Vol. VIII, № 1, Mayo (2000)
Matemáticas: 1–??

Matemáticas:
Enseñanza Universitaria
©Escuela Regional de Matemáticas
Universidad del Valle - Colombia

Process Calculi to Analyze Emerging Applications in Concurrency

Alejandro Arbeláez
aarbelaez@puj.edu.co

Andrés Aristizábal
aaaristizabal@puj.edu.co

Julian Gutiérrez
J.E.Gutierrez@ed.ac.uk

Hugo A. López
lopez@dit.unitn.it

Jorge A. Pérez
perez@cs.unibo.it

Camilo Rueda
crueda@cic.puj.edu.co

Frank D. Valencia
frank.valencia@lix.polytechnique.fr

Abstract

The notion of *computation* has significantly evolved in the last ten years or so. Modern computing systems (e.g., Internet) now exhibit *infinite behavior*, usually in the context of decentralized networks where interactions are inherently *concurrent*. The ubiquitous presence of this new kind of systems has led to the urgent need of counting with techniques for designing them in a reliable way. *Process calculi* are formal specification languages of concurrent systems in which the notions of *process* and *interaction* prevail. They are endowed with reasoning techniques that allow to rigorously determine whether a system exhibits some desirable *properties*. The generic nature of process calculi has made possible their successful application in very diverse areas. Based on recent work by the authors, this paper illustrates the use of process calculi in two emerging application areas: biology and security protocols. Basic notions of process calculi are introduced, real systems in the two areas are modeled and their properties are verified.

Keywords: Computer Science, Concurrency Theory, Verification of Concurrent Systems, Process Calculi.

1 Introduction

Recent years have seen the impressive development of Internet and the increase in flexibility and power of communication networks. We now find ourselves in "global computing" environments, in which a significant evolution of the notion of *computation* can be recognized. There are two main issues that evidence this change. First of all, software artifacts are now meant to be *infinite*, in the sense that the applications they are part of demand an uninterrupted execution. For instance, in the context of operating systems and Web-based applications (such as online banking services), interruptions are a highly undesirable defect and, in some cases, can be also catastrophic. Secondly, since modern computer devices (e.g., portable computers, cell phones) now interact in the context of decentralized communication networks, the type of the behavior underlying them is inherently *concurrent*, rather than sequential. Popular online services for instant communication and file-sharing communities exhibit concurrent interactions among the involved users. Concurrency is also central to critical e-commerce applications.

These new characteristics of computation have a direct impact in the way software is conceived. In particular, the non-terminating and concurrent nature of modern software applications constitute a serious challenge in their verification. The traditional approach for software verification, focused on the study of the resulting outputs with respect to a given set of inputs, is clearly inadequate in the context of software pieces that are supposed to run almost forever. Moreover, traditional techniques also fail in providing comprehensive mechanisms for describing and reasoning about complex interactions commonly occurring in applications running in concurrent/mobile environments.

The above discussion suggests a paradigm shift in the goals of software verification techniques. The main goal of such techniques should be determining whether software components hold a set of desirable *properties*, instead of merely studying their input/output behavior. Some examples of interesting properties that a system should exhibit include:

- *correctness* properties, that ensure that software components do what they are supposed to do;
- *safety* properties, that guarantee that nothing bad occurs during software execution;
- *liveness* properties, that guarantee that something good happens because of software execution.

Independently of the exact definition of the desired properties (which will depend on the particular features of each system), it is clear that one prefers *general properties*, valid for the whole system, instead of particular properties that are valid only for certain scenarios. This observation suggests that the disciplined use of formal verification techniques (i.e., those based on mathematical foundations) is a reasonable research direction.

Concurrency theory is the branch of Computer Science that aims at providing foundational techniques to describe and reason about concurrent systems and their behavior. A particularly important class of such techniques is represented by *process calculi*: these are “small” languages provided with a few operators that are intended to capture the essential features of the systems of interest. Several *reasoning techniques* on concurrent systems have been developed on top of process calculi.

Although process calculi were originally conceived for the study of distributed, mobile communication systems, recent research reveals an increasing interest on analyzing phenomena in other fields that also exhibit concurrent behavior. In this way, for instance, process calculi have been used to describe and analyze systems in such diverse areas as computer music [**?**, **?**, **?**], data integration on the Web [**?**], Web services [**?**], biology [**?**, **?**, **?**, **?**, **?**] and secure communications [**?**, **?**, **?**, **?**]. Several works have shown how process calculi may provide new insights on the behavior of systems in such areas. This opens the way for the development of software tools that put into practice the reasoning techniques associated to process calculi; this could constitute an alternative approach for property verification in the areas of interest.

This paper constitutes an introduction to the modeling of systems using process calculi. Based on recent work by the authors, this paper illustrates the use of process calculi in two emerging applications in concurrency: *systems biology* and *computer security*. More specifically, we show how a process calculus based on the notion of *constraint* as an element of partial information turns out to be appropriate in modeling and analyzing biological systems. In a similar way, we show how a process calculi proposed for describing security protocols is suitable for the analysis of protocols for Peer-to-Peer (P2P) communication systems. The most relevant properties in each application are discussed and formalized, and the reasoning techniques associated to each calculus are presented. Representative examples of systems specifications are given.

The rest of this paper is structured as follows. Next, a general introduction to process calculi is provided. In Section **??**, the use of process calculi in Biology and Security applications is discussed. Section **??** concludes.

2 Process Calculi

Process calculi (also known as *process algebras*) can be defined as formalisms devised for the description and analysis of the behavior of *concurrent systems*; i.e., systems consisting of multiple computing agents (*processes*) that interact with each other. As such, the goal of a process calculus is to provide a *rigorous framework* where complex systems can be accurately analyzed, including *reasoning techniques* to verify their essential properties. In this section we discuss some basic principles on process calculi, including several issues that distinguish them from other formal models for concurrency and the main approaches to give meaning to processes. In addition, some verification tools derived from process calculi will be mentioned.

The nature and features of concurrent systems occurring in the real world makes it difficult the task of finding a canonical model in which *every* system can be accurately represented. In fact, even in the context of a restricted field (say, distributed systems) a wide variety of different phenomena, occurring at different levels, can be recognized. The goal is then to identify a set of common set of *underlying principles* in the systems of interest, and to define suitable operators that capture them in a precise way. In other words, a process of *abstraction* is required to define meaningful calculi in the simplest possible way.

Process calculi are then *abstract* specification languages for concurrent systems. This implies that models of systems abstract from real but unimportant details that do not contribute in essential system interactions. Abstraction not only allow designers to better understand the core of a system, but it also turns out to be necessary for an effective use of reasoning techniques associated to the calculus.

In addition, process calculi follow a *compositional* approach for systems description. This implies that a process calculus model of a system is given in terms of models representing its subsystems. This favors an appropriate abstraction of the main components of the systems and, more importantly, allows to explicitly reason about the *interactions* among the identified subsystems. As we will see later, each calculus assumes a particular abstraction criteria over systems, which will have influence on the level of compositionality models will exhibit.

Process calculi also pay special attention to *economy.* There are few process constructors, each one with a distinct and fundamental role in capturing the behavior of systems. A reduced number of constructors in the language helps to maintain the theory underlying the calculus tractable as well as stimulates a precise definition of the abstraction criteria that the calculus intends to express.

Let us illustrate the interplay of the above issues by introducing one of the most representative process calculus for mobility.

2.1 A Process Calculus for Mobile Systems

The π-calculus [**?**, **?**], was proposed by Milner, Parrow and Walker in the early 90's for the analysis of mobile, distributed systems. The ability of representing *link mobility* is one of the main advances of the π-calculus with respect CCS (Calculus for Communicating Systems) [**?**], its immediate predecessor. In the π-calculus, the description of mobile systems and their interactions is based on the notion of *name.* In principle, a process (an abstraction of a mobile agent) should be capable of evolving in many different ways, but always maintaining its identity during the whole computation. In addition, a process should be capable of identifying other related processes. In the π-calculus a name also denotes a *communication channel*, in such a way that communication among two processes is possible provided that they share the same channel. As a consequence, in the π-calculus a name abstracts the identity of processes in an interaction by considering the communication channel each process is related to.

In the π-calculus, process capabilities are abstracted as *atomic actions.* They come in two main flavors:

- $x(z)$, representing the *reception* (or reading) of the datum z on the channel x. z is then ready for any subsequent computations.
- $\overline{x}d$, denoting the *transmission* of a datum d over the channel x.

Actions (denoted by α) are used in the context of *processes* that are constructed by the following syntax:

$$P, Q, \ldots \ ::= \ \mathbf{0} \mid \sum_{i \in I} \alpha_i.P_i \mid P \parallel Q \mid !P \mid (\nu x)P \,.$$

Some intuitions underlying the behavior of these processes follow.

- Process $\mathbf{0}$ represents the process that does *nothing*. It is meant to be the basis of more complex processes.

- The *interaction* of processes P and Q is represented by their *parallel composition* $P \parallel Q$. In addition to the individual actions of each process, their communication is possible, provided that they *synchronize* on a channel, as illustrated in the following example.

$$R = x(y).\overline{y}z.\mathbf{0} \parallel \overline{x}w.\mathbf{0}$$

  Here, R represents the interaction of two processes sharing a channel x. The transmission of w through x is complemented by its reception, which involves recognizing w as y. This is regarded as an atomic computational step. Afterwards, a datum z is sent, using the received name w as communication channel. Notice that in the context of R, there is no partner for w in its attempt of transmitting z.

- $\sum_{i \in I} \alpha_i.P_i$, usually known as a *summation* process, represents a choice on the involved P_i's, depending on the capabilities represented by each α_i. Only when any such processes is ready to interact with another one, a *choice* among all the possible interaction options takes place. For instance, in the process

$$(x(y).\overline{z}y.\mathbf{0} + z(y).\mathbf{0} + x(w).w(z).\mathbf{0}) \parallel \overline{x}r.\mathbf{0}$$

  the first and third components of the sum are ready to interact with $\overline{x}r.\mathbf{0}$. Depending on the choice, different resulting processes are possible. For instance, if the third component is selected, the resulting interaction would lead to the process $r(z).\mathbf{0}$.

- Process $!P$ represents the *infinite execution* (or *replication*) of process P. There will be an infinite number of copies of P executing: $!P = P \parallel P \parallel P \parallel \ldots$.

- Process $\nu x P$ is meant to describe *restricted* names. Name x is said to be *local* to P and is only visible to it. A disciplined use of restricted names is crucial in delimiting communication.

The π-calculus is thus a language based in a few simple, yet powerful, abstractions. In addition to the above-mentioned abstraction of name as communication channels that can be transmitted, in the π-calculus the behavior of mobile systems is reduced to a few representative phenomena: synchronization on shared channels, infinite behavior and restricted communication. The compositional nature of the calculus is elegantly defined by the parallel composition operator, which is the basis for representing interactions among processes and the construction of models.

2.2 Key Issues in Process Calculi

There are many different process calculi in the literature, mainly agreeing in the formal flavor they encourage, as well as in the principles of abstraction, compositionality and economy discussed before. Main distinctions among these calculi arise from issues such as the process *constructs* considered (i.e., the process languages they define), the methods used to give *meaning* to process terms (i.e., semantics), and the methods to reason about process behavior (i.e., tools for property verification). In what follows we discuss some of these issues, following [**?**].

Process Constructs

The role of process constructs is to faithfully capture the intended phenomena each process calculus wants to reason about. As a consequence, each calculus will define different process constructs. Nevertheless, the following elements are commonly found in process calculi:

- *Action* constructs for representing atomic, basic actions.
- *Composition* constructs for expressing the parallel composition of processes
- *Summation* constructs for expressing the possibility of diverse courses of action in a computation
- *Restriction* constructs, for delimiting the interactions of processes.
- *Infinite behavior* constructs.

It is interesting to observe how calculi that are in principle quite different coincide in these basic issues for their definitions. As an example, consider process calculi based on Concurrent Constraint Programming (CCP) [**?**], a model for concurrency based on *partial information*. As opposed to the π-calculus, which defines a point-to-point communication discipline, CCP is a model of *shared-memory* communication. Intuitively, this implies that a process in a communication *broadcasts* messages to every other agent in the system. In spite of this important difference, process constructs in CCP-based process calculi are very similar to those defined in π: there are constructors for parallel composition, local behavior and infinite execution. Not surprisingly, the only significant differences arise in the constructs related to communication and synchronization: in CCP-based calculi there are operations for *increasing* the knowledge of the pool of agents and for *querying* the current knowledge those agents posses. `ntcc`, a CCP-based process calculus will be discussed in Section **??**.

Meaning of Processes

Endowing process terms with a formal meaning is crucial in order to analyze process behavior. There are at least three main approaches used to give meaning to process terms.

- *Operational Semantics* An operational semantics interprets a process term by using *transitions* that define computational steps. A common practice is to capture the state of the system by means of *configurations*, succinct structures that, in addition to a process term, may include other relevant information. Transitions are usually *labeled* by the actions that originate evolution between configurations. This is commonly denoted as $P \xrightarrow{a} Q$, meaning that process P performs action a and then behaves as process Q. Operational semantics are then defined by a set of *(reduction) rules* that formally define the features of the relation $\xrightarrow{a}$. The set of reduction rules that constitute the operational behavior of a calculus is also known as its *labeled transition system* (or *LTS*).

  As an example, consider the rule that formalizes the communication of interacting processes in the π-calculus, informally discussed in the previous section:
$$x(y).P \parallel \overline{x}z.Q \longrightarrow P\{z/y\} \parallel Q.$$
  In this (labeled) rule, $P\{z/y\}$ denotes the syntactic replacement of all occurrences of the name y with the name z in the context of process P.

- *Denotational Semantics* A denotational semantics interprets processes by using a function $[\![\cdot]\!]$ which maps them into a mathematical object (e.g., a set or a category). Definitions of denotational semantics usually involve the identification of relevant objects that can be *observed* from process behavior. A process is then equated to the set of observations that can be made of it. As an example, consider $[\![\cdot]\!]_{obs}$, a simple interpretation for the π-calculus that characterizes a process P by the set of names that are transmitted during its evolution. For instance, the denotation of the process R defined before is given by $[\![R]\!]_{obs} = \{w, z\}$.

  Interestingly, the compositionality principle in process calculi also appears in denotational semantics definitions, as the meaning of processes is determined from the meaning of its sub-processes. As an example, consider the denotation of a binary summation:
$$[\![P + Q]\!]_{obs} = [\![P]\!]_{obs} \cup [\![Q]\!]_{obs}.$$

- *Algebraic Semantics* Algebraic semantics attempt to give meaning by stating a set of laws (or axioms) equating process terms. Processes and their operations are then interpreted as structures that obey these laws. In its more basic use, the algebraic approach can be used to formally equate processes with minor syntactic differences. This gives rise to a relation known as *structural congruence*, that allows for cleaner rule definitions in the operational semantics. As an example of the kind of "equality" that can be characterized by means of algebraic semantics, in the π-calculus it is safe to consider the following set of axioms for parallel composition:
$$P \parallel \mathbf{0} \equiv P, \quad P \parallel Q \equiv Q \parallel P, \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R.$$

These axioms succinctly define that parallel composition is a commutative and associative operation, and that its module is the process **0**.

It is important to remark that a process language can have several semantic interpretations. In fact, the combination of two or more approaches is a common practice, since for instance, an approach can be more appropriate for intuitive understanding of processes whereas other can be more suitable for mathematical proofs. This is usually the case of Operational Semantics and Denotational/Algebraic ones. The use of several semantics motivates a legitimate question, that of determining whether different semantics are equivalent to each other.

Property Verification

As argued in the introduction, obtaining solid guarantees about the behavior of systems makes property verification the ultimate goal when formalizing systems using process calculi. Using process calculi, there are two main approaches for property verification. The first one pertains to *comparing* process calculi specifications. The idea consists in determining whether two process specifications are equivalent up to some notion of *behavioral equivalence*. In this way, for instance, a specification representing a system's implementation is compared against a specification that is assumed to have the desired property. If both specifications are regarded as equivalent, then there are solid elements to consider that the implementation specification holds the property in question.

Naturally, this approach relies on the power and features of the selected behavioral equivalence. To get an idea of such an equivalence consider *bisimilarity*. Roughly speaking, two processes P and Q are said to be *bisimilar* if whenever P performs an action a evolving into P', then Q can also perform a and evolve into a Q' that is bisimilar to P', and similarly with P and Q interchanged. It is worth noticing that a great deal of what can be said about the comparison approach for property verification relies on the *discriminating power* of behavioral equivalences. This implies that there are some equivalences more strict than others (that is, accept less processes as being equal), and that one should select the most appropriate one for analysis.

The second approach for property verification is commonly referred to as *model checking*, and advocates the use of suitable modal logics to verify properties of a given (process) specification. The key aspect here is to find a formal relationship between process terms and formulas of the logic. Several ways of obtaining such a formal relationship between process terms and logic formulas have been proposed in the literature; in the next section we will give an in-depth description of how this is done in the case of `ntcc`, which is equipped with a proof system of a linear temporal logic (or LTL).

2.3 Verification Tools based on Process Calculi

The theoretical development of process calculi has led to the development of software tools and programming languages that implement their most representative results. This is certainly an attractive option for verification of critical systems, as engineers count with solid frameworks where design flaws can be discovered in very early development phases. There is a variety of tools that take process calculi into practice; they all differ in the kind of specification languages that are supported (and that are usually very similar to process calculi specification), the kind of properties they can verify and the user interfaces they provide, among other features. Here we summarize the main features of some of these tools, namely the programming language Pict, the Edinburgh Concurrency Workbench (CWB) and the Concurrency Workbench of the New Century (CWB-NC).

Pict [?] is a concurrent programming language based in the π-calculus. It is a functional programming language with static typing, based on a core language that corresponds to an asynchronous variant of the π-calculus. Several intuitive additions to this core language are available, including basic data structures and concurrent objects.

The Edinburgh Concurrency Workbench (CWB) [?], is a popular automated tool for manipulation and analysis of concurrent systems. The base description languages for CWB are CCS and its synchronous variant SCCS. Using CWB it is possible to:

- perform analysis of semantic equivalences among specifications;
- define propositions in a modal logic and check whether a given process satisfies a specification formulated in this logic;
- derive automatically logical formulas which distinguish nonequivalent processes;
- interactively simulate an agent's behavior.

Finally, the Concurrency Workbench of the New Century (CWB-NC) [?] is a tool that offers several approaches for verification, providing support for different languages, including CCS and LOTOS [?]. The simplest of such approaches is *reachability analysis*: the tool is provided with a formal specification of the system and with a logic formula describing an undesirable state of it. The tool then explores every possible state of the system and checks if such an state can be reached. If so, the user is provided with an execution sequence leading to such an state. The second approach is somehow related and consists in performing a *model checking* process over a system description. This description is now analyzed with respect to a temporal logic formula that describes a property that the system should exhibit during its execution. Finally, a third technique available pertains to the above-mentioned *equivalence-based* approach for property verification. Efficient algorithms for equivalence checking and routines for performing these types of verification have been implemented. The tool also provides diagnostic information for explaining why two systems fail to be related by a given semantic equivalence.

3 Applications

In this section we illustrate the use of process calculi in Systems Biology and Security in Communication Protocols. In order to do so, a particular calculus for each application area is presented and described. An example extracted from real scenarios is modeled and verified. Related work in using process calculi in each area is also mentioned.

3.4 Systems Biology

The recent progress in *molecular biology* has allowed to describe the structure of many components making up biological systems (e.g., genes and proteins) as *isolated* entities. Instead of being alone, these entities are part of complex biological networks present at the cellular environment (such as genetic regulatory networks) whose aim is to define and regulate cellular processes. The current challenge is to move from molecular biology to *systems biology* [**?**], in order to understand how these individual components or entities *integrate* among them in the networks they shape. Once this integration has been understood, it will be then possible to discover how these entities perform their tasks.

The complexity and size of biological systems has motivated the use of computational models that allow to abstract their behavior and make their study easier. In this way, the inherent *concurrent* behavior of biological systems has encouraged the use of process calculi as a suitable description language. Reasons supporting this claim could be inferred from the features of calculi, already discussed. Next we review some works in which process calculi have been exploited in the biological context.

Most of the work using process calculi in biology have used formalisms such as the π-calculus and the Ambient calculus [**?**], or some extensions of them. Some representative pieces of work in such a direction are [?, ?, **?**]. In other cases, new process calculi have been proposed for modeling of more particular biological phenomena and systems. For instance, calculi to reason about interactions among membranes [**?**], interactions among proteins [**?**], reversibility in bio-molecular processes [**?**] and hybrid biological systems [**?**] can be found in the literature.

However, apart from [?] which uses *hybrid* concurrent constraint programming, none of the above calculi have tackled the problem of modeling biological systems of which only *partial information* about their behavior at *system level* is available. This is a significant group of systems, if we consider that lots of biological phenomena are still being discovered and/or investigated. For this reason, we have explored the use of CCP as a possible computational model for representing this kind of information in the biological context. In this way, our interest is centered in the study of process calculi based on *constraints*, identifying advantages in their use that would make the study of biological systems easier.

In particular, in [?] we have argued for the use of `ntcc`, a temporal CCP process calculus, for describing biological systems. `ntcc` comprises a variety of features for describing, simulating and reasoning about complex biological systems. Some of these features include: the natural use of *concurrent agents* (i.e., processes) for modeling biological entities, the explicit notion of *time* for describing the evolution of dynamic biological systems, *constraints* as a formal mechanism for representing partial information in the state of systems, *asynchronous and non-deterministic* operators for modeling partial information about the behavior of systems and the possibility of including *quantitative* information for parameterizing models with actual values coming from experimentation.

A crucial advantage is that this *theoretical framework* for studying biological systems can be implemented in a CCP programming language such as Mozart [**?**]. This allows to observe and analyze, at system level, the behavior of models proposed using the calculus.

Background

In this section we give a concise introduction to CCP and `ntcc`, the process calculus that we have used for describing biological systems. We emphasize on the opportunities and advantages of using `ntcc` in this context.

Let us first define some basic notions of CCP. CCP is a computational formalism for describing the behavior of concurrent systems. In CCP all the (partial) information is *monotonically* accumulated in a so-called *store*. The store keeps the knowledge about the system in terms of *constraints*, or statements defining the possible values a variable can take (e.g., $x + y \geq 7$). Concurrent agents (i.e., processes) that are part of the system interact with each other using the store as a shared communication medium. They have two basic capabilities over the store, represented by *tell* and *ask* operations. While the former *adds* a piece of information about the system, the latter *queries* the store to determine if some piece of information can be inferred from its current content.

One of the most distinguishing features of CCP is that it provides a *unified framework* where processes have a dual perspective: the traditional *operational* view of *process calculi* and a *declarative* one based on *logic*. This allows CCP to take advantage of techniques and tools from both process calculi and logic to *model* and *reason* about concurrent systems.

The `ntcc` process calculus is a *temporal* extension of CCP. The process constructs of the calculus naturally capture the main features of timed and reactive systems. In particular, `ntcc` allows to model:

- *unit-delays* to explicitly model pauses in system execution.
- *time-outs* to execute a process in the next time unit if in the current one a piece of information cannot be inferred from the store.
- *synchrony* to control and coordinate the concurrent execution of multiple systems.
- *infinite behavior* to represent the persistent execution of a system.
- *asynchrony* to represent unbounded but finite delays in the execution of a system.

- *non-determinism* to express the diverse execution alternatives for a system from the same initial conditions.

We now proceed to summarize the main features that ntcc has to offer to systems biology.

Theoretical Opportunities There are some conceptual features of ntcc that are specially important in the biological context.

- *Time in Modeling of Complex Dynamic Systems* In some biological scenarios it is important to know the initial and final states of a system. Nevertheless, in other situations it is mandatory to be able to analyze the evolution of the system in time. This is why having a description language with an explicit notion of time is fundamental for both achieving the control of the modeled systems and supervising their evolution step-by-step. ntcc is equipped with constructs that allow to explicitly control temporal features of modeled systems.

- *Partial information* As mentioned before, this kind of information arises naturally in the biological context since the structure and behavior of many biological phenomena are nowadays a matter of research. Two main kinds of partial information can be identified in biological systems: *quantitative* and *behavioral*. While *partial quantitative information* usually involves incomplete information on the *state* of the system (e.g., the set of possible values for a variable, the probability for a system to evolve to a certain future state), *partial behavioral information* is related to uncertainty associated to the *behavior* of interactions (e.g., the unknown relative speeds on which two systems interact, the time interval during which a medicine is effective).

  In ntcc, whereas partial quantitative information is represented with the aid of *constraints*, partial behavioral information is described by means of the *asynchronous and non-deterministic* process constructs in the calculus. The appropriate use of these elements for describing biological systems can build up, with a certain *abstraction* degree, rather complex biological patterns of behavior.

- *Verification of Biological Properties* As discussed before, perhaps the most important feature of process calculi is that their solid mathematical foundations allow to verify properties of the systems they model. In the case of ntcc such properties can be verified following a logic-based approach, expressing properties using a *linear-temporal logic (LTL)* and deriving proofs using a *proof system* associated with the calculus.

Practical Opportunities Complementary to these advantages, ntcc presents a useful feature in practical terms, as systems represented by means of the calculus can be easily implemented and their behavior observed using ntccSim [?], a simulation tool of ntcc processes. Indeed, ntccSim uses Mozart (a CCP-based programming language) to implement the rules of the operational

Figure 1: Discrete Reactive Computation in ntcc

semantics of ntcc. Since these rules formalize process behavior, in ntccSim it is possible to execute ntcc specifications so to observe and analyze systems described using the calculus.

A Process Calculus with Explicit Time and Constraints

ntcc is a temporal concurrent constraint calculus suitable to model non-deterministic and asynchronous behavior. As such, it is particularly appropriate to model *reactive systems*, those that compute by reacting to stimuli coming from its environment. As mentioned in the previous section, one of the main features of ntcc is that it is equipped with a *proof system* for verifying linear-temporal properties of ntcc processes. In this section we briefly describe the syntax, operational semantics and proof system of ntcc, referring the reader to [?, ?] for further details.

A fundamental notion in CCP-based calculi is that of a *constraint system*. Basically, a constraint system provides a signature from which syntactically denotable objects in the language called *constraints* can be constructed, and an entailment relation ($\models$) specifying interdependencies among such constraints. More precisely, a constraint system is a pair (Σ, Δ) where Σ is a signature of function and predicate symbols, and Δ is a decidable theory over Σ (i.e., a decidable set of sentences over Σ with at least one model). The underlying language $\mathcal{L}$ of (Σ, Δ) contains the symbols $\neg, \wedge, \Rightarrow, \exists, \mathtt{true}$ and $\mathtt{false}$ which denote logical negation, conjunction, implication, existential quantification, and the always true and always false predicates, respectively. *Constraints*, denoted by $c, d, \ldots$ are first-order formulas over $\mathcal{L}$. We say that c *entails* d in Δ, written $c \models_\Delta d$ (or just $c \models d$ when no confusion arises), if $c \Rightarrow d$ is true in all models of Δ. For operational reasons we shall require $\models$ to be decidable.

In ntcc time is divided into discrete *intervals* (or *time units*), each of them having its own *(constraint) store*. In this way, each time unit can be understood as a reactive entity, where a process P_i receives a stimuli e_i (i.e., a constraint) from the environment. The process P_i is then executed considering this input, responding with some output r_i (that is, new constraints) once no further processing over the store is possible. Computation in the next time unit is then based on a residual process resulting from P_i and on new inputs provided by the environment. Figure **??** illustrates this kind of computation for three time units.

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

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

Below we provide some intuitions regarding the behavior of `ntcc` processes in the biological context. For the sake of space, some formal details are elided from this presentation.

Adding and Querying (Partial) Information Process **tell**(c), the simplest operation to express *partial information*, adds a constraint c into the current store, thus making it available to other processes in the same time interval.

In the biological context, **tell** operations allow to represent at least two kinds of *partial information* statements: *ground rules* and *state definition* statements. The first ones precisely state certain conditions that apply during the life of the biological system. A clear advantage here w.r.t. other calculi for biology is that these conditions can be expressed by exploiting the available (possibly incomplete) knowledge.

Remarkably, the *declarative flavor* in this kind of statements could favor the definition of essential properties in (biological) models. Complementary to ground rules, *state definition* statements refers to those constraints intended to define the exact values for the variables in the system. This is particularly useful when one exactly knows the set of possible states of the system at a given time; series of such statements (for different time units) thus constitute a detailed view of the behavior of the system.

Complementary to **tell** operations, *guarded operations* of the form **when** c **do** P constitute the basic means for *querying* (or *asking*) information about the state of a system. Intuitively, a process **when** c **do** P queries the current constraint store: if c is present in such a store then the execution of P is enabled. The “presence” of c depends on the inference capabilities associated with the store. That is, a particular constraint could not be explicitly present in the store, but it could be inferred from the available information.

Non-deterministic Choices Non-determinism is a valuable way of representing several possible courses of action from the same initial state without providing any information on how one of such courses is selected.

In `ntcc`, non-deterministic behavior is obtained by generalizing processes of the form **when** c **do** P: a *guarded-choice summation* $\sum_{i \in I}$ **when** c_i **do** P_i, where I is a finite set of indexes, represents a process that, in the current time interval, must non-deterministically choose one of the P_j $(j \in I)$ whose corresponding constraint c_j is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible then the summation is precluded. We shall use **skip** to denote the process $\sum_{i \in I}$ **when** c_i **do** P_i when $I = \emptyset$.

In the biological context, the combination of guarded choices and partial information represent an appropriate mechanism to formalize the inherent *unpredictability* in system interactions. In this sense, non-determinism is a way of explicitly representing *partial behavioral information.*

Communication The concurrent execution of two processes P and Q is represented by the parallel composition $P \parallel Q$. We use $\prod_{i \in I} P_i$, where I is a finite set of indexes, to denote the parallel composition of all P_i.

Local Information Process **local** x **in** 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.

Basic Timed Constructs `ntcc` provides two basic time operators, **next** (P) and **unless** c **next** (P). Let us analyze them separately. **next** (P) represents the activation of P in the next time unit. Hence, a move of **next** (P) is a unit-delay of P. **next** (P) can be also considered as the simplest way of expressing the dynamical behavior over time. This is fundamental in `ntcc`, since information is *not automatically* transferred from one time interval to the next. We shall use $\mathbf{next}^n\ (P)$ as an abbreviation for **next** (**next** (… **next** (P)) …)), where **next** is repeated n times.

In the context of partial information, to be able to reason about *absence* of information is both important and necessary. Although sometimes it is possible to predict some of the possible future states for a system, usually there is a strong need of expressing *unexpected behavior.* In this kind of scenarios, processes of the form **unless** c **next** P may come in handy: P will be activated only if c cannot be inferred from the current store. The **unless** processes thus add (weak) time-outs in the execution, i.e., they wait one time unit for a piece of information c to be present and if it is not, they trigger activity in the next time unit.

Asynchrony The $\star$ operator allows to express asynchronous behavior through time intervals. Process $\star P$ represents an arbitrary long but finite delay in the activation of P. This kind of behavior therefore constitutes another instance of partial information: in addition to the partial information *on the variables* that are part of the state of the system (and that is expressed by the operators discussed above), the $\star$ operator allows to express partial information *on the time units* where processes are executed. This is particularly interesting when describing (biological) processes that interact at *unknown relative speeds.*

Persistent Behavior Somehow opposed to the eventual behavior enforced by asynchronous behavior, *persistent* (or infinite) behavior serves to express conditions that are valid in every possible state of the system. The replicated process $!P$ represents $P \parallel \mathbf{next}\ (P) \parallel \mathbf{next}^2(P) \parallel \ldots$, i.e. unboundedly many copies of P but one at a time. As such, persistent behavior is an appropriate way of enforcing conditions stating ground rules of the systems of interest.

Operational Semantics The intuitive behavior of `ntcc` processes described above is formalized by means of an operational semantics that considers transitions between process-store configurations of the form $\langle P, c\rangle$ with stores represented as constraints. The transitions of the semantics are given by the relations $\longrightarrow$ and $\Longrightarrow$. They are formally defined in [?]. Intuitively, an *internal* transition $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ should be read as "P with store d reduces, in one internal step, to P' with store d' ". The *observable transition* $P \xRightarrow{(c,d)} R$ should be read as "P on input c, reduces in one *time unit* to R and outputs d". The observable transitions are obtained from terminating sequences of internal transitions.

Let us now consider an infinite sequence of observable transitions (or *run*) $P = P_1 \xRightarrow{(s_1,r_1)} P_2 \xRightarrow{(s_2,r_2)} P_3 \xRightarrow{(s_3,r_3)} \ldots$. This sequence can be interpreted as an *interaction* between the system P and an environment. At a time unit i, the environment provides a stimulus s_i and P_i produces r_i as a response. If $\alpha = s_1.s_2.s_3\ldots$ and $\alpha' = r_1.r_2.r_3\ldots$, then the above interaction is represented as $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$.

Alternatively, if $\alpha = \mathtt{true}^{\omega}$, we can interpret the run as an interaction among the parallel components in P without the influence of an external environment (i.e.,each component is part of the environment of the others). In this case α is called the *empty* input sequence and α' is regarded as a *timed observation* of such an interaction in P. We will say that the *strongest postcondition* of a process P, denoted $sp(P)$, denotes the set of all infinite sequences that P can possibly output. More precisely, $sp(P) = \{\alpha' \mid \text{for some } \alpha : P \xRightarrow{(\alpha,\alpha')}{}^{\omega}\}$.

A Logic Approach for Property Verification The Linear-time Temporal Logic associated with `ntcc` is defined as follows. Formulas $A, B, \ldots \in \mathcal{A}$ are defined by the grammar:

$$A, B, \ldots := c \mid A \mathbin{\dot{\Rightarrow}} A \mid \dot{\neg} A \mid \dot{\exists}_x A \mid \circ A \mid \Box A \mid \Diamond A.$$

Here c denotes an arbitrary constraint which acts as an atomic proposition. Symbols $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}_x$ represent linear-temporal logic implication, negation and existential quantification. These symbols are not to be confused with the logic symbols $\Rightarrow$, $\neg$ and $\exists_x$ of the constraint system. Symbols $\circ$, $\Box$ and $\Diamond$ denote the linear-temporal operators *next*, *always* and *eventually*. We use $A \mathbin{\dot{\vee}} B$ as an abbreviation of $\dot{\neg} A \mathbin{\dot{\Rightarrow}} B$ and $A \mathbin{\dot{\wedge}} B$ as an abbreviation of $\dot{\neg}(\dot{\neg} A \mathbin{\dot{\vee}} \dot{\neg} B)$. The standard interpretation structures of linear temporal logic are infinite sequences of states. In `ntcc`, states are represented with constraints, thus we consider as interpretations the elements of $\mathcal{C}^{\omega}$. When $\alpha \in \mathcal{C}^{\omega}$ is a model of A, we write $\alpha \models A$.

We shall say that P satisfies A if every infinite sequence that P can possibly output satisfies the property expressed by A. A relatively complete proof system for assertions $P \vdash A$, whose intended meaning is that P satisfies A, is given in Table **??**. We shall write $P \vdash A$ if there is a derivation of $P \vdash A$ in this system.

Example: Modeling a biological mutation using `ntcc`

This section illustrates the most important `ntcc` features for describing biological systems. We shall present the use of ntccSim to observe the behavior of an `ntcc` model of the system. Finally, a property of such a model will be verified.

In this example, we are interested in modeling the control system of a *biological network*, including a set of genes. To do so we define three `ntcc` processes: $StartControl$, $MutatedGene$ and $WildGene$. The first process indicates the number of molecules interacting with the control region at the start of the

LTELL	$\mathbf{tell}(c) \vdash c$	LSUM	$\dfrac{\forall i \in I \quad P_i \vdash A_i}{\sum_{i \in I} \mathbf{when}\ \ c_i\ \ \mathbf{do}\ \ P_i \vdash \dot{\bigvee}_{i \in I}(c_i \mathbin{\dot\wedge} A_i) \mathbin{\dot\vee} \dot{\bigwedge}_{i \in I} \dot\neg c_i}$		
LPAR	$\dfrac{P \vdash A \quad Q \vdash B}{P \parallel Q \vdash A \mathbin{\dot\wedge} B}$	LUNL	$\dfrac{P \vdash A}{\mathbf{unless}\ \ c\ \ \mathbf{next}\ \ P \vdash c \mathbin{\dot\vee} \bigcirc A}$		
LREP	$\dfrac{P \vdash A}{!P \vdash \Box A}$	LLOC	$\dfrac{P \vdash A}{\mathbf{local}\ x\ \mathbf{in}\ P \vdash \dot\exists_x\, A}$		
LSTAR	$\dfrac{P \vdash A}{\star P \vdash \Diamond A}$	LNEXT	$\dfrac{P \vdash A}{\mathbf{next}\ (P) \vdash \bigcirc A}$	LCONS	$\dfrac{P \vdash A}{P \vdash B}$ if $A \Rightarrow B$

Table 1: A proof system for (linear-temporal) properties of `ntcc` processes

Figure 2: Molecular concentration in a DNA region of a mutated gene

study of the system. The second one defines the system behavior under mutated conditions. The last one represents the system behavior in wild or normal conditions. These processes can be formalized in `ntcc` as follows:

$$\begin{array}{lcl}
\mathit{StartControl} & \stackrel{\text{def}}{=} & \mathbf{tell}(x = n) \\
\mathit{MutatedGene} & \stackrel{\text{def}}{=} & \star\ !\ (\mathbf{tell}(\mathit{mut} = 1) \parallel \mathbf{next}\ (\mathbf{tell}(x = f_m))) \\
\mathit{WildGene} & \stackrel{\text{def}}{=} & !\ \mathbf{unless}\ \ \mathit{mut} = 1\ \ \mathbf{next}\ \ \ \mathbf{tell}(x = f_w) \\
\mathit{ControlRegion} & \stackrel{\text{def}}{=} & \mathit{StartControl} \parallel \mathit{MutatedGene} \parallel \mathit{WildGene}
\end{array}$$

where x is a variable representing the cellular concentration of molecules interacting with the control region of the set of genes, and n a real number as a starting value.

On the one hand, the process *MutatedGene* establishes that a mutation will eventually occur in the gene in an undetermined future time unit and, as a consequence, the behavior of the system will be defined by the constraint $x = f_m$, where f_m is a function determining an incorrect behavior in the gene control region. On the other hand, the process *WildGene* states that the behavior of the control region is represented by the constraint $x = f_w$ unless the mutation occurs (i.e., which is represented by the constraint $\mathit{mut} = 1$). Function f_w represents the behavior of the system in wild conditions. Figure **??** illustrates the behavior of the system; it was obtained using ntccSim, with $n = 0$.

A logic-based approach for verifying biological properties In this section we will verify a system property using the inference system associated with `ntcc`. As a case of study, we will verify that when the mutation occur, variable x will be determined only by function f_m. Formally, we wish to verify the following formula:

$$\mathit{ControlRegion} \vdash \Diamond\Box x = f_m$$

The formulas for processes $StartControl$, $MutatedGene$ and $WildGene$ are:

$$\begin{array}{lcl} StartControl & \vdash & x = n \\ MutatedGene & \vdash & \Diamond\Box(mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \\ WildGene & \vdash & \Box(mut = 1 \mathbin{\dot\vee} \circ x = f_w) \end{array}$$

$$\dfrac{\dfrac{}{StartControl \vdash x = n}\;\mathsf{LTELL} \qquad \dfrac{}{MutatedGene \vdash \Diamond\Box(mut = 1 \mathbin{\dot\wedge} \circ x = f_m)}\;\mathsf{LRULES1}}{StartControl \parallel MutatedGene \vdash \; (\; x = n \;) \mathbin{\dot\wedge} (\; \Diamond\Box(mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \;)}\;\mathsf{LPAR}$$

where LRULES1 denotes the systematic application of rules LSTAR, LREP, LPAR, LNEXT and LTELL of the proof system over process $MutatedGene$. For the sake of space, we assume the following abbreviations: $WG = WildGene$, $SC = StartControl$ and $MG = MutatedGene$.

$$\dfrac{\dfrac{}{WG \vdash \; (\; \Box(mut = 1 \mathbin{\dot\vee} \circ x = f_w) \;)}\;\mathsf{LRULES2} \qquad \dfrac{}{SC \parallel MG \vdash \; (\; x = n \;) \mathbin{\dot\wedge} (\; \Diamond\Box(mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \;))}}{WG \parallel SC \parallel MG \vdash \; (\; \Box(mut = 1 \mathbin{\dot\vee} \circ x = f_w) \;) \mathbin{\dot\wedge} (\; x = n \;) \mathbin{\dot\wedge} (\; \Diamond\Box(mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \;)}\;\mathsf{LPAR}$$

where LRULES2 represents the application of rules LREP, LUNL and LTELL over process $WildGene$. Finally, we can perform the following deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{ControlRegion \vdash \; (\; \Box\;(mut = 1 \mathbin{\dot\vee} \circ x = f_w) \;) \mathbin{\dot\wedge} (\; x = n \;) \mathbin{\dot\wedge} (\; \Diamond\Box\;(mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \;)}}{ControlRegion \vdash \Box\;(mut = 1 \mathbin{\dot\vee} \circ x = f_w) \mathbin{\dot\wedge} \Diamond\Box\;(mut = 1 \mathbin{\dot\wedge} \circ x = f_m)}\;\mathsf{LCONS}}{ControlRegion \vdash \Diamond\Box\;(\;\; (mut = 1 \mathbin{\dot\vee} \circ x = f_w) \mathbin{\dot\wedge} (mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \;)}\;\mathsf{LCONS}}{ControlRegion \vdash \Diamond\Box\;(\; mut = 1 \mathbin{\dot\wedge} (mut = 1 \mathbin{\dot\wedge} \circ x = f_m) \;)}\;\mathsf{LCONS}}{ControlRegion \vdash \Diamond\Box\circ x = f_m}\;\mathsf{LCONS}}{ControlRegion \vdash \Diamond\Box x = f_m}\;\mathsf{LCONS}$$

The above logical expression verify that the constraint $x = f_m$ will define the behavior of the system in an undetermined future time, and that this behavior will continue forever.

In this way, we have shown how the behavior of a system can be formally analyzed in two ways: (i) following the `ntcc` operational semantics in a mechanical way by using ntccSim (see Figure **??**) and (ii) by means of a logical-temporal proof derived with the inference system associated with `ntcc`. Concerning logic-based proofs, a remarkable aspect to consider here is that certain aspects of systems might be difficult to study by just using simulations; in our example, it is possible that simulations do not reveal the presence of a mutation, as it could occur in a very long time. As a consequence, in this case the logic proof can be regarded as being more effective, as it can reveal the actual behavior of the system.

It is worth mentioning other works related to the use of `ntcc` and CCP for the study of biological systems. In [**?**] a study of the state of the art in the modeling of biological systems using process calculi is presented. The advantages of using CCP in biology are analyzed there. The paper [?] offers a detailed explanation of how `ntcc` process constructs can be used to model biological systems. A biological system for ion transport is also modeled and verified there. In [?, ?] a complete `ntcc` model of the lactose operon genetic regulatory network is proposed. Similarly to the example discussed here, a stability property that cannot be ensured by simulation is formally verified using the proof system associated with the calculus. Finally, [?] presents a summary of the work on `ntcc` in systems biology.

3.5 Security in Communication Protocols

The *security* of information has always been one of the main concerns in social behavior. The assurance of a personal secret which cannot be revealed to someone unauthorized, and the notion of trust have been relevant concerns since the beginnings of commerce and wars. The emergence of global communications, electronic processing, and distributed computation have increased the relevance of these concerns. Recent data from the Internet Fraud Crime report [?] is just but one example of the strong influence secure communications have in business: about 228.400 complaints (with quantitative losses of US$183,14 Millions) were reported to be related with threats including electronic fraud, identity theft and even hacking.

A wide variety of (automated) tools have been developed to overcome security risks, including firewalls, access control mechanisms and cryptographic-based software. Nevertheless, these mechanisms by themselves are not enough to provide security warranties; the open nature of the communications and the inherent vulnerabilities of distributed systems make it essential to provide higher levels of assurance for participants of privacy-sensitive communication processes. This is why *security protocols* were created: these are sets of routines that define a precise set of steps that participants (also known as principals, agents or parties) have to follow in order to establish some security goals during communications.

There exist diverse kinds of interesting properties related to the behavior of security protocols (*security properties*), and usually aim at achieving different objectives [?]. Some representative examples include:

- *secrecy*, or the guarantee that a secret message never appears unprotected on the medium;
- *authentication*, or the guarantee that no principal is impersonated by an unknown or malicious agent;
- *anonymity*, or the guarantee that the responsibles of actions cannot be identified.

In this context, the use of process calculi for the analysis of security protocols appears as a promising approach. Several facts support this claim. First, the open nature of network scenarios is naturally captured by process calculi models, allowing for the inclusion of malicious attackers inside the environment. Second, the abstraction principle of process calculi helps engineers and designers to focus in the communication components of the protocol, leaving aside unimportant implementation details. In addition, the compositional approach of process calculi specifications allows to accurately describe network agents, their capabilities and complex interactions. Finally, a process calculi approach for the modeling of protocols would allow to (automatically) verify their security properties by means of the reasoning techniques associated with the calculi.

The above intuitions have been widely studied in the literature. Below we provide a summary of the most representative efforts in this direction.

One of the first attempts involves the use of the CSP process algebra [**?**]. In CSP, systems of concurrent agents interact via message exchange. It is intended to be a multipurpose algebra: several specialized theories could be constructed on the top of its semantic model. In this way, concrete formalisms can be designed and verified using this theory, with an environment especially crafted for each purpose. Several approaches for analyzing security properties in protocols [**?**, **?**] have been developed. In such works, network models and attacker abilities are abstracted as processes, whereas security properties are defined as predicates over the execution traces of such processes.

Another significant approach is the one that uses the π-calculus as specification language. Here protocol participants are abstracted as concurrent processes that exchange messages through channels. Remarkably, secret generation is abstracted by means of restricted names. In this way, for instance, process $(\nu s)(A \parallel B)$ represents a secret s that is shared by interacting agents A and B. Although this approach is clearly incomplete for modeling purposes in the security context, it served as inspiration for several extensions to the calculus. In fact, the Spi calculus [?] —one of such extensions— allows for the expression of cryptographic operations, using the behavioral equivalence approach for property verification. Other extensions include the *applied* π-calculus [**?**], a variant where operations over functions and data-types are treated in a general framework, thus enabling the use of complex cryptographic functions within protocol specifications. Recent works ([?]) involve the combination of a version of this applied π-calculus, type systems and logic interpretations of processes to analyze security protocols from a logic perspective: formal specifications of protocols are translated into logic clauses to perform reachability analysis about their properties.

Other process calculi have adopted an operational view of processes by using different foundations such as Petri Nets. One of such calculi is the Security Protocol Language (SPL) [?]. Next we provide an in-depth introduction to it.

A Process Calculus for Security Protocols

SPL is a process calculus that models security protocols as the asynchronous exchange of messages between agents. It is based on a *persistent* network model, in which every transmitted message is remembered for an unlimited period of time; this represents the power of an attacker to infinitely collect information

Variables		
	$v ::= x \mid Y$	*(names)*
	$k ::= Pub(v) \mid Priv(v) \mid Key(\vec{v})$	*(keys)*
	$M ::= v \mid k \mid (M, M') \mid \{\!\vert M \vert\!\}_k$	*(messages)*
Processes		
(Binding)	$\mathbf{P} ::= \mathbf{out\,new}(\vec{x})\, M.\mathbf{P}$	*(secret generation)*
	$\mid \mathbf{in\,pat}\, \vec{x}\vec{\chi}\vec{\psi}\, M.\mathbf{P}$	*(pattern-matching input)*
(Replication)	$\mid \; \Vert_{i \in I}\, \mathbf{P_i}$	*(process composition)*
	$\mid \; !\mathbf{P}$	*(infinite behavior)*

Table 2: Syntax of SPL

from the network. SPL provides an *event-based* operational semantics, which allows to represent protocol evolutions in a clear and intuitive way, as well as an intuitive set of *proof techniques*, that allow to verify security properties by taking advantage of the semantics. SPL has been successfully used in the analysis of several communication protocols (see, e.g., [**?**, ?, **?**]). Next we provide a formal introduction to the SPL process calculus, following [**?**, ?].

Syntax Let us start by giving the syntactic sets of the calculus:

- An infinite set N of *names*, denoted by $\vec{x}$. Names range over newly-generated values (known as *nonces*) and agent identifiers. It is common to denote agent identifiers with capital letters.
- *Variables* over names ($\vec{x}$), keys ($\vec{\chi}$) and messages ($\vec{\psi}$).
- A set of *processes* $P, Q, R, \ldots$.

In addition to these elements, it is possible to give a complete notion of messages by defining message *tuples* (denoted (M_1, M_2)), messages *ciphered* with a key (denoted $\{\!\vert M \vert\!\}_x$, x being the key), and messages involving cryptographic primitives for handling public and private keys (denoted $Pub(v), Priv(v)$ and $Key(\vec{v})$, respectively).

SPL processes can be grouped in two types: *replication* constructions that allow for composition of processes and *binding* processes that restrict variables in different contexts. While the former group is defined in the expected way, in the latter the output process

$$\mathbf{out\,new}(\vec{x})\, M.\mathbf{P}$$

binds the vector $\vec{x}$ to a set of fresh values $\vec{n}$, outputting the message $M[\vec{n}/\vec{x}]$ to the network while defining the evolution of $\mathbf{P}$ into $\mathbf{P}[\vec{n}/\vec{x}]$[1]. In addition, the process

$$\mathbf{in\,pat}\, \vec{x}\vec{\chi}\vec{\psi}\, M.\mathbf{P}$$

[1]Notice that $P[\vec{n}/\vec{x}]$ has an analogous meaning to the $P\{x/z\}$ notation explained before: all the occurrences of the message $\vec{x}$ in the context of process P are replaced by $\vec{n}$.

acts as a *pattern-matching input*, receiving every message from the network that match the pattern M, binding the new variables $\vec{x}, \vec{\chi}, \vec{\psi}$ with the received contents. Table **??** summarizes these syntactic elements.

Operational Semantics One of the main characteristics of SPL is the inclusion of a dual operational semantics to analyze the evolution of a process during its execution. A labeled transition system for SPL is defined over a set of configurations of the form $\langle p, s, t \rangle$, where p is a closed (i.e., variable-free) process term, s a subset of names, and t a subset of variable-free messages (i.e., the messages available in the network). The reduction rules allow to analyze how the information is included and how process are affected by such an inclusion. They are defined as follows:

- The *output* rule, labeled by $\mathbf{out}\,\mathbf{new}(\vec{n})\ M[\vec{n}/\vec{x}]$, defines the evolution of a process $\mathbf{out}\,\mathbf{new}(x)\ M.\mathbf{P}$ as the process $\mathbf{P}[\vec{n}/\vec{x}]$, including the information in the fresh variables $\vec{n}$ in the set of names and augmenting the set of messages with the information transmitted in M.

- The *input* rule is labeled by $in\ M[\vec{n}/\vec{x}, \vec{k}/\vec{\chi}, \vec{N}/\vec{\psi}]$ and allows the evolution of a process $\mathbf{in}\,\mathbf{pat}\ \vec{x}\vec{\chi}\vec{\psi}\ M.\mathbf{P}$ as the process $\mathbf{P}$, with the new variables instantiated from the contents of M.

- The *parallel composition* rule (labeled by $j : \alpha$, being j the index of one of the processes involved in the composition) decomposes an indexed composition process $\|_{i \in I} P_i$ into each of its subprocesses for further analysis.

Although this transition semantics is an appropriate method of representing process behavior, it is not fine enough to describe *dependencies* between events, or to support typical proof techniques such as those based on maintaining invariants along the trace of the execution of protocols. For this reason, SPL provides an additional semantics based in events that makes protocol events and their dependencies more explicit.

SPL event-based semantics relies on a variant of persistent Petri nets, so-called *SPL-nets* [?], which define events in the way they affect conditions, and how the models stated *evolve* using a general methodology (so-called *token games*). The interested reader may find full details about Petri Nets and persistent SPL-nets in [**?**] and [?], respectively. With the foundations of SPL-nets, every action present in a protocol can be described as a transition that generates a set of typed conditions (Petri net events). These events can be grouped in three different kinds (see Figure **??**):

- *Control conditions (^{c}e and e^{c})*, representing the current state of execution of a given process.

- *Name conditions (^{n}e and e^{n})*, that denote the names generated along the network.

□

Figure 3: Events and transitions of SPL event based semantics: p_i and q_i denote control conditions, n_i and m_i name conditions and N_i, M_i output conditions.

- *Output conditions (^oe and e^o)*, the pieces of information sent through the network. An important characteristic is the natural persistence of these kind of events, which models the fact that information always can be accessed form the network once it is published.

Notice that all the actions that can be performed using the transition semantics can be related to transitions in the event-based semantics with three typical actions: output, input and parallel composition transitions.

To illustrate the elements of the event semantics, consider a simple output event $e = (\mathbf{Out}(\mathbf{out}\,\mathbf{new}(\vec{x})\,M); \vec{n})$, where $\vec{n} = n_1 \ldots n_t$ are the distinct names to match with variables $\vec{x} = x_1 \ldots x_t$. The action $act(e)$ corresponding to this event is the output action $\mathbf{out}\,\mathbf{new}\,\vec{n}\,M[\vec{n}/\vec{x}]$. Conditions associated to this event are:

$$\begin{array}{lll} {}^ce = \langle \mathbf{out}\,\mathbf{new}(\vec{x})\,M.p, a\rangle & {}^oe = \emptyset & {}^ne = \emptyset \\ e^c = \langle Ic(p[\vec{n}/\vec{x}])\rangle & e^o = \{M[\vec{n}/\vec{x}]\} & e^n = \{n_1, \ldots n_t\}, \end{array}$$

where ${}^*e$ and e^* represent the pre and postconditions of the event e, respectively. $Ic(p)$ stands for the initial control conditions of a closed process; it is defined inductively as $Ic(X) = \{X\}$ if X is an input or an output process, and as $Ic(\|_{i\in I} P_i) = \bigcup_{i\in I}\{i : c \mid c \in Ic(P_i)\}$ otherwise.

Proving Security Properties in SPL Proofs of security properties in SPL follow a general methodology. Once a protocol has been modeled, an attacker capable of altering its correct execution is included in the obtained model. This attacker relies on the Dolev-Yao threat model [**?**], in which the attacker is a malicious entity capable to overhear, intercept, introduce and synthesize messages over the network. An SPL representation of a powerful attacker can be seen in Table **??**.

After that, a property is defined using a general set of *proof principles*, which state a general set of theorems available for the verification of security properties. Counting with these general principles is one of the advantages SPL has over other process calculi in terms of property verification. Some of the available proof principles are the following (see [?] for a formal definition):

- *Well-Foundedness:* If a property P holds in the initial condition i and then is violated at a stage t, then there is an event that breaks the property between i and t.
- *Freshness:* Only one copy of a name is generated in a run of the protocol.
- *Control Precedence:* There exists a casual dependency between control conditions.

Attacker Capability	*SPL formalization*
Compose messages	$Spy_1 \equiv \textbf{in}\, \psi_1 . \textbf{in}\, \psi_2 . \textbf{out}\, \psi_1 , \psi_2$
Decompose a message in sub-components	$Spy_2 \equiv \textbf{in}\, \psi_1 , \psi_2 . \textbf{out}\, \psi_1 . \textbf{out}\, \psi_2$
Encrypt any message with available keys	$Spy_3 \equiv \textbf{in}\, x . \textbf{in}\, \psi . \textbf{out}\, \{\!\vert\, \psi \,\vert\!\}_{Pub(x)}$ $Spy_4 \equiv \textbf{in}\, Key(x,y) . \textbf{in}\, \psi . \textbf{out}\, \{\!\vert\, \psi \,\vert\!\}_{Key(x,y)}$
Decrypt messages with available keys	$Spy_5 \equiv \textbf{in}\, Priv(x) . \textbf{in}\, \{\!\vert\, \psi \,\vert\!\}_{Pub(x)} . \textbf{out}\, \psi$ $Spy_6 \equiv \textbf{in}\, Key(x,y) . \textbf{in}\, \{\!\vert\, \psi \,\vert\!\}_{Key(x,y)} . \textbf{out}\, \psi$
Sign messages with available keys	$Spy_7 \equiv \textbf{in}\, Priv(x) . \textbf{in}\, \psi . \textbf{out}\, \{\!\vert\, \psi \,\vert\!\}_{Priv(x)}$
Verify signatures with available keys	$Spy_8 \equiv \textbf{in}\, x . \textbf{in}\, \{\!\vert\, \psi \,\vert\!\}_{Priv(\times)} . \textbf{out}\, \psi$
Create new random values	$Spy_9 \equiv \textbf{out new}(\vec{n})\, \vec{n}$

Table 3: SPL spy model

- *Input/Output Precedence:* There exists a casual dependency between output conditions.

- *Message Surroundings:* There exists a relation between messages and submessages at a given level of encryption.

The final step in the proof consists in deriving a contradiction. First, an event in which the property does not hold is assumed. For instance, if we wish to proof secrecy of a message, we assume that it is accessed by an attacker. Then, using the operational semantics and following event dependencies, one tries to find an event that breaks the property. If such an event cannot be reached, then it can be concluded that the protocol holds the desired property.

Example: Modeling and Verifying P2P Systems using SPL

P2P architectures are one of the most revolutionary changes in communication networks. They allow for collaboration between two different entities no matter the barriers between them. In P2P systems, every node is connected to the network in a decentralized manner, acting simultaneously as a requester, forwarder, and source of information. This feature has broaden the applicability of these systems, constituting a novel approach for distributed computations. P2P have become a major force in nowadays computing world because of its benefits, such as architecture cost, scalability, viability, and resource aggregation of distributed management infrastructures. These benefits have been extensively exploited in multiple application areas, such as information retrieval, routing and content discovery, providing a huge set of tools based on P2P architectures (see, e.g., [**?**, **?**, **?**, **?**, **?**]). However, these architectures have to face several security risks: the open nature of the network brings the opportunity to attackers to inflict damage in the protocol, by, for instance, reading privacy-sensitive information in transmit or writing fake information that could affect the correct execution of the protocol. Also, the multiplicity of the roles present in each agent implies that the communication must ensure the same level of security at each phase, no matter who receive the information.

Recently, we have shown that SPL provides a useful approach to analyze these kind of systems [?]. The compositionality of the calculus allows to model complex scenarios where attackers can infringe damage in each of the phases of the protocol. Also, the persistent network model turns to be useful to express the possibility attackers have of listening for a message over an infinite amount of time. Further, process replication allows to model infinite scenarios, stressing the level of complexity in security.

We shall illustrate this by modeling MUTE, a P2P protocol for content-sharing networks. Application examples of this kind of networks include online communities such as Kazaa and those dedicated to music distribution on the Web. The MUTE protocol intends to provide reliable communication for the agents involved in the network (so-called *peers*), working as a tool to communicate requests of keywords through the network, so that an specific file can be found and then received [**?**]. This protocol is composed of two main phases: searching and routing parts. We will focus directly in its searching phase, due to its significance in terms of security: if the information of the search (which agent is requesting, and who has the correct information) is safeguarded, then the routing phase will guarantee that only the correct principals will know the information. Next we present a formalization of MUTE, followed by a proof of secrecy over shared keys along the protocol run.

Abstracting P2P systems Let us give some definitions for P2P systems. They will serve to abstract common behaviors and objects in such systems:

Definition 1 (Sets in Mute)**.** *Let $Files$ be the set of all the files in the P2P network and $Files(A)$ the set of files belonging to peer A. Let $Keywords$ be the set of keywords associated to the files $Files$, $Keywords(A)$ the keywords associated to the peer A and $Keys$ the relation $Files : Keywords$, representing the keywords associated to a particular file. Let $Headers$ be the set of headers of files, which is associated to $Files$, $Headers(A)$ the set directly related to $Files(A)$, such that each header which belongs to $Headers(A)$ will be associated to a unique file belonging to $Files(A)$.*

The following definition will provide a formal basis to define and reason about P2P networks:

Definition 2 (P2P Network model)**.** *We shall describe a P2P network as an undirected graph G whose nodes represent the peers and whose edges mean the direct connections among them. We use $Peers(G)$ to denote the set of all the nodes in G. Given a node $X \in Peers(G)$, Let $ngb(X)$ be the set of immediate neighbors of X.*

For example, consider a P2P network G with $A, B \in Peers(G)$. Suppose that A initiates the protocol by broadcasting a request to all of its neighbors in order to find a particular answer, and B is the agent which has the desired answer that A is searching for, deciding to send a response. In this case, B can be any node inside the network with the desired file on its store. A requests for a particular file it wishes to download, sending the request to the network

$$\begin{array}{lll}
\mathbf{Init(A)} & \equiv & (\|_{B \,\in\, ngb(A)}\ \mathbf{out}\,\mathbf{new}(n)\,(\{\!|\, n, Kw \,|\!\}_{Key(A,B)}, A, B). \\
 & & (\|_{Y \,\in\, ngb(A)}\ \mathbf{in}\,(\{\!|\, n, res, m \,|\!\}_{Key(Y,A)}, Y, A)) \\
 & & \\
\mathbf{Interm(A)} & \equiv & !(\|_{Y \,\in\, ngb(A)}\ \mathbf{in}\,(\{\!|\, M \,|\!\}_{Key(Y,A)}, Y, A)\,. \\
 & & \|_{B \,\in\, ngb(A)-\{Y\}}\ \ o(\{\!|\, M \,|\!\}_{Key(A,B)}, A, B)) \\
 & & \\
\mathbf{Resp(A)} & \equiv & \|_{Y \,\in\, ngb(A)\,,\, kw \,\in\, Keys(Files(A))}\ \mathbf{in}\,(\{\!|\, x, Kw \,|\!\}_{Key(Y,A)}, Y, A)\,. \\
 & & (\|_{B \,\in\, ngb(A)}\mathbf{out}\,\mathbf{new}(m)\,(\{\!|\, x, res, m \,|\!\}_{Key(A,B)}, A, B)) \\
 & & \\
\mathbf{Node(A)} & \equiv & \mathbf{Init(A)}\ \|\ \mathbf{Interm(A)}\ \|\ \mathbf{Resp(A)} \\
\mathbf{SecureMUTE} & \equiv & \|_{A \,\in\, Peers(G)}\mathbf{Node(A)}
\end{array}$$

Figure 4: MUTE specification on SPL

by broadcasting it to its neighbors. This request includes a keyword $kw \in Keywords$, which will match the desired file, and a nonce N which will act as the request identifier. Along the searching path an unknown amount of peers will forward the request until B is reached, the peer with the correct file such that $\exists f \in Files(B)$ and $kw \in Keys(f)$. Then, B sends its response by means of the header of the file RES, among with the identifier N and a new name M generated by it to recognize the message as an answer. This is done again by broadcasting the message through a series of forward steps, until reaching the actual sender A.

The following specification shows how a protocol for P2P systems can be constructed. Considering only the searching protocol of MUTE, the phases considered are the ones that involve the transmission of the keyword, the response message and the keys (leaving aside the phases of connection), and the sub-messages that include plaintext. We assume that the symmetric keys are equivalent (i.e., $key(A,B) = key(B,A)$). A formal model of this scenario is presented in Figure **??**[2].

Some intuitions behind the model follow. It is assumed that the topology of the net has already been established. The agent starts searching for an own keyword. Then this agent broadcasts the desired keyword to all of its neighbors. They receive the message and check whether the keyword matches one of their files. When at least one of the neighbors find the requested keyword, it will broadcast a response message, in such a way that eventually the one searching for the keyword will get it and understand it as an answer to its request. The message will be forwarded by all the agents until it reaches its destination. In the case that the keyword does not match any file of the agent, it will broadcast it to its neighbors asking them for the same keyword. The choice of having or not the right file is modeled in a non-deterministic way. Notice that the model abstracts away from performance issues such as, e.g., the search for the best path, concentrating the analysis only in the satisfaction of a secrecy property.

[2]Notation $(\|_{i\in I}\ \mathbf{P_i}).\mathbf{R}$, representing the execution of process $\mathbf{R}$ once parallel composition $\|_{i\in I}\ \mathbf{P_i}$ is fully executed, can be easily encoded using elements in the language. See [**?**] for more details.

Verifying a Typical Security Property: Secrecy Considering the guidelines stated before, the complete model must consider the spy presented in Table **??**. The final model could be seen as

$$\mathbf{MUTE} \equiv \mathbf{SecureMUTE} \| !(\|_{i\in\{1\ldots 8\}} \mathbf{Spy_i}) \tag{1}$$

To analyze secrecy of a given protocol in SPL, one considers arbitrary *runs* of the protocol.

Definition 3 (Run of a Protocol)**.** *A run of a process $p = p_0$ is a sequence*

$$\langle p_0, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_w} \langle p_w, s_w, t_w\rangle \xrightarrow{e_{w+1}} \ldots$$

Analyzing the specification, the set of events can be made explicit.

Events in MUTE Each role in the specification generates different roles according to the action they execute. The complete model must consider all the rules for a complete proof, as described in the following definition:

Definition 4 (Events in MUTE)**.** *The event e_w is an event in the set*

$$Ev(MUTE) = Init : Ev(p_{Init}) \cup Interm : Ev(p_{Interm}) \cup Resp : Ev(p_{Resp}) \cup Spy : Ev(p_{Spy})$$

We now proceed to explain the events.

- *Initiator Events:* The initiator events indicate the behavior of process $\mathbf{Init(A)}$. This process can be splitted in two main sub-processes: an *output process* that generates a new name n and a request message $(\{\!|\, n, kw \,|\!\}_{Key(A,B)}, A, B)$ over the store, and an *input process* that receives the answer message $(\{\!|\, n, res, m \,|\!\}_{Key(A,B)}, A, B)$ via an input action $\mathbf{in}\,(\{\!|\, n, res, m \,|\!\}_{Key(A,B)}, A, B)$. Figures **??** and **??** describe these sub-processes.

(a)(b)
IniIni-
tiatia-
tortor
OutIn-
putput
ac-ac-
tiontion

Figure 5: Initiator Events

- *Intermediator Events:* Each agent acting as an intermediator has to forward the received messages. Figure **??** illustrates the event in which the intermediator process receives the message $(\{\!|\, M \,|\!\}_{Key(Y,A)}, Y, A)$ via an input action $\mathbf{in}\,(\{\!|\, M \,|\!\}_{Key(Y,A)}, Y, A)$. The composition of a second subprocess (Figure **??**) completes the intermeditator behavior, forwarding the received messages M to one of the neighbors by means of an output $o(\{\!|\, M \,|\!\}_{Key(A,B)}, A, B)$.

(a)(b)
In-Out-
putput
ac-ac-
tiontion

Figure 6: Intermediator Events

- *Responder Events:* A responder agent is basically composed by two processes: an initial input (Figure **??**) that waits for a message request $(\{\!|\, n, kw \,|\!\}_{Key(Y,A)}, Y, A)$, and a subsequent output of the answer $(\{\!|\, n, res, m \,|\!\}_{Key(A,B)}, A, B)$ via an output action $o(\{\!|\, n, res, m \,|\!\}_{Key(A,B)}, A, B)$, with a new name m (Figure **??**).

(a)(b)
In-Out-
putput
ac-ac-
tiontion

Figure 7: Responder Events

The property: secrecy over shared keys The secrecy theorem for the MUTE protocol concerns the shared keys of neighbors. If they are not corrupted from the start and the peers behave as the protocol states then the keys will not be leaked during a protocol run. If we assume that the shared keys are not contained in the initial output conditions ($key(X, Y) \not\sqsubseteq t_0$, where $X, Y \in Peers$), then at the initial state of the run there is no danger of corruption.

Theorem 1. *Given a run of MUTE* $\langle MUTE, s_0, t_0\rangle \xrightarrow{e_1} \cdots \xrightarrow{e_v} \langle p_v, s_v, t_v\rangle \xrightarrow{e_{v+1}}$ $\ldots$ *and* $A_0, B_0 \in Peers(G)$*, if* $key(A_0, B_0) \not\sqsubseteq t_0$ *then for each* $w \geq 0$ *in the run* $key(A_0, B_0) \not\sqsubseteq t_w$

Proof. Suppose there is a run of MUTE in which $key(A_0, B_0)$ appears on a message sent over the network. This means, since $key(A_0, B_0) \not\sqsubseteq t_0$, that there is a stage $w > 0$ in the run such that

$$key(A_0, B_0) \not\sqsubseteq t_{w-1} \text{ and } key(A_0, B_0) \sqsubseteq t_w.$$

Where $e_w \in Ev(\text{MUTE})$ (Definition **??**). By the evolution of nets with persistent conditions (SPL-nets token game), we can infer that $key(A_0, B_0) \sqsubseteq e^o_w$. As can be easily checked by using the events defined before, the shape of every **Init** or **Interm** or **Resp** event

$$e \in \mathbf{Init} : Ev(p_{\mathbf{Init}}) \cup \mathbf{Interm} : Ev(p_{\mathbf{Interm}}) \cup \mathbf{Resp} : Ev(p_{\mathbf{Resp}})$$

does not fulfill that. $key(A_0, B_0) \sqsubseteq e^o$, so the event e_w can therefore only be a spy event. If $e_w \in Spy : Ev(p_{Spy})$, however by using the proof principle of control precedence and the token game for SPL-nets, there must be an earlier stage u in the run, $u < w$ such that $key(A_0, B_0) \sqsubseteq t_u$ which clearly is a contradiction. $\square$

This small but illustrative example has served as a basis for the study of more complex properties, such as considering the secrecy threats involved with outsider attackers in the protocol [?]. This approach using process calculi has allowed us to expand our analyzes, considering new, more powerful models of attackers. These attackers are capable to *infiltrate* inside the P2P network with greater knowledge about the requests, answers and keys involved. Also, our approach has allowed us to find security attacks within these models, and to propose new designs that correct the protocol, guaranteeing a full secrecy property [?].

In the same sense, other classes of P2P protocols have been studied. For instance, protocols for collaborative P2P applications (such as Microsoft Messenger [**?**, **?**] and collaborative searches [?]) aim to allow application-level collaboration between users. There exist high security risks in these applications: the transmission of *private data* through the network is an important issue so that attackers will not access that kind of secrets. We have demonstrated the applicability of the process calculi approach in the modeling and verification of collaborative P2P applications, extending the language to consider attackers in dynamic reconfiguration systems [?].

4 Concluding Remarks

In this paper we have introduced basic ideas underlying *process calculi*, a set of formalisms aimed to describe and analyze essential properties of concurrent systems. The design of a process calculus respects a series of basic principles (abstraction, compositionality and economy) and intends to constitute a rigorous framework for the study of a particular phenomenon. An introduction to the most relevant issues and components of a process calculus was also given.

Although process calculi were originally devised for the study of distributed and mobile computing systems, a recent research trend consists in using them for analyzing systems and phenomena in emerging applications in the arts, the sciences and the engineering. The generic nature of process calculi and their associated techniques are some of the reasons that have motivated such a trend.

Based on previous works by the authors, the paper focused on two of such emerging applications, namely *systems biology* and *computer security.* Two different calculi (`ntcc` and SPL) were presented in order to illustrate some examples of systems in each application area. The particular motivations and underlying ideas of these calculi were discussed and explained. Some properties of the modeled applications were verified using the reasoning techniques associated to each calculus. Since the modeled applications correspond to real systems, the presented proofs enjoy a great deal of significance.

It is important to remark that process calculi verification can *complement* usual methods for system design and construction. In systems biology, for instance, process calculi can contribute to study the behavior of systems that are difficult to analyze using conventional experimentation. Some important steps in this direction have been taken; nevertheless, both theoretical and practical research efforts are needed to build more effective tools for biologists and related experts. As for security, the increasing flexibility of communication networks certainly poses new challenges for the design of protocols that ensure relevant security properties. One approach that seems particularly promising is the one that seeks the translation of protocol descriptions as some kind of logic formulas. As a consequence, further efforts are required in order to automatically verify more sophisticated properties. Representing more powerful and smarter attackers is also an interesting challenge to undertake.

Another challenge pertains to the development of software tools. This is particularly urgent as process calculi are trying to constitute an alternative for property verification in domains where experts usually know little (or even know nothing at all) about Computer Science formalisms. Therefore, tools to be designed must involve intuitive notations and user interfaces as well as efficient simulation capabilities. Some reported initial efforts in this practical direction are the tools for biological simulation proposed in [?] and χ-Spaces [?, ?], a framework for the development of protocols that is close to SPL.

Acknowledgement

Research reported in this work was supported by the Pontificia Universidad Javeriana-Cali under the project "Modelamiento de Problemas de Ciencia y Tecnología usando Cálculos de Procesos Concurrentes (Fase II)". This work was carried out while authors Gutiérrez, López and Pérez were research assistants at the AVISPA Research Group of the Pontificia Universidad Javeriana, Cali.