Concurrent Constraint Programming: Calculi, Languages and Emerging Applications

Carlos Olarte\textsuperscript{1,2}, Camilo Rueda\textsuperscript{2,3}, and Frank D. Valencia\textsuperscript{1}

\textsuperscript{1} LIX, École Polytechnique, France
\{carlos.olarte, frank.valencia\}@lix.polytechnique.fr
\textsuperscript{2} Pontificia Universidad Javeriana Cali, Colombia
\texttt{crueda@cic.puj.edu.co}
\textsuperscript{3} IRCAM, France

Abstract. Process calculi treat concurrent processes much like the $\lambda$-calculus treat computable functions. They provide a language in which the structure of terms represents the structure of processes together with an operational semantics to represent computational steps. Concurrent Constraint Programming (CCP) is a process calculus which combines the traditional operational view of process calculi with a declarative one based upon logic. This combination allows CCP to benefit from the large body of techniques of both process calculi and logic. This paper presents a (non-complete) survey of CCP based process calculi. We also show the applicability of the CCP model summarizing the main application of these calculi in different scenarios: timed and reactive systems, biological systems, music interaction and analysis of security protocols.

1 Introduction

Concurrent Constraint Programming (CCP) \cite{52} has emerged as a simple but powerful paradigm for concurrency tied to logic. CCP extends and subsumes both concurrent logic programming \cite{57} and constraint logic programming \cite{31}. A fundamental issue in CCP is the specification of concurrent systems by means of constraints. A constraint (e.g. $x + y \geq 10$) represents partial information about certain variables. During the computation, the current state of the system is specified by a set of constraints called the store. Processes can change the state of the system by telling information to the store (i.e., adding constraints), and synchronize by asking information to the store (i.e., determining whether a given constraint can be inferred from the store).

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

- A \textit{tell} operator adding a constraint to the store.
- An \textit{ask} operator querying if a constraint can be deduced from the store.
- \textit{Parallel Composition} combining processes concurrently.
- A \textit{hiding} operator (also called \textit{restriction} or \textit{locality}) introducing local variables and thus restricting the interface a process can use to interact with others.
In this paper, we shall survey CCP-based calculi, languages and applications. We first recall the notion of constraint system and the basic constructs in CCP (Section 2). Next, in Section 3, we present different extensions of the CCP model dealing with notions such as time, asynchrony, non-determinism, mobility and stochastic behavior. Section 4 is devoted to summarize different programming languages whose basis are the CCP model. In section 5 we show the applicability of the CCP model and its extensions to model systems in a wide spectrum of scenarios including biological and physical systems, reactive systems, verification of security protocols and interaction in music. Section 6 concludes the paper.

2 Preliminaries

CCP calculi are parametric in a constraint system. A constraint system provides a signature from which the constraints can be constructed and an entailment relation $\vdash$ specifying inter-dependencies between these constraints. A constraint represents a piece of information (or partial information) upon which processes may act. For instance, processes modeling temperature controllers may have to deal with partial information such as $42 < \text{tsensor} < 100$ expressing that the sensor registers an unknown (or not precisely determined) temperature value between 42 and 100. The inter-dependency $c \vdash d$ expresses that the information specified by $d$ follows from the information specified by $c$, e.g. $(\text{tsensor} > 42) \vdash (\text{tsensor} > 0)$. We can set up the notion of constraint system by using First-Order Logic as it was done in [58]. Let us suppose that $\Sigma$ is a signature (i.e., a set of constant, functions and predicate symbols) and that $\Delta$ is a consistent first-order theory over $\Sigma$ (i.e., a set of sentences over $\Sigma$ having at least one model). Constraints can be thought of as first-order formulae over $\Sigma$. We can then decree that $c \vdash d$ if the implication $c \Rightarrow d$ is valid in $\Delta$. This gives us a simple and general formalization of the notion of constraint system as a pair $(\Sigma, \Delta)$.

As an example, take the finite domain constraint system (FD) [29]. In FD variables are assumed to range over finite domains and, in addition to equality, we may have predicates that restrict the possible values of a variable to some finite set. More formally, $\text{FD}[n]$ $(n > 0)$ is the constraint system where $\Sigma$ is given by the constant symbols $0, \ldots, n-1$ as well as by the equality $=$, and $\Delta$ is given by the axioms of equational theory $x = x$, $x = y \Rightarrow y = x$, $x = y \land y = z \Rightarrow x = z$, and $v = w \Rightarrow \text{false}$ for each two different constants $v, w \in \Sigma$.

Alternatively, the notion of constraint system can be given in terms of Scott’s information system without consistency structure as it was done in [52].

2.1 CCP Processes

We shall use the notation in [39] to give the syntax of CCP:
Definition 1 (CCP Syntax). Processes $P, Q, \ldots$ in CCP are built from constraints in the underlying constraint system by the following syntax:

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

The process $\text{tell}(c)$ adds the constraint $c$ to the store. The process $\text{when } c \text{ do } P$ 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 $c$. The parallel composition of $P$ and $Q$ is represented as $P \parallel Q$. The process $(\text{local } x) \ P$ behaves like $P$, except that all the information on $x$ produced by $P$ can only be seen by $P$ and the information on $x$ produced by other processes cannot be seen by $P$.

The process $q(y)$ is an identifier with arity $|y|$. We assume that every such an identifier has a unique (recursive) definition of the form

$$q(x) \overset{\text{def}}{=} Q$$

with $x$ pairwise distinct and $|x| = |y|$. The process $q(y)$ behaves then as $Q[y/x]$.

3 CCP-based calculi

Several extensions of the basic constructs presented above have been studied in the literature in order to provide settings for the programming and specification of systems with the declarative flavor of concurrent constraint programming. For example, temporal extensions to deal with reactive systems have been introduced in [50, 16, 39], non-deterministic behavior and asynchrony [39, 16], probabilistic behavior [25, 11, 40], mobility [42], etc. In this section we briefly describe some of these extensions.

Temporal CCP ($tcc$) The first timed CCP model was introduced in [50] as an extension of CCP. It is aimed at programming and modeling timed reactive systems. This model elegantly combines CCP with ideas from the paradigms of Synchronous Languages [5].

The $tcc$ model takes the view of reactive computation as proceeding deterministically in discrete time units (or time intervals). In other words, time is conceptually divided into discrete intervals. In each time interval, a deterministic CCP processes receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the initial store and when it reaches its resting point, it responds to the environment with the final store. Furthermore, the resting point determines a residual process, which is then executed in the next time interval.

This view of reactive computation is particularly appropriate for programming reactive systems such as robotic devices, micro-controllers, databases and reservation systems. These systems typically operate in a cyclic fashion, 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.

The $tcc$ calculus introduces constructs to (1) delay the execution of a process. And (2) time-out (or weak pre-emption) operations that waits 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.

In spite of its simplicity, the tcc extension to CCP is far-reaching. Many interesting temporal constructs can be expressed (see e.g. [50]). 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.

Definition 2 (Deterministic tcc syntax). Processes P,Q,... in tcc are built from constraints in the underlying constraint system by the following syntax:

\[
P, Q := \text{skip} | \text{tell}(c) | \text{when } c \text{ do } P | P \parallel Q | (\text{local } x) P | \text{next } P | \text{unless } c \text{ next } P \mid P
\]

The processes \text{tell}(c), \text{when } c \text{ do } P, P \parallel Q and (\text{local } x) P are similar to those in CCP. The process next \ P delays the execution of \ P to the next time interval. The time-out unless c next \ P is also a unit-delay, but \ P is executed in the next time unit iff \ c is not entailed by the final store at the current time interval. Finally, the replication ! \ P means P \parallel \text{next } P \parallel \text{next}^2 P \parallel ... , i.e., unboundedly many copies of \ P but one at a time.

Non-determinism and Asynchrony. The above syntax has been extended to deal with non-deterministic behavior and asynchrony in the ntcc calculus [39]:

Definition 3 (ntcc Processes). The ntcc processes result from adding to the syntax in Definition 2 the following constructs:

\[
\sum_{i \in I} \text{when } c_i \text{ do } P_i \mid \ast P
\]

The guarded-choice \sum_{i \in I} \text{when } c_i \text{ do } P_i where \ I is a finite set of indices, represents a process that, in the current time interval, must non-deterministically choose one of the \ P_j ( j \in I ) 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.

The operator “\ast” corresponds to the unbounded but finite delay operator \(\epsilon\) for synchronous CCS [35] and it allows to express asynchronous behavior through the time intervals. Intuitively, process \ast P represents \ P + \text{next } P + \text{next}^2 P + ... , i.e., an arbitrary long but finite delay for the activation of \ P.

Strong Pre-Emption in tcc. The work in [64] introduces the notion of strong pre-emption in tcc, i.e. the time-out operations can trigger activity in the current time interval. Strong pre-emption are useful when an action must be triggered immediately on the absence of a constraint \ c rather than delayed to the next interaction with the environment as in unless \ c \text{ next } P. In this case, it is assumed that \ c has not been
produced in the store, and will not be produced throughout system execution at the present time instant. This calculus is called Default \( tcc \) which syntax is obtained as follows:

**Definition 4 (Default \( tcc \) Processes).** The Default \( tcc \) processes result from adding in the syntax in Definition 2 the following construct:

\[
\text{if } c \text{ else } P
\]

Intuitively, if \( c \) cannot be deduced from the current store now and it will not be produced during the current time interval, the process \( P \) is executed.

The \( tccp \) [16] calculus is another timed extension of the CCP model. This calculus introduces a discrete global clock and assumes that \( \text{ask} \) and \( \text{tell} \) actions take one time-unit. Computation evolves in steps of one time-unit, so called clock-cycles, which are syntactically separated by action prefixing. Moreover maximal parallelism is assumed, that is at each moment every enabled agent of the system is activated.

**Continuous Time in CCP.** The Hybrid concurrent constraint programming model, **Hybrid cc** \( (hcc) \)[22], is a calculus where it is possible to express discrete and continuous evolution of time. More precisely, there are points at which discontinuous change may occur (i.e. the execution proceeds as burst of activity) and open intervals in which the state of the system changes continuously (i.e. the system evolves continuously and autonomously). The notion of **continuous constraint system** (a real-time extension of constraint systems) is introduced to describe the continuous evolution. The syntax of Default \( tcc \) is extended with the construct \( \text{hence } P \), asserting that \( P \) holds continuously beyond the current instant. In combination with the other constructs in Default \( tcc \), various patterns of temporal activity can be generated.

**Definition 5 (hcc Processes).** The hcc processes result from adding to the syntax in Definition 4 the following construct:

\[
\text{hence } P
\]

We point the reader to [23] which presents a survey of the development of the previous extensions and the relationships between their semantic models.

**Mobile Behavior** In [42] the authors introduced the **universal timed cc calculus** \( (utcc) \) by extending \( tcc \) with an abstraction operator. This calculus allows for mobility behavior in the sense of the \( \pi \)-calculus, i.e. generation and communication of private channels or links. Basically, \( utcc \) replaces the ask operation \( \text{when } c \text{ do } P \) by a parametric ask constructor of the form \((\text{abs } x; c) P\). This process can be viewed as an abstraction of the process \( P \) on the variables \( x \) under the constraint (or with the guard) \( c \). Operationally, \((\text{abs } x; c) P\) executes \( P[t/x] \) in the current time interval for all the terms \( t \) in the underlying constraint system s.t \( c[t/x] \) is entailed by the current store.
**Definition 6 (utcc Processes).** The utcc processes result from replacing in the syntax in Definition 2 the expression \texttt{when} \(c\) \texttt{do} \(P\) with
\[
(\text{abs } \mathbf{x}; c) P
\]
where variables in \(\mathbf{x}\) are pairwise distinct.

**Probabilistic Behavior** When modeling a physical system, it may be the case that some of its components are only partially known. Then, the model of the system must deal with an approximate or incomplete description of these components. A natural way to model this situation is to give a probability distribution to the set of outputs the partially known components may perform. For example, often information is available about failure models and their associated probabilities of a component in a mechanical device. In [25] a probabilistic model for CCP and tcc is studied where random variables with a given probability distribution are introduced.

In biological systems a similar situation arises. In this setting, a rate is associated to each active reaction, i.e. a real number representing the frequency of occurrence of a certain reaction. Works such as [44, 40, 11] study this kind of stochastic extensions for CCP.

Although the calculi presented in [25, 44, 40, 11] differ in the constructs provided and the semantics given, all of them aim at representing the stochastic nature of the system they model. As an example, in all of them is possible to specify a probabilistic choice of the form \(P + r Q\) where the process \(P\) is executed with a probability \(r\) and \(Q\) with a probability \(1 - r\).

**Linear Concurrent Constraint Programming.** Linear CCP [51, 20, 6] is a variant of CCP where the constraints are built from a linear constraint system (based on Girard’s intuitionistic linear logic) instead of a classic constraint system (Section 2) based on first-order logic.

Constraints in LCC are then linear formulae obtained from a signature \(\Sigma\) and the multiplicative conjunction \(\otimes\), existential quantification and the exponential connective \(!.\) The latter allows to recover the classical constraint system by writing all constraints preceded by “!”.

In LCC, the linear tell operator \texttt{tell}(c) increments the current store \(d\) to \(d \otimes c\) instead of \(d \land c\). Furthermore, the linear ask \texttt{when} \(c\) \texttt{do} \(P\) evolves to \(P\) iff there exists \(e\) s.t. the current store \(d\) entails \(c \otimes e\). When this evolution occurs, the resulting store is \(e\), i.e., the constraint \(c\) is removed. Due to the removal of information, this calculus is intrinsically non-deterministic. Nevertheless, being non-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.

**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 logic programming, replacing unification with constraint solving. An example is the language \texttt{cc(FD)} ([30]) that implements an efficient \textit{finite domains} constraint system.

In contrast with earlier constraint logic programming (CLP) languages, \texttt{cc(FD)} is (as CCP models) parametric in the constraint system. This means that the language can be tailored to specific problem domains without losing the “naturalness” of specifications, due to the need to accommodate pre-established constraints intended for other applications, as is the case with CLP languages. Programs in \texttt{cc(FD)} are written in a prolog-like syntax. For example, in a \texttt{cc(FD)} program for solving the so-called perfect square problem a “no overlap” constraint ensuring that squares do not intersect is the following:

\[
\text{non\_overlap}(X_1, Y_1, S_1, X_2, Y_2, S_2) : - \\
X_1 + S_1 \leq X_2 \lor X_2 + S_2 \leq X_1 \lor Y_1 + S_1 \leq Y_2 \lor Y_2 + S_2 \leq Y_1
\]

User constraints in \texttt{cc(FD)} are translated into canonical forms called \textit{indexicals} [13] 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.

\textbf{Janus}

The year seminal works on CCP were published, Saraswat and Kahn [54] proposed Lucy, a simple language were agents communicate only 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. Careful type restrictions ensure that no inconsistencies crop even when distributed agents perform \textit{tell} operations. An extension, called Janus, was proposed by the same authors. Janus was intended for distributed programming. It resembles concurrent logic programming languages such as Prolog. A Janus program is a network of agents which communicate by passing messages over a channel. Agents consist of \textit{rules} of the form

\[
p(t_1, \ldots, t_n) \leftarrow C | C_1, B.
\]

where \(t_1, \ldots, t_n\) are terms, \(C\) are \textit{ask} and \(C_1\) \textit{tell} constraints, and \(B\) is a conjunction of \textit{goals}. Messages with patterns matching the left hand side of a rule and trigger a behavior determined by its right hand. In accordance with the CCP model, variables are “logical”. In Janus, each variable has two aspects or \textit{faces}, corresponding to \textit{ask} and \textit{tell} annotations over the variable. Any of these can also be passed around. Careful design of restrictions on \textit{askers} and \textit{tellers} ensure the \textit{failure-free property}: Janus computations cannot abort because the store is inconsistent.

A graphical representation of Janus, pictorial Janus was also proposed by Saraswat, Kahn and Levy (see http://jerry.c-lab.de/ wolfgang/PJ/). The basic elements of a PJ program are graphical primitives, i.e., closed contours and connections. The exact shape of a contour has no semantics and can be chosen by the user. Agents and rules are closed
contours with external ports. Guards of rules are (smaller) contours located at the outside of the rule. Channels are represented by arrows. Lines depict links between objects. Rules are drawn inside agents. Figure 1 shows an agent with four rules implementing an AND gate. A visual debugging environment for Pictorial Janus, providing real-time animation of programs, was proposed in [18]. This paper also contains a nice introduction to PJ.

**JCC**

The language jcc [53] 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, where future refers to logical time, as in tcc (e.g. statement next {S}). Interaction of past and current agents follows the semantics of Default tcc processes. A strong point of jcc is its complete integration with a main-stream programming language such as Java.

**LMNtal**

The goal of LMNTAL [61] is to provide a scalable, uniform view of concurrent programming concepts such as processes, messages, synchronous and asynchronous computation. It inherits ideas from the concurrent constraint language GHC 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. 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 computation. An expression \( p(X_1, \ldots, X_m) \) defines an atomic process. Variables \( X_i \) are its links. LMNtal makes no distinction between processes and data. Atom \( X = Y \) denotes a connector. It connects one side of the link \( X \) and one side of the link \( Y \). \( \{ P \} \) denotes a process enclosed with the membrane \( \{ \} \); and \( T : -T \) (\( T \) is a term) a rewrite rule for processes. Links in the left part of the rule are consumed and on the right hand are produced. Rules may define so-called process contexts. For instance, in

\[
\{\text{exch}, \$a[X,Y][]\} : -\{\$a[Y,X][]\}
\]

expression \( \$a[X,Y][] \) specifies a context for process \( \text{exch} \). Rules can also specify contexts. Complex patterns can thus be defined for rule triggering. With these patterns, concepts such as mobility can be easily expressed. Operational semantics of the language is formally given by reduction rules, much like in process calculi. There exists a Java implementation of LMNtal (http://www.ueda.info.waseda.ac.jp/lmntal/).

**Oz**

Arguably the CCP language that has found more widespread use is **Oz**[59, 60]. 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 constraints store 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 kernel **Oz** is small and conceptually simple (see Figure 2) its rich semantics allows the implementation of several computation models: declarative, stateful, lazy declarative, lazy stateful, eager, in both sequential and concurrent settings. In Figure 2 sublanguages corresponding to each computation model are separated by horizontal lines. Many of the constructs are found in main-stream languages. Construct **WaitNeeded** is used for lazy computation. Cells are used to model stateful computation within a single-assignment variables setting. **IsDet** tests whether a single-assignment variable has already been bound. Traditional programming styles such as imperative, functional or object-oriented can homogeneously coexist within the language. A complete presentation and analysis of all **Oz** computation models, together with programming strategies for each can be found in [45].
By default, a constraint system over (infinite) feature trees is used for value assignment to logical variables, but 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 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 in the monotonic store. That is, cell values do not really change. What changes is the variable associated to its name.

The message passing concurrency is implemented with a port abstraction. Ports are channels were messages can be sent. Messages for a port accumulate in an associated (infinite) stream. Ports are implemented using cells and thus are not really primitives of the language. Oz includes also an asynchronous distribution model.

**CORDIAL**

CORDIAL [46] is a visual language intended as a user transparent integration of constraints and objects. The language is based on a CCP calculus extended with the notion of objects and classes. Methods are represented as windows. 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. Objects are identified by an associated constraint parametrized on a local variable (so-called 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 very complex patterns of communication and mobility.

5 Applications

CCP based languages and calculi have been extensively used to model, analyze and verify systems in different scenarios. The reactive model of the temporal extensions makes tcc-based calculi appropriate to specify and program reactive systems, e.g. electromechanical devices, and to design software to control such a systems. In the same way, we shall show that music interaction can be neatly modeled with tcc-based calculi. To describe biological phenomena, probabilistic extensions of CCP are used to model the fact that interactions in these systems follow a given rate, i.e. a probability that some interaction occurs. Finally, we shall show that CCP calculi can also model security protocols and the reasoning techniques provided by the calculus may aide to the verification of security properties on them.

5.1 Physical Systems

In [66], tcc is used as a language for constructing software for computationally-controlled electromechanical systems, such as photocopiers. These systems can be viewed as real-time reactive systems which react continuously with their environment at a rate controlled by the environment. In this way, tcc provides a declarative model for the components that make up the electromechanical device. The authors show that the software based on tcc benefits from the incorporation of timing constructs in tcc that express the pattern of interaction over time between the controller and the environment. Furthermore, since tcc programs can be compiled into real-time finite-state machines with minimal runtime overhead, the implementation of the system is straightforward and efficient. The strong connection of CCP calculi and logic also presents an important advantage in the use of tcc in this context. Namely, programs can be viewed as formulae in a (temporal) logic, thus making possible the use of standard techniques for proving properties over the software constructed.

A natural application of the hcc calculus is in 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 [26], 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 constructs in the calculus are used to create and destroy segments dynamically, without the code of each segment being aware of the creation and destruction.

The work in [63] 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 [39, 17] the authors show how the ntcc calculus allows for the modeling of autonomous robotic devices. In particular it is shown how tasks for an RCX programmable micro-controller can be described in the 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) 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 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 model is used to verify musical properties of a system before launching costly constraint processes.

In [47], 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 that have been prescribed some 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 the paper 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 [4] 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).

The ntcc system in [48] 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.
5.3 Biological Systems

Quantitative information is fundamental for biological systems. E.g., most bio-chemical reactions are highly dependent on the presence of a certain amount of the substances involved. Moreover this information is partial as it is difficult to obtain exact values for parameterizing models. Unpredictable behavior is an inherent condition of the biologic phenomena. One usually counts with “partial behavioral information” for describing interactions in models. This information not only ignores elements on “how” reactions occur, but also “when” such reactions commonly happen. The above-mentioned “partial-information” notions makes CCP a good alternative to model systems in the biological context. The notion of partial quantitative information is central to CCP via constraints. Partial behavioral information is actually the novelty of $\text{ntcc}$ via non-deterministic and asynchronous operators. In [3, 27] the authors propose a model in $\text{ntcc}$ of a mechanism for cellular transport; the Sodium-Potassium pump. In the same work, the connection of $\text{ntcc}$ and linear temporal logic (LTL) is exploited to facilitate reachability analysis. In particular given a LTL formula stating a property of the system, it is possible to know if the process modeling the system is able to reach a state satisfying that property.

In [40] a stochastic extension of $\text{ntcc}$ is proposed and a model of the gene expression mechanism of the $\lambda$-virus is given. Due to the representation as constraints of the quantitative information of the system, the model proposed is more compact than the stochastic $\pi$-calculus model proposed for the same system.

Another stochastic extension of CCP called $s\text{CCP}$ is proposed in [11]. With this extension, the authors can describe straightforwardly biological networks. The $s\text{CCP}$ calculus is then 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 $\text{hcc}$ has been exploited to model dynamic biological systems, e.g. in [19, 9]. For instance, in [9] it is shown that $\text{hcc}$ can naturally model a variety of biological phenomena, such as reaching thresholds, kinetics, genetic interaction and biological pathways.

Finally, [49] makes use of a linear CCP [20] language to model protein interaction. The work in [43] 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

Due to technological advances such as the Internet and mobile computing, security has become a serious challenge in Computer Science. In particular, to deal with the verification of security protocols, several process languages have been defined with remarkable similarities to CCP. For instance Crazzolara and Winskel’s SPL [14], the spi calculus variants by Abadi [1] and Amadio [2], and Boreale’s calculus in [10] 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 [2, 10, 21] 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.

As previously mentioned, the utcc calculus [42] extends tcc with a parametric ask operator of the form (abs $x; c) P$ so-called abstraction. Along with the locality operator ([local $x) P$), utcc is able to model generation and communication of private names, i.e. mobility in the sense of the $\pi$-calculus. This notion is central to the modeling of security protocols. The authors in [42] show how utcc is able to model and exhibit the secrecy flaw of the well known Needham-Schroeder [38] security protocol. Basically, nonces (or secrets) are modeled via the locality operator and inputs in the protocol are represented as abstractions. The cryptographic primitives and the message an attacker may infer are specified in a suitable constraint system.

A second technique for the verification problem of security protocols is presented in [41]. In that work, the authors present an encoding of a simple language for security into monotonic utcc processes (i.e. processes not including the “unless” constructor). Making use of the denotational semantics based on closure operators for the monotonic fragment of utcc, the authors show that it is possible to give a closure operator semantics to languages for security. A new reasoning technique in this field is then proposed using the fixed point of the semantics to characterize a flaw in a protocol.

In [33] an extension of the timed CCP language tccp [16] is studied as a language for modeling security protocols. Furthermore, in [32], a policy languages for role-based access control in the lines of Default tcc is proposed. This language allows for a declarative analysis of access control and trust management.

5.5 Other Applications

The applicability of CCP based calculi to reason about well-established formalisms for concurrency, e.g. Linear Temporal Logic (LTL), has been studied in [62, 41]. In [62], 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 authors prove the decidability of the satisfaction problem for the restricted negation formulae without rigid variables in the Manna and Pnueli’s [34] LTL.

In [41] the authors show how using a very simple and decidable constraint system involving only monadic predicates and no equality nor function symbols, utcc is able to encode Two Counter Machines. It proves the undecidability of the observable relation in this language. Then using the correspondence of utcc with first-order LTL, the authors show that the fragment of monadic first-order LTL without equality and function symbols is strongly incomplete.

The work in [65] 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 CCP programs, more precisely, into the CCP based language AKL [28]. 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. In the case of an inconsistent specification, the interpreter of AKL returns “fail”. Finally, if the specification is not executable, the interpreter either goes into an infinite loop or returns that the computation is suspended. It both cases it means that the program generated from the specification did not sufficiently define the post-state.

The work in [56] introduces a constraint system to handle equations and inequations over real numbers. The constraint system proposed along with the model of Linear CCP [20] 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 [36] the authors experiment with the use of ntcc as a language to describe dynamic enumeration strategies to solve constraint satisfaction problems (CSP). 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). The idea is then to implement dynamic enumeration strategies that speed up the resolution process to find a solution in a CSP.

The authors in [8] combines ideas from semiring-based constraints (soft constrains [7]) and tccp [16] leading to Timed Soft Concurrent Constraint Programming (tscpp). In this calculus, action-prefixing is interpreted as the next-time operator and the parallel execution of agents follows the scheduling policy of maximal parallelism. Adopting soft constraints instead of crisp ones gives more flexibility in application where partial information and preferences are present. The authors show an application of this framework modeling scenarios of communicating-agents that need to negotiate a desired Quality of Service (QoS).

The work in [12] proposes the so-called cc-pi calculus combining the mechanism of name-passing in the π-calculus and the ideas of CCP. In this case, the formalization of the constraint system is also given in terms of semiring-based constraints [7]. The main application of cc-pi is to specify QoS requirements and to conclude Service Level Agreement (SLA) contracts. The proposed language is also equipped with mechanisms for resource allocation and for joining different SLA requirements. Similarly to Linear CCP [20], cc-pi exhibits non-monotonic behavior since it introduces a retract construct, whose effect is to erase a previously told constraint. The move to non-monotonic behavior in this case allows to model allocation and deallocation of the same resource.
6 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 different reasoning techniques this model offers. For instance, closure operator semantics for fragments of CCP calculi have been studied in [52, 50, 39, 41, 55, 15]. These semantics allow to retrieve the input-output behavior of a process compositionally which is a salient property when analyzing complex systems. On the other hand, the declarative view of processes as formulae in certain logic allows for reachability analysis using standard techniques in first-order (temporal) logic.

There are a lot of theoretical and practical work to model more complex systems in the scenarios described in this paper. For example, in music interaction is important to give true-concurrency semantics to tcc-based calculi (see e.g. [24, 37]). We are particularly interested in giving an event structure semantics to the ntcc calculus. It may help to the analysis of systems in which parallel actions cannot be model as interleaving, for instance, when considering musical notes that must be played simultaneously. From the practical point of view, real-time interpreters of these calculi are required to develop real-life application in music-improvisation settings.

Similar to the work in [42] and [41] to verify secrecy properties in security protocols, we believe that the reasoning techniques in CCP have much to offer for the verification of different properties such as authentication and anonymity. In the latter case, the notion of probability will be fundamental.

For the modeling and simulation of biological system, the development of efficient mechanisms for including ordinary differential equations (ODEs) in CCP-based languages is necessary. Such a system, in combination with the existing constraint systems over real intervals and finite domains, will allow to take advantage of the knowledge currently held by biologists about the structure and behavior of molecular systems. It may contribute to fill the gap that prevents computer scientists from straightforwardly using some well studied models of biological networks. This constraint system certainly will be helpful also for the modeling of physical systems.

Acknowledgements

We would like to thank to Jorge A. Perez and Eric Monfroy for their valuable feedback on earlier versions of this paper.
References