{\mathbf{next}\ P \ \vdash \ \circ F}$$

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

LREP: $$\frac{P \ \vdash \ F}{!\,P \ \vdash \ \Box F}$$

LSTAR: $$\frac{P \ \vdash \ F}{\star\, P \ \vdash \ \Diamond F}$$

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

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

Process Verification.

We are now ready to define what it means for a process P to satisfy a specification F.

Definition 15 (Verification). P satisfies F, *written* $P \models_{\mathrm{CLTL}} F$, *iff* $sp(P) \subseteq [\![F]\!]$.

Thus, the intended meaning of $P \models_{\mathrm{CLTL}} F$ is that every sequence P can possibly output on inputs from arbitrary environments satisfies the temporal formula F. For example, $\star\,\mathbf{tell}(c) \models \Diamond c$, since in every infinite sequence output by $\star\,\mathbf{tell}(c)$ on arbitrary inputs, there must be an element entailing c.

Following the discussion above, notice that $P = \mathbf{tell}(c) + \mathbf{tell}(d) \models (c \mathbin{\dot\vee} d)$ as every constraint e output by P entails either c or d. In contrast, $Q = \mathbf{tell}(c \vee d) \not\models (c \mathbin{\dot\vee} d)$ in general since Q can output a constraint e which entails $c \vee d$, but neither c nor d.

4.5 Proof System for Verification.

[32] introduces a *proof (or inference) system* for assertions of the form $P \vdash F$, where $P \vdash F$ is intended to be the "counterpart" of $P \models F$ in the sense that $P \vdash F$ should approximate $P \models_{\mathrm{CLTL}} F$ as closely as possible (ideally, they should be equivalent). The system is presented in Table 3.

Definition 16 ($P \vdash F$). *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 briefly comment on (the soundness of) 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 reflecting the fact that every output of $\mathbf{tell}(c)$ on arbitrary input, indeed satisfies the atomic proposition c, i.e., $\mathbf{tell}(c) \models_{\mathrm{CLTL}} c$.

Consider now the rule for the choice operator:

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

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, i.e. (inductively) $P_i \models_{\mathrm{CLTL}} F_i$. Then we may conclude that every output of P on arbitrary input will satisfy either: (a) some of the guards c_i and their corresponding F_i (i.e., $\dot{\bigvee}_{i \in I}(c_i \,\dot{\wedge}\, F_i)$), 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

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

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 $F \,\dot{\wedge}\, G$.

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:

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

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

Assume that $P \vdash F$, i.e. (inductively) $P \models_{\mathrm{CLTL}} F$. Rule LNEXT reflects that we may then conclude that also the process $\mathbf{next}\, P$ 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 F is satisfied by P, then executing P in each time interval will imply F to be satisfied in each time interval, i.e. $!\, P$ satisfies $\Box F$. Rule LSTAR reflects that if P is executed in some time interval, then in that time interval F is satisfied, and hence $\star\, P$ satisfies $\Diamond F$.

Finally, we have the rule:

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

Notice that this rule refers to some unspecified way of inferring validity of **CLTL** formulae. We shall return to this point shortly. Rule LCONS simply says that if P satisfies a specification F then it also satisfies any weaker specification G. We shall also refer to LCONS as *the consequence rule*.

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 11. Recall Example 9. We have a process R which was repeatedly checking the state of $\mathtt{motor}_1$. If a malfunction is reported, R would tell that $\mathtt{motor}_1$ must be turned off. We also have a process S stating that motor $\mathtt{motor}_1$ is doomed to malfunction. Let $R\ =!\,\mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e)$ and $S\ =\ \star\,\mathbf{tell}(c)$ with the constraints $c = \mathtt{malfunction}(\mathtt{motor}_1\mathtt{\_status})$ and $e = (\mathtt{motor}_1\mathtt{\_speed} = 0)$. 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 $\mathtt{motor}_1$ is eventually turned off. The following is a derivation of the above assertion.

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

More complex examples of the use of the proof system for proving the satisfaction of processes specification can be found in [30]—in particular for proving properties of mutable data structures. $\square$

Let us now return to the issue of the relationship between $\vdash$ and $\models_{\text{CLTL}}$.

Theorem 2 (Relative Completeness, [30]). *If P is locally-independent then $P \vdash F$ iff $P \models_{\text{CLTL}} F$.*

Notice that this is indeed a “relative completeness” result, in the sense that, as mentioned earlier, one of our proof rules refer to the validity of temporal implication. 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 this, one may wonder about the decidability of the validity problem for our temporal logic. We look at this issue next.

Decidability Results. In [52] 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 of ntcc and negation-free **CLTL** formulae. Please recall that the locally-independent fragment of ntcc admits infinite-state processes. Also note that **CLTL** is first-order. Most first-order LTL's in computer science are not recursively axiomatizable, let alone decidable [1].

Furthermore, [52] 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**

A theoretical application of the theory of ntcc is presented in [52], stating a new positive decidability result for a first-order fragment of Pnueli's first-order **LTL** [22]. 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.

5 A Hierarchy of Timed CCP Languages

In the literature several timed ccp languages have been introduced, differing in their way of expressing infinite behavior. In this section we shall introduce a few fundamental representatives of mechanisms introducing infinite behavior, expressed as variants of the ntcc calculus. We shall also characterize their relative expressiveness following [29].

Since timed CCP languages are deterministic we shall confine our attention to the *deterministic* processes of ntcc as described in [30]. These are all the star-free processes with all summations having at most one guard. On top of this fragment we consider the following variants:

- rep: deterministic ntcc ; infinite behavior given by replication.
- $\mathtt{rec}_{\mathrm{p}}$: obtained from deterministic ntcc replacing replication by *parametric recursion*. In $\mathtt{rec}_{\mathrm{p}}$ each procedures body *has no free variables* other than its formal parameters.
- $\mathtt{rec}_{\mathtt{i}}$: same as $\mathtt{rec}_{\mathrm{p}}$, but where the actual parameters in recursive calls are *identical* to the formal parameters; i.e., we do not vary the parameters in the recursive calls.
- $\mathtt{rec}_{\mathrm{d}}$: obtained by using *parameterless recursion*, but *including* free variables in procedure bodies with *dynamic scope*.
- $\mathtt{rec}_{\mathrm{s}}$: same as $\mathtt{rec}_{\mathrm{d}}$ but with *static scope*.

In the following, the expressive power of these process languages is compared with respect to the notion of input-output behavior, as introduced in Section 4.2. More precisely, one language is considered at least as expressive as another, if any input-output behavior expressed by a process in the latter can be expressed also by a process in the former. The comparison results can be summarized as follows:

– $\mathtt{rec_p}$ and $\mathtt{rec_d}$ are equally expressive, and strictly more expressive than the other languages,
– $\mathtt{rep}$, $\mathtt{rec_s}$ and $\mathtt{rec_i}$ are equally expressive.

In fact, [29] shows a strong separation result between the languages $\mathtt{rec_p}$/$\mathtt{rec_d}$ and $\mathtt{rep}$/$\mathtt{rec_s}$/$\mathtt{rec_i}$: the input-output equivalence is undecidable for the languages in the first class, but decidable for the languages in the second class.

The undecidability results holds even if we fix an underling constraint system with a finite domain having at least one element. The undecidability result is obtained by a reduction from Post's correspondence problem [34] and an input-output preserving encoding between $\mathtt{rec_p}$/$\mathtt{rec_d}$.

The decidability results hold for arbitrary constraint systems, and follow from Büchi automata [3] representation of $\mathtt{ntcc}$ processes and input-output preserving encodings between the languages in $\mathtt{rep}$/$\mathtt{rec_s}$/$\mathtt{rec_i}$.

The expressiveness gaps illustrated above may look surprising to readers familiar with the π-calculus [27], since it is well known that the π-calculus correspondents of $\mathtt{rep}$, $\mathtt{rec_i}$ and $\mathtt{rec_p}$ all have the same expressive power. The reason for these differences can be attributed to the fact that the π-calculus has some powerful mechanisms (such as mobility), which compensate for the weakness of replication and the lower forms of recursion.

We start by formally defining our five classes of process languages.

5.1 Replication

We shall use $\mathtt{rep}$ to denote the deterministic fragment of $\mathtt{ntcc}$. The processes in the deterministic fragment are those star-free processes in which the cardinality of every summation index set is at most one. Thus, the resulting syntax of process in $\mathtt{rep}$ is given by:

$$\begin{array}{rll} P, Q, \ldots ::= \mathbf{skip} & \mid \mathbf{tell}(c) \mid \mathbf{when}\; c\; \mathbf{do}\; P \mid P \parallel Q \mid (\mathbf{local}\; x)\, P \\ \mid\; \mathbf{next}\, P & \mid\; \mathbf{unless}\; c\; \mathbf{next}\, P & \mid\; !\, P \end{array} \tag{9}$$

Infinite behavior in $\mathtt{rep}$ is provided by using replication. This way of expressing infinite behavior is also considered in [43]. To be precise, [43] uses the **hence** operator. However, $\mathbf{hence}\, P$ is equivalent to $\mathbf{next}\, !\, P$ and, similarly $!\, P$ is equivalent to $P \parallel \mathbf{hence}\, P$.

5.2 Recursion

Infinite behavior in tcc languages may also be introduced by adding recursion, as e.g. in [40, 41, 49]. Consider the process syntax obtained from replacing replication $!P$ with *process* (or *procedure*) calls $A(y_1, \ldots, y_n)$, i.e.:

$$\begin{array}{rll} P, Q, \ldots ::= \mathbf{skip} & \mid \mathbf{tell}(c) \mid \mathbf{when}\; c\; \mathbf{do}\; P \mid P \parallel Q \mid (\mathbf{local}\; x)\, P \\ \mid\; \mathbf{next}\, P & \mid\; \mathbf{unless}\; c\; \mathbf{next}\, P & \mid\; A(y_1, \ldots, y_n) \end{array} \tag{10}$$

The process $A(y_1, \ldots, y_n)$ is an *identifier* with arity n. We assume that every identifier has a (recursive) *process (or procedure) definition* of the form $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as P with y_i replacing x_i for each i.

We declare $\mathcal{D}$ to be the set of recursive definitions under consideration. We shall often use the notation $\boldsymbol{x}$ as an abbreviation of $x_1, x_2, \ldots, x_n$ if n is unimportant or obvious. We shall sometimes say that $A(\boldsymbol{y})$ is an *invocation* with *actual parameters* $\boldsymbol{y}$, and given $A(\boldsymbol{x}) \stackrel{\text{def}}{=} P$ we shall refer to P as its *body* and to $\boldsymbol{x}$ as its *formal parameters*.

Finite Dependency and Guarded Recursion Following [40], we shall require, for all the forms of recursion defined next, the following: (1) any process to depend only on finitely many definitions and (2) recursion to be "next" guarded. For example, given $A(\boldsymbol{x}) \stackrel{\text{def}}{=} P$, every invocation $A(\boldsymbol{y})$ in P must occur within the scope of a "next" or "unless" operator operator. This avoids non-terminating sequences of internal reductions (i.e., non-terminating computation within a time interval). Below we give a precise formulation of (1) and (2).

Given $A_1(\boldsymbol{x}_1) \stackrel{\text{def}}{=} P_1$ and $A_2(\boldsymbol{x}_2) \stackrel{\text{def}}{=} P_2$, we say that A_1 (directly) *depends* on A_2, written $A_1 \rightsquigarrow A_2$, if there is an invocation $A_2(\boldsymbol{y})$ in P_1. Requirement (1) can be then formalized by requiring the strict ordering induced by $\rightsquigarrow^*$ (the reflexive and transitive closure of $\rightsquigarrow$)[1] to be well founded.

To formalize (2), suppose that $A_1 \rightsquigarrow A_2 \rightsquigarrow \ldots \rightsquigarrow A_n \rightsquigarrow A_{n+1} = A_1$, where $A_i(\boldsymbol{x}_1) \stackrel{\text{def}}{=} P_i$. We shall require that for at least one i, $1 \leq i \leq n$, the occurrences of A_{i+1} in P_i are within the scope of a "next" or an "unless" operator.

Parametric Recursion

We consider a further restriction for the case of recursion involving parameters. *All the free variables in definitions' bodies must be formal parameters*; more precisely, for each $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$, we decree that $\mathit{fv}(P) \subseteq \{x_1, \ldots, x_n\}$.

We shall use $\mathtt{rec_p}$ to denote the tcc language with recursion with the above syntactic restriction. The operational rules for $\mathtt{rec_p}$ are obtained from Table 1 by replacing the rule for replication REP with the following rule for recursion:

$$\text{REC} \; \frac{}{\langle A(\boldsymbol{y}), d\rangle \longrightarrow \langle P[\boldsymbol{y}/\boldsymbol{x}], d\rangle} \; A(\boldsymbol{x}) \stackrel{\text{def}}{=} P \tag{11}$$

As usual $P[y_1, \ldots, y_n/x_1, \ldots, x_n]$, with all the x_i's being pairwise distinct, is the process that results from syntactically replacing every free occurrence of x_i by y_i using α-conversion wherever needed to avoid capture.

[1] The relation $\rightsquigarrow^*$ is a pre-ordering. By induced strict ordering we mean the strict component of $\rightsquigarrow^*$ modulo the equivalence relation obtained by taking the symmetric closure of $\rightsquigarrow^*$.

Identical Parameters Recursion. An interesting tcc language considered in [40] arises from $\texttt{rec}_\texttt{p}$ by restricting the parameters not to change through recursive invocations. In the π-calculus this restriction does not cause any loss of expressive power since such form of recursion can encode general recursion (see [27]).

An example satisfying the above restriction is $R_P(\boldsymbol{x}) \stackrel{\texttt{def}}{=} P \parallel \mathbf{next}\, R_P(\boldsymbol{x})$. Here the actual parameters of the invocation in the body of the definition are the same as the formal parameters of R_P. An example not satisfying the restriction is $R'_P(\boldsymbol{x}) \stackrel{\texttt{def}}{=} P \parallel \mathbf{next}\,(\mathbf{local}\,\boldsymbol{x})\, R'_P(\boldsymbol{x})$. Here the actual parameters are bound and therefore different from those of the formal parameters.

One can formalize the identical parameters restriction on a set of mutually recursive definitions as follows. Suppose that $A_1 \rightsquigarrow A_2$ and $A_2 \rightsquigarrow^* A_1$ with $A_1(\boldsymbol{x}_1) \stackrel{\texttt{def}}{=} P_1$ and $A_2(\boldsymbol{x}_2) \stackrel{\texttt{def}}{=} P_2$ in the underlying set of definitions $\mathcal{D}$. Then for each invocation $A_2(\boldsymbol{y})$ in P_1 we should require $\boldsymbol{y} = \boldsymbol{x}_2$ and $\boldsymbol{y} \notin bv(P_1)$. In other words the actual parameters of the invocation A_2 in P_1 (i.e., $\boldsymbol{y}$) should be syntactically the same as its formal parameters (i.e., $\boldsymbol{x_2}$). Furthermore, they should not be bound in P_1 to avoid cases such as $R'_P(\boldsymbol{x})$ above.

The processes of tcc with identical parameters are those of $\texttt{rec}_\texttt{p}$ that satisfy this requirement. We shall refer to this language as $\texttt{rec}_\texttt{i}$.

Parameterless Recursion.

Tcc with parameterless recursion have been studied in [40]. All identifiers have arity zero, and hence, for convenience, we omit the "( )" in $A(\,)$.

Given a parameterless definition $A \stackrel{\texttt{def}}{=} P$, requiring all variables in $fv(P)$ to be formal parameters, as in $\texttt{rec}_\texttt{p}$, would mean that the body P has no free variables, and the resulting class of process languages would be expressively weak. Hence, we now suggest to allow free variables in procedure bodies.

Now. assuming that the operational rules for parameterless recursion are the same as for $\texttt{rec}_\texttt{p}$, what are the resulting scope rules for free variables in procedure bodies? Traditionally, one distinguishes between *dynamic* and *static* scoping, as illustrated in the following example.

Example 12. Consider a constant identifier A with the following definition

$$\begin{aligned} A \stackrel{\texttt{def}}{=}\ & \mathbf{tell}(x = 1) \\ & \parallel \mathbf{next}\,(\mathbf{local}\,x)\,(A \parallel \mathbf{when}\ x = 1\ \mathbf{do}\ \mathbf{tell}(z = 1)) \end{aligned}$$

In the case of dynamic scoping, an outside invocation A causes the execution $\mathbf{tell}(z = 1)$ in the second time interval. The reason is that $(\mathbf{local}\,x)$ binds the x resulting from the unfolding of the A inside the definition's body. In fact, the telling of $x = 1$, in the second time unit, will not be visible in the store. In the case of static scoping, $(\mathbf{local}\,x)$ does not bind the x of the unfolding of A because such an x is intuitively a "global" variable, and hence $\mathbf{tell}(z = 1)$ will not be executed. In fact, the telling of $x = 1$, will also be visible in the store in the second time interval. $\square$

Parameterless Recursion with Dynamic Scoping. The rule LOC in Table 1 combined with REC causes the parameterless recursion to have dynamic scoping[2]. As illustrated in the example below, the idea is that since $(\mathbf{local}\, x)\, P$ reduces to a process of the form $(\mathbf{local}\, x)\, Q$, the free occurrences of x in the unfolding of invocations in P get bounded.

Example 13. Consider A as defined in Example 12. Let us abbreviate the definition of A as $A \stackrel{\text{def}}{=} \mathbf{tell}(x = 1) \parallel P$. Also let $Q = \mathbf{skip} \parallel P$. We have the following reduction of $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\langle \mathbf{tell}(x=1), \mathtt{true}\rangle \longrightarrow \langle \mathbf{skip}, x=1\rangle}\ \mathrm{TELL}}{\langle \mathbf{tell}(x=1) \parallel P, \mathtt{true}\rangle \longrightarrow \langle Q, x=1\rangle}\ \mathrm{PAR}}{\langle A, \mathtt{true}\rangle \longrightarrow \langle Q, x=1\rangle}\ \mathrm{REC}}{\langle (\mathbf{local}\, x, \mathtt{true})\, A, \mathtt{true}\rangle \longrightarrow \langle (\mathbf{local}\, x, x=1)\, Q, \mathtt{true}\rangle}\ \mathrm{LOC}$$

Thus, $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$ reduces to $(\mathbf{local}\, x, x = 1)\, (\mathbf{skip} \parallel P)$ in store $\mathtt{true}$. Notice that the free x in A's body become local to $(\mathbf{local}\, x, x = 1)\, (\mathbf{skip} \parallel P)$, i.e, it now occurs in the local store but not in the global one. □

We shall refer to the language allowing only parameterless recursion with free-variables in the procedure bodies as $\mathtt{rec}_{\mathtt{d}}$; parameterless recursion with dynamic scoping.

Remark 2. It should be noticed that, unlike in $\mathtt{rec}_{\mathtt{p}}$, we cannot freely α-convert processes in $\mathtt{rec}_{\mathtt{d}}$ without changing behavior. For example, we could α-convert the process $(\mathbf{local}\, x)\, A$ in the above example into $(\mathbf{local}\, z)\, A$ (since $A[z/x]$ is syntactically equal to A) but the behavior of $(\mathbf{local}\, z)\, A$ would not be the same as that of $(\mathbf{local}\, x)\, A$.

Parameterless Recursion with Static Scoping. From the previous section it follows that static scoping as in [40] requires an alternative to the rule for local behavior LOC .

The rule LOC' defines locality for the parameterless recursion with static scoping language henceforth referred to as $\mathtt{rec}_{\mathtt{s}}$.

$$\mathrm{LOC}' \ \frac{\langle P[y/x], d\rangle \longrightarrow \langle P', d'\rangle}{\langle (\mathbf{local}\, x)\, P, d\rangle \longrightarrow \langle P', d'\rangle} \text{ if } y \text{ is fresh} \tag{12}$$

As in [24], we use the notion of *fresh variable* meaning that it does not occur elsewhere in a process definition or in the store. It will be convenient to presuppose that the set of variables $\mathcal{V}$ is partitioned into two infinite sets $\mathcal{F}$ and $\mathcal{V} - \mathcal{F}$. We shall assume that the fresh variables are taken from $\mathcal{F}$ and that no input from the environment or processes, other than the ones generated when applying LOC', can contain variables in $\mathcal{F}$.

The fresh variables introduced by LOC' are not to be visible from the outside. We hide these fresh variables, as suggested in [43], by using existential quantification in the

[2] Rules LOC and REC are basically the same in ccp, hence the observations made in this section regarding dynamic scoping apply to ccp as well.

output constraint of observable transitions. More precisely, we replace in Table 1 the rule for the observable transitions OBS with the rule

$$\mathrm{OBS}' \; \frac{\langle P, c\rangle \;\longrightarrow^*\; \langle Q, d\rangle \not\longrightarrow}{P \;\xRightarrow{(c, \exists_{\mathcal{F}} d)}\; F(Q)} \tag{13}$$

where $\exists_{\mathcal{F}} d$ represents the constraint resulting from the existential quantification in d of free occurrences of variables in $\mathcal{F}$.

In order to see why LOC' causes static scoping in $\mathtt{rec}_\mathtt{s}$, suppose that P in Rule LOC' in Equation 12 contains an invocation A where $A \stackrel{\text{def}}{=} R$. When replacing x with y in P, A remains the same since $A[y/x]$ is A. Furthermore, since y is chosen from $\mathcal{F}$, there will be no capture of free variables in R when unfolding A. This causes the scoping to be static. Let us illustrate this by revisiting the previous example.

Example 14. Let A, P and Q as in the previous example. We have the following reduction of $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \mathbf{tell}(x=1), \mathtt{true}\rangle \;\longrightarrow\; \langle \mathbf{skip}, x=1\rangle}\ \mathrm{TELL}}{\langle \mathbf{tell}(x=1) \parallel P, \mathtt{true}\rangle \;\longrightarrow\; \langle Q, x=1\rangle}\ \mathrm{PAR}}{\langle A, \mathtt{true}\rangle \longrightarrow \langle Q, x=1\rangle}\ \mathrm{REC}}{\langle (\mathbf{local}\, x)\, A, \mathtt{true}\rangle \;\longrightarrow\; \langle Q, x=1\rangle}\ \mathrm{LOC}'$$

Thus, $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$ reduces to $\mathbf{skip} \parallel P$ in store $(x = 1)$ making the free x in A's body visible in the "global" store . $\square$

Remark 3. Notice that, as in $\mathtt{rec}_\mathtt{d}$, in $\mathtt{rec}_\mathtt{s}$ we do not need α-conversion since in the reductions of $\mathtt{rec}_\mathtt{s}$ we only use syntactic replacements of variables by fresh variables.

5.3 Summary of TCC Languages

We have described five classes of tcc languages with infinite behabior, based on the literature. We adopt the following convention.

Convention 3 *We shall use $\mathcal{L}$ to designate the set of tcc languages*

$$\{\mathtt{rep}, \mathtt{rec}_\mathtt{p}, \mathtt{rec}_\mathtt{i}, \mathtt{rec}_\mathtt{d}, \mathtt{rec}_\mathtt{s}\}.$$

Furthermore, we shall index sets and relations involving tcc processes with the appropriate tcc language name to make it clear what is the language under consideration. We shall omit the index when it is unimportant or clear from the context.

For example, $\longrightarrow_{\mathtt{rec}_\mathtt{p}}$ and $\xRightarrow{(.,.)}_{\mathtt{rec}_\mathtt{p}}$ refer to the (internal and observable) reduction of $\mathtt{rec}_\mathtt{p}$. Similarly, $\mathit{Proc}_{\mathtt{rec}_\mathtt{p}}$ denotes the set of processes in $\mathtt{rec}_\mathtt{p}$, $\sim_{io}^{\mathtt{rec}_\mathtt{p}}$ denotes the input-output equivalence (Definition 10) for processes in $\mathit{Proc}_{\mathtt{rec}_\mathtt{p}}$, and $\approx_{io}^{\mathtt{rec}_\mathtt{p}}$ denotes congruence induced by $\sim_{io}^{\mathtt{rec}_\mathtt{p}}$.

5.4 The TCC Equivalences

In this section we relate the equivalences and their congruences for the various tcc languages. Each behavioral equivalence (and congruence) for a tcc language ℓ is obtained by taking the $\mathtt{ntcc}$ transitions given in Definition 9 (and thus in Definition 10) to be those of ℓ (i.e., replace $\xrightarrow{(.,.)}$ with $\xrightarrow{(.,.)}_{\ell}$).

The theorem below states the relationship among the equivalences.

Theorem 3 (Equivalence Results, [29]). *For each $\ell \in \mathcal{L}$,*

1. *If $\ell = \mathtt{rec_s}$ then $\approx^{\ell}_{io} = \approx^{\ell}_{o} \subset \sim^{\ell}_{io} \subset \sim^{\ell}_{o}$.*
2. *If $\ell \neq \mathtt{rec_s}$ then $\approx^{\ell}_{io} = \approx^{\ell}_{o} = \sim^{\ell}_{io} \subset \sim^{\ell}_{o}$.*

The theorem says the input-output and output congruences coincide for all languages. It also states that the input-output behavior is a congruence for every tcc language but $\mathtt{rec_s}$. This reveals a distinction between $\mathtt{rec_s}$ and the other tcc languages and, in fact, between $\mathtt{rec_s}$ and the standard model of concurrent constraint programming [45].

In the following sections we shall classify the tcc languages based on the decidability of their input-output equivalence.

5.5 Undecidability Results

In [29] it is shown that $\sim^{\mathtt{rec_p}}_{io}$ is undecidable for processes with an underlying finite-domain constraint system. Recall that a finite-domain constraint system $\mathbf{FD}[n]$ (see Definition 3) provides a theory of variables ranging over a finite domain of values $D = \{0, 1, \ldots, n-1\}$ with syntactic equality over these values. We shall also prove a stronger version of this result establishing that $\sim^{\mathtt{rec_p}}_{io}$ is undecidable even for the finite-domain constraint system with one single constant $\mathbf{FD}[1]$, i.e., $|D| = 1$. In sections 5.7 we shall give an input-output preserving constructive encoding from $\mathtt{rec_p}$ into the parameterless recursion language $\mathtt{rec_d}$, thus proving also the undecidability of $\sim^{\mathtt{rec_d}}_{io}$.

Theorem 4 (Undec. of $\sim^{\mathtt{rec_p}}_{io}$, [29]). *The problem of deciding given $P, Q \in Proc_{\mathtt{rec_p}}$ in a finite-domain constraint system, whether or not $P \sim^{\mathtt{rec_p}}_{io} Q$, is undecidable.*

We find it convinient to recall the proof of the above theorem given in [29] since it decribes very well the computational power of $\mathtt{rec_p}$. The proof is a reduction from Post's correspondence problem (PCP) [34].

Definition 17 (PCP). *A Post's Correspondence Problem (PCP)* instance *is a tuple (V, W), where $V = \{v_0, \ldots, v_n\}$ and $W = \{w_0, \ldots, w_n\}$ are two lists of non-empty words over the alphabet $\{0, 1\}$. A* solution *to this instance is a sequence of indexes $i_0, \ldots, i_m$ in $I = \{0, \ldots, n\}$ with $i_0 = 0$ s.t.*

$$v_{i_0}.v_{i_2} \ldots v_{i_m} = w_{i_0}.w_{i_2} \ldots w_{i_m}.$$

PCP is the following problem: given a PCP instance (V, W), does it have a solution?

The Post's Correspondence Problem is known to be undecidable [34]. We reduce PCP to the problem of deciding input-output equivalence between $\mathtt{rec_p}$ processes, thus proving Theorem 4.

The Post's Correspondence Problem Reduction. Let (V, W) be a PCP instance where $V = \{v_0, \ldots, v_n\}$ and $W = \{w_0, \ldots, w_n\}$ are sets of non-empty words. Let $\mathbf{FD}[m]$ (Definition 3) be the underlying constraint system where $m = max(|V|, 2)$ (i.e., we need at least two constants in the encoding below).

For each $i \in I = \{0, \ldots, |V| - 1\}$, we shall a define process $A_i(a, b, index, x)$ which intuitively behaves as follows:

1. It waits until is told that $a = 1$ to start writing v_i, one symbol per time unit. Each such a symbol, say s, will be written in x by telling $x = s$. Similarly, it waits until $b = 1$ to start writing w_i, one symbol per time unit. Each such a symbol will also be written in x.
2. It spawns a process $A_j(a', b', index, x)$ when the environment inputs an index $index = j$ in I.
3. It sets $a = 0$ and $a' = 1$ when it finishes writing v_i, i.e., $|v_i|$ time units later after it started writing v_i (this way it announces that its job of writing v_i is done, and allows A_j to start writing v_j). Similarly, it sets $b = 0$ and $b' = 1$ when it finishes writing w_i.
4. It aborts unless the environment provides an *index* in I. It also aborts if an inconsistency arises: Either two symbols (one from a V word and another from a W word) are written in x in the same time unit and they do not match (thus generating `false`), or the environment itself inputs `false`.

Thus, intuitively the A_i's keep writing V and W words, as the environment dictates, as long as the symbols match and the environment keeps providing indexes in I at each time unit.

Auxiliary Constructs We use the following constructs:

$$\begin{aligned}&W_{c,P}(\boldsymbol{x}) \stackrel{\texttt{def}}{=} \mathbf{when}\ c\ \mathbf{do}\ P \parallel \mathbf{unless}\ c\ \mathbf{next}\ W_{c,P}(\boldsymbol{x})\\ &R_Q(\boldsymbol{y}) \stackrel{\texttt{def}}{=} Q \parallel \mathbf{next}\ R_Q(\boldsymbol{y})\end{aligned}$$

where $fv(P) \cup fv(c) = \{\boldsymbol{x}\}$ and $fv(Q) = \{\boldsymbol{y}\}$. We use the more readable notation $\mathbf{wait}\ c\ \mathbf{do}\ P$ and $\mathbf{repeat}\ Q$ for $W_{c,P}(\boldsymbol{x})$ and $R_Q(\boldsymbol{y})$, respectively. We also define $\mathbf{whenever}\ c\ \mathbf{do}\ P$ as an abbreviation of $\mathbf{repeat}\ \ \mathbf{when}\ c\ \mathbf{do}\ P$.

We now define $A_i(a, b, index, x)$ for each $i \in I$ according to Items 1-4. The local variable *ichosen* is used as flag to check whether the environment inputs an index.

$$\begin{aligned}&A_i(a, b, index, x) \stackrel{\texttt{def}}{=} (\mathbf{local}\, a'\ b'\ ichosen)\,(\\ &\quad \mathbf{wait}\ a = 1\ \mathbf{do}\ V_i\\ &\parallel \mathbf{wait}\ b = 1\ \mathbf{do}\ W_i\\ &\parallel \textstyle\prod_{j \in I} \mathbf{when}\ index = j\ \mathbf{do}\ (\mathbf{tell}(ichosen = 1)\\ &\qquad\qquad\qquad\qquad \parallel \mathbf{next}\ A_j(a', b', index, x))\\ &\parallel Abort\ \ )\end{aligned}$$

The process V_i writes, one by one, the v_i symbols in x (notation $v_i(n)$ denotes the $n-$th element of v_i). Furthermore it sets $a = 0$ and $a' = 1$ when it finishes writing v_i. The process W_i is defined analogously.

$$V_i = \prod_{0 \le k < |v_i|} \mathbf{next}^{\,k}\mathbf{tell}(x = v_i(k)) \parallel \mathbf{next}^{\,|v_i|}(\mathbf{tell}(a = 0) \parallel \mathbf{tell}(a' = 1))$$

$$W_i = \prod_{0 \le k < |w_i|} \mathbf{next}^{\,k}\mathbf{tell}(x = w_i(k)) \parallel \mathbf{next}^{\,|w_i|}(\mathbf{tell}(b = 0) \parallel \mathbf{tell}(b' = 1))$$

The process $Abort$ aborts, according to Item 4 above, by telling `false` thereafter (thus creating a constant inconsistency).

$$\begin{array}{l} Abort = \\ \parallel \mathbf{unless}\ ichosen = 1\ \mathbf{next}\,\mathbf{repeat}\ \mathbf{tell}(\mathtt{false}) \\ \parallel \mathbf{when}\ \mathtt{false}\ \mathbf{do}\ \mathbf{repeat}\ \mathbf{tell}(\mathtt{false}) \end{array}$$

Let us now define a process $B_i(a, b, index, x, ok)$ for each $i \in I$ that behaves exactly like $A_i(a, b, index, x)$, but in addition it outputs $ok = 1$ whenever it stops writing v_i and w_i exactly in the same time interval. [3] This happens when both a and b are set to zero in the same unit and it will imply that a solution of the form $v_{i_0}. \dots .v_i = w_{i_0}. \dots .w_i$ for the PCP (V, W) has been found.

$$\begin{array}{l} B_i(a, b, index, x, ok) \stackrel{\mathtt{def}}{=} (\mathbf{local}\,a'\ b'\ ichosen)\,(\\ \ \ \mathbf{wait}\ a = 1\ \mathbf{do}\ V_i \\ \parallel \mathbf{wait}\ b = 1\ \mathbf{do}\ W_i \\ \parallel \prod_{j \in I} \mathbf{when}\ index = j\ \mathbf{do}\ (\mathbf{tell}(ichosen = 1) \\ \qquad\qquad\qquad\qquad \parallel \mathbf{next}\ B_j(a', b', index, x, ok)) \\ \parallel Abort \\ \parallel \mathbf{whenever}\ a = 0 \wedge b = 0\ \mathbf{do}\ \mathbf{tell}(ok = 1)) \end{array}$$

Since we require the first index in a solution for PCP (V, W) to be 0, we define two processes $A(index, x)$ and $B(index, x, ok)$ which trigger A_0 and B_0 as follows .

$$\begin{array}{l} A(index, x) \stackrel{\mathtt{def}}{=} (\mathbf{local}\,a\ b)\,(\\ \ \ \mathbf{tell}(a = 1) \parallel \mathbf{tell}(b = 1) \parallel A_0(a, b, index, x)) \end{array}$$

$$\begin{array}{l} B(index, x, ok) \stackrel{\mathtt{def}}{=} (\mathbf{local}\,a\ b)\,(\\ \ \ \mathbf{tell}(a = 1) \parallel \mathbf{tell}(b = 1) \parallel B_0(a, b, index, x, ok)) \end{array}$$

One can verify that the only difference between the processes $A(index, x)$ and $B(index, x, ok)$ is that the latter eventually tells $ok = 1$ iff there is a solution to the PCP (V, W).

Since the PCP problem is undecidable, from the lemma above it follows that given $P, Q \in Proc_{\mathtt{rec_p}}$ in a finite-domain constraint system, the question of whether $P \sim_{io}^{\mathtt{rec_p}} Q$ or not is undecidable. This proves Theorem 4. $\square$

.

[3] The reader may wonder why the A_i's do not have the formal parameter ok as well. This causes no problem here, but you can think of A as having a dummy ok formal parameter if you wish

Undecidability Over Fixed Finite-Domains

Actually [29] gives a stronger version of the above theorem; input-output equivalence in undecidable in $\mathtt{rec_p}$ even if we fix the underlying constraint system to be $\mathbf{FD}[1]$, which is the finite-domain constraint system whose only constant is 0.

Theorem 5 ([29]). *Fix* $\mathbf{FD}[1]$ *to be the underlying constraint system. The question of whether* $P \sim_{io}^{\mathtt{rec_p}} Q$ *or not is undecidable.*

From Theorems 5 and 3, we also have that the input-output and default output congruences are undecidable for $\mathtt{rec_p}$ over a fixed finite-domain constraint system.

Theorem 6. *The input-output and output congruences* $\approx_{io}^{\mathtt{rec_p}}$ *and* $\approx_{o}^{\mathtt{rec_p}}$ *are undecidable for processes in the finite-domain constraint system* $\mathbf{FD}[1]$.

Notice that $\mathbf{FD}[1]$ is a very simple constraint system (i.e., only equality and one single constant). So, the undecidability results for other constraint systems providing theories with equality and an at least one constant symbol follow from Theorem 5. This includes almost all constraint system of interest (e.g. the Herbrand constraint system [39], the Kahn constraint system [45], Enumerated Types [39] and modular arithmetic [32]).

5.6 Decidability Results

In sharp contrast to the undecidability result for $\mathtt{rec_p}$, the equivalence of `rep` processes is decidable even for *arbitrary constraint systems* [29].

Theorem 7. *The following equivalences for processes in* `rep` *over arbitrary constraint system are decidable:*

1. *The input-output equivalence* $\sim_{io}^{\mathtt{rep}}$, *default output equivalence* $\sim_{o}^{\mathtt{rep}}$ *and strongest-postcondition equivalence* $\sim_{sp}^{\mathtt{rep}}$.
2. *The output congruences* $\approx_{io}^{\mathtt{rep}}$ *and* $\approx_{o}^{\mathtt{rep}}$.

In section 5.7 we shall show via constructive encodings that `rep`, $\mathtt{rec_i}$, $\mathtt{rec_s}$ have the same expressive power. We then conclude that the corresponding equivalences for $\mathtt{rec_i}$ and $\mathtt{rec_s}$ are also decidable. These decidability results in `rep` with arbitrary constraint system are to be contrasted to the undecidability results in $\mathtt{rec_p}$ with the simple finite-domain constraint system $\mathbf{FD}[1]$.

5.7 Classification of the Timed CCP Languages

In this section we discuss the relation between the various tcc languages, and we classify them on the basis of their expressive power.

Figure 2 shows the sub-language inclusions and the encodings preserving the input-output behaviour between the various tcc versions. To complete the picture, we have included the class $\mathtt{rec_0}$ denoting the language with neither parameters nor free variables in the bodies of definitions. Classes I, II, III represent a partition based on the

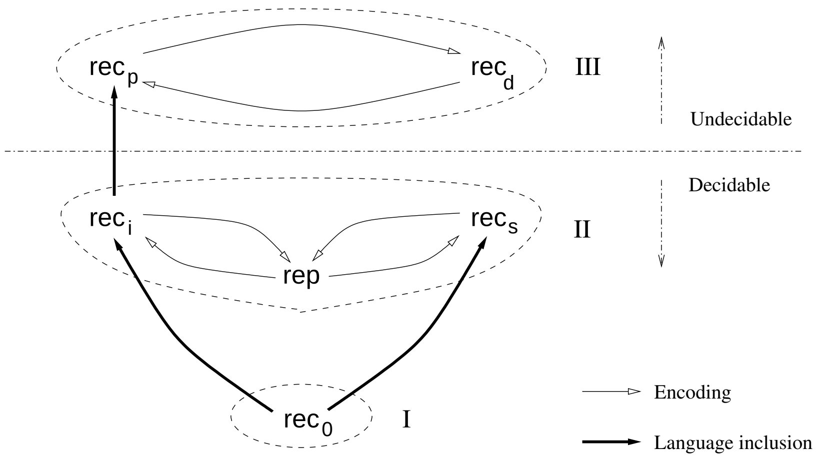

Figure2. Classification of the various tcc languages: The tcc hierarchy.

expressive power: two languages are in the same class if and only if they have the same expressive power. We will first discuss the separation results, and then the equivalences.

Given the input-output preserving encodings in [29], which we will recall in the next section, the separation between Classes II and III is already suggested by the results in Sections 5.6 and 5.5. From the proof of Theorem 4 it follows that $\mathtt{rec}_\mathtt{p}$ is capable of expressing the "behavior" of Post's correspondence problems, and hence clearly capable of expressing output behavior not accepted by Büchi automata. It turns out that the output (and input-ouput) behavior of every process in `rep` can be represented as a language accepted by a Büchi automata [29].

The separation between Classes I and II, on the other hand, follows from the fact that without parameters or free variables the recursive calls cannot communicate with the external environment, hence in $\mathtt{rec}_0$ a process can produce information on variables for a finite number of time intervals only.

The Encodings

Let us recall briefly the input-output preserving encodings among the various tcc languages in [29]. Henceforth, $[\![\cdot]\!] : \ell \to \ell'$ will represent the encoding function from class ℓ to class ℓ' We shall say that $[\![\cdot]\!]$ is *homomorphic* wrt to the parallel operator if $[\![P \parallel Q]\!] = [\![P]\!] \parallel [\![Q]\!]$, and similarly for the other operators.

.

Notation 3 *We shall use the following notation:*

- *We use* $\mathit{call}(x)$ *as abbreviation of* $x = 1$ *and declare, for each identifier* A*, a fresh variable* z_A *uniquely associated to it.*
- *We denote by* $I(P)$ *the set of identifiers on which* P *depends, i.e. the transitive closure of* $\leadsto$ *of the identifiers occurring in* P *(see Section 5.2).*

– *We often use $\mathcal{D}_\ell$ to denote the set of recursive definitions under consideration for processes in ℓ. As usual we omit ℓ when it is clear from the context.*

***Encoding* $\mathbf{rec_s} \rightarrow \mathbf{rep}$.** Here the idea is to simulate a procedure definition by a replicated process that activates (the encoding of) its body P each time it is called. The activation can be done by using a construct of the form $\mathbf{when}\ c\ \mathbf{do}\ P$. The call, of course, will be simulated by $\mathbf{tell}(c)$.

The key case is the local operator, since we do not want to capture the free variables in the bodies of procedures. Thus, we need to α-convert by renaming the local variables with fresh variables.

First we need two auxiliary encodings $[\![\cdot]\!]_{\mathcal{D}}$ and $[\![\cdot]\!]_0$: given by :

$$[\![A \stackrel{\mathtt{def}}{=} P]\!]_{\mathcal{D}} = !\mathbf{when}\ \mathit{call}(z_A)\ \mathbf{do}\ [\![P]\!]_0$$

$$[\![A]\!]_0 = \mathbf{tell}(\mathit{call}(z_A))$$

$$[\![(\mathbf{local}\,x)\,P]\!]_0 = (\mathbf{local}\,y)\,([\![P[y/x]]\!]_0) \quad \text{where } y \text{ is fresh}$$

with $[\![\cdot]\!]_0$ being homomorphic on all the other operators of $\mathtt{rec_s}$.

We are now ready to give our encoding of $\mathtt{rec_s}$ into $\mathtt{rep}$.

Definition 18. *The encoding $[\![\cdot]\!] : \mathtt{rec_s} \rightarrow \mathtt{rep}$ is given by:*

$$[\![A]\!] = (\mathbf{local}\,\boldsymbol{z})\,([\![P]\!]_0 \parallel \prod_{i=1}^{n} [\![A_i(\boldsymbol{x}_i) \stackrel{\mathtt{def}}{=} P_i]\!]_{\mathcal{D}})$$

with $I(P) = \{A_1, \ldots, A_n\}$ and $\boldsymbol{z} = z_{A_1} \ldots z_{A_n}$.

***Encoding* $\mathbf{rec_i} \rightarrow \mathbf{rep}$.** This encoding is similar to the encoding in the previous section, except that now we need to encode the passing of parameters as well. Let us give some intuition first.

A call $A(\boldsymbol{y})$, where $A(\boldsymbol{x}) \stackrel{\mathtt{def}}{=} P$, can occur in a process or in the definition of identifier B (possibly A itself). Consider the case in which there is no mutual dependency between A and B or A is a call in a process. Then, the actual parameters of A may be different from the formal ones (i.e., $\boldsymbol{y} \neq \boldsymbol{x}$). If so, we need to model the call by providing a copy of the replicated process that encodes the definition of A and by making the appropriate parameter replacements.

Now, consider the case in which there is a mutual dependency between A and B (i.e. if also A depends on B). From the restriction imposed on (the mutual) recursion of $\mathtt{rec_i}$ (see Section 5.2), we know that the actual parameters must coincide with the formal ones (i.e., $\boldsymbol{y} = \boldsymbol{x}$) and therefore we do not need to make any parameter replacement. Neither do we need to provide a copy of the replicated processes as it will be available at the top level.

As for the previous encoding, we first define the auxiliary encodings $[\![\cdot]\!]_{\mathcal{D}}$ and $[\![\cdot]\!]_0$:

$$[\![A(\boldsymbol{x}) \stackrel{\text{def}}{=} P]\!]_{\mathcal{D}} = !\mathbf{when}\ \mathit{call}(z_A)\ \mathbf{do}\ [\![P]\!]_0$$

$$\begin{aligned}[\![A(\boldsymbol{y})]\!]_0 &= \mathbf{tell}(\mathit{call}(z_A)) \\ &\quad \text{if } \boldsymbol{y} = \boldsymbol{x} \text{ and } A(\boldsymbol{x}) \stackrel{\text{def}}{=} P \in \mathcal{D}\end{aligned}$$

$$\begin{aligned}[\![A(\boldsymbol{y})]\!]_0 &= (\mathbf{local}\, z_A)\,(\\ &\quad \mathbf{tell}(\mathit{call}(z_A)) \parallel [\![A(\boldsymbol{x}) \stackrel{\text{def}}{=} (P[\boldsymbol{y}/\boldsymbol{x}])]\!]_{\mathcal{D}}) \\ &\quad \text{if } \boldsymbol{y} \neq \boldsymbol{x} \text{ and } A(\boldsymbol{x}) \stackrel{\text{def}}{=} P \in \mathcal{D}\end{aligned}$$

with $[\![\cdot]\!]_0$ homomorphic on all the other operators of $\texttt{rec}_\texttt{i}$.

It worth noticing that if we did not have the restriction on the recursion in $\texttt{rec}_\texttt{i}$ mentioned above, the encoding $[\![.]\!]_{\mathcal{D}}$ would not be well-defined. E.g., consider the definition $A(x) \stackrel{\text{def}}{=} \mathbf{next}\ (\mathbf{local}\, y)\ A(y)$ which violates the restriction, and try to compute $[\![A(x) \stackrel{\text{def}}{=} (\mathbf{local}\, y)\ A(y)]\!]_{\mathcal{D}}$.

We are now ready to give our encoding of $\texttt{rec}_\texttt{i}$ into $\texttt{rep}$.

Definition 19. *The encoding* $[\![\cdot]\!] : \texttt{rec}_\texttt{i} \to \texttt{rep}$ *is given by:*

$$[\![A(\boldsymbol{y})]\!] = (\mathbf{local}\, \boldsymbol{z})\,([\![P]\!]_0 \parallel \prod_{i=1}^{n} [\![A_i(\boldsymbol{x}_i) \stackrel{\text{def}}{=} P_i]\!]_{\mathcal{D}})$$

with $I(P) = \{A_1, \ldots, A_n\}$ *and* $\boldsymbol{z} = z_{A_1} \ldots z_{A_n}$.

Encoding $\texttt{rep} \to \texttt{rec}_\texttt{i}$**.** This encoding is rather simple. The idea is to replace $!\,P$ by a call to a new process identifier R_P, defined as a process that expands P and then calls itself recursively in the next time interval. The free variables of $!\,P$, $\boldsymbol{x}$, are passed as (identical) parameters.

Definition 20. *The encoding* $[\![\cdot]\!] : \texttt{rep} \to \texttt{rec}_\texttt{i}$ *is given by:*

$$\begin{aligned}[\![!\,P]\!] &= R_P(\boldsymbol{x}) \\ &\quad \textit{where } R_P(\boldsymbol{x}) \stackrel{\text{def}}{=} [\![P]\!] \parallel \mathbf{next}\ R_P \in \mathcal{D}_{\texttt{rec}_\texttt{i}}, \boldsymbol{x} = \mathit{fv}(P).\end{aligned}$$

with $[\![\cdot]\!]$ *homomorphic on all the other operators of* $\texttt{rep}$.

Encoding $\texttt{rec}_\texttt{d} \to \texttt{rec}_\texttt{p}$**.** Intuitively, if the free variables are treated dynamically, then they could equivalently be passed as parameters.

Definition 21. *The encoding* $[\![\cdot]\!] : \texttt{rec}_\texttt{d} \to \texttt{rec}_\texttt{p}$ *is given by*

$$\begin{aligned}[\![A]\!] &= A(\boldsymbol{x}) \\ &\quad \textit{where } A \stackrel{\text{def}}{=} P \in \mathcal{D}_{\texttt{rec}_\texttt{d}} \\ &\quad \textit{and } A(\boldsymbol{x}) \stackrel{\text{def}}{=} [\![P]\!] \in \mathcal{D}_{\texttt{rec}_\texttt{p}}, \boldsymbol{x} = \mathit{fv}(P)\end{aligned}$$

with $[\![\cdot]\!]$ *homomorphic on all the other operators of* $\texttt{rec}_\texttt{d}$

***Encoding* $\mathbf{rec_p} \rightarrow \mathbf{rec_d}$.** The idea is to establish the link between the formal parameters $\boldsymbol{x}$ and the actual parameters $\boldsymbol{y}$ by telling the constraint $\boldsymbol{x} = \boldsymbol{y}$. However, this operation has to be encapsulated within a $(\mathbf{local}\, \boldsymbol{x})$ in order to avoid confusion with other potential occurrences of $\boldsymbol{x}$ in the same context of the call.

Definition 22. *The encoding* $[\![\cdot]\!] : \mathtt{rec_p} \rightarrow \mathtt{rec_d}$ *is given by*

$$[\![A(\boldsymbol{y})]\!] = (\mathbf{local}\, \boldsymbol{x})\, (A \parallel E_{\boldsymbol{y}/\boldsymbol{x}})$$
$$\textit{where } A(\boldsymbol{x}) \stackrel{\mathtt{def}}{=} P \in \mathcal{D}_{\mathtt{rec_p}},\ A \stackrel{\mathtt{def}}{=} [\![P]\!] \in \mathcal{D}_{\mathtt{rec_d}},$$
$$\textit{and } \ E_{\boldsymbol{y}/\boldsymbol{x}} \stackrel{\mathtt{def}}{=} \mathbf{tell}(\boldsymbol{y} = \boldsymbol{x}) \parallel \mathbf{next}\, E_{\boldsymbol{y}/\boldsymbol{x}} \in \mathcal{D}_{\mathtt{rec_d}}$$

with $[\![\cdot]\!]$ *homomorphic on all the other operators of* $\mathtt{rec_d}$.

***Encoding* $\mathbf{rep} \rightarrow \mathbf{rec_s}$.** Here we take advantage of the automata representation of the input-output behavior of $\mathtt{rep}$ processes given in [29]. Basically, the idea is to use the recursive definitions as equations describing these input-output automata.

Let P be an arbitrary process in $\mathtt{rep}$. Let us recall the automaton $M_P = A_P^{io}$ in [29] representing the input-output behavior of P on the inputs of relevance for P. The start state of M_P is P. Let T_P be the set of transitions of M_P. Each transition from Q to R with label (c, d), written $\langle Q, (c, d), R\rangle \in T_P$, represents an observable transition $Q \xRightarrow{(c,d)} R$.

So, for each state Q of M_P we define an identifier A_Q as follows:

$$A_Q \stackrel{\mathtt{def}}{=} \prod_{\langle Q,(c,d),R\rangle \in T_P} \mathbf{when}\ c\ \mathbf{do}\ (\mathbf{tell}(d) \parallel O(\sqcup c, R))$$
$$\text{with } \sqcup c = \bigvee_{e \in \{c' \mid c' \neq c,\ c' \models c,\ \langle Q,(c',d'),R'\rangle \in T_P\}} e$$

where $O(\sqcup c, R)$ takes the form $\mathbf{unless}\ \sqcup c\ \mathbf{next}\ A_R$ if $c \neq \mathtt{false}$, otherwise it takes the form $\mathbf{next}\ A_R$.

Intuitively, A_Q expresses that if we are in state Q and c is the strongest constraint entailed by the input, then the next state will be R and the output will be d, with $\langle Q, (c, d), R\rangle \in T_P$.

Definition 23. *The encoding* $[\![\cdot]\!] : \mathtt{rep} \rightarrow \mathtt{rec_s}$ *is defined as* $[\![P]\!] = A_P$.

6 Related Work and Concluding Remarks

Saraswat el al proposed a proof system for tcc [40], based on an intuitionistic logic enriched with a next operator. The system is complete for hiding-free and finite processes. Also Gabrielli et al [4] introduced a proof system for the tccp model (see Section 3). 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 [22].

The decidability results for the ntcc equivalences here presented are based on reductions from ntcc processes into finite-state automata [31, 29, 52]. The work in [43] also shows how to compile tcc into finite-state machines thus providing an execution model of tcc.

In [49] Tini explores the expressiveness of tcc languages, focusing on the capability of tcc to encode synchronous languages. In particular, Tini shows that Argos [23] and a version of Lustre restricted to finite domains [16] can be encoded in tcc.

In the context of tcc, Tini [50] introduced a notion of bisimilarity with a complete and elegant axiomatization for the hiding-free fragment of tcc. The notion of bisimilarity has also been introduced for `ntcc` by Valencia in his PhD thesis [51].

On the practical side, Saraswat el al introduced Timed Gentzen [41], a particular tcc-based programming language for reactive-systems implemented in PROLOG. More recently, Saraswat el al released jcc [44], an integration of timed (default) ccp into the JAVA programming language. Rueda et al [38] demonstrated that essential ideas of computer generated music composition can be elegantly represented in `ntcc`. Hurtado and Muñoz [20] 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 [21]—the robotic devices chosen in Section 4 as motivating examples.

Future Work. Timed ccp is still under development and certainly much remain to be explored. In order to contribute to the development of timed ccp as a well-established model of concurrency, a good research strategy could be to address those issues that are central to other mature models of concurrency. In particular, the analysis and formalization of the `ntcc` behavioral equivalences, which at present time are still very immature (e.g., axiomatizations of process equivalences and automatic tools for behavioral analysis).

Furthermore, the decision algorithms for `ntcc` verification and satisfiability, are very inefficient, and of theoretical interest only. For practical purposes, it is important to conduct studies on the design and implementation of efficient algorithms for verification.

Acknowledgments. We owe much to Catuscia Palamidessi for her contributions to the development of the `ntcc` calculus.

References

1. M. Abadi. The power of temporal proofs. *Theoretical Computer Science*, 65:35–84, 1989.
2. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
3. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Cong. on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.
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. P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in `cc(fd)`. In *Proceedings of the 2000 ACM symposium on Applied computing 2000*, volume 910 of *LNCS*. Springer-Verlag, 1995.
19. C. A. R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.
20. 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.
21. H. H. Lund and L. Pagliarini. Robot soccer with LEGO mindstorms. *LNCS*, 1604:141–151, 1999.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer, 1991.
23. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR '92*, volume 630 of *LNCS*, pages 550–564. Springer-Verlag, 1992.
24. N.P. Mendler, P. Panangaden, P. J. Scott, and R. A. G. Seely. A logical view of concurrent constraint programming. *Nordic Journal of Computing*, 2(2):181–220, 1995.
25. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.
26. R. Milner. A finite delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1992.
27. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
28. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7, 1974.
29. 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.
30. M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(2):145–188, 2002.
31. M. Nielsen and F. Valencia. *Temporal Concurrent Constraint Programming: Applications and Behavior*, chapter 4, pages 298–324. Springer-Verlag, LNCS 2300, 2002.

32. C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proc. of CP'01*. Springer-Verlag, LNCS 2239, 2001.
33. C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress '62*, 1962.
34. E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.
35. G.M. Reed and A.W. Roscoe. A timed model for communication sequential processes. *Theoretical Computer Science*, 8:249–261, 1988.
36. F. Rossi and U. Montanari. Concurrent semantics for concurrent constraint programming. In *Constraint Programming: Proc. 1993 NATO ASI*, pages 181–220, 1994.
37. J.H. Réty. Distributed concurrent constraint programming. *Fundamenta Informaticae*, 34(3), 1998.
38. 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.
39. V. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.
40. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*, pages 71–80, 1994.
41. 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.
42. V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proc. of POPL'95*, pages 272–285, 1995.
43. V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, 1996.
44. 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.
45. V. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91*, pages 333–352, 1991.
46. E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510, 1990.
47. G. Smolka. A Foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *LNCS*, 1994. Invited Talk.
48. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.
49. S. Tini. On the expressiveness of timed concurrent constraint programming. *Electronics Notes in Theoretical Computer Science*, 1999.
50. S. Tini. An axiomatic semantics for the synchronous language gentzen. In *FOSSACS'01*, volume 2030 of *LNCS*. Springer-Verlag, 2001.
51. F. Valencia. *Temporal Concurrent Constraint Programming*. PhD thesis, BRICS, University of Aarhus, 2003.
52. F. Valencia. Timed concurrent constraint programming: Decidability results and their application to LTL. In *Proc. of ICLP'03*. Springer-Verlag, LNCS, 2003.
53. W. Yi. *A Calculus for Real Time Systems*. PhD thesis, Chalmers Institute of Technology, Sweden, 1991.
54. 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.