Models and Emerging Trends of Concurrent Constraint Programming

Carlos Olarte, Camilo Rueda, Frank D. Valencia

Models and Emerging Trends of Concurrent Constraint Programming [*][†]

Carlos Olarte[‡] Camilo Rueda [§] Frank D. Valencia [¶]

June 2013

Abstract

Concurrent Constraint Programming (CCP) has been used over the last two decades as an elegant and expressive model for concurrent systems. It models systems of agents communicating by posting and querying partial information, represented as constraints over the variables of the system. This covers a vast variety of systems as those arising in biological phenomena, reactive systems, net-centric computing and the advent of social networks and cloud computing. In this paper we survey the main applications, developments and current trends of CCP.

1 Introduction

Concurrent Constraint Programming (CCP) [181, 186, 187] is a well-established formalism for concurrency based upon the shared-variables communication model. The CCP model traces its origins back to the ideas of *computing with constraints* [142, 200, 202, 210], *Concurrent Logic Programming* [194, 123] and *Constraint Logic Programming* (CLP) [134, 116, 117]. It was designed for giving programmers and models explicit access to the concept of partial information, traditionally referred to as *constraints*. CCP is intended for reasoning, modeling and programming concurrent and reactive agents (or processes) that interact with each other and their environment by posting and asking information in a medium, a so-called *store*. The CCP formalism enables a unified representation of agents: they can be seen as both computing processes (behavioral imperative style) and as formulas in logic (logical declarative style). Due to this dual view of processes, CCP can benefit from the development and mathematical apparatus of well-established formalisms such as process calculi and logic. In addition, it has been shown as a flexible and versatile framework where several variants can be experimented with and validated.

[*]Preprint of the paper: *Carlos Olarte, Camilo Rueda, Frank D. Valencia: Models and emerging trends of concurrent constraint programming. Constraints 18(4): 535-578 (2013)*

[†]This work has been partially supported by the project ANR 12IS02001 PACE.

[‡]Pontificia Universidad Javeriana Cali, Colombia. carlosolarte@puj.edu.co

[§]Pontificia Universidad Javeriana Cali, Colombia. crueda@puj.edu.co

[¶]CNRS, LIX, École Polytechnique, France. frank.valencia@lix.polytechnique.fr

The CCP model has received a significant theoretical and implementational attention over the last two decades. In this survey, we shall provide a unified presentation of the most important developments, applications and state of the art in CCP. We aim at giving a broad perspective to the reader about the calculus, its variants, applications and reasoning techniques, as well as recent trends in the area. We start by recalling in Section 2.1 the notion of constraint system that makes CCP a flexible model able to adapt to different application domains. Thereafter, Section 2.2 presents the core language of CCP.

Similar to other mature models of concurrency, such as Petri nets [162, 170] and process calculi [47, 19, 140], CCP has been extended also to capture the behavior of different phenomena in emerging systems. Section 3 presents the developments of CCP calculi in order to deal with: discrete and continuous time and asynchrony for reactive systems; stochastic behavior for physical and biological systems; and linearity (consumption of constraints) for imperative data structures. Furthermore, we shall show the extensions of CCP to cope with: soft constraints where agents can express preferences; communication of local names or links (i.e. mobility) for net-centric computing and protocols; and epistemic and spatial modalities for social networks and cloud computing. In order to give a unified view of these developments, we shall use the same notation of the core language in Section 2.2 that may marginally differ from the original publications. We also present the operational semantics of the calculi and we follow a simple running example to improve the understanding.

Section 4 is devoted to summarize different programming languages whose bases are the CCP model and we also refer some implementations of interpreters for CCP calculi. In Section 5 we show the applicability of CCP and its extensions to model systems in a wide spectrum of scenarios including physical and reactive systems, biological phenomena, multimedia interaction, and net-centric computing.

An appealing feature of the concurrent constraint programming model is that it offers a large set of reasoning techniques for studying the modeled systems. Such techniques come from the theory of process calculi as well as from logic thanks to the declarative reading of CCP agents as logical formulas. In Section 6 we describe the elegant denotational semantics for CCP based on closure operators and the program analysis techniques developed for the language. Furthermore, we summarize the developments of logical characterizations of processes, inference systems and model checking techniques for proving properties of CCP models. We also describe the current efforts to endow CCP with bisimulation reasoning techniques as standardly done in process calculi. Section 7 concludes the paper.

The reader may also refer [95], an article included in the book that celebrates the 25th anniversary of the Italian Association for Logic Programming (GULP) [68]. It presents an overview of the contributions of the Italian research community to CCP.

2 Constraints and Agents

Concurrent Constraint Programming (CCP) [181, 186, 187] has emerged as a simple but powerful paradigm for concurrency tied to logic. Under this paradigm, the conception of *store as valuation* in the von Neumann model is replaced by the notion of *store as constraint*, and *processes* are seen as *information transducers*. The CCP model of computation makes use of *ask* and *tell* operations instead of the classical *read* and *write*. An ask operation tests if a given piece of information (i.e., a constraint as in $x > 42$) can be deduced from the store. A *tell* operation conjoins the store with a new constraint to augment the information in it. This is a very general paradigm that extends and subsumes both Concurrent Logic Programming [194, 123] and Constraint Logic Programming [134, 116, 117].

A fundamental issue in CCP is then the specification of concurrent systems by means of constraints that represent partial information about certain variables. The state of the system is specified by the *store* (i.e., a constraint) that is *monotonically refined* by adding new information.

In the spirit of process calculi [47, 19, 140], the language of processes in the CCP model is given by a small number of primitive operators or combinators. A typical CCP process language is equipped with the following operators:

- A *tell* operator adding a constraint to the store.
- An *ask* operator querying if a constraint can be deduced from the store.
- *Parallel Composition* combining processes concurrently.
- A *hiding* operator (also called *restriction* or *locality*) introducing local variables and thus restricting the interface a process can use to interact with others.

Additionally, infinite computations can be described by means of *recursion* or *replication* as we shall see later.

Central to CCP is the notion of *constraint system*. Roughly, a constraint system specifies the basic constraints agents can tell and ask during computation. This makes the language parametric and hence, versatile to be used in different contexts. Next we recall this idea to later introduce the language of CCP processes.

2.1 Constraint Systems

CCP calculi are parametric in a *constraint system*. A constraint system provides a signature from which the constraints can be constructed as well as an entailment relation, notation $\vdash$, specifying inter-dependencies between these constraints. Recall that a constraint represents a piece of information (or partial information) upon which processes may act.

In the literature, the notion of constraint system has been set up in two alternative ways: 1) in terms of Scott's information systems [193] without consistency structure as in [187] and 2) as first-order logic (FOL) formulas as in [180, 195]. In the following we explain these two formulations.

2.1.1 Cylindric Constraint Systems

Processes in CCP can add constraints that lead to an *inconsistent* store. Therefore, it is necessary to represent the possibility of inconsistent information. Constraint systems in this approach are then formalized as algebraic structures with operators to express conjunction of constraints, absence of information, inconsistent information, hiding of information and parameter passing. More precisely, following the presentation in [41]:

Definition 1 (Constraint System) *A cylindric constraint system is a structure* $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathtt{true}, \mathtt{false}, \mathit{Var}, \exists, D \rangle$ *such that*

- $\langle \mathcal{C}, \leq, \sqcup, \mathtt{true}, \mathtt{false} \rangle$ *is a lattice with* $\sqcup$ *the lub operation (representing the logical* and*), and* $\mathtt{true}$*,* $\mathtt{false}$ *the least and the greatest elements in* $\mathcal{C}$*, respectively. Elements in* $\mathcal{C}$ *are called* constraints *with typical elements* $c, d \ldots$*.*

- Var *is a denumerable set of variables and for each* $x \in \mathit{Var}$ *the function* $\exists x : \mathcal{C} \to \mathcal{C}$ *is a cylindrification operator satisfying: (1)* $\exists x(c) \leq c$*; (2) if* $c \leq d$ *then* $\exists x(c) \leq \exists x(d)$*; (3)* $\exists x(c \sqcup \exists x(d)) = \exists x(c) \sqcup \exists x(d)$*; and (4)* $\exists x \exists y(c) = \exists y \exists x(c)$*.*

- *For each* $x, y \in \mathit{Var}$*,* $d_{xy} \in D$ *is a* diagonal element *and it satisfies: (1)* $d_{xx} = \mathtt{true}$*; (2) if* z *is different from* x, y *then* $d_{xy} = \exists z(d_{xz} \sqcup d_{zy})$ *and (3) If* x *is different from* y *then* $c \leq d_{xy} \sqcup \exists x(c \sqcup d_{xy})$*.*

The concepts of *cylindrification operators* and *diagonal elements* were borrowed from the theory of cylindric algebras [110]. The cylindrification operator models a sort of existential quantification, helpful for defining the CCP hiding (local) operator and it is assumed to be a continuous function [187, 41]. The diagonal elements are useful to model parameter passing in procedure calls The constraint d_{xy} can be thought of as the equality $x = y$.

The entailment relation ($\vdash$) is the reverse of the ordering $\leq$, i.e., we say that d *entails* c iff $c \leq d$ and we write $d \vdash c$. For operational reasons $\vdash$ is often required to be decidable. The entailment relation is defined in [187] only on the compact elements of $\mathcal{C}$ (finite sets of *tokens*, or basic constraints) while this restriction is not required in [41]. Nevertheless, in both cases, processes are only allowed to ask and tell basic (finite) constraints.

2.1.2 Constraint System as FOL formulas

The notion of constraint system as first order logic (FOL) formulas can be seen as an instance of the previous definition. It gives a logical flavor that has found widespread use in the literature (see e.g., [195, 77, 153, 157]).

Definition 2 (Constraint System as FOL formulas) *A constraint system is a pair* (Σ, Δ) *where* Σ *is a signature of constant, function and predicate symbols, and* Δ *is a first order theory over* Σ *(i.e., a set of non-logical axioms). Let* $\mathcal{L}$ *be the underlying first-order language under* Σ *with variables* $x, y, \ldots$*, and logic symbols* $\wedge, \exists, \mathtt{true}$ *and* $\mathtt{false}$*. Constraints are first-order formulas over* $\mathcal{L}$*.*

Under this definition, we say that c entails d iff the implication $c \Rightarrow d$ is true in all models of Δ. In the rest of the paper, for the sake of presentation, we shall make use only of this definition of constraint system.

Other realizations of constraint systems have given rise to CCP idioms. For instance, constraints as Girard's linear logic [97] formulas allowed to develop the theory of linear CCP [77] where agents can *consume* information from the store. Semiring-based constraints [25] allowed to define Soft Concurrent Constraint Programming [29] where agents can post and ask *soft constraints* in order to express preferences, fuzziness, probabilities, etc. Moreover, by adding space functions into the structure of the constraint system, CCP calculi devised for epistemic and spatial reasoning where studied in [124]. We will come back to these definitions in Section 3 where we give an account of different CCP-based calculi.

2.1.3 Examples of Constraint Systems

Let P be a process modeling a temperature controller. Hence, P may have to deal with partial information such as $tsensor > 42$ expressing that the sensor registers an unknown (or not precisely determined) temperature value above 42. Furthermore, P may query if $tsensor > 0$. Then, the constraint system must provide an entailment relation to assert that this information can be inferred, for instance, from the information $tsensor > 42$ (i.e., $tsensor > 42 \vdash tsensor > 0$). In order to deal with such type of constraints, one can consider the finite domain constraint system (FD) [111]. FD variables are assumed to range over finite domains and, in addition to equality, one may have predicates that restrict the possible values of a variable to some finite set.

Another example is the Herbrand constraint system [187] underlying logic programming where a first-order language with equality is assumed. The entailment relation is the one we expect from equality, for instance, $f(x, y) = f(a, g(z))$ must entail $x = a$ and $y = g(z)$.

The reader may find in [187] more examples of constraint systems such as the Kahn Constraint System underlying data-flow languages and the Rational Interval Constraint System. We also point to [65] and [149] that describe, respectively, an implementation of a real interval constraint system and a constraint system to deal with finite sets. Modern constraint solvers as Gecode (`http://www.gecode.org/`) can indeed be used as basis for the implementation of interpreters for CCP languages (see Section 4.7).

2.2 The Language of CCP Processes

Now that we have set up the notion of constraint system, we are ready to define the language of processes. By using the notation in [153]:

Definition 3 (Syntax of CCP [187]) *Processes $P, Q, \ldots$ in CCP are built from constraints in the underlying constraint system by the following syntax:*

$$P, Q := \mathbf{skip} \mid \mathbf{tell}(c) \mid \mathbf{when}\, c\, \mathbf{do}\, P \mid P \parallel Q \mid (\mathbf{local}\, \overline{x}; c)\, P \mid q(\overline{x})$$

The process $\mathbf{skip}$ does nothing and represents inaction. The process $\mathbf{tell}(c)$ adds the constraint c to the store. The process $\mathbf{when}\ c\ \mathbf{do}\ P$ *asks* if c can be deduced from the store. If so, it behaves as P. In other case, it waits until the store contains at least as much information as c. This way, ask agents define a powerful synchronization mechanism based on entailment of constraints.

The parallel composition of the processes P and Q is represented as $P \parallel Q$. The process $(\mathbf{local}\,\overline{x}; c)\,P$ behaves like P, except that all the information on the variables $\overline{x}$ produced by P (represented as c) can only be seen by P and the information on $\overline{x}$ produced by other processes cannot be seen by P. In fact, what the other processes can observe from c is $\exists\overline{x}(c)$ and what P can observe from the store d is $\exists\overline{x}(d)$. If there is no local information (i.e., $c = \mathtt{true}$), it is customary to write $(\mathbf{local}\,\overline{x})\,P$ instead of $(\mathbf{local}\,\overline{x}; \mathtt{true})\,P$.

The process $q(\overline{y})$ is an *identifier* with arity $|\overline{y}|$. We assume that every such an identifier has a unique (recursive) definition of the form $q(\overline{x}) \stackrel{\mathtt{def}}{=} Q$ where $\overline{x}$ are pairwise distinct and $|\overline{x}| = |\overline{y}|$. Then, $q(\overline{y})$ behaves as $Q[\overline{y}/\overline{x}]$. The process $Q[\overline{y}/\overline{x}]$ is actually modeled, with the help of diagonal elements, as $\Delta^{\overline{y}}_{\overline{x}} = (\mathbf{local}\,\overline{z}; d_{\overline{yz}})\,(\mathbf{local}\,\overline{x}; d_{\overline{xz}})\,(Q)$ where $d_{\overline{xy}}$ is the constraint $\bigsqcup_{i\in 1\ldots|\overline{x}|} d_{x_i y_i}$.

2.3 Operational Semantics

The structural operational semantics (SOS) of CCP can be described by means of configurations of the form $\langle P, c\rangle$ where P is a process and c represents the current store. Then, a transition of the form $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$ dictates that P under the store c evolves into P' and produces the store c'. Remember that processes in CCP can only add information. Then, by monotonicity, it must be the case that $c' \vdash c$. Figure 1 shows the SOS for the CCP processes in Definition 3.

Let us explain the Rule $\mathrm{R_L}$ as it may seem somewhat complex. Consider the process $Q = (\mathbf{local}\,x; c)\,P$. The global store is d and the local store is c. We distinguish between the *external* (corresponding to Q) and the *internal* points 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 first hide the information about x that Q may have in d by existentially quantifying x in d. Similarly, from the external point of view, the observable information about x that the reduction of internal agent P may produce (i.e., c') cannot be observed. Thus we hide it by existentially quantifying x in c' before adding it to the global store. Additionally, we make c' the new private store of the evolution of the internal process.

The semantics of the local operator can also be given by using *fresh* variables, i.e., variables that do not appear elsewhere in the processes as in [138, 77, 105]. In this case, configurations are augmented with a set of *eigenvariables*.

Observe that parallel composition is defined above in terms of two rules, noted $\mathrm{R_{PL}}$ and $\mathrm{R_{PR}}$. It is also possible to define parallel composition with a single rule: this is done by internalizing into the semantics a structural congruence relation which equates processes with minor syntactic differences, such as $P \parallel Q$ and $Q \parallel P$ (see e.g., [153]).

An observable (or output) of a process P under input c is then a constraint d such that $\langle P, c\rangle \longrightarrow^* \langle P', d\rangle \not\longrightarrow$ where $\longrightarrow^*$ is the transitive and reflexive closure of the relation $\longrightarrow$. Depending on the semantic framework, one may be interested only in finite computations and/or the limit of infinite ones [187, 41].

We conclude this section with the well known example of concatenation of lists written in CCP taken from [187]:

$$\mathrm{R_T}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle}$$

$$\mathrm{R_A}\ \frac{d \vdash c}{\langle \mathbf{when}\ c\ \mathbf{do}\ P, d\rangle \longrightarrow \langle P, d\rangle}$$

$$\mathrm{R_{PL}}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\mathrm{R_{PR}}\ \frac{\langle Q, c\rangle \longrightarrow \langle Q', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P \parallel Q', d\rangle}$$

$$\mathrm{R_L}\ \frac{\langle P, c \wedge (\exists \overline{x} d)\rangle \longrightarrow \langle P', c' \wedge (\exists \overline{x} d)\rangle}{\langle (\mathbf{local}\,\overline{x}; c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\,\overline{x}; c')\, P', d \wedge \exists \overline{x} c'\rangle}$$

$$\mathrm{R_C}\ \frac{q(\overline{x}) \stackrel{\mathtt{def}}{=} Q \text{ is a process definition}}{\langle q(\overline{y}), c\rangle \longrightarrow \langle Q[\overline{y}/\overline{x}], c\rangle}$$

Figure 1: Operational semantics for the CCP language in Definition 3.

Example 1 (Append) *Assume the Herbrand constraint system where constraints are existentially quantified conjunctions of equations over a given set of terms (e.g., $L = []$ represents that L is the empty list and $\exists x \exists y (L = [x \mid y])$ that L is a list with head x and tail y). The process below concatenates the lists L_1 and L_2 into L_3:*

$$\begin{array}{lcl} \mathit{append}(L_1, L_2, L_3) & \stackrel{\mathtt{def}}{=} & \mathbf{when}\ L_1 = []\ \mathbf{do}\ \mathbf{tell}(L_3 = L_2) \ \parallel \\ & & \mathbf{when}\ \exists x \exists y (L_1 = [x \mid y])\ \mathbf{do}\ \ (\mathbf{local}\, x, y, z)\, (\\ & & \quad \mathbf{tell}(L_1 = [x \mid y]) \parallel \mathbf{tell}(L_3 = [x \mid z]) \parallel \mathit{append}(y, L_2, z)) \end{array}$$

2.4 Non-determinism

Non-determinism arises in CCP by introducing *guarded choices*.

Definition 4 (Non-deterministic CCP [187]) *Non-determinism is obtained by replacing the ask operator* $\mathbf{when}\ c\ \mathbf{do}\ P$ *with the guarded choice* $\sum\limits_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ *where I is a finite set of indexes.*

The process $\sum\limits_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ non-deterministically chooses one of the P_j whose corresponding guard (constraint) c_j is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible then the summation remains blocked until more information is added to the store:

$$\mathrm{R_{SUM}}\ \frac{d \vdash c_j \quad j \in I}{\langle \sum\limits_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i, d\rangle \rightarrow \langle P_j, d\rangle}$$

Example 2 (Consumer-producer streams) *Let* $\mathtt{append}$ *be as in the Example 1 and assume the following process definitions*

$$\begin{array}{lcl} \mathit{prod}_a(x) & \stackrel{\mathtt{def}}{=} & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ (\mathbf{local}\, x')\,(\mathbf{tell}(x = [a|x']) \parallel \mathit{prod}_a(x')) + \\ & & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\, x = [] \\ \mathit{prod}_b(x) & \stackrel{\mathtt{def}}{=} & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ (\mathbf{local}\, x')\,(\mathbf{tell}(x = [b|x']) \parallel \mathit{prod}_b(x')) + \\ & & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\, x = [] \end{array}$$

The process $\mathit{prod}_a(x) \parallel \mathit{prod}_b(y) \parallel \mathit{append}(x, y, z)$ *binds* x *and* y *to a (possibly infinite) list of* a*'s and* b*'s respectively and* z *to the concatenation of* x *and* y*.*

3 Constraint-Based Concurrent Calculi

Several extensions of the CCP model have been studied in order to provide frameworks for the programming, specification and verification of systems with the declarative flavor of concurrent constraint programming. This section is devoted to describe these developments.

3.1 Timed Concurrent Constraint Programming

Reactive systems [20, 107] are those that react continuously with their environment at a rate controlled by the environment. For example, a controller or a signal-processing system, receives a stimulus (input) from the environment, computes an output and then waits for the next interaction with the environment.

In CCP, the shared store of constraints grows monotonically, i.e., agents cannot drop information (constraints) from it. Then, a system that changes the state of a signal (i.e., the value of a variable) cannot be straightforwardly modeled: the conjunction of the constraints "$\mathit{signal} = \mathit{on}$" and "$\mathit{signal} = \mathit{off}$" leads to an inconsistent store [1].

Timed CCP (`tcc`) [182] extends CCP for reactive systems and elegantly combines CCP with ideas from the paradigm of Synchronous Languages [20]. Time in `tcc` is conceptually divided into *time intervals* (or *time-units*). In a particular time interval, a CCP process P gets an input 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 also a residual process Q that is then executed in the next time-unit. The resulting store d is not automatically transferred to the next time-unit. This way, computations during a time-unit proceed monotonically but outputs of two different time-units are not supposed to be related. Therefore, the variable signal in the example above may change its value when passing from one time-unit to the next one.

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

[1] It is possible however to make use of streams to represent changes in the value of a variable.

The `tcc` calculus extends the language of CCP processes with constructs that: (1) *delay* the execution of a process; and (2) *time-out* (or weak pre-emption) operations that wait during the current time interval for a given piece of information to be present, if it is not, they trigger a process in the next time interval.

Definition 5 (Deterministic `tcc` [182]) *The syntax of `tcc` is obtained by adding to Definition 3 the processes* $\mathbf{next}\, P$ *and* $\mathbf{unless}\ c\ \mathbf{next}\ P$.

The process $\mathbf{next}\, P$ delays the execution of P to the next time interval. The *time-out* $\mathbf{unless}\ c\ \mathbf{next}\ P$ is also a unit-delay, but P is executed in the next time-unit only if c is not entailed by the final store at the current time interval. Notice that, in general, $P = \mathbf{unless}\ c\, \mathbf{next}\ R$ is not the same as $Q = \mathbf{when}\ \neg c\ \mathbf{do}\ \mathbf{next}\ R$. Take for instance $x = 1$ as the constraint c. From the store `true`, one cannot deduce neither $x = 1$ nor $x \neq 1$. In this particular case, the process P executes R (provided that the final store is `true`) while Q remains blocked.

In `tcc`, recursive calls must be guarded by a **next** operator to avoid infinite computations during a time-unit. If the temporal language does not consider recursive definitions, it is customary to add the *replication* $!\, P$ in order to give processes the possibility to be executed in infinitely many time-units. The *replication* $!\, P$ 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. The reader may refer to [152] for a detailed account of the expressiveness power of temporal CCP languages with and without recursion and replication.

In spite of its simplicity, the `tcc` extension to CCP is far-reaching. Many interesting temporal constructs can be expressed as it is shown in [182]. 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 SOS of `tcc` considers *internal* and *observable* transitions. The internal transitions are similar to those in Figure 1 and correspond to the operational steps that take place during a time-unit. As for the timed operators, the **unless** process evolves into **skip** if its guard can be entailed from the current store:

$$\mathrm{R_U}\ \frac{d \vdash c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

The seemingly missing rule for the **next** operator will be clarified soon.

The *observable transition* $P \overset{(c,d)}{\Longrightarrow} Q$ should be read as "P on input c, reduces in one *time-unit* to Q and outputs d". The observable transitions are obtained from finite sequences of internal ones, i.e., $P \overset{(c,d)}{\Longrightarrow} Q$ whenever $\langle P, c\rangle \longrightarrow^* \langle R, d\rangle \not\longrightarrow$. The process Q (the continuation for the next time-unit) is obtained from:

$$F(R) = \begin{cases} \mathbf{skip} & \text{if } R = \mathbf{skip} \text{ or } R = \mathbf{when}\ c\ \mathbf{do}\ R' \\ F(R_1) \parallel F(R_2) & \text{if } R = R_1 \parallel R_2 \\ (\mathbf{local}\, \overline{x})\, F(R') & \text{if } P = (\mathbf{local}\, \overline{x}; c)\, R' \\ Q & \text{if } R = \mathbf{next}\, R' \text{ or } R = \mathbf{unless}\ c\ \mathbf{next}\ R' \end{cases}$$

The function $F(R)$ (the future of R) unfolds *next* and *unless* expressions. Notice that an *ask* process reduces to **skip** if its guard was not entailed by the final store d. Notice also that F is not defined for $\mathbf{tell}(c)$, $!\,Q$ or $p(\overline{x})$ processes since they must occur within a $\mathbf{next}$ or $\mathbf{unless}$ expression.

3.1.1 Strong Pre-Emption in Timed CCP

Modeling some systems may require detecting negative information (i.e., the *absence* of information) and react *instantaneously*. Take for instance a process P that needs to be aborted to *immediately* start a process Q when a specific event occurs. This is usually called pre-emption of P. The work in [183] introduces the notion of strong pre-emption in `tcc`: on the absence of a constraint, time-out operations can trigger activity in the current time interval rather than delaying it to the next interaction with the environment as in `tcc`. Borrowing ideas from default logics [171], **Default** `tcc` is defined as follows:

Definition 6 (Default `tcc` [183]) *The syntax of* **Default** `tcc` *is obtained by adding to Definition 5 the constructor* $\mathbf{when}\ c\ \mathbf{else}\ P$.

Unlike $\mathbf{unless}\ c\ \mathbf{next}\ P$ in `tcc`, the process $\mathbf{when}\ c\ \mathbf{else}\ P$ reduces to P at the present time instant if c has not been produced in the store, and will not be produced throughout system execution. The transition system is then parametrized with a constraint e, representing the final "guess" that is used to evaluate defaults:

$$\frac{d \vdash c}{\langle \mathbf{when}\ c\ \mathbf{else}\ P, d\rangle \longrightarrow_e \langle \mathbf{skip}, d\rangle}\ \mathrm{R_{DA}} \qquad \frac{e \not\vdash c}{\langle \mathbf{when}\ c\ \mathbf{else}\ P, d\rangle \longrightarrow_e \langle P, d\rangle}\ \mathrm{R_{DA'}}$$

Example 3 (Default values for variables [183]) *The following program:*

$$\mathtt{default}(x, v) \overset{\mathrm{def}}{=} \mathbf{when}\ x \neq v\ \mathbf{else}\ x = v$$

sets the value of x to be v unless the current value of x is different from v.

3.1.2 Non-Determinism and Asynchrony in `tcc`

The notion of *non-determinism* and *asynchrony* is ubiquitous in concurrent systems. As for non-determinism, a system may exhibit different behaviors when reacting to the same input. As for asynchrony, we may have processes that occur at some point in the future time but we do not know exactly when as in a failure model for a component that is doomed to fail. The `ntcc` calculus [153] extends `tcc` by adding guarded-choices for modeling non-deterministic behavior and an unbounded finite-delay operator for asynchronous behavior. Computation in this language progresses as in `tcc`, except for the non-determinism induced by the new constructs:

Definition 7 (`ntcc` Processes [153]) *The* `ntcc` *processes result from adding to the syntax in Definition 5 two constructs,* $\sum\limits_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ *and* $\star P$.

The guarded-choice is similar to that in Definition 4. The operator "$\star$" corresponds to the unbounded but finite delay operator ϵ for synchronous CCS [139] and it allows to express asynchronous behavior through the time intervals. Intuitively, the process $\star P$ represents $P + \mathbf{next}\, P + \mathbf{next}^{\,2} P + ...$, i.e., an arbitrary long but finite delay for the activation of P:

$$\mathrm{R_S}\ \frac{n \geq 0}{\langle \star P, d\rangle \rightarrow \langle \mathbf{next}^{\,n} P, d\rangle}$$

Example 4 (Controller for a Robot) *Assume a controller for a robot where the environment sends signals when the device has to turn. Let*

$$\begin{aligned} \mathtt{TurnR} \stackrel{\text{def}}{=}\ & \mathbf{when}\ dir = \mathtt{N}\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(dir = \mathtt{E}) + \mathbf{when}\ dir = \mathtt{E}\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(dir = \mathtt{S}) + \\ & \mathbf{when}\ dir = \mathtt{S}\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(dir = \mathtt{W}) + \mathbf{when}\ dir = \mathtt{W}\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(dir = \mathtt{N}) \end{aligned}$$

be a process that changes in the next time-unit the value of dir according to its current value. The process `TurnL` *can be defined similarly. The process below specifies that the robot turns left or right when the environment provides the signal* `turn`*:*

$$\begin{aligned} \mathtt{Robot} \stackrel{\text{def}}{=}\ & \mathbf{next}\ \mathtt{Robot} \parallel \mathbf{when}\ \mathtt{turn}\ \mathbf{do}\ \mathtt{TurnR} + \mathbf{when}\ \mathtt{turn}\ \mathbf{do}\ \mathtt{TurnL} \parallel \\ & \sum_{i \in \{\mathtt{N},\mathtt{E},\mathtt{S},\mathtt{W}\}} \mathbf{when}\ dir = i\ \mathbf{do}\ \mathbf{unless}\ \mathtt{turn}\ \mathbf{next}\ dir = i \end{aligned}$$

The summation above specifies that the direction in the next time-unit is the same as in the current one unless the constraint `turn` *can be deduced.*

3.1.3 The `tccp` Language

In the `tcc` and `ntcc` models of computation, stores are not automatically transferred to the next time-unit. Therefore, the store is monotonically refined in each time-unit but outputs of two different time-units are not supposed to be related to each other. The `tccp` process calculus [34] is an orthogonal timed non-deterministic extension of CCP. In this language, time is identified with the time needed to ask and tell information to the store. Furthermore, unlike `tcc`, the information in the store is carried through the time-units. Then, streams are used in `tccp` to model the evolution of variable values along the time.

Definition 8 (`tccp` Processes [34]) *The syntax of* `tccp` *is obtained by adding the construct* $\mathbf{when}\ c\ \mathbf{then}\ P\ \mathbf{else}\ Q$ *to the language in Definition 4.*

The `tccp` calculus introduces a discrete global clock and assumes that ask and $tell$ actions take one time-unit. Computation evolves in steps of one time-unit, so called clock-cycles, that are syntactically separated by action prefixing. Moreover, maximal parallelism is assumed, that is, at each moment every enabled agent of the system is activated:

$$\mathrm{R_P}\ \frac{\langle P_1, c\rangle \longrightarrow \langle P_1', c_1\rangle \quad \langle P_2, c\rangle \longrightarrow \langle P_2', c_2\rangle}{\langle P_1 \parallel P_2, c\rangle \longrightarrow \langle P_1' \parallel P_2', c_1 \wedge c_2\rangle}$$

Let $P = \mathbf{when}\ c\ \mathbf{then}\ Q\ \mathbf{else}\ R$. This process queries if c can be deduced from the current store. If so, Q is executed. Otherwise, unlike the **unless** process in `tcc`,

P reduces to R *instantaneously*:

$$\mathrm{R}_{\mathrm{NE}}\ \frac{\langle Q, d\rangle \longrightarrow \langle Q', d'\rangle \quad d \vdash c}{\langle \mathbf{when}\ c\ \mathbf{then}\ Q\ \mathbf{else}\ R, d\rangle \longrightarrow \langle Q', d'\rangle}$$

$$\mathrm{R}'_{\mathrm{NE}}\ \frac{\langle R, d\rangle \longrightarrow \langle R', d'\rangle \quad d \not\vdash c}{\langle \mathbf{when}\ c\ \mathbf{then}\ Q\ \mathbf{else}\ R, d\rangle \longrightarrow \langle R', d'\rangle}$$

Analogous rules are given when Q and R cannot evolve on the store d.

Example 5 (Time-outs) *Assume now that the controller in Example 4 must emit the signal* `stop` *when the environment does not provide the signal* `turn` *after* n *time-units. This can be done by defining inductively* $\sum\limits_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i\ \mathbf{time\text{-}out(m)}\ Q$ *that behaves as* Q *if after* m *time-units none of the guards* c_i *can be entailed. Let* $P = \sum\limits_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$*. When* $m = 0$*, the time-out constructor is defined as*

$$\mathbf{when}\ c_1\ \mathbf{then}\ P\ \mathbf{else}\ \mathbf{when}\ c_2\ \mathbf{then}\ P\ \mathbf{else}\ ...\mathbf{when}\ c_n\ \mathbf{then}\ P\ \mathbf{else}\ Q$$

As for $m > 0$*, we have:*

$$\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i\ \mathbf{time\text{-}out(m)}\ (\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i\ \mathbf{time\text{-}out(m-1)}\ Q)$$

In our particular case, I *is a singleton,* $c_i =$ `turn`*,* $m = n$ *and* $Q = \mathbf{tell}(\mathtt{stop})$*.*

`tccp` programs usually require to perform arithmetic calculations in order to output a signal to the environment. Even though the underlying constraint system may support some basic arithmetic, it is not realistic to assume that it implements any computable function. One may think of implementing such functions as `tccp` processes. Nevertheless, these computations would consume an unspecified amount of time-units, making the synchronization of processes more difficult. In [4], `tccp` is extended to consider *external* functions written in a functional language. Processes can evaluate *instantaneously* a function by means of an external implementation, thus avoiding the burden of computing it as a `tccp` process.

3.1.4 Continuous Time in CCP

The Hybrid concurrent constraint programming model, *Hybrid cc* (`hcc`) [101], is a calculus that can express discrete and continuous evolution of time. More precisely, there are points at which discontinuous change may occur (i.e. the execution proceeds as a burst of activity) and open intervals in which the state of the system changes continuously (i.e. the system evolves continuously and autonomously).

In `hcc` the notion of constraint system is extended with differential equations and the entailment relation is defined in order to solve initial value problems. Hence, *continuous constraint systems* allows for describing the continuous evolution of time. For instance, the constraint `init`$(x = 0)$ means that the initial value of x is 0 and `cont`$(dot(x) = 1)$ that the first derivative of x is 1. Then, it is possible to infer that $x = t$ in time t.

Definition 9 (hcc Processes [101]) *The* hcc *processes result from adding to the syntax in Definition 6 the construct* $\mathbf{hence}\ P$.

The constructor $\mathbf{hence}\ P$ specifies a process where P holds continuously beyond the current instant. Therefore, if $\mathbf{hence}\ P$ is invoked at time t, a copy of P is created at each time in the time interval (t, ∞). In combination with the other constructs in **Default** tcc, various patterns of temporal activity can be generated.

Example 6 (Controlling the speed of the robot) *Assume that the variable v controls the speed of the robot in Example 4. The following process reduces the speed of the robot when it receives the signal* stop*:*

$$\mathtt{speed\text{-}control} \stackrel{\text{def}}{=} \mathbf{when}\ \mathtt{stop}\ \mathbf{do\ hence\ when}\ v > 0\ \mathbf{do\ tell}(\mathtt{cont}(dot(v) = -1))$$

The reader can find in [99] a survey of the developments of the temporal extensions of tcc, **Default** tcc and hcc and the relationships between their semantic models. We also point the reader to [189] where a CCP calculus based on the notion of real time is proposed. The rtcc calculus allows to deal with strong preemption and delay declarations in the lines of ntcc. Furthermore, the transition system is extended to specify true concurrency and the resources that processes require to be executed.

3.2 Probabilistic Behavior

Non-deterministic choices in a model leave completely unspecified the way the system may react to a given input. In some cases, models can be refined when information about the propensity of an action to occur is available. For instance, we could have information about the probability of a system component to fail. In [100] a probabilistic model for CCP and tcc is studied where random variables with a given probability distribution are introduced. The resulting languages, *probabilistic* CCP and *probabilistic* tcc (pcc, ptcc), allow programs to make stochastic moves during execution, so that they may be seen as stochastic processes.

Definition 10 (Probabilistic CCP and tcc [100]) *The syntax of* pcc *(resp.* ptcc*) are obtained by adding to Definition 3 (resp. 5) the constructor* $\mathbf{new}\ (x, f)\ \mathbf{in}\ P$ *where x is a variable and f is its probability mass function.*

The constructor introduced above chooses a value for x according to f:

$$\mathrm{R}_{\mathrm{PROB}}\ \frac{f(r) > 0 \qquad y \text{ is a fresh variable}}{\langle \mathbf{new}\ (x, f)\ \mathbf{in}\ P, d\rangle \longrightarrow \langle P[y/x], d \wedge y = r\rangle}$$

Example 7 (Random Zigzag) *Assume that the robot in Example 4 takes autonomously the decision of turning right or left:*

$$\begin{aligned} \mathtt{choice} \stackrel{\text{def}}{=}\ & \mathbf{new}\ (x, f : f(0) = f(1) = 0.25, f(2) = 0.5)\ \mathbf{in} \\ & \quad \mathbf{when}\ x = 0\ \mathbf{do}\ \mathtt{TurnR} \parallel \mathbf{when}\ x = 1\ \mathbf{do}\ \mathtt{TurnL} \end{aligned}$$

The agent above with a probability of 0.25 calls the procedure TurnR*. Similarly for* TurnL*. Moreover, with a probability of 0.5, the directions does not change.*

An interesting feature of `pcc` is the use of constraints to add information about random variables. This can be used to declaratively modify the probabilities of the possible execution paths as the following example shows.

Example 8 (Random Zigzag Revisited) *Consider the process*

$$\begin{array}{rcl} \mathtt{choice} & \stackrel{\mathtt{def}}{=} & \mathbf{new}\ (x, f : f(0) = f(1) = 0.25, f(2) = 0.5)\ \mathbf{in}\ \mathbf{tell}(x \in \{0,1\}) \parallel \\ & & \quad \mathbf{when}\ x = 0\ \mathbf{do}\ \mathtt{TurnR} \parallel \mathbf{when}\ x = 1\ \mathbf{do}\ \mathtt{TurnL} \end{array}$$

Here we restrict the variable x to take a value in the set $\{0,1\}$. Then, the choice $x = 2$ is precluded and, normalizing f, we observe the calls `TurnL` and `TurnR` with the same probability of 0.5*.*

The results of [100] are extended in [98] where `pcc` processes with recursion are considered. The authors show that due to the combination of probabilities and constraints (as in Example 8), the interpretations of procedure calls are not necessarily monotonic and then, the semantics cannot be compositional. The authors extend the operational semantics with labels indicating the needed constraint to perform a transition and a denotational semantics based on weak bisimulation on such transition system is shown to be a congruence. The works in [161, 44, 9] study also stochastic extensions for CCP. Although the calculi developed in those works differ in the constructs provided and the semantic treatment, all of them aim at representing the stochastic nature of the modeled system. The semantics of probabilistic extensions of CCP has been also studied in [164, 165]. Furthermore, a framework for the analysis of probabilistic programs is developed in [166].

3.3 Linear Concurrent Constraint Programming

Linear CCP (`lcc`) [179, 77, 22, 23, 105] is a variant of CCP where constraints are built from a *linear constraint system* based on Girard's intuitionistic linear logic (ILL) [97].

Definition 11 (Linear Constraint Systems [77]) *A linear cs is a pair $(\mathcal{C}, \vdash)$ where $\mathcal{C}$ is a set of formulas (linear constraints) built from a signature Σ, a denumerable set of variables $\mathcal{V}$ and the following ILL operators:* multiplicative *conjunction ($\otimes$) and its neutral element (1), the existential quantifier ($\exists$) and the exponential bang ($!$). Let Δ be a (possibly empty) subset of $\mathcal{C} \times \mathcal{C}$ defining the non-logical axioms of the constraint system (i.e, a theory). Then the entailment relation $\vdash$ is the least set containing Δ and closed by the rules of ILL.*

The bang connective allows to recover the classical constraint system by writing all constraints preceded by "$!$".

The language of `lcc` processes is similar to that of CCP but variables in ask agents can be universally quantified.

Definition 12 (Linear CCP [77]) *Processes in `lcc` are built from constraints in a linear constraint system and the syntax in Definition 4 where ask agents are of the form $\forall \overline{x}(\mathbf{when}\ c_i\ \mathbf{do}\ P_i)$.*

The linear tell operator $\mathbf{tell}(c)$ augments the current store d to $d \otimes c$. Furthermore, the linear ask $\mathbf{when}\ c\ \mathbf{do}\ P$ evolves into P whenever there exists e s.t. d entails $c \otimes e$. When this happens, the constraint c is *consumed*:

$$\mathrm{R_{LA}}\ \frac{d \vdash c[t/x] \otimes d'}{\langle \forall x(\mathbf{when}\ c\ \mathbf{do}\ P), d\rangle \longrightarrow \langle P[t/x], d'\rangle}$$

Due to the removal of information, `lcc` is intrinsically non-deterministic and the constraint d' above has to be carefully chosen to avoid the unwanted weakening of the store as in $!c \vdash c$ [106, 137, 105]. Nevertheless, since `lcc` is not monotonic (in the sense that the information in the store can be dropped), this model introduces some forms of imperative programming particularly useful for reactive systems. For instance, imperative data structures are encoded directly with linear constraints instead of streams.

Example 9 (Access Permissions) *Assume a constraint system with the constant symbols* `unq` *(unique) and* `shr` *(share) and with the ternary predicate* $\mathtt{acc}(x, r, p)$ *(agent* x *is currently accessing the resource* r *with permission* $p \in \{\mathtt{unq}, \mathtt{shr}\}$*). Assume also that the constraint system is equipped with the axiom* $\Delta = \mathtt{acc}(x, r, \mathtt{unq}) \vdash \mathtt{acc}(x, r, \mathtt{shr})$*. In words, if* x *makes use of* r *with permission* `unq`*, it can* downgrade *this permission to* `shr`*. Let* $R = P \parallel Q$ *where*

$$P \;=\; \mathbf{tell}(\mathtt{acc}(x, r, \mathtt{unq})) \qquad\qquad Q \;=\; \forall y(\mathbf{when}\ \mathtt{acc}(x, y, \mathtt{shr})\ \mathbf{do}\ Q')$$

Roughly speaking, P *adds to the store the information required to state that* x *uses* r *and has a unique permission on it. Thereafter,* Q consumes *this information (by using* Δ*), leading to the store* $\top$ *(i.e.,* x *has no longer access to* r*) where the agent* $Q'[r/y]$ *is executed.*

3.4 Soft Concurrent Constraint Programming

Soft constraints [25, 30] have been introduced in constraint programming in order to deal with different levels of consistency. The framework of semiring-based constraints [25] gives a general setting where, according to an algebraic structure, it is possible to represent preferences, fuzziness, probabilities and uncertainty in constraint satisfaction problems (CSP).

The authors in [29] propose Soft Concurrent Constraint Programming (`scc`). The key idea is to replace the (crisp) constraint system in Definition 1 by a semiring-based constraint system. Let us first recall the notion of c-semirings.

Definition 13 (C-Semiring [25]) *A semiring is a tuple* $\langle \mathcal{A}, +, \times, 0, 1\rangle$ *where (1)* $\mathcal{A}$ *is a set and* $0, 1 \in \mathcal{A}$*. (2)* $+$ *is commutative, associative and* 0 *is its unit element. (3)* $\times$ *is associative, distributes over* $+$*,* 1 *is its unit element and* 0 *is its absorbing element. A c-semiring is a semiring such that* $+$ *is idempotent,* 1 *is its absorbing element and* $\times$ *is commutative.*

Intuitively, $+$ is the lub operator and it is used to choose the *best* constraint: $a \leq b$ iff $a + b = b$. The $\times$ operator allows for combining constraints. Here it is important to notice that combining more constraints leads to a *worse* level of consistency, that is,

$c \times c' \leq c$ unlike in Definition 1 where $c \leq c \sqcup c'$ (i.e., $c \sqcup c' \vdash c$). As an example, fuzzy constraint satisfaction problems [69] can be modeled and solved by using the c-semirings $\langle [0,1], max, min, 0, 1\rangle$.

By defining suitable operations on the semiring, it is possible to define a cylindric algebra leading to a *soft constraint system* as shown in [29]. Then, CCP agents can tell and ask soft constraints. Furthermore, such agents can define thresholds that express the level of consistency of the store. This allows the programmer to specify that an action (tell or ask) is executed only if it does not decrease the consistency level to a given lower bound.

3.4.1 Timed Soft Concurrent Constraint Programming

Following the approach of `tccp`, a timed extension of `scc` (`stcc`) was proposed in [28]. As in `tccp`, action-prefixing in `stcc` is interpreted as the next-time operator and the parallel execution of agents follows the scheduling policy of maximal parallelism.

Definition 14 (`stcc` Processes [28]) *Agents in* `stcc` *are built from:*

$$\begin{array}{lll} P, Q & := & \mathbf{skip} \mid \mathbf{tell}(c) \rightsquigarrow P \mid \sum\limits_{i \in I} \mathbf{when}\, c_i \rightsquigarrow P_i \mid P \parallel Q \mid (\mathbf{local}\, \overline{x}; c)\, P \mid q(\overline{x}) \\ & & \mathbf{when}_{\Phi}\, c\, \mathbf{then}\, P\, \mathbf{else}\, Q \mid \mathbf{when}^{a}\, c\, \mathbf{then}\, P\, \mathbf{else}\, Q \end{array}$$

Here, $\rightsquigarrow$ *can be either* $\rightarrow_{\Phi}$ *or* $\rightarrow^{a}$ *where* a *is a semiring element and* Φ *a constraint.*

Intuitively, the semiring value a and the constraint Φ are used as a cut level to prune branches of computation that are not satisfactory. For instance, the process $\mathbf{tell}(c) \rightarrow^{a} P$ adds c to the current store d, if the conjunction (based on the $\times$ operator of the semiring) of d and c is *better* (with respect to $+$) than a.

Example 10 (Bounded Zigzag) *Let us choose the Fuzzy c-semiring and assume now that the robot in Example 7 zigzags according to* preferences*:*

$$\begin{array}{lll} \mathit{choice} & \stackrel{\text{def}}{=} & \mathbf{when}\ \mathtt{true} \rightarrow^{0.1} \mathit{TurnR} + \mathbf{when}\ \mathtt{true} \rightarrow^{0.3} \mathit{TurnL} + \\ & & \mathbf{when}\ \mathtt{true} \rightarrow \mathbf{skip} \end{array}$$

The process above can always do nothing ($\mathbf{skip}$*) and then, the direction of the robot remains the same. Moreover, assume that each time it turns left or right, we add a constraints and a* penalization *(in terms of the semiring value) is paid. Then, the number of times the robot chooses to turn is confined according to the thresholds* 0.1 *and* 0.3.

3.5 Mobile Behavior

Process calculi such as the π-calculus [140] allow to specify *mobile* systems, i.e., systems where agents can communicate their local names. Unlike the π-calculus (that is based on point-to-point communication), interaction in CCP is *asynchronous* as communication takes place thorough the shared store. In the CCP model it is possible to specify *mobility* in the sense of *reconfiguration* of the communication structure of the program. This can be done by using logical variables that represent communication channels and unification to bind messages to channels [181]. Since logical variables can be bound to a value only once, if two messages are sent through the same channel, then they must be equal to avoid an inconsistent store. This problem was addressed in [127] by considering *atomic* tells where the constraint c in $\mathbf{tell}(c)$ is added to the store d if the conjunction $c \wedge d$ is consistent. Channels are then represented as imperative style variables by binding them to streams. Therefore, a protocol is required since messages must compete for a position in such a stream.

The following two sections describe two alternative approaches to endow CCP with mechanisms to communicate private channels or links.

3.5.1 The `cc-pi` Calculus

The `cc-pi` calculus [50] results from the combination of the CCP model with a name-passing calculi. More precisely, `cc-pi` extends CCP by adding synchronous communication and by providing a treatment of names in terms of restriction and structural axioms closer to nominal calculi than to variables with existential quantification.

Definition 15 (`cc-pi` Processes [50]) *Processes in* `cc-pi` *are built from c-semiring based constraints as follows:*

$$\begin{array}{lll} P, Q & := & \mathbf{skip} \mid P \parallel Q \mid \sum\limits_{i \in I} \pi_i.P_i \mid (\mathbf{local}\, x; c)\, P \mid p(x) \\ \pi & := & \tau \mid \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{y}) \mid \mathbf{tell}(c) \mid \mathbf{when}\ c \mid \mathbf{retract}\ c \mid \mathbf{check}\ c \end{array}$$

In this calculus, tell and ask actions are prefixing much like in `stcc`. The name passing discipline of `cc-pi` is reminiscent to that in the pi-F calculus [211] whose synchronization mechanism is global and, instead of binding formal names to actual names, it yields explicit fusions, i.e., simple constraints expressing name equalities.

Example 11 (Name passing) *Assume two components P and Q of a system such that P creates a local variable that must be shared with Q. This system can be modeled as:*

$$P = (\mathbf{local}\, y)\, (\bar{x}\langle y\rangle.P') \qquad\qquad Q = x\langle z\rangle.Q'$$

In $P \parallel Q$, P sends *the* private *name y on channel x and synchronizes with Q leading to* $(\mathbf{local}\, x)\, (P' \parallel Q' \parallel \mathbf{tell}(y = z))$.

Similar to the non-monotonic extension of `scc` reported in [31], `cc-pi` also introduces retraction of constraints ($\mathbf{retract}\ c$) whose effect is to erase a previously told constraints. Furthermore, it is possible to check if a given constraint c is consistent with the current store though the prefix $\mathbf{check}\ c$.

Another line of development in this direction was the π^+-calculus [66]. This language is an extension of the π-calculus [140] with constraint agents that can perform tell and ask actions. Similarly as in `cc-pi`, mobility of π^+comes from the operands inherited from the π-calculus.

3.5.2 Universal Timed CCP

Universal Timed CCP (`utcc`) was proposed in [158] as an orthogonal extension of `tcc` for the specification of mobile reactive systems as security protocols. Basically,

$\texttt{utcc}$ replaces the ask operation $\textbf{when}\ c\ \textbf{do}\ P$ by a *parametric ask* constructor of the form $(\textbf{abs}\ \overline{x}; c)\ P$. This process can be viewed as an *abstraction* of the process P on the variables $\overline{x}$ under the constraint (or with the *guard*) c.

Definition 16 ($\texttt{utcc}$ Processes [158]) *The $\texttt{utcc}$ processes result from replacing in the syntax in Definition 5 the expression* $\textbf{when}\ c\ \textbf{do}\ P$ *with* $(\textbf{abs}\ \overline{x}; c)\ P$ *where variables in $\overline{x}$ are pairwise distinct.*

Operationally, $(\textbf{abs}\ \overline{x}; c)\ P$ executes $P[\overline{t}/\overline{x}]$ in the current time interval for *all the terms* $\overline{t}$ s.t $c[\overline{t}/\overline{x}]$ is entailed by the current store. This construct is akin to replicated asks in $\texttt{lcc}$ that can replace process declarations [106, 105].

Example 12 (Mobile behavior in $\texttt{utcc}$) *Assume an uninterpreted binary predicate* $\texttt{out}$. *The system in Example 11 can be specified in $\texttt{utcc}$ as:*

$$P = (\textbf{local}\, y)\ (\textbf{tell}(\texttt{out}(x, y)) \parallel P') \qquad\qquad Q = (\textbf{abs}\ z; \texttt{out}(x, z))\, Q'$$

The SOS of $\texttt{utcc}$ dictates that the process above evolves into $(\textbf{local}\, x)\ (P' \parallel Q'[y/z])$ *where P' and Q' share the local variable y created by P. Then, any information produced by P' on y can be seen by Q' and viceversa.*

The reader may also refer [172] where CCP is endowed with an asynchronous message-based communication mechanism to model distributed systems. Agents can then communicate messages by using *send* and *receive* primitives. The work in [96] defines a model of *process mobility* for CCP. In this context, localities (or sites) are defined and agents are allowed to have their own local store. Sites are organized in a hierarchical way and then, it is possible for an agent to have sub-agents. A primitive *migrate* is added to the calculus in order to allow processes to move to another location and carry their local store. A distributed and probabilistic extension of CCP where a network of computational nodes, each of them with their own local store, is studied in [45]. Nodes can send and receive through communication channels constraints, agents (processes) and channels themselves. We also point the reader to [132] where the authors study an encoding of the $\texttt{utcc}$ calculus into $\texttt{tccp}$. Such encoding has to deal with the $\textbf{abs}$ operator in $\texttt{utcc}$ and also with the fact that the notion of time differs from the source and target calculi.

3.6 Epistemic and Spatial Modalities in CCP

In some situations, the centralized notion of store makes CCP unsuitable for systems where information and processes can be shared or spatially distributed among certain groups of agents. In particular, agents posting and querying information in the presence of spatial hierarchies for sharing information and knowledge. For instance, friend circles and shared albums in social networks or shared folders in cloud storage provide natural examples of managing information access. These domains raise important problems such as the design of models to predict and prevent privacy breaches, which are commonplace nowadays.

In [124] the authors enhance and generalize the theory of CCP for systems with spatial distribution of information. More precisely, the underlying theory of constraint systems is extended by adding space functions to their structure. Take for instance the constraint $d = \mathfrak{s}_i(c) \wedge \mathfrak{s}_j(c')$. Intuitively, d asserts that the local store of the agent i (resp. j) is c (resp. c'). Functions $\mathfrak{s}_i$, $\mathfrak{s}_j$,... can be seen as topological and closure operators and they allow for specifying spatial and epistemic information.

Definition 17 (Spatial-Epistemic CCP [124]) *Processes in Spatial-Epistemic CCP are obtained by adding the operator $[P]_i$ in the Syntax of Definition 3.*

The spatial operator can specify a process, or a local store of information, that resides within the space of the agent i (e.g., an application in some user account, or some private data shared with a specific group). This operator can also be interpreted as an epistemic construction to specify that the information computed by a process will be known to a given agent. It is worth noticing that the CCP concept of local variables cannot faithfully model local spaces since in the spatial constraint systems defined in [124], it is possible to have inconsistent local stores without propagating their inconsistencies towards the global store.

4 Programming Languages Based on the CCP Model

Several CCP programming languages have been designed. These cover a wide spectrum going from syntactic sugar over a particular CCP calculus, to graphical representations of the calculus primitives and to full fledged general purpose multiparadigm languages. Early CCP languages took inspiration in constraint logic programming (CLP), where unification was replaced by constraint solving. An example is the language $cc(FD)$ [111] that implements an efficient *finite domains* constraint system. The CLP model was implemented in various languages, each with a suitable constraint system (finite domains, booleans, real numbers). Nevertheless, these languages lacked flexibility since problem solving had to be tailored to the specific fix set of predefined constraints. In contrast, $cc(FD)$ offered general-purpose combinators, applicable to any constraint system, such as constructive disjunction and blocking implication [111]. These, together with entailment, allow the language to be tailored to specific user domains without losing the "naturalness" of specifications.

Programs in $cc(FD)$ are written in a Prolog-like syntax and user constraints are translated into canonical forms called *indexicals* [56] that can be implemented very efficiently.

Even though solving constraint problems remains an important goal of CCP languages, they have mostly evolved into powerful ways to define complex synchronization schemes in concurrent and distributed settings. In the rest of this section we describe some of the systems and programming languages built on the ideas of the CCP model. We also point to some implementation of frameworks and interpreters for the calculi described in Section 3. The reader may also refer to [88], a survey of the developments of (concurrent) languages that integrate constraint reasoning and solving.

4.1 Janus

Lucy [185] is a simple language were agents communicate by posting constraints over a mailbox called a "bag". Processes can merge and pass around bags in a kind of distributed version of the CCP model. Janus is an extension of Lucy intended for distributed programming and it resembles a concurrent logic programming language. A program is a network of agents that communicate by passing messages over a channel. Agents consist of *rules* of the form $p(t_1, \ldots, t_n) \leftarrow C \mid C_1, B$ where $t_1, \ldots, t_n$ are terms, C are ask and C_1 $tell$ constraints, and B is a conjunction of *goals*. When a message matches the pattern on the left, it triggers the behavior determined by the right hand side of the rule. In accordance with the CCP model, variables are logical and they have two aspects or *faces*, corresponding to *ask* and *tell* annotations. Any of these can also be passed around. Careful design of restrictions on *askers* and *tellers* ensure the *failure-free property*: Janus computations never abort due to the store becoming inconsistent.

A graphical representation of Janus, *Pictorial Janus* was proposed in [122]. The basic elements of a Pictorial Janus program are graphical primitives, i.e., closed contours, connections and links between objects. Rules like the one above can be specified inside agents. A visual debugging environment for Pictorial Janus, providing real-time animation of programs can be found in [72].

4.2 JCC

The language `jcc` [184] was designed as an integration of **Default** `tcc` into Java. `jcc` is intended for embedded reactive systems and for simulation and modeling in robotics and system biology. It implements bounded-time execution of the `tcc` calculus constructs. In `jcc` users can define their own constraint system and thus tune the language to particular domains. The main purpose of the language is to provide a model of loosely-coupled concurrent programming in Java. The model introduces the notion of a vat. A vat may be thought of as encapsulating a single synchronous, reactive `tcc` computation. A computation consists of a dynamically changing collection of interacting $vats$, communicating with each other through shared, mutable objects called $ports$. *Asker* and *teller* objects read from and write into the port. Constructs from the underlying `tcc` model allows an object to specify code that should be executed in the future.

4.3 LMNtal

The goal of LMNtal [205] is to provide a scalable and uniform view of concurrent programming concepts such as processes, messages, synchronous and asynchronous computation. It inherits ideas from the concurrent constraint language of Guarded Horn Clauses (GHC) [93] and from Janus. Basic components of the language are *links*, *multisets*, *nodes* and *transformations*. Links represent both communication channels between logically neighboring processes and logical neighborhood relations between data cells. Links are bi-directional.

Communication is based on constraints over logical variables. Processes sharing variables are thought of as been "connected", as in the CCP model. Multisets of nested nodes and links are a first-class notion in LMNtal. These organize into a hierarchy (called a *membrane* structure) and thus provide a kind of *ambient*, as in the Ambient calculus [55]. Transformations are rules, much like in Janus. LMNtal provides both channel mobility and process mobility: it allows dynamic reconfiguration of process structures as well as the migration of nested computations. An expression $p(x_1, ..., x_m)$ defines an atomic process. Variables x_i are its links. LMNtal makes no distinction between processes and data. Atom $x = y$ denotes a $connector$ between one side of the link x and one side of the link y. $\{P\}$ denotes a process enclosed within the membrane $\{\}$; and $T : -T$ a rewrite rule for processes. Links in the left part of the rule are $consumed$ and on the right hand are $produced$. Complex patterns can thus be defined for rule triggering and concepts such as mobility can be easily expressed. An implementation of LMNtal is available at `http://www.ueda.info.waseda.ac.jp/lmntal/`.

4.4 Constraint Handling Rules

Constraint Handling Rules (CHR) [89, 90] was originally conceived as a language for extending CLP systems with elegant mechanisms to define new constraint solvers, thus fulfilling the aim of CLP languages to be truly parametric in (user-built) constraint systems. CHR found afterwards widespread use as a general purpose multi-paradigm programming language. CHR provides users with declarative (multi-headed) rules for implementing simplification, propagation and so-called *simpagation* of constraints. Rules can be guarded with conditions that must hold for them to be applied. The rules act on a (multiset) constraint store. Simplification rules replace constraints in the store by simpler ones, propagation rules add redundant constraints (useful for additional simplifications) and simpagation rules do both. CHR is embedded in some host language $\mathcal{H}$, written CHR($\mathcal{H}$). The host language provides data types and some primitive constraints.

The declarative (classical logic) semantics of CHR [199] is given in terms of the constraint theory of the host language together with the logical formulas for each rule. A simplification rule $H \iff G \mid B$, for example, corresponds to the formula $\tilde{\forall}(G \rightarrow (H \leftrightarrow \exists \bar{x}.B))$, where $\tilde{\forall}$ quantifies over all free variables. This semantics, however, does not comply with the intended meaning of some CHR programs. The problem comes from the multiset store and from the unidirectionality and committed-choice nature of CHR rules. A better declarative semantics based on intuitionistic linear logic [97] was later proposed in [24].

CHR can encode a basic CCP language, as shown in [91]. The committed-choice feature of CHR is used to represent each $\mathbf{when}\ c_i\ \mathbf{do}\ P_i$ process in a non-deterministic choice $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ as a simplification rule of the form $summation(\sum(...)) \iff c_i \mid P_i$. The committed choice ensures that when this rule is chosen (c_i holds) none of the other rules for *ask* constructs in the summation can be used.

As mentioned, a key feature of CHR is the possibility of its embedding in many different types of programming languages (logic, functional, explicit-state), such as CHR(Prolog), CHR(Haskell), CHR(C). Various applications, ranging from Multi-Agent systems to language processing (CHR grammars) or software testing have been developed in CHR. A summary of these can be found in [199].

The simplicity and expressivity of CHR have attracted the attention of several researchers. The meaning of CHR has evolved from *theoretical* semantics [90], which are highly non-deterministic, to more refined notions of transition systems that eliminate some of the non-determinism [70]. The latter is the basis for implementing systems based on CHR. Similar to the developments of probabilistic extensions of CCP, CHR has been also extended to replace non-deterministic choices (in the rule to be applied) with probabilistic choices [92]. Probabilistic CHR (PCHR) allows for an explicit control of the rule to be applied and fairness can be directly expressed by choosing an appropriate probability distribution on the rules. Other extensions dealing with distributed constraint stores [188] and user-definable rule priorities [125] have been also proposed. In [137] the connection between `lcc` and CHR is studied. The authors show that a semantic preserving encoding in both directions is possible. Moreover, properties as confluence [71], termination [168] as well as general abstract interpretation techniques [192] for the analysis and optimization of CHR programs have been established. The reader may refer [199] and [91] for a more complete account on these topics.

4.5 Oz

Arguably the CCP-based language that has found more widespread use is Oz [196, 197]. The Oz model takes inspiration from CCP and CLP and from its predecessor AKL [87]. It builds up from a kernel language consisting of a first-order structure defining the values and constraints Oz computes with, a CCP calculus (called the Oz calculus) over this structure and the *actor model*, a (non-formal) computation model introducing high level concurrent notions such as computation spaces (for speculative computation) and threads. In this model the usual store of constraints coexists with a so-called *predicate store* that includes a non-monotonic mutable store. The last one is used to model shared state and message passing concurrent computation via the notions of *cells* and *ports*.

Although the Oz kernel is small and conceptually simple, its rich semantics allows for the implementation of several computation models: declarative, stateful, lazy declarative, lazy stateful, eager, in both sequential and concurrent settings. The Oz kernel is based on a calculus comprising the ρ calculus [151] plus some additions for modeling functional, object-oriented, constraint-based and logic programming. The ρ calculus is a relational calculus parametrized by a logical constraint system. Its basic constructs are constraints, parallel composition, local declarations, conditionals and abstractions. Constraints are taken from the underlying constraint system, conditionals test for entailment of constraints from the store and abstractions are used to encode procedures.

As mentioned, traditional programming styles such as imperative, functional or object-oriented can homogeneously coexist within the Oz language. Furthermore, constraint programming, message passing concurrency and an asynchronous distribution model are also supported. A complete presentation and analysis of all Oz computation models, together with programming strategies for each, can be found in [173].

By default, Oz relies on a constraint system over (infinite) feature trees for value assignment to logical variables. Extensions to finite domains, finite sets and real intervals are also provided. Oz has been successfully used in many different problem domains. A strong point of the language is the coherent combination of the declarative pure CCP model with the traditional shared state scheme. A cell variable is assigned to a unique name in the monotonic (i.e. CCP) store and this name is associated with another variable in the mutable store. The latter represents the value of the cell. Changing the value of a cell amounts to changing the association of its name to a different variable. This variable, in turn, may have a different value from the previous one in the monotonic store. That is, cell values do not really change. What changes is the variable associated to its name.

4.6 CORDIAL

CORDIAL [174] is a visual language based on the π^+-calculus [66] and provides transparent integration of constraints and objects. Objects within methods are represented by closed contours. Object methods launch CCP processes that, in addition to the usual ask and $tell$ operations, can send messages to other objects. Messages are objects connected by links to object mailboxes. In CORDIAL objects are not located at some reference but "float" over a constraints medium and they are identified by an associated constraint parametrized on the local variable $self$. Senders willing to invoke some object method post a constraint involving some variable, say x, and then send the message to x. Any object such that its associated constraint can be entailed by the store conjoined with the constraint $self = x$, is eligible to accept the message. Some eligible object is then non-deterministically chosen to handle the message. This scheme allows for very complex patterns of communication and mobility.

4.7 Interpreters for CCP Calculi

Similar to `jcc`, some other interpreters for the calculi described in Section 3 have been implemented. In the context of `lcc`, SiLCC (`http://contraintes.rocq.inria.fr/~tmartine/silcc/`) is an implementation of the language with a module system as described in [106]. Furthermore, ALCOVE *Aeminium Linear COnstraints VErifier* (`http://avispa.puj.edu.co`) is a `lcc`-based tool for the analysis of access permissions in concurrent-by-default programs written in Aeminium [201].

The `ntcc` calculus has been implemented as `ntccSim` (`http://avispa.puj.edu.co`), an interpreter written in the Oz language. An important feature of this tool is that several constraint systems can be included in the same model. For instance, constraints over finite domains (FD) and real intervals (XRI) have been used to implement computational models of biological system (see Section 5.3). The `ntccrt` interpreter for the `ntcc` calculus (`http://sourceforge.net/projects/ntccrt/`) is written in C++ and specifications can be made in Common Lisp or in *OpenMusic* (`http://repmus.ircam.fr/openmusic/`), and then translated to C++. This framework can be also integrated as a plugin for either *Pure Data (*PD*) or Max/MSP* [169] to take advantage of the facilities offered by those languages to implement, for example, sound processors. `ntccrt` makes use of the state of the art propagation techniques in Gecode (`http://www.gecode.org/`) to implement the underlying constraint system. Hence, the system is able to deal with real-time requirements for the execution of `ntcc` models in the context of computer-based musical improvisation (see Section 5.2). In [130] an interpreter for `tccp` written in Maude is described (`http://users.dsic.upv.es/~villanue/tccp-func/`). As for `hcc`, at `http://www-cs-students.stanford.edu/~vgupta/hcc/` the reader may find an interpreter for this language. Finally, k-stores [16] is an interpreter for the epistemic and spatial calculus in Definition 17.

In the context of CHR, implementations of Prolog such as SWI-Prolog (`http://www.swi-prolog.org`) and SICStus Prolog (`http://sicstus.sics.se`) feature modules for CHR. WebCHR is a web tool that allows the execution of Prolog and CHR programs (`http://chr.informatik.uni-ulm.de/~webchr/`). Implementations of CHR for different programming languages such as Java, C and Haskell can be found at `http://chr.informatik.uni-ulm.de/`. The system CHRat [76] implements a modular version of CHR that allows for reusing built-in constraints, defined in a constraint system, as a constraint solver in another CHR program (`http://contraintes.inria.fr/~tmartine/chrat/`). CHRiSM [198] (`http://people.cs.kuleuven.be/~jon.sneyers/chrism/`) integrates CHR and PRISM (PRogramming In Statistical Modeling) [190], a probabilistic extension of Prolog for symbolic-statistical modeling.

5 Emergent Applications for CCP

Nowadays concurrent systems are ubiquitous in several domains and applications. They pervade different areas in science (e.g. biological and chemical systems), engineering (e.g., security protocols, mobile and service oriented computing and social networks) and even the arts (e.g. tools for multimedia interaction). CCP based languages and calculi have been extensively used to model, analyze and verify concurrent systems in different scenarios such as the aforementioned. The simplicity and the expressivity of this model attracts the attention of modelers mainly due to: (1) the parameterization of CCP in a constraint system provides a very flexible way to tailor data structures to specific domains and applications; (2) The declarative synchronization mechanism based on entailment of constraints eases the modeling of complex interactions between subsystems; (3) The ability to deal with partial information allows for modeling and studying such systems even when one is not fully aware of the behavior of all the components involved; (4) As we shall show in Section 6, CCP enjoys several reasoning techniques; finally, (5) the underlying model of CCP based on a common store of partial information is akin to several systems where components post information asynchronously.

This section is devoted to show some of the most relevant applications of CCP. We shall show that the reactive model of temporal CCP allows for the declarative specification of reactive systems such as electromechanical devices, software control and multimedia interaction systems. The temporal and probabilistic extensions of CCP have found application in system biology and physical systems. Finally, the declarative nature of CCP and its reasoning techniques have been used to specify and verify, for instance, security protocols and service oriented computing systems.

5.1 Physical Systems

The work in [212] shows the applicability of `tcc` as programming language to specify controllers for electromechanical systems. In this setting, `tcc` provides a declarative model for the components that comprise the device. The authors show that the timing constructs in `tcc` can neatly express the pattern of interaction over time between the controller and the environment. Furthermore, since `tcc` programs can be compiled into finite-state machines [182], the implementation of the system is straightforward and efficient. The strong connection of CCP calculi and logic is also an advantage in this context since it is possible to use standard techniques for proving properties over the software constructed.

A natural application of the `hcc` calculus is the modeling of physical systems. In this scenario, one is interested in observing the change of the state of the system when interacting with the environment (discrete change) and also when evolving autonomously (continuous change). In [103], a compositional model of a photocopier paper path in `hcc` is presented. The declarative nature of `hcc` is particularly useful in this setting, since for each fragment of the model, it is only necessary to state the laws of physics applicable, e.g. equilibrium laws, boundary condition, etc.

The work in [207] makes use of CCP for the design of reprographic machines. In this case, the CCP model allows to capture in a declarative and compositional way the model of the machine in an appropriate level of abstraction, thus providing support for the requirement specification and design activities.

Finally, in [153, 167] the authors show how tasks for an RCX programmable micro-controller can be specified in the `ntcc` calculus.

5.2 Music Interaction and Composition

Many systems for music composition and interaction have been proposed in the past. These are based in general either on dataflow models and languages inspired in digital sound processing systems, for interaction, and on existing (mostly functional) general-purpose programming languages, for music composition. The purpose of the former is controlling musical devices (e.g. sound synthesizers) in real-time performance settings. The latter aim at providing composers with tools for supporting the structuring and controlled evolution of complex musical material. CCP-based calculi have been proposed recently in both domains. What is intended in the first case is to take advantage of the natural synchronization mechanism provided by blocking ask processes to model complex concurrent interactions in a precise and simple way. In the second case, the logical nature of the calculus is used to verify musical properties of a system before launching costly constraint processes.

In [175], `ntcc` models of various musical problems are described. These problems involve relations between harmonic and rhythmic properties. What (harmonic) information is output at each time unit determines rhythm properties. The problem consists in finding out whether two musical voices with specific melodic evolution rules can comply with some given harmonic relations when played together. This problem pops up frequently in music composition in many different forms. The particular instance of this problem described in [175] is the following: two voices are constructed in such a way that the second one reproduces the first (up to transposition) with a time gap of p. The upper and lower voices play notes in the sets S_1 and S_2, respectively. A transposition function f gives for each upper voice note the lower note that is to be played p time units later. Additional constraints state that time units that are either contiguous or separated by p units should not play the same two notes (chords). Finally, all chords thus formed between the two voices must be chosen among the elements of a given set C. The strategy is to construct a weaker ntcc model of the problem and then, use the linear temporal logic associated with ntcc to find conditions for the problem to be solvable.

In [13] an entirely different domain is explored using ntcc, that of live improvisation of an interpreter and the computer. The computer must first learn the musical *style* of the human interpreter and then begin to play jointly in the same style. A *style* in this case means some set of meaningful sequences of musical material (notes, durations, etc.) the interpreter has played. A graph structure called *factor oracle* (FO) is used to efficiently represent this set. The ntcc models define processes that construct in real-time the FO (i.e. learn the style) and then synchronize with the interpreter to travel through different paths in the FO graph (i.e. improvise).

Interactive Multimedia deals with scenarios where multimedia content and interactive events are related via computer programs. A recent trend is to express the relationships between contents and interactions in a precise way by integrating both in a general model, thus providing a kind of enriched score for composers. One such system, called *interactive scores* [2], allows the specification of contents whose temporal occurrence is not given in advance but is the result of temporal constraints (in the form of Allen interval relations) defined for them. The occurrence of external interactions (whose window of observation is also subject to constraints) is thought to transform (or instantiate) the temporal structure of the piece by imposing further constraints. The score thus defines a collection of possible temporal occurrences of the audio/visual events in a performance. In [204], ntcc has been proposed to extend this model so that external interactions can condition paths in the score. The calculus is also used to implement a precise synchronization of processes in this model. In [156] the ability of utcc to express mobile behavior was exploited to define interactive scores where interactive points can be defined to adapt the hierarchical structure of the score depending on the information inferred from the environment.

The ntcc system in [176] is proposed as a framework for constructing sound processing models in a precise and compact way. Processes in a given time unit define (data flow) transformations of a sound sample supplied by the environment (or a past process). The resulting sample is then output and can also be transmitted to the next time unit. Shared variables are used to represent links between processes. Sample delay units are straightforwardly represented with the ntcc *next* primitive. The compositionality of the calculus is used to represent hierarchies of sound processing boxes.

5.3 Biological Systems

The study of biological systems has found a fertile substrate in the CCP model, mainly due to: (1) constraints can naturally express quantitative information as well as partial information on the available reactants in the system; (2) synchronization via constraint entailment allows for triggering actions when some information can be derived from the system. For instance, it is natural to express that a given reaction occurs only when certain component is present in the system; (3) the ability of CCP to build up models (i.e., components) by parallel composition leads to a robust modeling strategy: one can study separately components of a system and then observe the behavior of the whole system; (4) timed operators in temporal extensions of CCP allows for describing actions (more precisely reaction in this context) that can take several time-units to be completed; and (5), probabilistic constructs as in `pcc` allow for choosing, according to a given probability distribution function, among different reaction that may have different propensities to occur.

In [10, 104] the authors propose a model in `ntcc` of a mechanism for cellular transport: the Sodium-Potassium pump. In the same work, the connection of `ntcc` and linear temporal logic is exploited to facilitate reachability analysis.

BioWayS (`http://avispa.javerianacali.edu.co`) [57] is a web tool for the modeling of biochemical networks based on the `ntcc` calculus. In [112], it is shown that BioWayS allows for the compositional modeling and simulation of biological systems.

The stochastic extension `sCCP` proposed in [44] allows for describing stochastic duration by means of functional rates. This calculus has been used to describe biological networks and it has been shown to be a general and extensible framework to describe a wide class of dynamical behaviors and kinetic laws.

The discrete and continuous nature of `hcc` has been exploited to model dynamic biological systems, e.g. in [74, 32]. For instance, in [32] it is shown that `hcc` can naturally model a variety of biological phenomena, such as reaching thresholds, kinetics, genetic interaction and biological pathways.

In [46], the authors carry out a comparative study of `sCCP`, `hcc`, and Biocham [75] as languages for the modeling of biochemical reactions. In [26] the authors compare the `sCCP`, `ntcc`, and `hcc` models for the blood coagulation process. Experimental results are shown when using the interpreters of `hcc` and `ntcc` to simulate the system.

Finally, [178] makes use of a linear CCP language to model protein interaction. The work in [160] uses CCP techniques for the protein structure prediction problem, which consists in predicting the 3D native conformation of a protein, when its sequence of amino acids is known. The authors also provide a prototype in the Oz language showing the feasibility of the approach proposed.

5.4 Security and Service Oriented Computing

Due to technological advances such as the Internet and mobile computing, security has become a serious challenge in Computer Science. Several process calculi have been proposed in order to deal with the verification of security protocols. Some of the features of those calculi are reminiscent of CCP. For instance, SPL [61], the spi calculus variants in [1] and the calculi presented in [8] and [42] are all operationally defined in terms of configurations containing items of information (messages) which can only increase during evolution. Such monotonic evolution of information is akin to the notion of monotonic store in CCP. Moreover, the calculi in [8, 42, 86, 18] are parametric in an entailment relation over a logic for reasoning about protocol properties very much like CCP is parametric in an entailment relation over an underlying constraint system.

The notion of mobility in `utcc` is used in [158] to model and exhibit the secrecy flaw of the well known Needham-Schroeder [150] security protocol. The cryptographic primitives and the messages an attacker may infer are specified in a suitable constraint system. In [157] the authors describe an encoding of a simple language for security into monotonic `utcc` processes (i.e. processes not including the **unless** constructor). Then, by using the denotational semantics of `utcc`, the authors show that it is possible to give a closure operator semantics to languages for security. Moreover, [82] develops an abstract interpretation framework to approximate the semantics of a security protocol for verification purposes.

A type system for restricting the behavior of agents in `utcc` is studied in [113]. This system gives guarantees that a channel name and encrypted values are only extracted by agents that are able to infer the channel or the non-encrypted value from the store.

An extension of `tccp` [34] is studied in [129] as a language for modeling security protocols. The authors show how the language can naturally express the behavior of the principals in the protocol and a Dolev-Yao attacker [67] for verification purposes.

In [118], a policy language for role-based access control in distributed systems along the lines of **Default** `tcc` is proposed. The authors combine constraint reasoning and temporal logic model-checking to verify whether a given resource (e.g. a directory in a file system) can be accessed.

Soft Concurrent Constraint Programming based languages have been used in the modeling and verification of service oriented computing systems. The `cc-pi` calculus, for instance, has been used to specify Quality of Service (QoS) and to conclude Service Level Agreement (SLA) contracts [50]. The language is equipped with mechanisms for resource allocation and for joining different SLA requirements to reach an agreement between agents (clients and servers) in a service oriented computing scenario. In [53], the non-deterministic choice in `cc-pi` is replaced by a prioritized guarded choice. Alternatives are labeled with constraints and the chosen one corresponds to the constraint with a higher priority over the constraints of the alternative branches. The prioritized calculus is then used to express richer QoS negotiations where agents may state preferences between a set of possible alternatives. See also [52] where an overview of `cc-pi` and prioritized `cc-pi` is given. That work describes also the application of these languages in the specification of service negotiation in telecommunication and financial domains in the context of the SENSORIA project (`http://www.sensoria-ist.eu`).

The work in [31] shows the application of a non-monotonic extension of `scc` (where constraints can be *retracted* from the store) in the modeling of negotiation processes where different parties have to agree on a contract specifying QoS requirements expressed as semiring values. Moreover, in [27], the authors specify with the same language an access control mechanism with granularity at the level of constraints. Then, it is possible to control the way the different agents of the system post and consume information.

Sessions and sessions types [203, 64] were introduced with the aim of guaranteeing structured communication between agents. Type rules statically ensure that a predefined communication scheme/protocol (typically based on duality) is respected along process execution. The work in [133] studies an encoding of the language for structured communications proposed in [115] into `utcc`. The framework allows for the declarative analysis of sessions in network protocols. Furthermore, due to the timed nature of `utcc`, it is possible to reason about session duration and expiration in structured communications. In [59], the authors combine CCP and name passing in the style of `cc-pi` together with sessions. The resulting calculus aims at specifying QoS requirements where safe interactions between clients and servers are assured. The primitive used to open a session makes use of constraints whose satisfaction is necessary for starting and conducting the session interaction. Hence, constraints and the underlying type system guarantee bilinearity, i.e., channels are private and they are exclusively used to carry on the communication prescribed by the session. The calculus is also endowed with a primitive for delegation (also restricted by constraints) that allows agents in a protocol to delegate a service to a third party.

In [49] a different approach for guaranteeing structured communication is devised. In this work, the client-service interaction is decomposed in three phases: *negotiation* where agents negotiate certain desired behaviors, but without any guarantee of success; once the agents agree, they *commit* and the choosing behaviors must be respected; finally, the protocol is *executed* upon the agreed properties and deadlock-freeness is guaranteed. The communication scheme is specified in a source calculus close to the π-calculus establishing communication patterns between clients and services. This model is then compiled into a target calculus close to the prioritized `cc-pi` calculus [53] where named constraint semirings [50] encode the behaviors of agents. More precisely, the choices in the compiled model are guarded by check (ask) constructs that enable the corresponding continuation only if the global store allows it. The novelty of this approach relies on the fact that constraints are used to choose the right interaction and to avoid deadlock in the execution phase. That is, the combination of the constraints of the client and the server leads only to executions of the client-service system that do not stuck.

Propositional Contract Logic (PCL) [17] extends intuitionistic logic with a *contractual* form of implication. This logic aims to model SLA contracts to formalize the duties of the client and the server in a service oriented scenario. The execution model of this logic is based on a calculus of contracting processes which relies on a CCP language (plus primitives for a name passing discipline) where agents tell and asks formulas in PCL. The calculus provides also a *fuse* operation that, unlike `cc-pi`, allows for simultaneous multiparty agreements.

5.5 Other Applications and Results

In [206], a Büchi finite state automata characterization of the strongest postcondition of the local independent fragment of the `ntcc` calculus is given. Using this characterization, the author proves the decidability of the satisfaction problem for the restricted negation formulas without rigid variables in linear temporal logic [135].

The work in [209] studies the execution of formal specifications in SPECS-C++, a model-based formal specification language designed for specifying the interfaces of C++ classes. Since this specification language was not designed to be executed, the approach proposed by the authors is to translate such a formal specifications into the CCP based language AKL [109]. A subset of the specifications in SPECS-C++ can be then executed in AKL. If the specification is consistent (and executable), it is possible to find the set of post-states satisfying the specification.

The connection between CCP languages parametrized with a finite domain (FD) constraint system and query languages in finite model theory such as first-order logic over relational vocabulary, fixpoint logics, and Datalog is studied in [78]. This work presents complexity results for CCP(FD) when considering complete and open constraint systems (those that do not fix the interpretation of all relation symbols); flat and deep guards, i.e. guards that can be an arbitrary process instead of a constraint; and terminal and success observables (where there are no blocking asks in the final configuration).

The work in [191] introduces a constraint system to handle equations and inequations over real numbers. This constraint system along with the model of `lcc` provides a general and extensible foundation for linear programming algorithms design. The authors show that it is possible to build a version of the (constraint solver) simplex algorithm in this framework and additionally, that it is possible to specify non-trivial concurrent algorithms on it.

In [141] the authors experiment with the use of `ntcc` as a language to describe dynamic enumeration strategies to solve constraint satisfaction problems. In this case, the reactivity of the calculus allows to design enumeration strategies that adapt themselves according to information issued from the resolution process and from external solvers such as an incomplete solver (e.g. local search).

A `lcc` model of the access permission mechanism in Aeminium [201], an object-oriented concurrent-by-default programming language, is presented in [155]. The logical interpretation of `lcc` as formulas in intuitionistic linear logic allows for the automatic verification of Aeminium's program properties such as deadlock detection, correctness with respect to the access permission specification, and the ability of methods to run concurrently.

6 Verification and Reasoning Techniques

In this section we survey different reasoning techniques developed for CCP. As we stated before, CCP is a model of concurrency tied to logic. Then, CCP benefits from verification techniques coming from both process calculi and logic. In the forthcoming section, we give an account of semantics, program analyses, logic characterizations, model checking and equivalences developed for CCP calculi.

$\mathrm{D}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]_I$	$=$	$\mathcal{C}$
$\mathrm{D}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]_I$	$=$	$\{d \mid d \vdash c\}$
$\mathrm{D}_{\mathrm{ASK}}$	$[\![\mathbf{when}\, c\, \mathbf{do}\, P]\!]_I$	$=$	$\{d \mid d \not\vdash c\} \cup [\![P]\!]_I$
$\mathrm{D}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]_I$	$=$	$[\![P]\!]_I \cap [\![Q]\!]_I$
$\mathrm{D}_{\mathrm{LOCAL}}$	$[\![(\mathbf{local}\, \overline{x}; c)\, P]\!]_I$	$=$	$\{d \mid$ there is d' s.t. $d' \vdash c, \exists\overline{x}(d) = \exists\overline{x}(d')$ and $d' \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{CALL}}$	$[\![q(\overline{x})]\!]_I$	$=$	$I(q(\overline{x}))$

Figure 2: Semantic equations for the CCP calculus in Definition 3. $\mathcal{C}$ denotes the set of constraints in the underlying constraint system.

6.1 Semantic Frameworks and Program Analysis

The first semantic characterizations of CCP were inspired by methods and techniques from concurrency theory such as failure sets and bisimulation. For instance, [37] and [38] made use of tree-like structures labeled with functions on substitutions. Simpler tree-like structures labeled by constraints are used in [94], and in [186], similar structures modulo equivalence relations based on bisimulation are considered. The work in [39] showed that it is possible to give a simpler compositional semantics for CCP since communication is asynchronous and actions are triggered only depending on the current store. The proposed semantics consists of sequences of constraints labeled by assume/tell modes.

In [187], deterministic CCP processes are identified with Scott's closure operators (idempotent, extensive and monotonic functions). Such functions can be retrieved from their set of fixpoints and then, the meaning of a process is given by the set of constraints upon which the process cannot add any information. This set is also known as the *strongest postcondition* and it corresponds to the *quiescent* constraints for the process. This semantics characterization is quite elegant and simple. Figure 2 shows the semantic equations for the CCP calculus in Definition 3. Let us elaborate on them. Notice that all constraints are quiescent for $\mathbf{skip}$ ($\mathrm{D}_{\mathrm{SKIP}}$), i.e., $\mathbf{skip}$ cannot add any information to any constraint. A process $\mathbf{tell}(c)$ cannot add any information to d if d entails c ($\mathrm{D}_{\mathrm{TELL}}$). A constraint d is quiescent for $\mathbf{when}\ c\ \mathbf{do}\ P$ either if d does not entail c or d is quiescent for P ($\mathrm{D}_{\mathrm{ASK}}$). The constraint d is quiescent for $P \parallel Q$ if neither P nor Q can add any information to d ($\mathrm{D}_{\mathrm{PAR}}$). If d' cannot add any information to P and d and d' differ only on the information about the variables in $\overline{x}$ (i.e., $\exists\overline{x}(d) = \exists\overline{x}(d')$), then d cannot add any information to $(\mathbf{local}\, \overline{x}; c)\, P$ ($\mathrm{D}_{\mathrm{LOCAL}}$). Process calls are interpreted according to the *interpretation* I that gives meaning to the process definition ($\mathrm{D}_{\mathrm{CALL}}$). The semantics is then obtained by a standard fixpoint construction.

For the case of non-deterministic CCP, [187] denotes processes with bounded traced operators that recall the path the processes followed to reach the fixpoint, i.e., the sequence of tell/ask interactions with the environment. This semantics was proven to be equivalent [40] to that in [39]. The result follows from the fact that both semantics are fully abstract with respect to the notion of observables for finite computations. Furthermore, [80] studied restricted fragments of (non-deterministic) CCP that can be characterized as closure operators on sets of constraints. More precisely, the authors show that this is possible for *structurally confluent* CCP processes, i.e., processes whose outputs do not depend on the scheduling policy of the system. This fragment includes, for instance, *angelic*-CCP where only *local choices* are allowed (i.e, all the guards are the same) and *mutually exclusive choice* (i.e., one guard excludes the others). The semantic characterization for CCP with local choice found application in [36, 81] to establish the semantic foundations and a verification system for CLP with delay by means of closure operators. Confluence in CCP has been also studied in [148] where a confluent operational semantics for CCP with blind and angelic choice is proposed. In the same lines, [136] proposes a confluent calculus for CCP that considers blind and guarded choices. This calculus is later used for the analysis of CCP programs. To effectively deal with guarded choices, the semantics in [136] keeps the precluded alternatives when selecting a branch of execution. Those alternatives are guarded in such a way that they reduce to "failure" on termination.

For the case of infinite computations and non-deterministic behavior, the semantic foundations in [187] were extended in [119] and [126] to give meaning to *angelic*-CCP processes. Later on, [41] showed that the domain used in [119, 126] is not closed under set intersection and then, the semantics of parallel composition is not well defined. In [154] the authors considered the Lehmann's powerdomain [128] over set of traces. Then, a (compositional) fixpoint semantics can be obtained in order to retrieve the outputs of infinite computations where fairness is assumed, i.e., all enabled agents are eventually executed. Relying on these ideas, [41] shows that the construction in [154] can be used to capture both infinite computations and non-determinism when considering sets of constraints (instead of traces). Fairness requirements have been also studied in [54], where the operational semantics of the parallel operator makes use of quantitative metrics to provide a more accurate way to establish which of the processes in a parallel composition can succeed. This thus guarantees a fair criterion on the selection of processes.

The elegant semantic characterization of CCP has been extended to its subcalculi. In [182], `tcc` processes are denoted as closure operators on sequences of constraints. Then, for instance, the sequence $c_1.c_2.c_3....$ is quiescent for $\mathbf{next}\, P$ only if the subsequent $c_2.c_3...$ is quiescent for P. Similarly, [153] gives a denotational semantics to the `ntcc` calculus. This idea is also present in [157] where processes are identified with closure operators on sequences of linear temporal logic formulas [135] for the case of the `utcc` calculus. As for `tccp`, in [34] it is shown that the full abstraction problem for this language cannot be reduced to that one of CCP. Then, a semantics based on reactive sequences is proposed to be correct with respect to the notion of observables and fully abstract. Following these ideas, [28] proposes a semantics for `stcc`.

The aforementioned semantic characterizations of CCP rely on the idea of the strongest postcondition, i.e., the quiescent inputs of a process. A finer-grain characterization consists in determining the *minimal* requirements from the environment to produce an output, this is, observe the *causality* relation between inputs and outputs. These ideas have been developed to endow CCP with true concurrency semantics, i.e., semantics that interpret the parallel operator as a concurrent execution (instead of an interleaving execution) of processes. In [143, 144, 145], the operational rules of CCP are augmented with the context required for the reaction to occur. From such rules, the authors show that a contextual net [146] (a Petri Net that considers contexts representing the conditions required for an event to occur) can be constructed to capture all possible computations of a given program. It allows also to capture causal dependencies, mutual exclusion and the concurrency among processes. The contextual net semantics of CCP has been extended in [147, 48] with an inconsistency dependency relations to deal with atomic tells (see Section 3.5). This semantics is exploited to derive safe parallelization of CLP computation steps. The ideas in [143, 145] found also application in [102] where a semantics for CCP based on *contexted tokens* of the form c^d is proposed. Intuitively, c^d means d is the cause for the effect c. In this work it is also shown that the contexted tokens a process may output can be retrieved compositionally.

In the context of program analysis and transformations, unfold/fold transformations in CCP have been studied in [73]. The proposed transformation system, besides folding and unfolding, includes other new operations, namely backward instantiation, ask and tell simplification, branch elimination, conservative ask elimination and distribution. This framework has found application for proving deadlock freeness of CCP programs. Furthermore, [21] investigated transformation techniques for CCP based on the *replacement* (see [163] for a survey of transformation techniques in logic languages). Abstract semantic characterizations for CCP have been studied in [79, 213]. Those works proved that it is not possible to give a sound approximation, in the sense of abstract interpretation [60], for the ask operator if one considers only abstract domain values. The main problem is that *weaker* constraints are needed to over-approximate the program outputs but ask-synchronization requires *stronger* constraints to guarantee that suspension in the abstract model implies suspension in the concrete model. Then, an entailment relation between abstract and concrete constraints is used to give a safe approximation of the semantics. These ideas have been extended in [82] to consider temporal extensions of CCP. The proposed abstract semantics have been used to prove properties such as groundness and suspension freeness and they have served as the foundation for abstract diagnosis and debugging techniques for `tccp` [58] and `ntcc` programs [83].

6.2 Logics, Specifications and Verification

CCP-based languages have been shown to have a strong connection to logic that distinguishes this model from other formalisms for concurrency. The work in [138] shows that CCP processes can be viewed as logic formulas, constructs in the language as logical connectives and simulations (runs) as proofs. In [126], the semantic characterization of CCP processes is used to show that the logical view of the program and its denotation correspond to each other. Then, proving that P satisfies a given property F amounts to show that the semantics of P is included in the semantic objects satisfying F. Here the author considered infinite computations and then, liveness properties (i.e., something *good* eventually happens) may be proved. In [33], a calculus for proving correctness of CCP programs is introduced. In this framework, the specification of the program is given in terms of a first-order formula. The authors pointed out that some problems arise when representing non-deterministic choices by disjunction and when considering the representation of this logical connective in the constraint system. For example, assume that process P satisfies certain property F and consider the agent $Q = \mathbf{when}\ x = 0\ \mathbf{do}\ P + \mathbf{when}\ x > 0\ \mathbf{do}\ P$. One would like to state that the above process satisfies the formula F' defined as $(x = 0 \vee x > 0) \Rightarrow F$. Logically speaking,

$x \geq 0$ implies $x = 0 \vee x > 0$. Nevertheless, if we run Q in parallel with $\mathbf{tell}(x \geq 0)$, none of the guards of Q is enabled and then, P is not executed. Therefore, the authors enrich the logic of the constraint system and a property (represented by a constraint) is thus interpreted as the set of constraints that entails it. Consequently, logical operators are interpreted in terms of the corresponding set-theoretic operations. For instance, the formula F' above is interpreted as the union of the set of constraints that entails $x = 0$ and F and the set of constraints that entails $x > 0$ and F.

As for `lcc`, [77] shows that the observable behavior of CCP and `lcc` processes can be characterized as proofs in intuitionistic linear logic. This characterization is shown to be useful to verify *liveness* properties of systems. Furthermore, the language is endowed with a phase semantics to verify *safety* properties.

In the context of timed CCP calculi, [182] proposes a proof system for `tcc` based on an intuitionistic logic enriched with a next operator. Judgments in the proof system have the form $P_1, ..., P_n \vdash P$ where $P_1, ..., P_n$ and P are processes. Such judgments are valid if and only if the intersection of the denotations of the agents $P_1, ..., P_n$ is contained in the denotation of P; equivalently, any observation that can be made from the parallel system of agents $P_1, ..., P_n$ can also be made from P. The results in [33] are extended and strengthened in [153], where a linear temporal logic characterization of `ntcc` is studied along with a proof system to verify properties. In the same lines, [158] gives a logic characterization for the `utcc` calculus. A temporal logic based on the epistemic modalities of *knowledge* and *belief* is proposed in [35]. The assertions in the logic capture what a process *assumes* from the input (the environment) and what a process *commits* to, i.e., the outputs in a given time-unit. Then, this logic provides a language for the specification of the reactive behavior of processes. A sound a complete proof system to reason about the correctness of `tccp` programs based on this logic is reported in [35].

Model checking techniques have been also explored in the context of temporal CCP languages. In [3, 85], the behavior of `tccp` processes is modeled by means of a so called `tccp` structure. Such structure is like a Kripke structure where, following the CCP philosophy, the state of the system is represented as a conjunction of constraints. This allows for the specification of the possible states and transitions the system may exhibit. Then, the modal logic in [35] is used to specify the property to be verified and the model checking graph is analyzed to decide if the process satisfies or not the property. A tool implementing the construction of the `tccp` structure is described in [131]. The work of [85] was based on the ideas developed for the automatic verification of `tcc` programs in [84].

In order to mitigate the state explosion problem, [5] considers an abstract model-checking technique where both the model and the property to be verified are abstracted. Following a data-abstraction approach, abstract operations are defined to over approximate the behavior of the original program. The authors identify that an over-approximation is not sufficient to give an accurate approximation of the synchronization and timed mechanisms of `tccp`. Then, under-approximations of the semantics are considered to improve accuracy. The authors prove the total correctness of the abstract semantics which models precisely the suspension of processes. The abstract semantics is also implemented as a source-to-source transformation that compiles the abstract program back into `tccp`. Abstract refinements are also proposed to improve accuracy in the verification process. The work in [5] is extended in [7] where the authors study a general framework for abstract verification and analysis of concurrent programs. The programs considered in this study are imperative (states as variable valuations) and declarative (states as conjunction of constraints as in `tccp`) style. The semantics is approximated and implemented as a source-to-source transformation that interprets the abstract actions into the original language. Finally, we mention the work of [6] that proposes a new semantics for `tccp` able to recognize the time instant when some piece of information is added to the store. The logic in [35] is also extended with discrete-time marks to model synchronous real-time properties. Thus, real time is introduced in `tccp` and it is shown how model checkers for this language can be extended to deal with real-time properties.

6.3 Equivalences

Bisimilarity is one of the main representative of process equivalences. It captures our intuitive notion of process equivalence; two processes are equivalent if they can match each other moves. Furthermore, it provides an elegant co-inductive proof technique based on bisimulation [177]. Despite the relevance of bisimilarity, there have been few attempts to define a notion of bisimilarity for CCP. The work in [11] provides a labeled transition semantics and a notion of bisimilarity for CCP. A labeled transition $\langle P, c\rangle \overset{d}{\to} \langle Q, e\rangle$ says that d is the minimal piece of information that needs to be joint with c to perform a reduction from P. From these transitions, a derived notion of bisimulation following standard lines is obtained: Two configurations $\langle P, c\rangle$ and $\langle P', c'\rangle$ are bisimilar iff whenever $\langle P, c\rangle \overset{d}{\to} \langle Q, e\rangle$ then must exist a transition $\langle P', c' \wedge d\rangle \to \langle Q', e'\rangle$ so that $\langle Q, e\rangle$ and $\langle Q', e'\rangle$ are also bisimilar. The authors also showed a strong correspondence with existing CCP notions by providing a fully-abstract characterization of a standard observable behavior. Furthermore, in [12] the authors provided an algorithm for the automatic verification of bisimilarity. In the same lines, [105] studies a labeled bisimulation for Linear CCP processes. The latter equivalence is shown to coincide with a may-testing equivalence and the barbed congruence. Finally, a notion of open bisimulation is proposed in [51] for `cc-pi`. Essentially, two processes are open bisimilar if they have the same stores of constraints - which can be statically checked - and if their moves can be mutually simulated.

6.4 CCP and other Computational Models

The Fusion calculus [208] is a π-calculus variant that, rather than substitution, uses an implicit notion of equality, a fusion, between variables/names. So, in the Fusion calculus instead of replacing a parameter x of an input with that of an output, say z, an implicit fusion is given between the parameters involved in the communication by imposing $x = z$. This idea results in a calculus that is simpler and yet as expressive as the π-calculus. In fact, the Fusion calculus has only one binding operator where the π-calculus has two (input and restriction) and it has a complete symmetry between input and output actions. The authors in [208] gave an encoding from CCP into Fusion calculus as a compelling demonstration of the expressivity of their calculus. This makes the reasoning techniques and tools of the Fusion calculus available for CCP. However, the encoding is only intended for equality (inequality) based constraint systems such as the Herbrand constraint system. This may not come as a surprise since a fusion can be thought of as an equality constraint between two variables. In fact, an encoding from CCP with equality constraints into Explicit Fusion [211], an alternative presentation of the Fusion calculus, should be almost immediate.

CCP offers reasoning techniques substantially different from those from π-based calculi. CCP also focuses on the notion of partial information while it abstracts away from channel and point-to-point communication. It is worth noticing that some variants of the π-calculus include logic assertions in their process language (see e.g., the ψ-calculus [18] and `cc-pi` [50]) as well the use of parametric signatures (see e.g., the applied π-calculus [86]). These recent additions to the machinery of the π-calculus variants bear witness to the importance of the concepts singled out in CCP. We note also that CCP has been shown to be expressive enough to encode other models of computation, different from its predecessors Concurrent Logic Programming and Constraint Logic Programming. For instance, different (fragments of) asynchronous concurrent formalisms such as the asynchronous π-calculus, Actor models, Linda, Petri nets have been encoded as (linear and higher order) CCP processes [121, 181, 179, 127, 105]. Furthermore, sequential models of computations such as the the untyped λ-calculus and Minsky machines have been encoded into CCP [179, 157].

7 Concluding Remarks

The simplicity and elegance of the CCP model have attracted the attention of both practitioners and theoreticians in Computer Science. It can be seen in the large number of extensions proposed in the literature to cope with different notions such as time, non-determinism, mobility, etc. Being parametric in an underlying constraint system, the CCP model has offered the flexibility needed to be adopted as a formal basis for several programming languages and practical applications.

Another appealing feature of CCP is the set of reasoning techniques the model offers. For instance, closure operator semantics, logical characterization, proof and type systems, model checking, and more recently, equivalences and bisimulation techniques.

A current line of research is the development of a more principled notion of time in CCP. This is central to applications that require to impose real-time constraints on the execution of processes as in multimedia interaction systems. In the same lines, building interpreters that guarantee reliable responses in time is required.

From the verification point of view, there are ongoing works in defining more robust proof systems for CCP calculi and static analyzers for CCP programs. Developing machine-assisted tools relying on those techniques for the automatic verification of system properties is also desirable.

Finally, epistemic and spatial constraint systems open a new window for the specification of emergent systems such as cloud computing and social networks in ways that provide more flexible views of information hiding/sharing and where properties such as privacy could be specified.

Acknowledgments. We thank the anonymous reviewers for their detailed comments that helped us to improve this paper. Special thanks to Jorge A. Perez for his careful remarks and suggestions. This work has been partially supported by grant 1251-521-28471 from Colciencias (Colombia), and by Digiteo and DGAR (École Polytechnique) funds for visitors.

References

[1] Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. **148**(1), 1–70 (1999)

[2] Allombert, A., Desainte-Catherine, M., Assayag, G.: Iscore: a system for writing interaction. In: S. Tsekeridou, A.D. Cheok, K. Giannakis, J. Karigiannis (eds.) DIMEA, *ACM International Conference Proceeding Series*, vol. 349, pp. 360–367. ACM (2008)

[3] Alpuente, M., Falaschi, M., Villanueva, A.: A symbolic model checker for tccp programs. In: N. Guelfi (ed.) RISE, *LNCS*, vol. 3475, pp. 45–56. Springer (2004)

[4] Alpuente, M., Gramlich, B., Villanueva, A.: A framework for timed concurrent constraint programming with external functions. ENTCS **188**, 143–155 (2007)

[5] Alpuente, M., del Mar Gallardo, M., Pimentel, E., Villanueva, A.: A semantic framework for the abstract model checking of tccp programs. Theor. Comput. Sci. **346**(1), 58–95 (2005)

[6] Alpuente, M., del Mar Gallardo, M., Pimentel, E., Villanueva, A.: Verifying real-time properties of tccp programs. J. UCS **12**(11), 1551–1573 (2006)

[7] Alpuente, M., del Mar Gallardo, M., Pimentel, E., Villanueva, A.: An abstract analysis framework for synchronous concurrent languages based on source-to-source transformation. ENTCS **206**, 3–21 (2008)

[8] Amadio, R.M., Lugiez, D., Vanackère, V.: On the symbolic reduction of processes with cryptographic functions. Theor. Comput. Sci. **290**(1), 695–740 (2003)

[9] Aranda, J., Pérez, J.A., Rueda, C., Valencia, F.D.: Stochastic behavior and explicit discrete time in concurrent constraint programming. In: de la Banda and Pontelli [14], pp. 682–686

[10] Arbelaez, A., Gutierrez, J., Perez, J.A.: Timed concurrent constraint programming in systems biology. Newsletter of the ALP **19**(4) (2006)

[11] Aristizábal, A., Bonchi, F., Palamidessi, C., Pino, L.F., Valencia, F.D.: Deriving labels and bisimilarity for concurrent constraint programming. In: M. Hofmann (ed.) FOSSACS, *LNCS*, vol. 6604, pp. 138–152. Springer (2011)

[12] Aristizábal, A., Bonchi, F., Valencia, F.D., Pino, L.F.: Partition refinement for bisimilarity in ccp. In: Ossowski and Lecca [159], pp. 88–93

[13] Assayag, G., Dubnov, S., Rueda, C.: A concurrent constraints factor oracle model for music improvisation. In: Proc. of CLEI 2006 (2006)

[14] de la Banda, M.G., Pontelli, E. (eds.): Logic Programming, 24th Int. Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings, *LNCS*, vol. 5366. Springer (2008)

[15] Barahona, P., Felty, A.P. (eds.): Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal. ACM (2005)

[16] Barco, A., Knight, S., Valencia, F.: K-stores A spatial and epistemic concurrent constraint interpreter. In: Proc. of WFLP'12 (2012)

[17] Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: LICS, pp. 332–341. IEEE Computer Society (2010)

[18] Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: a framework for mobile processes with nominal data and logic. Logical Methods in Computer Science **7**(1) (2011)

[19] Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theor. Comput. Sci. **37**, 77–121 (1985)

[20] Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)

[21] Bertolino, M., Etalle, S., Palamidessi, C.: The replacement operation for ccp programs. In: A. Bossi (ed.) LOPSTR, *LNCS*, vol. 1817, pp. 216–233. Springer (1999)

[22] Best, E., de Boer, F., Palamidessi, C.: Concurrent constraint programming with information removal. In: First Workshop on Concurrent Constraint Programming (1995)

[23] Best, E., de Boer, F.S., Palamidessi, C.: Partial order and sos semantics for linear constraint programs. In: D. Garlan, D.L. Métayer (eds.) COORDINATION, *LNCS*, vol. 1282, pp. 256–273. Springer (1997)

[24] Betz, H., Frühwirth, T.W.: A linear-logic semantics for constraint handling rules. In: P. van Beek (ed.) CP, *LNCS*, vol. 3709, pp. 137–151. Springer (2005)

[25] Bistarelli, S.: Semirings for Soft Constraint Solving and Programming, *LNCS*, vol. 2962. Springer (2004)

[26] Bistarelli, S., Bottalico, M., Santini, F.: Constraint-based languages to model the blood coagulation cascade. In: S. Ferilli, D. Malerba (eds.) Logic-Based Approaches in Bioinformatics, pp. 32–41 (2009)

[27] Bistarelli, S., Campli, P., Santini, F.: A secure coordination of agents with nonmonotonic soft concurrent constraint programming. In: Ossowski and Lecca [159], pp. 1551–1553

[28] Bistarelli, S., Gabbrielli, M., Meo, M.C., Santini, F.: Timed soft concurrent constraint programs. In: D. Lea, G. Zavattaro (eds.) COORDINATION, *LNCS*, vol. 5052, pp. 50–66. Springer (2008)

[29] Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. ACM Trans. Comput. Log. **7**(3), 563–589 (2006)

[30] Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., Fargier, H.: Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. Constraints **4**(3), 199–240 (1999)

[31] Bistarelli, S., Santini, F.: A nonmonotonic soft concurrent constraint language to model the negotiation process. Fundam. Inform. **111**(3), 257–279 (2011)

[32] Bockmayr, A., Courtois, A.: Using hybrid concurrent constraint programming to model dynamic biological systems. In: P.J. Stuckey (ed.) ICLP, *LNCS*, vol. 2401, pp. 85–99. Springer (2002)

[33] de Boer, F.S., Gabbrielli, M., Marchiori, E., Palamidessi, C.: Proving concurrent constraint programs correct. ACM Trans. Program. Lang. Syst. **19**(5), 685–725 (1997)

[34] de Boer, F.S., Gabbrielli, M., Meo, M.C.: A timed concurrent constraint language. Inf. Comput. **161**(1), 45–83 (2000)

[35] de Boer, F.S., Gabbrielli, M., Meo, M.C.: Proving correctness of timed concurrent constraint programs. ACM Trans. Comput. Log. **5**(4), 706–731 (2004)

[36] de Boer, F.S., Gabbrielli, M., Palamidessi, C.: Proving correctness of constraint logic programs with dynamic scheduling. In: R. Cousot, D.A. Schmidt (eds.) SAS, *LNCS*, vol. 1145, pp. 83–97. Springer (1996)

[37] de Boer, F.S., Kok, J.N., Palamidessi, C., Rutten, J.J.M.M.: Control flow versus logic: A denotational and a declarative model for guarded horn clauses. In: A. Kreczmar, G. Mirkowska (eds.) MFCS, *LNCS*, vol. 379, pp. 165–176. Springer (1989)

[38] de Boer, F.S., Kok, J.N., Palamidessi, C., Rutten, J.J.M.M.: Semantic models for a version of parlog. In: ICLP, pp. 621–636 (1989)

[39] de Boer, F.S., Palamidessi, C.: A fully abstract model for concurrent constraint programming. In: S. Abramsky, T.S.E. Maibaum (eds.) TAPSOFT, Vol.1, *LNCS*, vol. 493, pp. 296–319. Springer (1991)

[40] de Boer, F.S., Palamidessi, C.: On the semantics of concurrent constraint programming. In: ALPUK, pp. 145–173 (1992)

[41] de Boer, F.S., Pierro, A.D., Palamidessi, C.: Nondeterminism and infinite computations in constraint programming. Theor. Comput. Sci. **151**(1), 37–78 (1995)

[42] Boreale, M.: Symbolic trace analysis of cryptographic protocols. In: F. Orejas, P.G. Spirakis, J. van Leeuwen (eds.) ICALP, *LNCS*, vol. 2076, pp. 667–681. Springer (2001)

[43] Borning, A. (ed.): Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings, *LNCS*, vol. 874. Springer (1994)

[44] Bortolussi, L., Policriti, A.: Modeling biological systems in stochastic concurrent constraint programming. Constraints **13**(1-2), 66–90 (2008)

[45] Bortolussi, L., Wiklicky, H.: A distributed and probabilistic concurrent constraint programming language. In: M. Gabbrielli, G. Gupta (eds.) ICLP, *Lecture Notes in Computer Science*, vol. 3668, pp. 143–158. Springer (2005)

[46] Bottalico, M., Bistarelli, S.: Constraint based languages for biological reactions. In: Hill and Warren [114], pp. 561–562

[47] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984)

[48] Bueno, F., Hermenegildo, M.V., Montanari, U., Rossi, F.: Partial order and contextual net semantics for atomic and locally atomic cc programs. Sci. Comput. Program. **30**(1-2), 51–82 (1998)

[49] Buscemi, M.G., Coppo, M., Dezani-Ciancaglini, M., Montanari, U.: Constraints for service contracts. In: R. Bruni, V. Sassone (eds.) TGC, *LNCS*, vol. 7173, pp. 104–120. Springer (2011)

[50] Buscemi, M.G., Montanari, U.: CC-Pi: A constraint-based language for specifying service level agreements. In: R. De Nicola (ed.) ESOP, *LNCS*, vol. 4421, pp. 18–32. Springer (2007)

[51] Buscemi, M.G., Montanari, U.: Open bisimulation for the concurrent constraint pi-calculus. In: S. Drossopoulou (ed.) ESOP, *LNCS*, vol. 4960, pp. 254–268. Springer (2008)

[52] Buscemi, M.G., Montanari, U.: CC-Pi: A constraint language for service negotiation and composition. In: M. Wirsing, M.M. Hölzl (eds.) Results of the SENSORIA Project, *LNCS*, vol. 6582, pp. 262–281. Springer (2011)

[53] Buscemi, M.G., Montanari, U.: Qos negotiation in service composition. J. Log. Algebr. Program. **80**(1), 13–24 (2011)

[54] Campli, P., Bistarelli, S.: Capturing fair computations on concurrent constraint language. In: Hill and Warren [114], pp. 559–560

[55] Cardelli, L., Gordon, A.D.: Mobile ambients. In: M. Nivat (ed.) FoSSaCS, *LNCS*, vol. 1378, pp. 140–155. Springer (1998)

[56] Carlson, B., Haridi, S., Janson, S.: AKL(FD) - a concurrent language for FD programming. In: SLP, pp. 521–535 (1994)

[57] Chiarugi, D., Falaschi, M., Olarte, C., Palamidessi, C.: Compositional modelling of signalling pathways in timed concurrent constraint programming. In: A. Zhang, M. Borodovsky, G. Özsoyoglu, A.R. Mikler (eds.) BCB, pp. 414–417. ACM (2010)

[58] Comini, M., Titolo, L., Villanueva, A.: Abstract diagnosis for timed concurrent constraint programs. TPLP **11**(4-5), 487–502 (2011)

[59] Coppo, M., Dezani-Ciancaglini, M.: Structured communications with concurrent constraints. In: C. Kaklamanis, F. Nielson (eds.) TGC, *LNCS*, vol. 5474, pp. 104–125. Springer (2008)

[60] Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Log. Program. **13**(2&3), 103–179 (1992)

[61] Crazzolara, F., Winskel, G.: Petri nets in cryptographic protocols. In: IPDPS, p. 149. IEEE Computer Society (2001)

[62] Dahl, V., Niemelä, I. (eds.): Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings, *LNCS*, vol. 4670. Springer (2007)

[63] Demoen, B., Lifschitz, V. (eds.): Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings, *LNCS*, vol. 3132. Springer (2004)

[64] Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and session types: An overview. In: C. Laneve, J. Su (eds.) WS-FM, *LNCS*, vol. 6194, pp. 1–28. Springer (2009)

[65] Díaz, J.F., Gutierrez, G., Olarte, C.A., Rueda, C.: Using constraint programming for reconfiguration of electrical power distribution networks. In: P.V. Roy (ed.) MOZ, *LNCS*, vol. 3389, pp. 263–276. Springer (2004)

[66] Díaz, J.F., Rueda, C., Valencia, F.D.: Pi+- calculus: A calculus for concurrent processes with constraints. CLEI Electron. J. **1**(2) (1998)

[67] Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2), 198–207 (1983)

[68] Dovier, A., Pontelli, E. (eds.): A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP, *LNCS*, vol. 6125. Springer (2010)

[69] Dubois, D., Fargier, H., Prade, H.: The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In: Proc. 2nd IEEE Conference on Fuzzy Systems, pp. 1131–1136 vol.2. San Francisco, CA (1993)

[70] Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: Demoen and Lifschitz [63], pp. 90–104

[71] Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for constraint handling rules. In: Dahl and Niemelä [62], pp. 224–239

[72] Dücker, M., Lehrenfeld, G., Müller, W., Tahedl, C.: A generic system for interactive real-time animation. In: ECBS, pp. 263–270. IEEE Computer Society (1997)

[73] Etalle, S., Gabbrielli, M., Meo, M.C.: Transformations of CCP programs. ACM Trans. Program. Lang. Syst. **23**(3), 304–395 (2001)

[74] Eveillard, D., Ropers, D., de Jong, H., Branlant, C., Bockmayr, A.: A multi-scale constraint programming model of alternative splicing regulation. Theor. Comput. Sci. **325**(1), 3–24 (2004)

[75] Fages, F., Batt, G., Maria, E.D., Jovanovska, D., Rizk, A., Soliman, S.: Computational systems biology in biocham. ERCIM News **2010**(82), 36 (2010)

[76] Fages, F., de Oliveira Rodrigues, C.M., Martinez, T.: Modular CHR with ask and tell. In: Proc. of Fifth Workshop on Constraint Handling Rules (2008)

[77] Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: Operational and phase semantics. Inf. Comput. **165**(1), 14–41 (2001)

[78] Fages, F., Soliman, S., Vianu, V.: Expressiveness and complexity of concurrent constraint programming: a finite model theoretic approach. Tech. Rep. 98-14, LIENS (1998)

[79] Falaschi, M., Gabbrielli, M., Marriott, K., Palamidessi, C.: Compositional analysis for concurrent constraint programming. In: LICS, pp. 210–221. IEEE Computer Society (1993)

[80] Falaschi, M., Gabbrielli, M., Marriott, K., Palamidessi, C.: Confluence in concurrent constraint programming. Theor. Comput. Sci. **183**(2), 281–315 (1997)

[81] Falaschi, M., Gabbrielli, M., Marriott, K., Palamidessi, C.: Constraint logic programming with dynamic scheduling: A semantics based on closure operators. Inf. Comput. **137**(1), 41–67 (1997)

[82] Falaschi, M., Olarte, C., Palamidessi, C.: A framework for abstract interpretation of timed concurrent constraint programs. In: A. Porto, F.J. López-Fraguas (eds.) PPDP, pp. 207–218. ACM (2009)

[83] Falaschi, M., Olarte, C., Palamidessi, C., Valencia, F.: Declarative diagnosis of temporal concurrent constraint programs. In: Dahl and Niemelä [62], pp. 271–285

[84] Falaschi, M., Policriti, A., Villanueva, A.: Modeling concurrent systems specified in a temporal concurrent constraint language-i. ENTCS **48**, 197–210 (2001)

[85] Falaschi, M., Villanueva, A.: Automatic verification of timed concurrent constraint programs. TPLP **6**(3), 265–300 (2006)

[86] Fournet, C., Abadi, M.: Hiding names: Private authentication in the applied pi calculus. In: M. Okada, B.C. Pierce, A. Scedrov, H. Tokuda, A. Yonezawa (eds.) ISSS, *LNCS*, vol. 2609, pp. 317–338. Springer (2002)

[87] Franzén, T., Haridi, S., Janson, S.: An overview of the andorra kernel language. In: L.H. Eriksson, L. Hallnäs, P. Schroeder-Heister (eds.) ELP, *LNCS*, vol. 596, pp. 163–179. Springer (1991)

[88] Frühwirth, T., Michel, L., Schulte, C.: Chapter 13 - constraints in procedural and concurrent languages. In: F. Rossi, P. van Beek, T. Walsh (eds.) Handbook of Constraint Programming, *Foundations of Artificial Intelligence*, vol. 2, pp. 453 – 494. Elsevier (2006)

[89] Frühwirth, T.W.: Constraint handling rules. In: Constraint Programming, pp. 90–107 (1994)

[90] Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. **37**(1-3), 95–138 (1998)

[91] Frühwirth, T.W.: Constraint Handling Rules. Cambridge University Press (2009)

[92] Frühwirth, T.W., Pierro, A.D., Wiklicky, H.: Probabilistic constraint handling rules. ENTCS **76**, 115–130 (2002)

[93] Furukawa, K., Ueda, K.: Ghc - a language for a new age of parallel programming. In: K.V. Nori, S. Kumar (eds.) FSTTCS, *Lecture Notes in Computer Science*, vol. 338, pp. 364–376. Springer (1988)

[94] Gabbrielli, M., Levi, G.: Unfolding and fixpoint semantics of concurrent constraint logic programs. In: H. Kirchner, W. Wechler (eds.) ALP, *LNCS*, vol. 463, pp. 204–216. Springer (1990)

[95] Gabbrielli, M., Palamidessi, C., Valencia, F.D.: Concurrent and reactive constraint programming. In: Dovier and Pontelli [68], pp. 231–253

[96] Gilbert, D., Palamidessi, C.: Concurrent constraint programming with process mobility. In: J.W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, P.J. Stuckey (eds.) Computational Logic, *LNCS*, vol. 1861, pp. 463–477. Springer (2000)

[97] Girard, J.Y.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)

[98] Gupta, V., Jagadeesan, R., Panangaden, P.: Stochastic processes as concurrent constraint programs. In: A.W. Appel, A. Aiken (eds.) POPL, pp. 189–202. ACM (1999)

[99] Gupta, V., Jagadeesan, R., Saraswat, V.A.: Models for concurrent constraint programming. In: U. Montanari, V. Sassone (eds.) CONCUR, *LNCS*, vol. 1119, pp. 66–83. Springer (1996)

[100] Gupta, V., Jagadeesan, R., Saraswat, V.A.: Probabilistic concurrent constraint programming. In: A.W. Mazurkiewicz, J. Winkowski (eds.) CONCUR, *LNCS*, vol. 1243, pp. 243–257. Springer (1997)

[101] Gupta, V., Jagadeesan, R., Saraswat, V.A.: Computing with continuous change. Sci. Comput. Program. **30**(1-2), 3–49 (1998)

[102] Gupta, V., Jagadeesan, R., Saraswat, V.A.: Truly concurrent constraint programming. Theor. Comput. Sci. **278**(1-2), 223–255 (2002)

[103] Gupta, V., Saraswat, V., Struss, P.: A model of a photocopier paper path. In: Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning (1995)

[104] Gutierrez, J., Pérez, J.A., Rueda, C., Valencia, F.D.: Timed concurrent constraint programming for analysing biological systems. ENTCS **171**(2), 117–137 (2007)

[105] Haemmerlé, R.: Observational equivalences for linear logic concurrent constraint languages. TPLP **11**(4-5), 469–485 (2011)

[106] Haemmerlé, R., Fages, F., Soliman, S.: Closures and modules within linear logic concurrent constraint programming. In: V. Arvind, S. Prasad (eds.) FSTTCS, *LNCS*, vol. 4855, pp. 544–556. Springer (2007)

[107] Halbwachs, N.: Synchronous programming of reactive systems. In: A.J. Hu, M.Y. Vardi (eds.) CAV, *LNCS*, vol. 1427, pp. 1–16. Springer (1998)

[108] Hankin, C. (ed.): Programming Languages and Systems - ESOP'98, *LNCS*, vol. 1381. Springer (1998)

[109] Haridi, S., Janson, S., Montelius, J., Franzén, T., Brand, P., Boortz, K., Danielsson, B., Carlson, B., Keisu, T., Sahlin, D., Sjöland, T.: Concurrent constraint programming at sics with the andorra kernel language (extended abstract). In: PPCP, pp. 107–116 (1993)

[110] Henkin, L., J.D., M., Tarski, A.: Cylindric Algebras, Part I. North-Holland (1971)

[111] Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). J. Log. Program. **37**(1-3), 139–164 (1998)

[112] Hermith, D., Olarte, C., Rueda, C., Valencia, F.D.: Modeling cellular signaling systems: An abstraction-refinement approach. In: M.P. Rocha, J.M.C. Rodríguez, F. Fdez-Riverola, A. Valencia (eds.) PACBB, *Advances in Intelligent and Soft Computing*, vol. 93, pp. 321–328. Springer (2011)

[113] Hildebrandt, T., López, H.A.: Types for secure pattern matching with local knowledge in universal concurrent constraint programming. In: Hill and Warren [114], pp. 417–431

[114] Hill, P.M., Warren, D.S. (eds.): Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings, *LNCS*, vol. 5649. Springer (2009)

[115] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin [108], pp. 122–138

[116] Jaffar, J., Lassez, J.L.: Constraint logic programming. In: POPL, pp. 111–119. ACM Press (1987)

[117] Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. **19/20**, 503–581 (1994)

[118] Jagadeesan, R., Marrero, W., Pitcher, C., Saraswat, V.A.: Timed constraint programming: a declarative approach to usage control. In: Barahona and Felty [15], pp. 164–175

[119] Jagadeesan, R., Saraswat, V., Shanbhogue, V.: Angelic non-determinism in concurrent constraint programming. Tech. rep., Xerox Parc (1991)

[120] Jouannaud, J.P. (ed.): Constraints in Computational Logics, First International Conference, CCL'94, Munich, Germant, September 7-9, 1994, *LNCS*, vol. 845. Springer (1994)

[121] Kahn, K.M., Saraswat, V.A.: Actors as a special case of concurrent constraint programming. In: A. Yonezawa (ed.) OOPSLA/ECOOP, pp. 57–66. ACM (1990)

[122] Kahn, K.M., Saraswat, V.A.: Complete visualization of concurrent programs and their executions. In: LPE, pp. 30–34 (1990)

[123] de Kergommeaux, J.C., Codognet, P.: Parallel logic programming systems. ACM Comput. Surv. **26**(3), 295–336 (1994)

[124] Knight, S., Palamidessi, C., Panangaden, P., Valencia, F.D.: Spatial and epistemic modalities in constraint-based process calculi. In: M. Koutny, I. Ulidowski (eds.) CONCUR, *LNCS*, vol. 7454, pp. 317–332. Springer (2012)

[125] Koninck, L.D., Schrijvers, T., Demoen, B.: User-definable rule priorities for chr. In: M. Leuschel, A. Podelski (eds.) PPDP, pp. 25–36. ACM (2007)

[126] Kwiatkowska, M.Z.: Infinite behaviour and fairness in concurrent constraint programming. In: J.W. de Bakker, W.P. de Roever, G. Rozenberg (eds.) REX Workshop, *LNCS*, vol. 666, pp. 348–383. Springer (1992)

[127] Laneve, C., Montanari, U.: Mobility in the cc-paradigm. In: I.M. Havel, V. Koubek (eds.) MFCS, *LNCS*, vol. 629, pp. 336–345. Springer (1992)

[128] Lehmann, D.J.: Categories for fixpoint-semantics. In: FOCS, pp. 122–126. IEEE Computer Society (1976)

[129] Lescaylle, A., Villanueva, A.: Using tccp for the Specification and Verification of Communication Protocols. In: Proc. of WFLP 07 (2007)

[130] Lescaylle, A., Villanueva, A.: The tccp interpreter. ENTCS **258**(1), 63–77 (2009)

[131] Lescaylle, A., Villanueva, A.: A tool for generating a symbolic representation of tccp executions. ENTCS **246**, 131–145 (2009)

[132] Lescaylle, A., Villanueva, A.: Bridging the gap between two concurrent constraint languages. In: J. Mariño (ed.) WFLP, *LNCS*, vol. 6559, pp. 155–173. Springer (2010)

[133] López, H.A., Olarte, C., Pérez, J.A.: Towards a unified framework for declarative structured communications. In: A.R. Beresford, S.J. Gay (eds.) PLACES, *EPTCS*, vol. 17, pp. 1–15 (2009)

[134] Maher, M.J.: Logic semantics for a class of committed-choice programs. In: ICLP, pp. 858–876 (1987)

[135] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag (1991)

[136] Marriott, K., Odersky, M.: A confluent calculus for concurrent constraint programming. Theor. Comput. Sci. **173**(1), 209–233 (1997)

[137] Martinez, T.: Semantics-preserving translations between linear concurrent constraint programming and constraint handling rules. In: T. Kutsia, W. Schreiner, M. Fernández (eds.) PPDP, pp. 57–66. ACM (2010)

[138] Mendler, N.P., Panangaden, P., Scott, P.J., Seely, R.A.G.: A logical view of concurrent constraint programming. Nord. J. Comput. **2**(2), 181–220 (1995)

[139] Milner, R.: A finite delay operator in synchronous CCS. Tech. Rep. CSR-116-82, University of Edinburgh (1992)

[140] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Parts I and II. Inf. Comput. **100**(1), 1–40 (1992)

[141] Monfroy, E., Olarte, C., Rueda, C.: Process calculi for adaptive enumeration strategies in constraint programming. Research in Computer Science (2007)

[142] Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Inf. Sci. **7**, 95–132 (1974)

[143] Montanari, U., Rossi, F.: True concurrency in concurrent constraint programming. In: ISLP, pp. 694–713 (1991)

[144] Montanari, U., Rossi, F.: Contextual occurence nets and concurrent constraint programming. In: H.J. Schneider, H. Ehrig (eds.) Dagstuhl Seminar on Graph Transformations in Computer Science, *LNCS*, vol. 776, pp. 280–295. Springer (1993)

[145] Montanari, U., Rossi, F.: A concurrenct semantics for concurrent constraint programming via contextual nets. In: V. Saraswat, P.V. Hentenryck (eds.) Principles and Practice of Constraint Programming, pp. 3–27. MIT Press (1995)

[146] Montanari, U., Rossi, F.: Contextual nets. Acta Inf. **32**(6), 545–596 (1995)

[147] Montanari, U., Rossi, F., Bueno, F., de la Banda, M.J.G., Hermenegildo, M.V.: Towards a concurrent semantics based analysis of cc and clp. In: Borning [43], pp. 151–161

[148] Montanari, U., Rossi, F., Saraswat, V.A.: Cc programs with both in- and non-determinism: A concurrent semantics. In: Borning [43], pp. 162–172

[149] Müller, T., Müller, M.: Finite set intervals in oz. In: WLP, pp. 17–19 (1997)

[150] Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978)

[151] Niehren, J., Smolka, G.: A confluent relational calculus for higher-order programming with constraints. In: Jouannaud [120], pp. 89–104

[152] Nielsen, M., Palamidessi, C., Valencia, F.D.: On the expressive power of temporal concurrent constraint programming languages. In: PPDP, pp. 156–167. ACM (2002)

[153] Nielsen, M., Palamidessi, C., Valencia, F.D.: Temporal concurrent constraint programming: Denotation, logic and applications. Nord. J. Comput. **9**(1), 145–188 (2002)

[154] Nyström, S.O., Jonsson, B.: Indeterminate concurrent constraint programming: A fixpoint semantics for non-terminating computations. In: ILPS, pp. 335–352 (1993)

[155] Olarte, C., Pimentel, E., Rueda, C., Cataño, N.: A linear concurrent constraint approach for the automatic verification of access permissions. In: D.D. Schreye, G. Janssens, A. King (eds.) PPDP, pp. 207–216. ACM (2012)

[156] Olarte, C., Rueda, C.: A declarative language for dynamic multimedia interaction systems. In: E. Chew, A. Childs, C.H. Chuan (eds.) Mathematics and Computation in Music, *Communications in Computer and Information Science*, vol. 38, pp. 218–227. Springer Berlin Heidelberg (2009)

[157] Olarte, C., Valencia, F.D.: The expressivity of universal timed ccp: undecidability of monadic fltl and closure operators for security. In: S. Antoy, E. Albert (eds.) PPDP, pp. 8–19. ACM (2008)

[158] Olarte, C., Valencia, F.D.: Universal concurrent constraint programing: symbolic semantics and applications to security. In: R.L. Wainwright, H. Haddad (eds.) SAC, pp. 145–150. ACM (2008)

[159] Ossowski, S., Lecca, P. (eds.): Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012. ACM (2012)

[160] Palù, A.D., Dovier, A., Fogolari, F.: Protein folding simulation in ccp. In: Demoen and Lifschitz [63], pp. 452–453

[161] Pérez, J.A., Rueda, C.: Non-determinism and probabilities in timed concurrent constraint programming. In: de la Banda and Pontelli [14], pp. 677–681

[162] Petri, C.A.: Fundamentals of a theory of asynchronous information flow. In: IFIP Congress, pp. 386–390 (1962)

[163] Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. J. Log. Program. **19/20**, 261–320 (1994)

[164] Pierro, A.D., Wiklicky, H.: A banach space based semantics for probabilistic concurrent constraint programming. In: X. Lin (ed.) CATS, *Australian Computer Science Communications*, vol. 20, pp. 245–260. Springer-Verlag Singapore Pte. Ltd. (1998)

[165] Pierro, A.D., Wiklicky, H.: Probabilistic concurrent constraint programming: Towards a fully abstract model. In: L. Brim, J. Gruska, J. Zlatuska (eds.) MFCS, *LNCS*, vol. 1450, pp. 446–455. Springer (1998)

[166] Pierro, A.D., Wiklicky, H.: Concurrent constraint programming: towards probabilistic abstract interpretation. In: PPDP, pp. 127–138. ACM (2000)

[167] del Pilar Muñoz, M., Hurtado, A.R.: Programming robotic devices with a timed concurrent constraint language. In: M. Wallace (ed.) CP, *LNCS*, vol. 3258, p. 803. Springer (2004)

[168] Pilozzi, P., Schreye, D.D.: Improved termination analysis of chr using self-sustainability analysis. In: G. Vidal (ed.) LOPSTR, *LNCS*, vol. 7225, pp. 189–204. Springer (2011)

[169] Puckette, M., Apel, T., Zicarelli, D.: Real-time audio analysis tools for Pd and MSP. In: Proceedings, International Computer Music Conference., pp. 109–112 (1998)

[170] Reisig, W.: Petri Nets: An Introduction, *Monographs in Theoretical Computer Science. An EATCS Series*, vol. 4. Springer (1985)

[171] Reiter, R.: A logic for default reasoning. Artif. Intell. **13**(1-2), 81–132 (1980)

[172] Réty, J.H.: Distributed concurrent constraint programming. Fundam. Inform. **34**(3), 323–346 (1998)

[173] Roy, P.V., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)

[174] Rueda, C., Alvarez, G., Quesada, L., Tamura, G., Valencia, F.D., Díaz, J.F., Assayag, G.: Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. Constraints **6**(1), 21–52 (2001)

[175] Rueda, C., Valencia, F.: On validity in modelization of musical problems by CCP. Soft Comput. **8**(9) (2004)

[176] Rueda, C., Valencia, F.D.: A temporal concurrent constraint calculus as an audio processing framework. In: Sound and Music Computing conference (2005)

[177] Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge Universtity Press (2012)

[178] Saraswat, V.: Euler: an applied lcc language for graph rewriting. Tech. rep., IBM TJ Watson Research Center (2004)

[179] Saraswat, V., Lincoln, P.: Higher-order Linear Concurrent Constraint Programming. Tech. rep., Xerox Parc (1992)

[180] Saraswat, V.A.: The category of constraint systems is cartesian-closed. In: LICS, pp. 341–345. IEEE Computer Society (1992)

[181] Saraswat, V.A.: Concurrent Constraint Programming. MIT Press (1993)

[182] Saraswat, V.A., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: LICS, pp. 71–80. IEEE Computer Society (1994)

[183] Saraswat, V.A., Jagadeesan, R., Gupta, V.: Timed default concurrent constraint programming. J. Symb. Comput. **22**(5/6), 475–520 (1996)

[184] Saraswat, V.A., Jagadeesan, R., Gupta, V.: jcc: Integrating timed default concurrent constraint programming into java. In: F. Moura-Pires, S. Abreu (eds.) EPIA, *LNCS*, vol. 2902, pp. 156–170. Springer (2003)

[185] Saraswat, V.A., Kahn, K.M., Levy, J.: Janus: A step towards distributed constraint programming. In: NACLP, pp. 431–446 (1990)

[186] Saraswat, V.A., Rinard, M.C.: Concurrent constraint programming. In: F.E. Allen (ed.) POPL, pp. 232–245. ACM Press (1990)

[187] Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: D.S. Wise (ed.) POPL, pp. 333–352. ACM Press (1991)

[188] Sarna-Starosta, B., Ramakrishnan, C.R.: Compiling constraint handling rules for efficient tabled evaluation. In: M. Hanus (ed.) PADL, *LNCS*, vol. 4354, pp. 170–184. Springer (2007)

[189] Sarria, G.: Real-time concurrent constraint calculus: The complete operational semantics. Engineering Letters **19**(1), 38–45 (2011)

[190] Sato, T.: A glimpse of symbolic-statistical modeling by prism. J. Intell. Inf. Syst. **31**(2), 161–176 (2008)

[191] Schächter, V.: Linear concurrent constraint programming over reals. In: M.J. Maher, J.F. Puget (eds.) CP, *LNCS*, vol. 1520, pp. 400–416. Springer (1998)

[192] Schrijvers, T., Stuckey, P.J., Duck, G.J.: Abstract interpretation for constraint handling rules. In: Barahona and Felty [15], pp. 218–229

[193] Scott, D.S.: Domains for denotational semantics. In: M. Nielsen, E.M. Schmidt (eds.) ICALP, *LNCS*, vol. 140, pp. 577–613. Springer (1982)

[194] Shapiro, E.: The family of concurrent logic programming languages. ACM Comput. Surv. **21**(3) (1989)

[195] Smolka, G.: A foundation for higher-order concurrent constraint programming. In: Jouannaud [120], pp. 50–72

[196] Smolka, G.: The Oz programming model. In: J. van Leeuwen (ed.) Computer Science Today, *LNCS*, vol. 1000, pp. 324–343. Springer (1995)

[197] Smolka, G.: Concurrent constraint programming based on functional programming (extended abstract). In: Hankin [108], pp. 1–11

[198] Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: Chr(prism)-based probabilistic logic learning. TPLP **10**(4-6), 433–447 (2010)

[199] Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint handling rules. TPLP **10**(1), 1–47 (2010)

[200] Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artif. Intell. **9**(2), 135–196 (1977)

[201] Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: S. Arora, G.T. Leavens (eds.) OOPSLA Companion, pp. 933–940. ACM (2009)

[202] Sussman, G.J., Jr., G.L.S.: Constraints - a language for expressing almost-hierarchical descriptions. Artif. Intell. **14**(1), 1–39 (1980)

[203] Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: C. Halatsis, D.G. Maritsas, G. Philokyprou, S. Theodoridis (eds.) PARLE, *LNCS*, vol. 817, pp. 398–413. Springer (1994)

[204] Toro-Bermúdez, M., Desainte-Catherine, M.: Concurrent constraints conditional-branching timed interactive scores. In: Sound and Music Computing conference. Barcelona, Spain (2010)

[205] Ueda, K., Kato, N., Hara, K., Mizuno, K.: Lmntal as a unifying declarative language: Live demonstration. In: S. Etalle, M. Truszczynski (eds.) ICLP, *LNCS*, vol. 4079, pp. 457–458. Springer (2006)

[206] Valencia, F.D.: Decidability of infinite-state timed ccp processes and first-order ltl. Theor. Comput. Sci. **330**(3), 577–607 (2005)

[207] Varejao, F.M., Fromherz, M.P., Garcia, A.C.B., de Souza, C.S.: An integrated framework for the specification and design of reprographic machines. In: Thirteenth Int. Conf. on Applications of Artificial Intelligence in Engineering. Computational Mechanics Publications (1998)

[208] Victor, B., Parrow, J.: Concurrent constraints in the fusion calculus. In: K.G. Larsen, S. Skyum, G. Winskel (eds.) ICALP, *LNCS*, vol. 1443, pp. 455–469. Springer (1998)

[209] Wahls, T., Leavens, G.T., Baker, A.L.: Executing formal specifications with concurrent constraint programming. Autom. Softw. Eng. **7**(4), 315–343 (2000)

[210] Waltz, D.L.: Gene freuder and the roots of constraint computation. Constraints **11**(2-3), 87–89 (2006)

[211] Wischik, L., Gardner, P.: Explicit fusions. Theor. Comput. Sci. **340**(3), 606–630 (2005)

[212] Wong, H.C., Fromherz, M., Gupta, V., Saraswat, V.: Control-based programming of electro-mechanical controllers. In: IJCAI Workshop on Executable Temporal Logics (1995)

[213] Zaffanella, E., Giacobazzi, R., Levi, G.: Abstracting synchronization in concurrent constraint programming. Journal of Functional and Logic Programming **1997**(6) (1997)