Decidability of Infinite-State Timed CCP Processes and First-Order LTL

Frank D. Valencia [1]

Dept. of Information Technology
Uppsala University
Box 337 SE-751 05 Uppsala
SWEDEN

Abstract

The `ntcc` process calculus is a timed *concurrent constraint programming* (ccp) model equipped with a first-order *linear-temporal logic* (LTL) for expressing process specifications. A typical behavioral observation in ccp is the *strongest postcondition (sp).* The `ntcc` sp denotes the set of all infinite output sequences that a given process can exhibit. The *verification problem* is then whether the sequences in the sp of a given process satisfy a given `ntcc` LTL formula.

This paper presents new positive decidability results for timed ccp as well as for LTL. In particular, we shall prove that the following problems are decidable: (1) The *sp equivalence* for the so-called *locally-independent* `ntcc` fragment; unlike other fragments for which similar results have been published, this fragment can specify *infinite-state systems.* (2) *Verification* for locally-independent processes and negation-free *first-order* formulae of the `ntcc` LTL. (3) *Implication* for such formulae. (4) *Satisfiability* for a first-order fragment of Manna and Pnueli's LTL. The purpose of the last result is to illustrate the applicability of ccp to well-established formalisms for concurrency.

Key words: Process Calculi, Timed Concurrent Constraint Programming, Infinite-State Systems, Temporal Logic, First-Order LTL, Decidability.

1 Introduction

The notion of *constraint* is certainly not rare in concurrency. After all, concurrency is about the interaction of agents and such an interaction often involves

[1] Supported by PROFUNDIS FET-Global Computing Project

constraints of some sort (e.g., synchronization constraints, access-control, actions that must eventually happen, actions that cannot happen, etc).

Saraswat's *concurrent constraint programming* (ccp) [37] is a well-established formalism for concurrency based upon the shared-variables communication model where interaction arises via constraint-imposition over shared-variables. In ccp, agents can interact by *adding* (or *telling*) partial information in a medium, a so-called *store*. Partial 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 [9,30,12], stochastic behavior [13], and most prominently *time* [33,5,35,14]. *Timed* ccp extends ccp by allowing agents to be constrained by time requirements and it has been studied extensively as a model for reactive systems [5,11,25–28,33,34,36,41]. A very distinctive feature of timed ccp is that it combines in one framework an *operational and algebraic* view based upon process calculi with a *declarative* view based upon temporal logic. So, processes can be treated as computing agents, algebraic terms and temporal formulae. At this point it is convenient to quote Robin Milner:

> *I make no claim that everything can be done by algebra ... It is perhaps equally true that not everything can be done by logic; thus one of the outstanding challenges in concurrency is to find the right marriage between logic and behavioral approaches.*
>
> — Robin Milner, [20]

In this paper we shall see that the combination in one framework of the alternative views of processes mentioned above allows timed ccp to benefit from the large body of techniques of well established theories used in the study of concurrency. For instance, we shall work with finite-state automata representations; a classic technique used in temporal logic. Furthermore, the combination may allow timed ccp to be used for proving new results for these theories. In fact, by using results for timed ccp proved in this paper, we also prove new decidability results for the standard linear-time temporal logic (LTL) [18].

1.1 Contributions.

The `ntcc` process calculus [26] is a generalization of the timed ccp model tcc [33]. The calculus can represent timed concepts such as unit delays, unbounded finite delays, time-outs, pre-emption, synchrony and asynchrony. Furthermore, `ntcc` is equipped with an LTL to specify timed properties and with an *inference system* for the verification problem (i.e., for proving whether a given process fulfills a given LTL specification).

In this paper we shall present new decidability results for infinite-state `ntcc` processes, the `ntcc` LTL (here called *constraint* LTL or **CLTL** for short), and for the standard first-order LTL (here called **LTL** for short) described by Manna and Pnueli in [18]. The description and relevance of these results are outlined next:

- *On the sp equivalence and verification problem.* The strongest-postcondition (sp) behavior of a given `ntcc` process P denotes the set of all infinite sequences of outputs that P can exhibit. Thus, P fulfills a given specification (i.e., a **CLTL** formula) F iff each sequence in its sp satisfies F. In Section 4, we show that for a substantial fragment of `ntcc` and the negation-free first-order fragment of **CLTL** : (1) *the sp equivalence is decidable* and (2) *the verification problem is decidable.*
  - · A noteworthy aspect of these two results is that the `ntcc` fragment above admits *infinite-state processes.* All other `ntcc` fragments for which similar results have been published [25,27] are restricted to finite-state processes.
  - · Another noteworthy aspect is that **CLTL** is first-order. Most first-order LTLs in computer science are not recursively axiomatizable let alone decidable [1].
- *On the* `ntcc` *LTL.* In Section 4 we prove that: (3) *the validity of implication is decidable for the negation-free first-order fragment of* **CLTL**.
  - · As for Hoare logic, the `ntcc` inference system [26] mentioned above has the so-called consequence rule which queries an oracle about (**CLTL**) implication. This causes the completeness of the system to be relative to the capability of determining the validity of implication, thus making our third result of relevance to `ntcc`.
  - · As a corollary of this result, we obtain the decidability of *satisfiability* for the negation-free first-order fragment of **CLTL**. This is relevant for specification purposes since, as remarked in [44], a specification is "interesting" only if it is satisfiable.
- *On the standard first-order LTL.* In Section 5 we prove that: (4) *the satisfiability problem in* **LTL** *is decidable for all negation-free first-order formulae without rigid variables.* This result is obtained from a reduction to **CLTL** satisfiability.
  - · Since first-order **LTL** is not recursively axiomatizable [1], satisfiability is undecidable for the full language of **LTL** . Recent work [17] and also [19], however, have taken up the task of identifying first-order decidable fragments of **LTL** . Our fourth result contributes to this task.
  - · The reduction from the standard **LTL** satisfiability to **CLTL** satisfiability also contributes to the understanding of the relationship between (timed) ccp and (temporal) classic logic.

In brief, this paper argues for timed ccp as a convenient framework for reactive systems by providing positive decidability results for behavior, specification and verification (1–3), and by illustrating its applicability to the well-established theory of **LTL** (4). This paper is the extended and revised version of [43].

1.2 Organization

The paper is organized as follows. Section 2 gives a brief introduction to the basic principles and central developments of ccp and timed ccp. Section 3 describes in detail the timed ccp model *ntcc*. Section 4 presents the decidability results for ntcc. Finally, as an application of the results in Section 4, Section 5 shows the decidability result for **LTL**.

2 Background

2.1 Concurrent Constraint Programming

In his seminal PhD thesis [32], 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 [23] leading to constraint programming and Shapiro's concurrent logic programming [38]. The ccp model has received a significant theoretical and implementational attention: The works in [37] and [6] gave a fixed-point denotational semantics to ccp, whilst [29] gave a (true-concurrent) Petri-Net semantics (using the formalism of contextual nets); in [7] the authors developed an inference system for proving properties of ccp processes; Oz [40] as well as AKL [16] 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 [32] 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 2.1 *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*

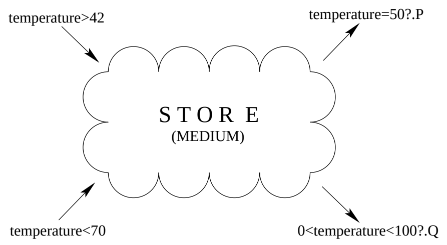

Figure 1. A simple ccp scenario

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:

- *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 [37].

2.2 Timed Concurrent Constraint Programming

The first timed ccp model was introduced in [33] 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 [33] 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 [36], 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] 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]).

The `ntcc` process calculus [26] is a generalization of the tcc model. 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 [26] and music applications [31].

The major difference between the 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 the `ntcc` calculus, tccp provides for arbitrary recursion and does not have an operator for specifying unbounded but finite delays.

In this paper we shall work with the generalization of tcc, the `ntcc` calculus which is described in detail in the next section.

3 The `ntcc` Calculus: Syntax and Operational Semantics

In `ntcc`, time is conceptually divided into *discrete intervals (or time units).* In a particular timed interval, a process P gets an input (an item of information represented as a *constraint*) c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines a residual process Q, which is then executed in the next time interval. In the rest of this section we shall recall `ntcc` concepts given in [26].

3.1 Constraint Systems

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

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

c, e.g., $(42 < \texttt{tsensor} < 100) \models (0 < \texttt{tsensor} < 120)$.

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

Definition 3.1 (Constraint Systems) *A* constraint system *is a pair* (Σ, Δ) *where* Σ *is a signature specifying constants, functions and predicate symbols, and* Δ *is a consistent first-order theory over* Σ *(i.e., a set of first-order sentences over* Σ *having at least one model).*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$. Here $\mathcal{V}$ is a countable set of variables $x, y, \ldots$, and $\mathcal{S}$ is the set of logic symbols $\neg, \wedge, \vee, \Rightarrow, \exists, \forall, \texttt{true}$ and $\texttt{false}$ which denote logical negation, conjunction, disjunction, implication, existential and universal quantification, and the always true and always false predicates, respectively. *Constraints,* denoted by $c, d, \ldots$, are first-order formulae over $\mathcal{L}$. We say that c *entails* d in Δ, written $c \models_\Delta d$ iff the formula $c \Rightarrow d$ is true in all models of Δ. We write $\models$ instead of $\models_\Delta$ when Δ is unimportant or can be inferred from the context. For operational reasons, *we shall require* $\models$ *to be decidable.* We say that c is equivalent to d, written $c \approx d$, iff $c \models d$ and $d \models c$.

Convention 3.2 *Henceforth,* $\mathcal{C}$ *denotes the set of constraints modulo* $\approx$ *under consideration in the underlying constraint system. So, we write* $c = d$ *iff they have the same representative in* $\mathcal{C}$*.*

The classical example of a constraint system is that of Herbrand (or finite trees) [32]:

Example 3.3 (Herbrand) *The Herbrand constraint system is such that:*

- Σ *is the set with infinitely many function symbols of each arity and equality* $=$.
- Δ *is given by Clark's Equality Theory with the schemas*

$$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$$

$$f(x_1, \ldots, x_n) = g(y_1, \ldots, y_n) \Rightarrow \texttt{false}, \text{ if } f, g \text{ are distinct symbols}$$

$$x = f(\ldots x \ldots) \Rightarrow \texttt{false}\,.$$

The importance of the Herbrand constraint system is that it underlies conventional logic programming and most first-order theorem provers. Its value lies in the Herbrand Theorem, which reduces the problem of checking unsatisfiabil-

ity of a first-order formula to the unsatisfiability of a quantifier-free formula interpreted over finite trees. □

3.2 Process Syntax

The process constructions in the `ntcc` calculus are given by the following syntax:

Definition 3.4 (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:*

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

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

The process $P \parallel Q$ represents the *parallel execution* of P and Q. The product $\prod_{i \in I} P_i$, where $I = \{i_1, \ldots, i_n\}$, denotes $((P_{i_1} \parallel P_{i_2}) \parallel \ldots P_{i_{n-1}}) \parallel P_{i_n}$.

The process $(\mathbf{local}\, x)\, P$ declares an x *local* to P, and thus we say that it *binds* x in P. The *bound variables* $bv(Q)$ (*free variables* $fv(Q)$) are those with a bound (a not bound) occurrence in Q.

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

The operator "$\star$" represents an *arbitrary (or unknown) but finite delay* (as "ϵ" in SCCS [21]) and allows asynchronous behavior across the time intervals. Intuitively, $\star\, P$ means $P + \mathbf{next}\, P + \mathbf{next}^2 P + \ldots$, i.e., an unbounded finite delay of P. The *replication* operator "!" is a delayed version of that of the π-calculus [22]: $!\, P$ means $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e., unboundedly many copies of P but one at a time.

For technical and unification purposes we add to the syntax of `ntcc` in [26] the tcc process **abort** [33] which causes all interactions with the environment to cease.

We conclude our informal description of processes with a simple example.

Example 3.5 *The following process repeatedly checks the state of* $\mathtt{motor}_1$*:*

$$R = !\,\textbf{when}\ \mathit{malfunction}(\mathit{motor}_1\_\mathit{status})\ \textbf{do}\ \textbf{tell}(\mathit{motor}_1\_\mathit{speed} = 0)$$

If a malfunction is reported, R *tells that* $\mathtt{motor}_1$ *must stop. Thus, in* $R \parallel S$*, where* $S = \star\,\textbf{tell}(\mathit{malfunction}(\mathit{motor}_1\_\mathit{status}))$*,* $\mathtt{motor}_1$ *must eventually stop.* $\square$

3.3 Structural Operational Semantics.

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

Intuitively, the relation $\equiv$ describes irrelevant syntactic aspects of processes. Its definition basically states that $(\mathit{Proc}/\equiv, \parallel, \textbf{skip})$ is a commutative monoid.

Definition 3.6 (Structural Congruence) *Let* $\equiv$ *be the smallest congruence over processes satisfying the following axioms:*

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

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

Convention 3.7 *We extend the syntax with a construct* $\textbf{local}\,(x, d)\,\textbf{in}\,P$*, to represent the evolution of a process of the form* $\textbf{local}\,x\,\textbf{in}\,Q$*, where* d *is the local information (or store) produced during this evolution. Initially* d *is "empty", so we regard* $(\textbf{local}\,x)\ P$ *as* $(\textbf{local}\,x, \mathtt{true})\ P$*.*

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".

Table 1
Rules for internal reduction $\longrightarrow$ (upper part) and observable reduction $\Longrightarrow$ (lower part). The assertion $\gamma \not\longrightarrow$ in OBS holds iff for no γ', $\gamma \longrightarrow \gamma'$. The relation $\equiv$ and F are given in Definitions 3.6 and 3.8, respectively.

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

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

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

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

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

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

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. The store d is not automatically transferred to the next time interval. Information in d can only be transfered to the next time unit by P itself.

We shall only describe some of the rules in Table 1 (see [26] for further details). As clarified below, the seemingly missing cases for "next" and "unless" processes are given by OBS. The rule STAR specifies an arbitrary delay of P. REP says that $!P$ creates a copy of P and then persists in the next time unit. ABORT realizes the intuition of **abort** causing the interactions with the environment to cease by generating infinite sequences of internal transitions.

Let us dwell a little upon the description of Rule LOC as it may seem somewhat complex. Let us consider the process

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

in Rule LOC. The global store is d and the local store is c. We distinguish between the *external* (corresponding to Q) and the *internal* point of view (corresponding to P). From the internal point of view, the information about x, possibly appearing in the "global" store d, cannot be observed. Thus, before reducing P we should first hide the information about x that Q may have in d. We can do this by existentially quantifying x in d. Similarly, from the external point of view, the new 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). The process $F(Q)$ is obtained by removing from Q summations that did not trigger activity and any local information which has been stored in Q, and by "unfolding" the sub-terms within "next" and "unless" expressions.

Definition 3.8 (Future Function) *Let $F : Proc \rightharpoonup Proc$ be defined by*

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

Remark 3.9 *The function F need no to be total since whenever we need to apply F to a Q (*OBS *in Table 1), every process of the form* $\mathbf{tell}(c)$, $\mathbf{abort}$, *$\star R$ and $!\, R$ in Q will occur within a "next" or "unless" expression.*

We conclude this section by revisiting our previous process example to illustrate a sequence of observable transitions.

Example 3.10 *Recall Example 3.5. We had $!\, \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e)$ in parallel with the process $\star \mathbf{tell}(c)$ where $c = \mathtt{malfunction}(\mathtt{motor_1\_status})$ and $e = (\mathtt{motor_1\_speed} = 0)$. For any* arbitrary $m > 0$, *the following is a valid sequence of observable transitions:*

$$\begin{array}{lcl} \star\mathbf{tell}(c) \parallel !\, \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) & \xRightarrow{(c, c\wedge e)} & \mathbf{next}^{\,m}\mathbf{tell}(c) \parallel !\, \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) \\ & \xRightarrow{(\mathtt{true},\mathtt{true})} & \mathbf{next}^{\,m-1}\mathbf{tell}(c) \parallel !\, \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) \\ & \vdots & \\ & \xRightarrow{(\mathtt{true},\mathtt{true})} & \mathbf{tell}(c) \parallel !\, \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) \\ & \xRightarrow{(\mathtt{true}, c\wedge e)} & !\, \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e) \\ & \vdots & \end{array}$$

Intuitively, in the first time interval the environment tells c, and thus the component $!\,\mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e)$ *tells e. The output is then $c \wedge e$. Furthermore, the other component* $\star\,\mathbf{tell}(c)$ *creates a process* $\mathbf{tell}(c)$ *which is to be triggered in $m+1$ times units, for some arbitrary m. In the following time units the environment does not provide any input whatsoever. In the $m+1$-th time unit c is told c and then* $!\,\mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(e)$ *tells e again.* $\square$

3.4 Observable Behavior: The Strongest Postcondition.

We now recall the notions of observable behavior for `ntcc` introduced in [27], in particular that of the *strongest postcondition* (sp), central to this paper.

Notation 3.11 *Throughout this paper $\mathcal{C}^\omega$ denotes the set of infinite (or ω) sequences of constraints in the underlying set of constraints $\mathcal{C}$. We use $\alpha, \alpha', \ldots$ to range over $\mathcal{C}^\omega$.*

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

IO and Output Behavior. Let $\alpha = c_1.c_2.\ldots$ and $\alpha' = c'_1.c'_2.\ldots$ be infinite sequences of constraints. If $P \xRightarrow{(\alpha,\alpha')}{}^\omega$, it means that at the time unit i, the environment *inputs* c_i to P_i which then responds with an output c'_i. As observers, we can see that on α, P responds with α'. We refer to the set of all (α, α') such that $P \xRightarrow{(\alpha,\alpha')}{}^\omega$ as the *input-output (io) behavior* of P. Alternatively, if $\alpha = \mathtt{true}^\omega$, we interpret the run as an interaction among the parallel components in P *without the influence of any (external) environment*; as observers what we see is that P produces α on its own. We refer to the set of all α' such that $P \xRightarrow{(\mathtt{true}^\omega,\alpha')}{}^\omega$ as the *output* behavior of P.

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

In [26] it is shown that the set of quiescent sequences of a given P can be alternatively characterized as *the set of infinite sequences that P can possibly output under arbitrary environments*; the strongest postcondition (sp) of P.

Definition 3.12 (SP) *The* strongest postcondition *of P, $sp(P)$, is given by $sp(P) = \{\alpha' \mid P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$ for some $\alpha\}$ and its induced observational equivalence $\sim_{sp}$ is given by $P \sim_{sp} Q$ iff $sp(P) = sp(Q)$.*

The above-mentioned match between the set of quiescent sequences and the SP is stated in the following theorem from [26].

Theorem 3.13 (Quiescent-SP Match, [26]) *For every P, $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$ iff $P \xRightarrow{(\alpha',\alpha')}{}^{\omega}$.*

We conclude this section by illustrating the difference between the sp and input-output observables with the following example:

Example 3.14 *Let $P = \mathbf{tell}(\mathtt{true}) + \mathbf{tell}(c)$ and $Q = \mathbf{tell}(\mathtt{true})$. Notice that they do not exhibit the same input-output (or ouput) behaviour: Unlike P, process Q cannot ouput $c.\mathtt{true}^{\omega}$ on input $\mathtt{true}^{\omega}$. Nevertheless, the reader can verify that P and Q have the same sp behavior.* $\square$

3.5 LTL Specification and Verification

We now look at the `ntcc` LTL [26]. This particular LTL expresses properties over sequences of constraints and we shall refer to it as **CLTL**. We begin by giving the syntax of LTL formulae and then interpret them with the **CLTL** semantics.

Definition 3.15 (LTL Syntax) *The formulae $F, G, \ldots \in \mathcal{F}$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by:*

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

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 needed as in **CLTL** these operators may have slight differences with 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*. Intuitively $\circ\ F$, $\Diamond\ F$ and $\Box\ F$ means that the property F must hold next, eventually and always, respectively. We use $F \dot{\Rightarrow} G$ for $\dot{\neg} F \dot{\vee} G$.

The standard interpretation structures of linear temporal logic are infinite sequences of states [18]. In the case of `ntcc` , it is natural to replace states by constraints, and consider therefore as interpretations the elements of $\mathcal{C}^{\omega}$.

The **CLTL** semantics of the `ntcc` logic is given in Definition 3.18. Following [18] we introduce the notion of *x-variant.* But first we need some little notation.

Notation 3.16 *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 3.17 (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 α and c, respectively, if they are the same except for the information about x. For example, $x = 1 \wedge y = 0$ is an x-variant of $x = 42 \wedge y = 0$.

We can now give the **CLTL** semantics of our `ntcc` logic.

Definition 3.18 (CLTL Semantics) *We say that α satisfies (or that it is a model of) F in* **CLTL** *, written $\alpha \models_{\mathrm{CLTL}} F$, iff $\langle \alpha, 1 \rangle \models_{\mathrm{CLTL}} F$, where:*

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

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. We ought to clarify the role of constraints as state formulae in our logic to justify our dotted notation. A temporal formula F expresses properties over sequences of constraints. As a state formula, c expresses a property which is satisfied only by those $e.\alpha'$ such that $e \models c$ holds. Therefore, the state formula `false` (and consequently $\Box$`false`) has at least one sequence that satisfies it (e.g. $\mathtt{false}^\omega$). On the contrary the temporal formula $\dot{\mathtt{false}}$ has no models whatsoever.

Similarly, 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 state formula (constraint) $c \vee d$ may have different models since $e \models c \vee d$ may hold while neither $e \models c$ nor $e \models d$ hold. Thus, in general $[\![c \dot{\vee} d]\!] \neq [\![c \vee d]\!]$. The same holds true for $\neg c$ and $\dot{\neg}\, c$.

Example 3.19 *Let $e = c \vee d$ with $c = (x = 42)$ and $d = (x \neq 42)$. One can verify that $\mathcal{C}^{\omega} = [\![c \vee d]\!] \ni e^{\omega} \notin [\![c \dot{\vee} d]\!]$ and also that $[\![\neg c]\!] \ni \mathtt{false}^{\omega} \notin [\![\dot{\neg}\, c]\!]$.* $\square$

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 $\dot{\vee}$.

Process Verification.

Let us now recall what it means for a process P to satisfy a specification F.

Definition 3.20 (Verification) *We say that the process P* satisfies *the formula F, written $P \models_{\mathrm{CLTL}} F$, iff $sp(P) \subseteq [\![F]\!]$.*

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.

Example 3.21 *Recall Example 3.5. We had a process R which was repeatedly checking the state of motor_1. If a malfunction is reported, R would tell that motor_1 must stop. We also had a process S stating that motor motor_1 was doomed to malfunction. Intuitively, this means that the parallel execution of R and S satisfies the specification $\Diamond(\mathit{motor}_1\_\mathit{speed} = 0)$ stating that motor_1 must eventually stop. In other words,*

$$R \parallel S \ \models_{\mathrm{CLTL}} \ \Diamond(\mathit{motor}_1\_\mathit{speed} = 0)$$

$\square$

Remark 3.22 *Notice that $P = \mathbf{tell}(c) + \mathbf{tell}(d) \models_{\mathrm{CLTL}} (c \dot{\vee} d)$ as every constraint e output by P entails either c or d. In contrast, $Q = \mathbf{tell}(c \vee d) \not\models (c \dot{\vee} d)$ in general since Q can output a constraint e which certainly entails $c \vee d$ and still entails neither c nor d – e.g. consider $e = (x = 1 \vee x = 2)$, $c = (x = 1)$ and $d = (x = 2)$. Notice, however, that $Q \models (c \vee d)$. In other words the formula $c \dot{\vee} d$ distinguishes P from Q while the (state) formula $c \vee d$ does not.*

The reader may now see why we distinguished the temporal formula $c \dot{\vee} d$ from the state formula (i.e., constraint) $c \vee d$.

⋆ ⋆ ⋆

4 Decidability Results for `ntcc`

We first present our decidability result for the strongest-postcondition (sp) equivalence. We then show, with the help of this first result, our decidability result for the `ntcc` verification problem. Finally, we present the decidability results for validity and satisfiability in **CLTL**.

The theory of Büchi FSA [3] is central to our results. These FSA are ordinary automata with an acceptance condition for infinite (or ω) sequences: an ω sequence is accepted iff the automaton can read it from left to right while visiting a final state infinitely often. The language recognized by a Büchi automaton A is denoted by $L(A)$. Regular ω-languages are those recognized by Büchi FSA.

4.1 Previous Approaches

For a better exposition of our results, it is convenient to look first into previous approaches to the decidability of the `ntcc` observational equivalences. First, we need the following definition.

Definition 4.1 (Derivatives) *Define $P \Longrightarrow Q$ iff $P \xRightarrow{(c,d)} Q$ for some c, d. A process Q is a* derivative *of P iff $P = P_1 \Longrightarrow \dots \Longrightarrow P_n = Q$ for some $P_1, \dots, P_n$.*

Restricted Nondeterminism: Finitely Many States. In [27] we show the decidability of output equivalence for the *restricted-nondeterministic* fragment of `ntcc` which only allows *$\star$-free processes* whose *summations are not in the scope of local operators*. First, the authors show that each process in this fragment has a *finite number of derivatives* (up-to output equivalence). Then, they show how to construct for any given restricted-nondeterministic P, a Büchi automaton that recognizes P's output behavior as an $\omega-$ language. Since language equivalence for Büchi FSA is decidable [39] the decidability of output equivalence follows. In his PhD dissertation [42] the author proved the decidability of the sp (and input-output) equivalence for restricted-nondeterministic processes by a reduction to that of output equivalence.

More Liberal Nondeterminism: Infinitely Many States. The above-mentioned FSA have states representing processes and transitions representing observable reductions. The automata are generated by an *algorithm* that uses the `ntcc` *operational semantics* to generate all possible observable reductions. Hence, the finiteness of the set of derivatives is crucial to guarantee termination. In fact, the algorithm may diverge if we allow arbitrary $\star$ processes, or summations within local operators because, as illustrated below, they can have *infinitely many derivatives* each with different observable behavior.

Example 4.2 *(1) Notice that $\star P$ has* infinitely many derivatives *of the form* $\textbf{next}^n P$ *and each of them may exhibit different observable behavior. (2) Let* $R = !!P$ *with* $P = \textbf{when}\ x = 1\ \textbf{do}\ \star\ \textbf{tell}(c)$. *Notice that we could have the following sequence* $R \Longrightarrow !P \parallel !!P \Longrightarrow !P \parallel !P \parallel !!P \Longrightarrow \ldots$. *Now, for any* $n > 0$, *let* $D_n = \prod_n !P \parallel !!P$ *(i.e., the n-th derivative). One can verify that on input* $(x = 1).\texttt{true}^\omega$, D_k *can tell c at* $k + 1$ *different time units but* D_{k-1} *can only do it at k different units. (3) A similar situation occurs if* $P = \textbf{when}\ x = 1\ \textbf{do}\ \delta\textbf{tell}(c)$ *where* δQ *denotes an arbitrary* possibly infinite *delay of Q. This delay operator can be recursively defined as* $\delta Q \stackrel{\text{def}}{=} Q + \textbf{next}\ \delta Q$ *and such a kind of definitions can be derived in* `ntcc` *using replication together with* blind choice summations within local processes *[26].* $\square$

4.2 Decidability of SP Equivalence: The Approach

We shall show a Büchi FSA characterization of the sp for processes that, unlike in previous approaches, can exhibit infinitely many, observationally different, derivatives—of the kind illustrated in Example 4.2. Another difference with the work previously mentioned is that, to get around the algorithmic problems illustrated above, the characterizations will be guided by the *sp denotational semantics* of `ntcc` [25] rather than its operational semantics.

SP Denotational Representation.

Table 2 shows a sp denotational semantics $[\![\cdot]\!] : \mathit{Proc} \to \mathcal{P}(\mathcal{C}^\omega)$ of `ntcc` . In fact, this is simply the semantics of `ntcc` given in [28] extended to take into account the **abort** construct.

Nevertheless, from [7] we know that there cannot be a $f : \mathit{Proc} \to \mathcal{P}(\mathcal{C}^\omega)$, compositionally defined, such that $f(P) = \mathit{sp}(P)$ for all P. In [28], however,

Table 2
A SP Denotation. Below $\beta.\alpha'$ is the concatenation of the finite sequence β followed by α'. The sequence $\exists_x\alpha$ results from applying $\exists_x$ to each constraint in α. In DSUM if $I = \emptyset$, the indexed union and intersection are taken to be $\emptyset$ and S^ω, respectively.

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

we showed that $[\![P]\!] = sp(P)$ for all P in the so-called *locally-independent* fragment. This fragment forbids non-unary summations (and "unless" processes) whose guards depend on local variables.

Definition 4.3 (Locally-Independent Processes) *A process P is said to be* locally-independent *iff for every* **unless** c **next** Q *and* $\sum_{i\in I}$ **when** c_i **do** Q_i *($|I| \geq 2$) in P, neither c nor the c_i's contain an occurrence of a variable bound by some local operator in P.*

Theorem 4.4 (Completeness) *If P is locally-independent, $[\![P]\!] = sp(P)$.*

Proof. By induction on structure of P. The case $P = \textbf{abort}$ is straightforward. The other cases follow as in the proof of Completeness in [28]. $\square$

Expressiveness. The locally-independent fragment allows $\star$ processes and also blind-choice within local operators which may exhibit infinitely many observationally different derivatives, as illustrated in Example 4.2. Furthermore, every summation whose guards are either all equivalent or mutually exclusive can be encoded in this fragment [42]. The applicability of this fragment is witnessed by the fact all the `ntcc` application the author is aware of [26,27,42] can be model as locally-independent processes. Furthermore, the (parameterless-recursion) tcc model [33] can be expressed in this fragment as, from the expressiveness point of view, the local operator is redundant in tcc with parameterless-recursion [25].

SP Büchi FSA Representation.

We shall give a Büchi characterization of $sp(P)$ from its denotation $[\![P]\!]$. We then benefit from the simple set-theoretical and compositional nature of $[\![\cdot]\!]$ as well as the closure properties of regular $\omega-$languages.

A technical problem arises: There can be infinitely many c's such that $c.\alpha$ is in the sp of a given P since $\mathcal{C}$ may be infinite, but the alphabets of Büchi FSA are finite. To overcome this problem we shall confine the sp of P (or P) to a suitable finite set $S \subseteq_{fin} \mathcal{C}$ including the so-called *relevant constraints* of P. The intuition is that for each input c for P, we can find a constraint c' in S entailed by c and build from the constraints in P, so that P behaves on c' as it behaves on c. We can then think of c' as containing just the information of c that is relevant to P.

Relevant Constraints. Before giving the actual definition of relevant constraints, let us give some more intuition about this notion with an example.

Example 4.5 *Let us consider the process*

$$P = \textbf{tell}(0 < x \wedge x < y) \parallel \textbf{when } prime(x) \textbf{ do tell}(z = 0)$$

where $prime(x)$ holds iff x is a prime number. Note that P tells $(0 < x \wedge x < y)$ no matter what, and on input $prime(x)$, which occurs as a constraint in P, it *tells $z = 0$. But P can also tell $z = 0$ on other inputs, e.g., $y = 4$ or $x = 7$; none of which appears in P.*

Let c be an arbitrary constraint on input of which P tells $z = 0$. From the operational semantics, we conclude that $(c \wedge 0 < x \wedge x < y) \models prime(x)$ holds. But such an assertion holds iff $c \models (0 < x \wedge x < y) \Rightarrow prime(x)$.

Let us then declare $c' = ((0 < x \wedge x < y) \Rightarrow prime(x))$. We notice that P on c' can also tell $z = 0$. So, c' is of relevance for P as it characterizes P on every input c causing $z = 0$ to be told. Furthermore, it can be obtained from the constraints in P using implication.

Now, let $Q = (\mathbf{local}\, x)\, P$. Notice, that like P, the process Q tells $z = 0$ on inputs such as $y = 4$. But because x is a local variable, unlike P, Q does not tell $z = 0$ on inputs such as $prime(x)$. For the same reason, Q does not tell $z = 0$ on c' above.

So, let us define $e = \forall_x c'$. Now Q on input e tells $z = 0$. Notice that e can be obtained from the constraints in Q using implication *and* universal quantification.

Furthermore, notice that $\mathbf{tell}(0 < x \wedge x < y)$ and $(\mathbf{local}\, z)\, \mathbf{tell}(z < x)$ are equivalent to $\mathbf{tell}(0 < x) \parallel \mathbf{tell}(x < y)$ and $\mathbf{tell}(\exists_z z < x)$, respectively. So, we also need conjunction *and* existential *quantification for obtaining the relevant constraints of a given process.* $\square$

The above example illustrates how to obtain relevant constraints for processes using implication, conjunction, and if local operators are involved, also universal and existential quantification. In the following we formalize the above intuition.

Definition 4.6 (Relevant Constraints) *Given $\mathcal{S} \subseteq \mathcal{C}$, let $\overline{\mathcal{S}}$ be the closure under conjunction and implication of $\mathcal{S}$. Let $C : Proc \to \mathcal{P}(\mathcal{C})$ be defined as:*
$C(\mathbf{skip}) = \{\mathtt{true}\}$
$C(\mathbf{tell}(c)) = \{c\}$
$C(\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i) = \bigcup_{i \in I} \{c_i\} \cup C(P_i)$
$C(\mathbf{unless}\ c\ \mathbf{next}\, P) = \{c\} \cup C(P)$
$C(P \parallel Q) = C(P) \cup C(Q)$
$C(!\, P) = C(\star\, P) = C(\mathbf{next}\, P) = C(P)$
$C((\mathbf{local}\, x)\, P) = \{\exists_x c, \forall_x c \mid c \in \overline{C(P)}\} \cup C(P)$

Define the relevant constraints *of P as the set $\mathcal{C}(P) = \overline{C(P)}$.*

Clearly $\mathcal{C}(P)$ is finite. Moreover, in what follows we shall show that $\mathcal{C}(P)$ provides the suitable finite set of constraints to characterize $sp(P)$. First, we need the following notation.

Definition 4.7 (Relevant Constraint Information) *Let $S \subset_{fin} \mathcal{C}$. Define $c(S)$ as the conjunction of all $e \in S$ entailed by c, i.e., $c(S) = \bigwedge_{e \in S, c \models e} e$.*

Intuitively, $c(S)$ contains the necessary information about c wrt a finite restricted universe of constraints $S \subset \mathcal{C}$.

The next lemma states that to characterize the sp of P we can use any set that containing at least its relevant constraints. Please recall that the sp of a given process can be equivalently characterized by its set of quiescent sequences—see Theorem 3.13.

Lemma 4.8 *Assume that S is such that $\mathcal{C}(P) \subseteq S \subset_{fin} \mathcal{C}$. Then*

$$P \xRightarrow{(c,c)} P' \text{ iff } P \xRightarrow{(c(S),c(S))} P'.$$

Let us illustrate how the above lemma applies to the previous example.

Example 4.9 *Let P be defined as in Example 4.5. Furthermore, suppose that $S = \mathcal{C}(P) = \overline{\{0 < x \wedge x < y, prime(x), z = 0\}}$ and that $c \models c' = ((0 < x \wedge x < y) \Rightarrow prime(x))$.*

Consider the "if" direction of the Lemma 4.8. Assume that

$$P \xRightarrow{(c(S),c(S))} P'.$$

It must then be the case that $c(S) \models (0 < x \wedge x < y)$, thus $c \models (0 < x \wedge x < y)$. Then $c \models prime(x)$ and since $prime(x) \in \mathcal{C}(P)$ then also $c(s) \models prime(x)$. So, on input $c(s)$, process P must tell $z = 0$ and thus, from the assumption, $c(s) \models z = 0$. Therefore, $c \models z = 0$. All in all, $c \models (0 < x \wedge x < y) \wedge prime(x) \wedge z = 0$. It is then is easy to check that

$$P \xRightarrow{(c,c)} P'.$$

The reader may care to apply the "only-if" direction of the lemma to our particular P. $\square$

Proof of Lemma 4.8 From Rule OBS in Table 1, it suffices to show that for every P,

i $\langle P, c\rangle \longrightarrow^* \langle Q, c\rangle$ iff $\langle P, c(S)\rangle \longrightarrow^* \langle Q, c(S)\rangle$, and
ii $\langle P, c\rangle \not\longrightarrow$ iff $\langle P, c(S)\rangle \not\longrightarrow$.

To simplify the presentation, we assume that P contains no nesting of local operators. The most interesting case is the "only if" direction of (i).

- The "only if" direction of (i). Assume the reduction sequence $\langle P, c\rangle \longrightarrow^* \langle Q, c\rangle$. The sequence can be represented as a sequence of the form

$$\begin{aligned}&\langle P_0, c\rangle \longrightarrow^* \langle P_1, c\rangle \longrightarrow \langle P_1', c\rangle \longrightarrow^* \ldots \longrightarrow^* \langle P_i, c\rangle \\ &\quad \longrightarrow \langle P_i', c\rangle \longrightarrow^* \langle P_{i+1}, c\rangle \longrightarrow \left\langle P_{i+1}', c\right\rangle \longrightarrow^* \ldots \end{aligned} \tag{1}$$

(with $P = P_0$ and $Q = P_n$) satisfying the following: (1) The (zero or more) reductions $\langle P_i, c\rangle \longrightarrow \langle P_i', c\rangle$ are obtained from derivations whose topmost (or root) rule is either SUM or UNL—i.e. they represent the execution of either a summation or an unless operator. (2) Each $\langle P_i, c\rangle \longrightarrow^* \langle P_{i+1}, c\rangle$ involves no application of SUM or UNL.

Now let g_i be the guard of the summation whose branch was selected (or the guard of the unless operator) when deriving $\langle P_i, c\rangle \longrightarrow \langle P_i', c\rangle$. From Rules SUM, UNL and LOC we must have:

$$e_i \wedge \exists_{\vec{x}_i} c \models g_i \tag{2}$$

where $\vec{x}_i$ is vector of at most one variable and e_i represents local information possibly introduced by rule LOC (the vector can be empty and e_i can be `true` meaning that LOC was not applied; see Convention 3.7).

By manipulating the assertion in Equation 2 we obtain the following transformation central to our proof:

$$\begin{aligned} e_i \wedge \exists_{\vec{x}_i} c \models g_i \ &\text{iff}\ \exists_{\vec{x}_i} c \models (e_i \Rightarrow g_i) \\ &\text{iff}\ \exists_{\vec{x}_i} c \models \forall_{\vec{x}_i}(e_i \Rightarrow g_i) \\ &\text{iff}\ c \models \forall_{\vec{x}_i}(e_i \Rightarrow g_i) \\ &\text{iff}\ c \models \forall_{\vec{x}_i}(e_i \Rightarrow g_i) \end{aligned} \tag{3}$$

Then, let $d_i = \forall_{\vec{x}_i}(e_i \Rightarrow g_i)$. From Definition 4.6, $g_i \in \mathcal{C}(P)$ because g_i is a guard in P. Since e_i represents local information, using Rule LOC we can verify that it can be constructed via conjunction and existential quantification out of the constraints in tell operators within the corresponding local process. So, $e_i \in \mathcal{C}(P)$ and hence, from Definition 4.6, $d_i \in \mathcal{C}(P)$.

Let $c' = \bigwedge_{i\in\{1,\ldots,n\}} d_i$. From Equation 3 $c \models c'$. Moreover, $c' \in \mathcal{C}(P)$ since each $d_i \in \mathcal{C}(P)$ and $\mathcal{C}(P)$ is closed under conjunction. Thus,

$$c(S) \models c' \tag{4}$$

Claim 1 *Given the P_i's and their primed versions in Equation 1, one can construct the sequence of the form*

$$\begin{aligned}&\langle P_0, c(s)\rangle \longrightarrow^* \langle P_1, c(s)\rangle \longrightarrow \langle P_1', c(s)\rangle \longrightarrow^* \ldots \longrightarrow^* \langle P_i, c(s)\rangle \\ &\longrightarrow \langle P_i', c(s)\rangle \longrightarrow^* \langle P_{i+1}, c(s)\rangle \longrightarrow \left\langle P_{i+1}', c(s)\right\rangle \longrightarrow^* \ldots\end{aligned}$$

satisfying conditions analogous that of Equation 1: Each $\langle P_i, c(S)\rangle \longrightarrow \langle P_i', c(S)\rangle$ is obtained from a derivation with topmost rule SUM *or* UNL*,* *and each sequence* $\langle P_i, c(S)\rangle \longrightarrow^* \langle P_{i+1}, c(S)\rangle$ *involves no application of* SUM *or* UNL. *It then follows that* $\langle P, c(S)\rangle \longrightarrow^* \langle Q, c(S)\rangle$ *as wanted.*

To prove the claim, let us first show that $\langle P_i, c(S)\rangle \longrightarrow \langle P_i', c(S)\rangle$ for each $i > 1$. Notice that $c(S) \models c' \models \forall_{\vec{x}_i}(e_i \Rightarrow g_i)$ for each i (Equation 4). So, just like $e_i \wedge \exists_{\vec{x}_i} c \models g_i$ (Equation 2), we get

$$e_i \wedge \exists_{\vec{x}_i} c(S) \models e_i \wedge \exists_{\vec{x}_i}\forall_{\vec{x}_i}(e_i \Rightarrow g_i) \models e_i \wedge \forall_{\vec{x}_i}(e_i \Rightarrow g_i) \models g_i. \qquad (5)$$

With the help of Equation 5, we conclude that $\langle P_i, c_i(S)\rangle$ can execute the summation or unless operation executed by $\langle P_i, c_i\rangle$. More precisely, we can obtain a derivation of $\langle P_i, c(S)\rangle \longrightarrow \langle P_i', c(S)\rangle$ by applying in the same order the rules used in $\langle P_i, c_i\rangle \longrightarrow \langle P_i', c_i'\rangle$ and the fact that $e_i \wedge \exists_{\vec{x}_i} c(S) \models g_i$ (Equation 5).

Furthermore, the conjunction of all constraints told during $\langle P_i, c\rangle \longrightarrow^* \langle P_{i+1}, c\rangle$ is entailed by c and it must be in S, so it is entailed by $c(S)$. From this we verify that $\langle P_i, c(S)\rangle \longrightarrow^* \langle P_{i+1}, c(S)\rangle$ (by using in same order the rules in $\langle P_i, c\rangle \longrightarrow^* \langle P_{i+1}, c\rangle$), thus concluding the proof of the claim.

- The "if" direction of (i) is similar to the previous item; it uses the fact that $c \models c(S)$.
- The 'if" direction of (ii). By means of contradiction let us suppose we have $\langle P, c(S)\rangle \not\longrightarrow$ and $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$. But then $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$ is obtained by a derivation whose topmost rule is SUM or UNL (i.e., either a summation or an unless is executed)—otherwise $\langle P, c(S)\rangle$ would reduce as well.

  We now proceed as we did in the first item of this proof. Let g be the guard of the summation whose branch was selected (or the guard of the unless operator) when deriving $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$. We must then have

$$e \wedge \exists_{\vec{x}} c \models g$$

  where $\vec{x}$ is vector of at most one variable and e represents local information possibly introduced by rule LOC. As in the first item we conclude that $c \models d = \forall_{\vec{x}}(e \Rightarrow g)$ (See Equation 3), and that $d \in S$. Then, $c(S) \models d$ by definition.

  From the above it follows that $e \wedge \exists_{\vec{x}} c(S) \models e \wedge \exists_{\vec{x}} d \models e \wedge \forall_{\vec{x}}(e \Rightarrow g) \models g$. We can then show that $\langle P, c(S)\rangle$ can execute the summation or unless operation executed by $\langle P, c\rangle$. I.e., $\langle P, c(S)\rangle \longrightarrow \langle P', c(S)\rangle$ by using the fact that $e \wedge \exists_{\vec{x}} c(S) \models g$ and applying in the same order the rules for deriving $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$. This contradicts our initial assumption.
- The 'only if" direction of (ii) is similar to the "if" direction but using the fact that $c \models c(S)$. □

Corollary 4.10 (Relevant SP) *Assume that* S *is such that* $\mathcal{C}(P) \subseteq S \subset_{fin} \mathcal{C}$. *Then*

$$c_1.c_2.\ldots \in sp(P) \text{ iff } c_1(S).c_2(S).\ldots \in sp(P)$$

Proof. From Definition 3.12, $c_1.c_2.\ldots \in sp(P)$ iff for some $P_1, P_2, \ldots$ we have $P = P_0 \xRightarrow{(c_1,c_1)} P_1 \xRightarrow{(c_2,c_2)} P_2 \xRightarrow{(c_3,c_3)} \ldots$. By case analysis on the structure of Q, it is easy to verify that if $Q \xRightarrow{(c,d)} Q'$, $\mathcal{C}(Q') \subseteq \mathcal{C}(Q)$. Hence $\mathcal{C}(P_i) \subseteq \mathcal{C}(P) \subseteq S$ for each $i > 0$. The result follows by applying Lemma 4.8 to each P_i. $\square$

Relevant-Constraints Denotational Representation. In the previous section we proved that we can restrict the sp of a given process to a finite set of constraints S. Therefore we can restrict as well its denotational semantics $[\![P]\!]$ to S.

Consequently, we shall work with a function $[\![\cdot]\!]^S : Proc \to \mathcal{P}(S^\omega)$ in Table 3 which is meant to capture $sp(P)$ but restricted to a set of relevant constraints S. The idea is to think that the underlying set of constraints $\mathcal{C}$ to be S. In fact, we can re-state the theorem in [28] as follows:

Theorem 4.11 (Completeness Revisited) *Let P be local independent and S be such that $\mathcal{C}(P) \subseteq S \subset_{fin} \mathcal{C}$. Then $[\![P]\!]^S = sp(P) \cap S^\omega$.*

Proof. The proof proceeds as the proof of Theorem 4.4 but taking the underlying set of constraints $\mathcal{C}$ to be S. $\square$

Now we have all what we need to reduce sp equivalence to the denotational equivalence over a finite set of constraints.

Theorem 4.12 (Relevant Characterization of SP Equivalence) *Let P and Q be locally-independent and let $S = \mathcal{C}(P) \cup \mathcal{C}(Q)$. Then*

$$[\![P]\!]^S = [\![Q]\!]^S \quad \textit{iff} \quad P \sim_{sp} Q.$$

Proof. From Corollary 4.10 and Theorem 4.11. $\square$

Büchi Constructions. Having identified our finite set of relevant constraints for the sp equivalence characterization, we now proceed to construct for each P and $S \subseteq_{fin} \mathcal{C}$, a Büchi automaton A_P^S recognizing $[\![P]\!]^S$. Each A_P^S will be *compositionally* constructed in the sense that it will be built solely from information about FSA of the form A_Q^S where Q is a subprocess of P. The most interesting case of this construction is replication, as described in the proof of the lemma below.

Table 3
The $[\![\cdot]\!]^S : \mathit{Proc} \to \mathcal{P}(S^\omega)$ SP Denotation. Below $S \subseteq \mathcal{C}$ and $\beta.\alpha'$ is the concatenation of the finite sequence β followed by α'. The sequence $\exists_x \alpha$ results from applying $\exists_x$ to each constraint in α. In DSUM if $I = \emptyset$, the indexed union and intersection are taken to be $\emptyset$ and S^ω, respectively.

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

Lemma 4.13 *Given P and $S \subseteq_{fin} \mathcal{C}$ one can effectively construct a Büchi automaton A_P^S over the alphabet S such that $L(A_P^S) = [\![P]\!]^S$.*

Proof. Let us construct A_P^S by case analysis on P.

- Skip and Tell Automata. $P = \mathbf{skip}$: $A^S_{\mathbf{skip}}$ has a single (initial, accepting) state, and for each $c \in S$, there is a transition labeled with c from this single state to itself. $P = \mathbf{tell}(c)$: $A^S_{\mathbf{tell}(c)}$ has exactly two states: one is its initial state and the other is its accepting one: the unique state of $A^S_{\mathbf{skip}}$. The transitions of $A^S_{\mathbf{tell}(c)}$ are those of $A^S_{\mathbf{skip}}$ plus a transition from the initial state to the state of $A^S_{\mathbf{skip}}$ labeled with d for each $d \in S$ such that $d \models c$.

- Parallel Automaton. $P = Q \parallel R$: From the theory of Büchi FSA we know that given A^S_Q and A^S_R one can construct an automaton for $L(A^S_Q) \cap L(A^S_R)$ [4]. Take $A^S_{Q\parallel R}$ to be such an automaton.
- Local Automaton. $P = (\mathbf{local}\, x)\, Q$: The states of $A^S_{(\mathbf{local}\, x)\, Q}$ are those of A^S_Q. Its initial and final states are those of A_Q. For each $c \in S$, $A^S_{(\mathbf{local}\, x)\, Q}$ has a transition from p to q labeled with c iff A^S_Q has a transition from p to q labelled with d for some $d \in S$ such that $\exists_x d = \exists_x e$.
- Replication Automaton. $P = !Q$: We have $\alpha \in [\![!Q]\!]^S$ iff every suffix of α is in $[\![Q]\!]^S$. We then need to construct a $A^S_{!P}$ that accepts an infinite sequence α iff *every suffix* of it is accepted by A^S_Q. At first, such a construction may seem somewhat complex to realize. Nevertheless, notice that the process $!Q$ is *dual* to $\star Q$ in the sense expressed in [28]:

  $$[\![!Q]\!] = \nu_X([\![Q]\!] \cap \{d.\alpha \mid \alpha \in X\}) \quad \text{while} \quad [\![\star Q]\!] = \mu_X([\![Q]\!] \cup \{d.\alpha \mid \alpha \in X\})$$

  where ν, μ are the greatest and least fixpoint (resp.) in the lattice $(\mathcal{P}(\mathcal{C}^\omega), \subseteq)$. In fact, $\alpha \in [\![\star Q]\!]^S$ iff *there is a suffix* of α in $[\![Q]\!]^S$. So, given an automaton B define $\star B$ as the automaton that accepts α iff *there is a suffix* of α accepted by B. The construction of $\star B$ is simple and similar to that of $A^S_{\star Q}$ below. Now, given A, one can construct the automaton $\overline{A}$ for the complement of $L(A)$ [39]. Hence, keeping duality in mind, we can take $A^S_{!Q}$ to be $\overline{\star \overline{A^S_Q}}$.
- Unbounded but Finite Delay Automaton. $P = \star Q$: The states of $A^S_{\star Q}$ are those of A^S_Q plus an additional state s_0. Its final states are those of A^S_Q. Its initial states are those of A^S_Q plus s_0. The transitions are those of A^S_Q plus transitions labelled with c, for each $c \in S$, from s_0 to itself and from s_0 to each initial state of A^S_Q. The transitions "looping" into the initial state s_0 model the "arbitrary but finite delay" of $\star Q$. Notice that any sequence to be accepted for $A^S_{\star Q}$ has to eventually leave the initial state s_0.

The simple summation case is left for the reader. The correctness of A^S_P and its effective construction can be easily verified by induction on P. □

We can now prove our decidability result for sp equivalence:

Theorem 4.14 (Decidability of $\sim_{sp}$) *Given the locally-independent processes P and Q, the question whether $P \sim_{sp} Q$ is decidable.*

Proof. From Theorem 4.12, Lemma 4.13 and the decidability of language equivalence for Büchi FSA [39]. □

Here we show the decidability results for the verification problem (i.e., given P and F whether $P \models_{\mathrm{CLTL}} F$, see Definition 3.20) as well as for **CLTL** (Definition 3.18).

Recall the **CLTL** models of a given formula F are in $\mathcal{C}^{\omega}$. Thus, for our decidability results we may attempt to give a Büchi characterization of **CLTL** formulae facing therefore the problem pointed out in the previous section: $\mathcal{C}$ may be infinite but Büchi FSA only have finite alphabets. One could try to restrict $[\![F]\!]$ to its "relevant" constraints as we did for $[\![P]\!]$. This appears to be possible for negation-free formulae but we do not know yet how to achieve this for the full language.

Nevertheless, for negation-free formulae there is an alternative to obtain the results by appealing to Theorem 4.14 and the correspondence between processes and LTL formulae. More precisely, we show that one can construct a locally-independent R_F whose sp corresponds to $[\![F]\!]$ if F is a restricted-negation formula in the following sense:

Definition 4.15 (Restricted-Negation LTL) *A formula F is a* restricted-negation formula *iff whenever $\dot{\neg}\, G$ appears in F then G is a state formula (i.e., $G = c$ for some c).*

Recall that in **CLTL** $\dot{\neg}\, c$ and the state formula $\neg c$ do not match (Example 3.19). In fact, we need $\dot{\neg}\, c$ should we want to express the minimal implication form $c \dot{\Rightarrow} D = \dot{\neg}\, c \,\dot{\vee}\, D$.

Lemma 4.16 *Let F be a restricted-negation formula. One can effectively construct a locally-independent R_F such that $[\![F]\!] = sp(R_F)$.*

Proof. Take $R_F = h(F)$ where h is the map from restricted-negation formulae to locally-independent processes given by

$$\begin{array}{ll}
h(\dot{\mathtt{true}}) = \mathbf{skip} & h(\dot{\mathtt{false}}) = \mathbf{abort} \\
h(c) = \mathbf{tell}(c) & h(\dot{\neg}\, c) = \mathbf{when}\ c\ \mathbf{do\ abort} \\
h(F \,\dot{\wedge}\, G) = h(F) \parallel h(G) & h(F \,\dot{\vee}\, G) = \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ h(F) \\
 & \qquad\qquad + \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ h(G) \\
h(\dot{\exists}\, xF) = (\mathbf{local}\, x)\, h(F) & h(\circ F) = \mathbf{next}\, h(G) \\
h(\Box F) = !\, h(F) & h(\Diamond F) = \star h(F)
\end{array}$$

Obviously, $h(F)$ can be effectively constructed. One can verify that $[\![F]\!] = [\![h(F)]\!]$ by induction on the structure of F. From Theorem 4.4 we obtain $[\![F]\!] = sp(h(F))$. $\square$

Notice that the map h above reveals the close connection between `ntcc` and LTL. We can now state the decidability of the verification problem for `ntcc`.

Theorem 4.17 (Decidability of Verification) *The question whether the assertion $P \models_{\mathrm{CLTL}} F$ holds is decidable for any given restricted-negation formula F and locally-independent process P.*

Proof. From Theorem 4.14 by using the following reduction to sp equivalence:

$$\begin{array}{rrcll}
P \models_{\mathrm{CLTL}} F \text{ iff } & sp(P) & \subseteq & [\![F]\!] & \text{(Definition 3.20)} \\
\text{iff } & sp(P) & \subseteq & [\![R_F]\!] & \text{(Lemma 4.16)} \\
\text{iff } & [\![P]\!] & \subseteq & [\![R_F]\!] & \text{(Theorem 4.4)} \\
\text{iff } & [\![P]\!] & = & [\![R_F]\!] \cap [\![P]\!] & \\
\text{iff } & [\![P]\!] & = & [\![R_F \parallel P]\!] & \text{(Definition of } [\![\cdot]\!]) \\
\text{iff } & P & \sim_{sp} & R_F \parallel P & \text{(Theorem 4.12).} \quad \square
\end{array}$$

We can reduce the validity of implication to the verification problem. Therefore,

Theorem 4.18 (Decidability for Validity of Implication) *Let F and G be restricted-negation formulae. The question of whether $F \dot{\Rightarrow} G$ is* **CLTL** *valid is decidable.*

Proof. $F \dot{\Rightarrow} G$ iff $[\![F]\!] = sp(R_F) \subseteq [\![G]\!]$ by Definition 3.18 and Lemma 4.16. Then $F \dot{\Rightarrow} G$ iff $R_F \models_{\mathrm{CLTL}} G$ by Definition 3.20. The result follows from Theorem 4.17. $\square$

As an immediate consequence of the above theorem we obtain the following:

Corollary 4.19 *Given any restricted-negation formula F, the questions of whether F is* **CLTL** *valid and whether F is* **CLTL** *satisfiable are both decidable.*

Proof. F is **CLTL** valid iff $\texttt{true} \dot{\Rightarrow} F$ is **CLTL** valid, and F is **CLTL** satisfiable iff $F \dot{\Rightarrow} \texttt{false}$ is not **CLTL** valid. The result follows from Theorem 4.18. $\square$

5 Application: Manna and Pnueli's LTL

We now apply the previous results on our `ntcc` LTL (**CLTL**) to standard first-order LTL, henceforth called **LTL**, as presented by Manna and Pnueli in [18]. Namely, we obtain a new positive decidability result on the satisfiability of a first-order fragment of **LTL** by a reduction to that of **CLTL**. The relevance of our result is that **LTL** is not recursively axiomatizable [1] and, therefore, the satisfiability problem is undecidable for the full language of **LTL**. We confine ourselves to having $\circ$, $\square$, and $\diamondsuit$ as modalities. This is sufficient for making a recursive axiomatization impossible [19].

We shall recall briefly some LTL notions given in [18]. We presuppose an underlying first-order language $\mathcal{L}$ (including equality) with its (non-logical) symbols interpreted over some concrete countable domains such as the natural numbers. Furthermore, we assume that for each v in the interpreting domain, $\mathcal{L}$ has a constant term $\bar{v}$ whose interpretation is v. For instance, as in [18], for the natural numbers we may have the constants $0, 1, \ldots$ in the language.

States and Models. A *state* s is an interpretation that assigns to each variable x in $\mathcal{L}$ a value $s[x]$ over the appropriate domain. The interpretation is extended to $\mathcal{L}$ expressions in the usual way. For example, if f is a function symbol of arity 1, $s[f(x)] = f(s[x])$. We write $s \models c$ iff c is true wrt s in the given interpretation of the $\mathcal{L}$ symbols. For instance, if $+$ is interpreted as addition over the natural numbers and $s[x] = 42$ then $s \models \exists_y (x = y + y)$. We say that c is *state valid* iff $s \models c$ for every state s.

A *model* is an infinite sequence of states. We shall use σ to range over models. The variables of $\mathcal{L}$ are partitioned into *rigid* and *flexible* variables. Each model σ must satisfy the *rigidity* condition: If x is rigid and s, s' are two states in σ then $s[x] = s'[x]$. In other words flexible variables are those, that unlike the rigid ones, can change their assignment from one state to another.

LTL Syntax and Semantics. The syntax of **LTL** is that of **CLTL** given in Definition 3.15. In this case, x is a variable in $\mathcal{L}$ and c represents a first-order formula over $\mathcal{L}$.

The semantics of **LTL** is similar that of **CLTL** (Definition 3.18) except that now the formulae are satisfied by sequences of states. We then need to extend the notion of x-variant (Definition 3.17) to states: s is *x-variant* of s' iff $s[y] = s'[y]$ for every variable y in $\mathcal{L}$ different from x.

Definition 5.1 (LTL Semantics) *A model σ* satisfies *F in* **LTL***, notation $\sigma \models_{\text{LTL}} F$, iff $\langle \sigma, 1\rangle \models_{\text{LTL}} F$ where $\langle \sigma, i\rangle \models_{\text{LTL}} F$ is obtained from Definition 3.18 by replacing α and $\models_{\text{CLTL}}$ with σ and $\models_{\text{LTL}}$, respectively. We say that F is* **LTL** satisfiable *iff $\sigma \models_{\text{LTL}} F$ for some σ, and that F is* **LTL** valid *iff $\sigma \models_{\text{LTL}} F$ for all σ.*

5.1 **LTL** *Decidability*

In order to prove our decidability result, *we assume that state validity (the set of valid state formulae) is decidable.* From [17] we know that even under this assumption the **LTL** defined above is undecidable. In contrast, under the assumption, **LTL** satisfiability is decidable for the fragment in which temporal operators are not allowed within the scope of quantifiers as it can be reduced to that of propositional LTL [1].

Example 5.2 *Let us now illustrate the interaction between quantifiers, modalities, flexible and rigid variables in* **LTL***. The formula $\Box \dot{\exists}_u (x = u \mathbin{\dot{\wedge}} \circ x = u{+}1)$, where x is flexible and u is rigid, specifies sequences in which x increases by 1 from each state to the next. This example also illustrates that existential quantification over rigid variables provides for the specification of counter computations, so we may expect their absence to be important in our decidability result. In fact, we shall state the* **LTL** *decidability for the* restricted-negation *fragment (Definition 4.15) with flexible variables only.* $\Box$

Removing Existential Quantifiers. One might be tempted to think that, without universal quantification and without rigid variables, one could remove the existential quantifiers rather easily: Pull them into outermost position with the appropriate α-conversions to get a prenex form, then remove them since $\dot{\exists}_x F$ is **LTL** satisfiable iff F is **LTL** satisfiable. But this procedure does not quite work; it does not preserve satisfiability:

Example 5.3 *Let $F = (x = 42 \mathbin{\dot{\wedge}} \circ x \neq 42)$, $G = \dot{\exists}_x \Box F$ and $H = \Box \dot{\exists}_x F$ where x is flexible. One can verify that unlike H, $\Box F$ and thus G are not* **LTL** *satisfiable. Getting rid of existential quantifiers is not as obvious as it may seem.* $\Box$

Relating CLTL and LTL Satisfiability. Let us give some intuition on how to obtain a *reduction* from **LTL** satisfiability to **CLTL** satisfiability. In what follows we confine ourselves to restricted-negation formulae without rigid variables. One can verify that $\neg c$ and $\dot{\neg} c$ have the same **LTL** models. So, in the reduction we can assume wlg that F has no $\dot{\neg}$ symbols. Notice that $F = (x = 42 \dot{\vee} x \neq 42)$ is **LTL** valid but not **CLTL** valid (Example 3.19). However, F is satisfiable in both logics.

In the general case, it is easy to see that if a temporal formula F is **LTL** satisfiable then F is **CLTL** satisfiable. The idea is that an assignment $s[x] = v$ can be represented by the constraint $x = \bar{v}$ where the term $\bar{v}$ is interpreted as v—recall that from the assumptions about $\mathcal{L}$ such a $\bar{v}$ must exist. The other direction of the implication does not necessarily hold. For instance, $\Diamond\,\texttt{false}$ is not **LTL** satisfiable but it is **CLTL** satisfiable—recall from Section 3.5 that in **CLTL** $\texttt{false}$ is not the same as $\dot{\texttt{false}}$. For example, $\texttt{false}^{\omega} \models_{\text{CLTL}} \Diamond\,\texttt{false}$. Nevertheless, as shown in the next lemma, we can get around this mismatch by using $\Box\,\dot{\neg}\,\texttt{false}$ to exclude **CLTL** models containing $\texttt{false}$—i.e., $\beta.\,\texttt{false}\,.\alpha \not\models_{\text{CLTL}} \Box\,\dot{\neg}\,\texttt{false}$ for any $\beta \in \mathcal{C}^{\infty}$ and $\alpha \in \mathcal{C}^{\omega}$.

Recall that (Σ, Δ) (Definition 3.1) denotes the underlying constraint system and $\mathcal{L}$ denotes the underlying first-order language of state formulae. Also recall that both Δ and the set of valid state formulae are required to be decidable. The next lemma reduces **LTL** satisfiability to that of **CLTL**.

Lemma 5.4 *Assume that (Σ, Δ) has $\mathcal{L}$ as first-order language and Δ is the set of valid state formulae. Then*

$$F \textit{ is } \textbf{LTL} \textit{ satisfiable} \quad \textit{iff} \quad F \,\dot{\wedge}\, \Box\,\dot{\neg}\,\texttt{false} \textit{ is } \textbf{CLTL} \textit{ satisfiable,}$$

if F is a restricted-negation formula with no occurrences of $\dot{\neg}$ and with no rigid variables.

Proof.

- "If" direction. Let $\alpha = c_1.c_2.\dots$ such that $\alpha \models_{\text{CLTL}} F \,\dot{\wedge}\, \Box\,\dot{\neg}\,\texttt{false}$. By using induction on F one can verify that:

  $$\text{For every } \sigma = s_1.s_2.\dots \text{ such that } s_i \models c_i, \text{ we have } \sigma \models_{\text{LTL}} F.$$

  We show the case $F = \dot{\exists}_x F'$; the others are easier. Let $\sigma = s_1.s_2.\dots$ be an arbitrary sequence such that $s_i \models c_i$ for each $i > 0$.

  By the definition of $\models_{\text{CLTL}}$, there must exist an x-variant $\alpha' = c'_1.c'_2.\dots$ of α such that $\alpha' \models_{\text{CLTL}} F'$. So, for each $i > 0$, c_i and c'_i must only differ in the information about x; i.e. $\exists_x c_i = \exists_x c'_i$. Thus, for each $i > 0$, there must be a $s'_i \models c'_i$ that differs from s_i only in the assignment to x; i.e., s'_i is an

x-variant of s_i. But from the induction we know that for any $\sigma' = s'_1.s'_2.\dots$ such that $s'_i \models c'_i$ $(i \geq 1)$, we must have $\sigma' \models_{\mathrm{LTL}} F'$. From the definition of $\models_{\mathrm{LTL}}$, it follows that $\sigma \models F$.

We still have to show that there must exist at least one $\sigma = s_1.s_2.\dots$ such that $s_i \models c_i$ for each $i > 0$. To see this, notice that since $\alpha \models_{\mathrm{CLTL}} \Box \dot{\neg}\,\texttt{false}$ then $c_i \neq \texttt{false}$ for every $i > 0$. Thus, each c_i must have at least a satisfying assignment. Hence, F is **LTL** satisfiable as wanted.

- The "only if" direction. Suppose that $\sigma = s_1.s_2 \ldots \models_{\mathrm{LTL}} F$. Let $\alpha = c_1.c_2\ldots$ with each $c_i = (x_1 = \overline{v_1} \wedge \ldots \wedge x_n = \overline{v_n})$ where $x_1, \ldots, x_n$ are the free variables of F, and $s_i[x_1] = v_1, \ldots, s_i[x_n] = v_n$. It is easy to verify that $\alpha \models_{\mathrm{CLTL}} F$ using induction on the structure of F. Furthermore, each c_i is clearly not equivalent to $\texttt{false}$, and hence, $\alpha \models_{\mathrm{CLTL}} \Box \dot{\neg}\,\texttt{false}$. Thus, $F \dot{\wedge} \Box \dot{\neg}\,\texttt{false}$ is satisfiable. $\Box$

We can now state the decidability result we claimed for first-order LTL.

Theorem 5.5 (Decidability of LTL Satisfaction) *Let F be a restricted-negation formula without rigid variables. The question of whether F is* **LTL** *satisfiable is decidable.*

Proof. From Lemma 5.4 and Corollary 4.19, and the fact that one can freely replace $\dot{\neg}$ by $\neg$ in F and the resulting formula will have the same **LTL** models than the original F. $\Box$

6 Concluding Remarks

We presented positive decidability results for the sp behavioral equivalence, the `ntcc` verification problem, and the `ntcc` specification first-order temporal logic **CLTL** . These results apply to *infinite-state* processes. A somewhat interesting aspect is that for proving the results it turned out to be convenient to work with the `ntcc` *denotational semantics* rather than with its operational counterpart. Also the use of Büchi automata-theoretic techniques in these results highlights the automata-like flavor of `ntcc`.

Furthermore, by using a reduction to **CLTL** satisfiability, we identified a first-order fragment of the standard LTL [18] for which satisfiability is decidable. The result contributes to the understanding of the relation between (timed) ccp and (temporal) classic logic and also illustrates the applicability of timed ccp to other theories of concurrency.

6.1 Related Work

We already discussed previous related work in Section 4.1. The work in [25] proves the decidability of the sp equivalence and other behavioral equivalences for several deterministic timed ccp languages. Another work [27] shows that output equivalence is decidable for a restricted nondeterministic `ntcc` fragment. The results in [25,27] are obtained by showing that the processes in these languages are indeed *finite-state*. In contrast, in this paper we dealt with *infinite-state* processes.

Saraswat et al [37] showed how to compile parameterless recursion tcc processes (basically finite-state deterministic `ntcc` processes) into FSA in a compositional way. Such FSA provide a simple and useful execution model for tcc but not a direct way of verifying sp (or input-output) equivalence. In fact, unlike our FSA constructions, the standard language equivalence between these FSA does not necessarily imply sp equivalence (or input-output) of the processes they represent.

Another interesting approach to timed ccp verification is that in [11]. The authors show how to construct structures (models) of tcc processes which then, by restricting the domains of variables to be finite, can be used for model-checking. Notice that in our results we make no assumptions about the domains of variables being finite.

The notion of constraint in other declarative formalisms such as Constraint Logic Programming (CLP) and Constraint Programming (CP) has also been used for the verification of infinite-state systems. The work in [8] shows how to translate infinite-state systems into CLP programs to verify safety and liveness properties. Esparza and Melzer [10] used CP in a semi-decision algorithm to verify 1-safe Petri Nets.

Merz [19] and Hodkinson et al [17] identified interesting decidable first-order fragments of LTL. These fragments are all monadic and without equality. A difference with our work is that these fragments do not restrict the use of negation or rigid variables, and our fragment is not restricted to be monadic or equality-free.

6.2 Future Work

The results in this paper depend on the translation from formulae into processes (which are then translated into FSA), hence our restriction on the occurrences of negation. Nevertheless, the author believes that we can dispense with this restriction in our decidability results. An approach for proving this claim could be to find a finite representation of the infinitely many constraints that may hold at any time unit for a given formula. If this can be done, we can then translate formulae directly into FSA. If the claim is true, then the inference system for `ntcc` [28] would be complete for locally-independent processes (and not just relative complete). This is because we would be able to determine the validity of arbitrary `ntcc` LTL implication as required by the consequence rule. It will be, therefore, interesting to be able to prove this claim.

Despite their theoretical interest, our Büchi constructions are very inefficient—see the complementation construction for the replication automaton in Section 4.2. For practical purposes, it is important to conduct studies to obtain more efficient constructions. In particular, preliminary studies indicate that by using *alternating* Büchi FSA [24] we can avoid complementation when representing the sp of processes and obtain a much better complexity.

Acknowledgments. The author is indebted to Catuscia Palamidessi, Mogens Nielsen, Joachim Parrow, Jiri Srba, Richard Mayr, Gerardo Schneider and Martin Leucker for helpful suggestions on the presentation of this paper. Thanks also goes to Igor Walukiewicz, Martin Abadi, Ian Hodkinson, and Stephan Merz for providing information about the decidability of First-Order LTL.

Many thanks to anonymous referees. Their constructive criticism was very valuable and each one of their suggestions was of great help for improving the previous version of this work.

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] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Computer and System Sciences*, 10:19–35, 1974.

[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] G. Delzanno and A. Podelski. Model checking in clp. In *Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579, pages 223–239. LNCS, 1999.

[9] J.F. Diaz, C. Rueda, and F. Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), 1998.

[10] J. Esparza and S. Melzer. Model checking LTL using constraint programming. In *18th International Conference on Application and Theory of Petri Nets*, volume 1248, pages 1–20. Springer-Verlag, 1997.

[11] M. Falaschi, A. Policriti, and A. Villanueva. Modelling timed concurrent systems in a temporal concurrent constraint language - i. In *Selected Papers from 2000 Joint Conference on Declarative Programming*, volume 48. Elsevier, 2000.

[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] 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.

[17] I. Hodkinson, F. Wolter, and M. Zakharyasche. Decidable fragments of first-order temporal logic. *Ann. Pure. Appl. Logic*, 106:85–134, 2000.

[18] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer, 1991.

[19] S. Merz. Decidability and incompleteness results for first-order temporal logics of linear time. *Journal of Applied Non-Classical Logic*, 2(2), 1992.

[20] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[21] R. Milner. A finite delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1992.

[22] R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.

[23] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7, 1974.

[24] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theor. Comput. Sci.*, 54(2-3):267–276, 1987.

[25] M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of concurrent constraint programming languages. In *Proc. of the 4th International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 156–167. ACM Press, 2002.

[26] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(2):145–188, 2002.

[27] M. Nielsen and F. Valencia. Temporal concurrent constraint programming: Applications and behavior. In *Formal and Natural Computing: Essays Dedicated to Grzegorz Rozenberg*, chapter 4, pages 298–324. Springer-Verlag, LNCS 2300, 2002.

[28] C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming (CP'01).* Springer-Verlag, LNCS 2239, 2001.

[29] F. Rossi and U. Montanari. Concurrent semantics for concurrent constraint programming. In *Constraint Programming: Proc. 1993 NATO ASI*, pages 181–220, 1994.

[30] J.H. Réty. Distributed concurrent constraint programming. *Fundamenta Informaticae*, 34(3), 1998.

[31] 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.

[32] V. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[33] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*, pages 71–80, 1994.

[34] 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.

[35] V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proc. of POPL'95*, pages 272–285, 1995.

[36] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, 1996.

[37] V. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91*, pages 333–352, 1991.

[38] E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510, 1990.

[39] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[40] G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.

[41] S. Tini. On the expressiveness of timed concurrent constraint programming. *Electronics Notes in Theoretical Computer Science*, 1999.

[42] F. Valencia. *Temporal Concurrent Constraint Programming.* PhD thesis, BRICS, University of Aarhus, 2003.

[43] F. Valencia. Timed concurrent constraint programming: Decidability results and their application to LTL. In *Proc. of ICLP'03*. Springer-Verlag, LNCS, 2003.

[44] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.