Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 351 (2020) 25–50

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Constraint-based Language for Multiparty Interactions[1]

Linda Brodo[2]

Università degli Studi di Sassari, Italy.

Carlos Olarte[3]

ECT, Universidade Federal do Rio Grande do Norte

Abstract

Multiparty interactions are common place in today's distributed systems. An agent usually communicates, in a single session, with other agents to accomplish a given task. Take for instance an online transaction including the vendor, the client, the credit card system and the bank. When specifying this kind of system, we probably observe a single transaction including several (binary) communications leading to changes in the state of all the involved agents. Multiway synchronization process calculi, that move from a binary to a multiparty synchronization discipline, have been proposed to formally study the behavior of those systems. However, adopting models such as Bodei, Brodo, and Bruni's Core Network Algebra (CNA), where the number of participants in an interaction is not fixed a priori, leads to an exponential blow-up in the number of states/behaviors that can be observed from the system. In this paper we explore mechanisms to tackle this problem. We extend CNA with constraints that declaratively allow the modeler to restrict the interaction that should actually happen. Our extended process algebra, called CCNA, finds application in balancing the interactions in a concurrent system, leading to a simple, deadlock-free and fair solution for the Dinning Philosopher problem. Our definition of constraints is general enough and it offers the possibility of accumulating costs in a multiparty negotiation. Hence, only computations respecting the thresholds imposed by the modeler are observed. We use this machinery to neatly model a Service Level Agreement protocol. We develop the theory of CCNA including its operational semantics and a behavioral equivalence that we prove to be a congruence. We also propose a prototypical implementation that allows us to verify, automatically, some of the systems explored in the paper.

Keywords: Concurrency theory, constraints, multiparty interactions

1 Introduction

Nowadays concurrent and mobile systems are ubiquitous in several domains and applications. They pervade different areas in science (biological and chemical sys-

[1] The joint work of the authors was possible thanks to the Visiting Scientist grant of University of Sassari. Carlos Olarte was also funded by CNPq. Linda Brodo was also funded by "*fondo di Ateneo per la ricerca 2019*" of University of Sassari.

[2] Email: brodo@uniss.it

[3] Email: carlos.olarte@gmail.com

https://doi.org/10.1016/j.entcs.2020.08.003

tems), engineering (security protocols and mobile and service oriented computing) and even the arts (tools for multimedia interaction). In general, concurrent systems exhibit complex forms of interaction, not only among their internal components, but also with the surrounding environment. Hence, a legitimate challenge is to provide computational models allowing us to understand the nature and the behavior of such complex systems. As an answer to this challenge, process algebra such as CCS [17], the π-calculus [18] and CSP [13] among several others have arisen as mathematical formalisms to model and reason about concurrent systems.

It is worth noticing that there is no unified model for concurrency, as in the case of the λ-calculus for sequential computations. Concurrency is, in fact, a relatively young area in computer science, and there are many models that accurately capture some behaviors but ignore/abstract some others. For instance, CCS focuses on the synchronization of processes: by exhibiting complementary actions two processes handshake and synchronize. However, no message is indeed sent among the agents. Data-passing extensions of CCS allow to overcome this problem, but the underlying communication network remains invariant during computation since processes cannot create and communicate new/private communication channels. The π-calculus gives a step forward and allows the communication of names (representing data and also communication links) that can later be used in other interactions. Many other process algebras have emerged as extensions of existing ones to cope with specific behaviors or they have taken inspiration from particular systems as in the case of calculi for system biology. For instance, the Spi Calculus [1] incorporates cryptographic primitives into the π-calculus for the specification and verification of security protocols and the Brane Calculi [10] took inspiration from the interaction of cell's membranes to model biological interactions. The beauty of all these formalisms relies on their simplicity (few operators), formal semantics and reasoning techniques, including behavioral equivalences (e.g., bisimulation), modal logics and model checking for specifying and verifying system's properties.

Most of the process algebras in the literature focus on binary interactions. Consider for instance CCS where a process $a.P$ (P prefixed by the *input* action a) can synchronize with $\overline{a}.Q$ (Q prefixed by the *output* action $\overline{a}$) when they are in a parallel composition (e.g., as in $a.P \mid \overline{a}.Q$). Such synchronization is made explicit via a special action τ (called the silent action) and what we observe is the synchronization of these processes in the transition $a.P \mid \overline{a}.Q \xrightarrow{\tau} P \mid Q$ where, after the handshake, P and Q continue their executions.

In this paper we shall focus on a multiparty extension of CCS, the so called Core Network Algebra (CNA) [4,6,5], a multiparty process algebra where the number of participants in each synchronization is not fixed a priori. In CNA, the binary interaction of CCS is extended and the usual input and output prefixes are generalized to *links*, e.g., ${}^{a}\backslash_{b}$, that can be thought of as the forwarding of a message received on channel a (the input channel) to another channel b (the output channel). The standard input and output prefixes of CCS are recovered when links expose only an output (${}^{\tau}\backslash_{b}$), or an input (${}^{a}\backslash_{\tau}$); these particular actions are the ends of a *link chain* where τ is the *silent* action (as in CCS). A link chain is the mechanism

in CNA by which $n \geq 2$ entities can synchronize. Each entity must offer a link that has to match with an adjacent link offered by another entity. For instance, if three processes offer, respectively, the links ${}^{a}\backslash_{b}$, ${}^{b}\backslash_{c}$ and ${}^{c}\backslash_{d}$, they can synchronize and produce the link chain ${}^{a}\backslash_{b}^{b}\backslash_{c}^{c}\backslash_{d}$, where information flows from a to d through b and c.

In [7,8] the authors showed that the multiparty and open (arbitrary number of participants) nature of CNA poses interesting problems for the point of view of verification. In particular, while the number of possible successor states from a CCS process is quadratic on the number of its outermost prefixes, it is exponential in the case of a CNA process. Moreover, it is possible to specify a graph of agents as a CNA process P (some examples in Section 4.2) and we can check that there is a Hamiltonian path in the graph iff there is exactly one immediate successor of P. We explored in [7,8] symbolic techniques aiming to tame the inherent complexity of the CNA transition system.

The goal of the present paper is to define a suitable extension of CNA that allows the specifier to control, in a declarative way, the behavior of processes. More precisely, we provide mechanisms that allow us to include in a CNA specification certain restrictions that appear naturally in the modeled system at hand. For instance, we may be interested in transitions/paths in a graph to have certain length or to specify upper bounds on the number of participants in an open interaction. This is the purpose of introducing constrains and values in prefixes: the process ${}^{a}\backslash_{b}\langle !v_p\rangle(?c_p).P$ offers the link ${}^{a}\backslash_{b}$ at a given cost/value v_p and checks whether the interaction with some other processes satisfies the constraint c_p. When this process interacts with, e.g., ${}^{b}\backslash_{c}\langle !v_q\rangle(?c_q).Q$, the synchronization ${}^{a}\backslash_{b}^{b}\backslash_{c}$ has a final cost $v = v_p + v_q$ and it can actually happen if v satisfies both c_p and c_q. As we shall see, the *control* mechanism due to constraints have interesting properties and, in the quest of defining such extensions, we found solutions to distributed problems that do not have a simple solution in other process calculi. Quoting José Meseguer [16], "*increased expressiveness is not a theoretical luxury, but an eminently practical goal, since formal specification languages should describe as simply and naturally as possible the widest possible class of systems.*". This is precisely our goal here.

Contributions and organization. We start in Section 2 recalling the CNA framework. In Section 3 we introduce the notion of constraints based on complete lattices and monoids. Such structure allows us to compare values (e.g, $c < 42$) and also to accumulate values (e.g., adding the length/cost of two paths in a graph). We then extend the language of CNA to allow processes to communicate such values and also to query constraints on them. Hence, a transition can happen only if all the involved agents satisfy their own constraints. We call this new calculus constrained CNA (CCNA). We shall show that CCNA is a conservative extension of CNA (Notation 3.7): CNA processes can be seen as CCNA processes that exchange the less restrictive value and query the always-true constraint. In Section 3.2 we endow CCNA with a suitable operational semantics. This semantics has some novelties wrt the standard one for CNA: we follow a *late* approach in contrast to the *early* one

used in [5]. Our semantics is then a good middle point between the inherent non-deterministic early semantics and the (more involved) symbolic semantics in [8]. Section 3.3 introduces the notion of (network) bisimulation for CCNA process and we prove that this equivalence is a congruence (thus allowing us to replace equals by equals in larger systems). In Section 4.1 we show that the use of constraints leads to a simple solution for the Dinning Philosopher problem where concurrent processes compete for the use of some resources. Our solution is deadlock-free and fair: all the agents can progress infinitely often. Fairness is usually imposed as an external condition but here, the constrained transition system satisfies such property. In Section 4.2 we model a graph representing a transportation system and we show how constraints allow us to discard some (undesired) behaviors. Moreover, in Section 4.3, we present an application in the context of a Service Level Agreement (SLA) protocol where constraints naturally represent restrictions such as the upper bound for the price of the service to be paid and the minimal quality (bandwidth) the client is expecting. Section 5 concludes the paper and present related work. We have also implemented a tool using the Maude system [16]. We shall not describe in depth such system here but we shall exemplify its use in Section 5. We thus contribute with a formal framework to specify constrained multiparty interactions and a prototypical tool showing its appropriateness as (automatic) reasoning technique.

2 Background on link chained interactions

Let $\mathcal{C}$ be the set of channel names, ranged over by $a, b, c, ...$, and $\tau, \Box$ two distinguished symbols not in $\mathcal{C}$. $\mathcal{A} = \mathcal{C} \cup \{\tau\} \cup \{\Box\}$ is the set of actions, ranged over by $\alpha, \beta, ...$, where the symbol τ denotes a *silent* action, while the symbol $\Box$ denotes a *virtual* (non-specified) action.

Definition 2.1 (Links: solid, virtual and valid) *A* link *is a pair* $\ell = {}^{\alpha}\backslash_{\beta}$*, with* $\alpha, \beta \in \mathcal{A}$*. We call* α *the* source site *of* ℓ *and* β *the* target site *of* ℓ*. A link* ${}^{\alpha}\backslash_{\beta}$ *is* solid *if* $\alpha, \beta \neq \Box$*; the link* ${}^{\Box}\backslash_{\Box}$ *is called* virtual*. A link is* valid *if it is solid or virtual. We shall use* $\mathcal{L}$ *to denote the set of valid links.*

Intuitively, ${}^{a}\backslash_{b}$ records an action a (or equivalently an input on channel a) and a co-action b (or an output on b), such that the input on a (receiving from the source of a communication) can be forwarded (to the target of the communication) along channel b. The τ-action is used when no interaction is required on the left (as in ${}^{\tau}\backslash_{a}$), on the right (as in ${}^{a}\backslash_{\tau}$) or on both sides (as in ${}^{\tau}\backslash_{\tau}$). The link ${}^{\tau}\backslash_{\tau}$ is called *τ-link*. A virtual link ${}^{\Box}\backslash_{\Box}$ represents a non-specified interaction that will be later completed. Examples of valid links are ${}^{\Box}\backslash_{\Box}$, ${}^{a}\backslash_{a}$, ${}^{\tau}\backslash_{a}$, ${}^{b}\backslash_{a}$, ${}^{\tau}\backslash_{\tau}$. The links ${}^{\Box}\backslash_{a}$, ${}^{a}\backslash_{\Box}$ are not valid.

Links can be combined in *link chains*. Intuitively, a link chain $s = \ell_1 ... \ell_n$ represents a multiparty interaction where each ℓ_i records the source and the target sites of each hop.

Definition 2.2 (Link Chain) *A* link chain *is a finite sequence* $s = \ell_1 ... \ell_n$ *of valid links* $\ell_i = {}^{\alpha_i}\backslash_{\beta_i}$*. We say that* s *is valid if:*

(i) *for all* $i \in [1,n)$, $\begin{cases} \beta_i, \alpha_{i+1} \in \mathcal{C} & \textit{implies } \beta_i = \alpha_{i+1} \\ \beta_i = \tau & \textit{iff } \alpha_{i+1} = \tau \end{cases}$ *; and*

(ii) $\exists i \in [1,n].\ \ell_i \neq {}^{\Box}\backslash_{\Box}$.

We shall use VC to denote the set of valid link chains. The length *of a link chain s (i.e., the number of links in s) will be denoted as $|s|$.*

Condition (i) says that two adjacent solid links must match on their adjacent sites. In order to highlight such a matching, we shall write link chains as, e.g., ${}^{a}\backslash^{b}_{i}\backslash^{c}_{c}\backslash_{d}$ instead of sequences of links as in ${}^{a}\backslash_{b}\ {}^{b}\backslash_{c}\ {}^{c}\backslash_{d}$. Condition (i) also says that the silent action τ cannot be matched by a virtual action $\Box$. As we shall see, this is required since a τ-action can be only matched with τ when processes synchronize on restricted channels. Condition (ii) says that a valid link chain must have at least one solid link (i.e., the chain ${}^{\Box}\backslash^{\Box}_{\Box}\backslash_{\Box}$ is not valid). Some examples of valid link chains are: ${}^{\Box}\backslash^{a}_{\Box}\backslash^{b}_{b}\backslash_{\tau}$, ${}^{a}\backslash^{\Box}_{b}\backslash^{c}_{\Box}\backslash_{d}$, and ${}^{\tau}\backslash^{a}_{a}\backslash_{\tau}$. The first chain represents an interaction where there is a pending synchronization on the left of ${}^{a}\backslash_{b}$; similarly, the second chain represents an interaction where a third-party process must offer a link joining b and c (i.e., ${}^{b}\backslash_{c}$). Finally, the last chain is the result of a binary interaction between a process performing the output ${}^{\tau}\backslash_{a}$ and a process performing the input ${}^{a}\backslash_{\tau}$. The following are examples of link chains that are not valid: ${}^{a}\backslash^{c}_{b}\backslash_{d}$, ${}^{\Box}\backslash^{\tau}_{\Box}\backslash_{a}$, and ${}^{a}\backslash^{c}_{\tau}\backslash_{d}$. Hereafter, we only consider valid links and valid link chains. We shall use $\perp$ to denote non-valid links and chains.

Now we introduce the merge operator $s \bullet s'$ that acts on two link chains of the same length, i.e. $|s| = |s'|$. Intuitively, $s \bullet s'$ makes the two link chains collapse into one link chain where some of the virtual links in s (resp. s') have been substituted with the corresponding solid links in s' (resp. s).

Definition 2.3 (Merge) *Let $\alpha, \beta \in \mathcal{A}$ be actions. The merge operator on actions is defined as follow:*

$$\alpha \bullet \beta = \alpha \textit{ if } \beta = \Box \qquad \alpha \bullet \beta = \beta \textit{ if } \alpha = \Box \qquad \alpha \bullet \beta = \perp \textit{ otherwise}$$

For links, let $\ell_1 = {}^{\alpha_1}\backslash_{\beta_1}$ and $\ell_2 = {}^{\alpha_2}\backslash_{\beta_2}$ be valid links and $\alpha_1 \bullet \alpha_2 = x_\alpha$, $\beta_1 \bullet \beta_2 = x_\beta$. If $x_\alpha, x_\beta \neq \perp$, then $\ell_1 \bullet \ell_2 = {}^{x_\alpha}\backslash_{x_\beta}$. Otherwise, $\ell_1 \bullet \ell_2 = \perp$. For the merge on chains, let $s = \ell_1 ... \ell_n$ and $s' = \ell'_1 ... \ell'_n$ be valid chains with $\ell_i = {}^{\alpha_i}\backslash_{\beta_i}$ and $\ell'_i = {}^{\alpha'_i}\backslash_{\beta'_i}$. If $\ell_i \bullet \ell'_i \neq \perp$ for all $i \in [1,n]$ and $(\ell_1 \bullet \ell'_1)...(\ell_n \bullet \ell'_n)$ is a valid chain, then $s \bullet s' = (\ell_1 \bullet \ell'_1)...(\ell_n \bullet \ell'_n)$. Otherwise, $s \bullet s' = \perp$.

As an example, the chains ${}^{\Box}\backslash^{\Box}_{\Box}\backslash^{a}_{\Box}\backslash_{b}$ and ${}^{c}\backslash^{\Box}_{a}\backslash_{\Box}$ cannot merge, as they have different length; ${}^{a}\backslash^{\Box}_{b}\backslash_{\Box}$ and ${}^{\Box}\backslash^{c}_{\Box}\backslash_{d}$ cannot merge since ${}^{a}\backslash^{c}_{b}\backslash_{d}$ is not a valid chain; a chain s cannot merge with itself; finally, ${}^{c}\backslash^{\Box}_{a}\backslash^{b}_{\Box}\backslash_{d}$ and ${}^{\Box}\backslash^{a}_{\Box}\backslash^{\Box}_{b}\backslash_{\Box}$ merge into ${}^{c}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_{d}$.

As usual in process calculi, names are restricted in order to force interactions. Let $s = \ell_1 ... \ell_n = \ldots {}^{\alpha_i}\backslash^{\alpha_{i+1}}_{\beta_i}\backslash_{\beta i+1} \ldots$ be a link chain. We say that a is *matched* in s if both

(1) $a \neq \alpha_1$ and $a \neq \beta_n$ (i.e., a cannot occur in the endpoints), and

(2) for any $i \in [1,n)$, either $\beta_i = \alpha_{i+1} = a$ or $\beta_i, \alpha_{i+1} \neq a$.

Otherwise, we say that a is *unmatched* (or *pending*) in s. For example, a and b are matched in ${}^{\tau}\backslash_{a}^{a}\backslash_{b}^{b}\backslash_{d}$. Instead, neither a nor b are matched in ${}^{d}\backslash_{a}^{\square}\backslash_{\square}^{c}\backslash_{b}$.

Definition 2.4 (Restriction) *Let $\alpha, \beta \in \mathcal{A}$ be actions and $a \in \mathcal{C}$ be a channel name. We define the following operations on actions and links:*

$$(\nu\, a)\alpha = \begin{cases} \tau & \text{if } \alpha = a \\ \alpha & \text{otherwise} \end{cases} \qquad \text{and} \qquad (\nu\, a)\ {}^{\alpha}\backslash_{\beta} = {}^{((\nu\, a)\alpha)}\backslash_{((\nu\, a)\beta)}$$

We lift those operations to link chains as follow:

$$(\nu\, a)s = \begin{cases} ((\nu\, a)\ell_1)\dots((\nu\, a)\ell_n) & \text{if } a \text{ is matched in } s \\ \bot & \text{otherwise} \end{cases}$$

For instance, in $s = {}^{\tau}\backslash_{a}^{a}\backslash_{b}^{\square}\backslash_{\square}$, the name a is matched and $(\nu a)s = {}^{\tau}\backslash_{\tau}^{\tau}\backslash_{b}^{\square}\backslash_{\square}$; whereas $(\nu b)s$ is undefined since b is pending in s.

3 CNA with constraints

In this section we present the CNA calculus equipped with data passing and constrains on those data. Unlike the link-calculus [5] (and also the π-calculus in that respect), we will consider values which are not channel names, thus they will have a separate definition set. Our goal is to perform some checks on the data that participants in a interaction can share. Values will be taken from an algebraic structure that allows us to compare ($\preceq$) and also to accumulate ($\otimes$) those values. The following definitions are from [11] that simplifies at some extent the presentation of c-semiring [3] (another algebraic structure used for accumulating and comparing constraints).

Recall that a partial order is a pair $\langle \mathcal{D}, \preceq \rangle$ such that $\mathcal{D}$ is a set and $\preceq$ is a reflexive, transitive and antisymmetric relation on $\mathcal{D}$. We use $\prec$ to denote the strict version of $\preceq$ (i.e., $v \prec v'$ if $v \preceq v'$ and $v \neq v'$). We also use $\succeq$ and $\succ$ with the expected meaning. A complete lattice is a partial order where any subset $X \subseteq \mathcal{D}$ has a *least upper bound* denoted as $\sqcup X$. By duality, the *greatest lower bound*, denoted as $\sqcap X$, also exists. As usual, we use $\top$ and $\bot$ to denote, respectively, $\sqcup\mathcal{D}$ and $\sqcup\emptyset$. An abelian (or commutative) monoid is a triple $\langle \mathcal{D}, \otimes, \top \rangle$ where $\otimes : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a commutative and associative operator and $\top$ is its identity (i.e., $\forall v \in \mathcal{D}. v \otimes \top = \top \otimes v = v$).

Definition 3.1 (CLM [11]) *A complete lattice monoid (for short CLM) is a tuple $\mathcal{K} = \langle \mathcal{D}, \preceq, \otimes \rangle$ s.t. $\langle \mathcal{D}, \preceq \rangle$ is a complete lattice, $\bot$ and $\top$ are, respectively, the least and the greatest elements of $\mathcal{D}$ and $\langle \mathcal{D}, \otimes, \top \rangle$ is a commutative monoid. We assume that $\otimes$ distributes over glb's, i.e.,*

$$\forall v \in \mathcal{D}. \forall X \subseteq \mathcal{D}. v \otimes \sqcap X = \sqcap\{v \otimes x \mid x \in X\}$$

Let us explain the above definition with a concrete instance that we shall use in the following sections. Consider the structure $\mathcal{K}_{\mathbb{N}} = \langle \mathbb{N}_+^\infty, \geq, + \rangle$ where $\mathbb{N}_+^\infty$ is the set of natural numbers completed with ∞ and $\geq$ is the usual "greater than" relation on $\mathbb{N}_+^\infty$ (e.g., $5 \geq 2$). We can think of the elements in $\mathbb{N}_+^\infty$ as costs. Note that we consider 0 as the "best" ($\top$) cost ($x \preceq 0$, i.e., $x \geq 0$ for any $x \in \mathbb{N}_+^\infty$). Also, ∞ is the worst ($\bot$) cost we can assume. We accumulate costs with $+$ (addition on $\mathbb{N}$). When costs are added, we get "worse costs" (e.g., $3 + 2 \geq 3$). More generally, we can show that $\otimes$ is an *intensive operator*: $\forall v, v' \in \mathcal{D}, v \otimes v' \preceq v$. It is also possible to prove that $\bot$ is absorbing for $\otimes$ (i.e., $\forall v \in \mathcal{D}. v \otimes \bot = \bot$). In our concrete example, $v + \infty = \infty$.

Definition 3.2 (Residuation) *Let $\mathcal{K} = \langle \mathcal{D}, \preceq, \otimes \rangle$ be a CLM and $v, v' \in \mathcal{D}$. The residuation of v with respect to v' is defined as $v \div v' = \sqcap \{x \in \mathcal{D} \mid v \preceq v' \otimes x\}$.*

As expected, $v \preceq v' \otimes (v \div v')$ (due to distributivity). Moreover, if $v' \preceq v$ then $v \div v' = \top$. In $\mathcal{K}_{\mathbb{N}}$, $\div$ is subtraction on $\mathbb{N}_+^\infty$ ($v \div v' = v - v'$) if $v \geq v'$ (i.e., $v \preceq v'$) and 0 otherwise. (See [2] for a discussion on residuation in constrain-based formalisms).

It is possible to combine different CLMs in order to measure/compare different entities in a single operation.

Lemma 3.3 (Combining CLMs) *Let $\mathcal{K}_1 = \langle \mathcal{D}_1, \preceq_1, \otimes_1 \rangle$ and $\mathcal{K}_2 = \langle \mathcal{D}_2, \preceq_2, \otimes_2 \rangle$ be CLMs. Define $\mathcal{K} = \mathcal{K}_1 \times \mathcal{K}_2 = \langle \mathcal{D}_1 \times \mathcal{D}_2, \prec, \otimes \rangle$ where $(v, w) \preceq (v', w')$ iff $v \preceq_1 v'$ and $w \preceq_2 w'$; and $(v, w) \otimes (v', w') = (v \otimes_1 v', w \otimes_2 w')$. Then, $\mathcal{K}$ is a CLM.*

Proof. The existence of arbitrary lubs is an easy consequence of the existence of lubs in $\mathcal{K}_1$ and $\mathcal{K}_2$. Distributivity also follows easily. Note that we can also define (point-wise) residuation on $\mathcal{K}$. □

3.1 Constrained multiparty interactions

Now we are ready to introduce the syntax of constrained CNA, from now on denoted as CCNA, which is parametric with respect to a CLM. In the following, we fix the CLM $\mathcal{K} = \langle \mathcal{D}, \preceq, \otimes \rangle$ with residuation operator $\div$. We shall use u, v to range over elements of $\mathcal{D}$. We shall call *data-variables* to variables (usually denoted as $x, y, ...$) that take values from $\mathcal{D}$.

Definition 3.4 (Syntax of CCNA) *Given a set of channel names $\mathcal{C}$ (ranged over by a, b) and a CLM $\mathcal{K}$, processes in CCNA are built from the syntax*

exp	$::= \mathfrak{a} \mid v \mid x \mid exp \otimes exp \mid exp \div exp$	*expressions*
atm	$::= exp \star exp' \quad$ *where* $\star \in \{\preceq, \prec, =, \neq, \succeq, \succ\}$	*atomic constraints*
c, c'	$::= atm \mid c \wedge c'$	*constraints*
p, q	$::= \mathbf{0} \mid \ell\langle !e \rangle (?c).P \mid p + q$	*sequential processes*
P, Q	$::= p \mid P \mid Q \mid (\nu\, a)P \mid A\langle a_1, \ldots, a_n; e_1, \ldots, e_m \rangle$	*processes*

where $\mathfrak{a}$ *is a distinguished data-variable representing the accumulated value of an interaction;* $v \in \mathcal{D}$*;* x *is a data-variable;* e *is an expression where* $\mathfrak{a}$ *does not occur;* $\ell = {}^{\alpha}\backslash_{\beta}$ *is a solid link built from* $\mathcal{C} \cup \{\tau\}$ *(i.e.* $\alpha, \beta \neq \square$*);* A *is a process identifier for which we assume a (possibly recursive) definition of the form* $A(\boldsymbol{b}; \boldsymbol{x}) \triangleq P$ *where* $\boldsymbol{b}$ *is a list of pair-wise distinct names and* $\boldsymbol{x}$ *is a list of pair-wise distinct data-variables; and* e_i *is an expression where* $\mathfrak{a}$ *does not occur.* □

In the following paragraphs, we explain in detail each of the components of this definitions. The symbol $\mathfrak{a}$ denotes a special data-variable that accumulates (using $\otimes$) the values added by each of the participants in an interaction (Definition 3.10 below). In $\mathcal{K}_{\mathbb{N}}$, examples of valid expressions are 3, $5 - y$, $5 + x + \mathfrak{a}$, etc.

Constraints will be used to check if the agents *agree* on the values of a given interaction. In $\mathcal{K}_{\mathbb{N}}$, examples of atomic constraints are $5 \geq 3$, $x \geq 4$, $x \geq y + 3$, $\mathfrak{a} \geq 10$, etc. A constraint is just a conjunction of such atoms. We use $\mathtt{tt}$ to denote the (always true) atomic constraint $\bot \preceq \top$.

Before explaining the meaning of processes, we need an extra definition. The only binder for (data-) variables is the process definition $A(\boldsymbol{b}; \boldsymbol{x}) \triangleq P$ where the variables in $\boldsymbol{x}$ occur bound in P. We shall use $fv(P)$ to denote the set of free (data-) variables in the process P. We shall also use $fv(c)$ to denote the set of free (data-) variables in c (including $\mathfrak{a}$).

Definition 3.5 (Entailment) *We say that a constraint* c *is ground if it does not contain data-variables nor the symbol* $\mathfrak{a}$ *(i.e.,* $fv(c) = \emptyset$*). Given a ground atomic constraint* $c = e \star e'$*, we say that* c *holds, notation* $\mathcal{K} \models c$*, if* e *and* e' *reduce (performing the operations* $\otimes$ *and* $\div$ *in* $\mathcal{K}$*), respectively, to* v *and* v' *and the relation* $v \star v'$ *holds in* $\mathcal{K}$*. Otherwise, we write* $\mathcal{K} \not\models c$*. We extend this notion to ground constraints as follows:* $\mathcal{K} \models c_1 \wedge \cdots \wedge c_n$ *whenever* $\mathcal{K} \models c_i$ *for all* $i \in 1..n$*. Let* c, c' *be constraints s.t.* $fv(c), fv(c') \subseteq \{\mathfrak{a}\}$*. We say that* c, c' *are equivalent, notation* $c \approx c'$ *if* $\forall v \in \mathcal{D}. \mathcal{K} \models c[v/\mathfrak{a}]$ *iff* $\mathcal{K} \models c'[v/\mathfrak{a}]$*.*

As an example, we have the following entailments: $\mathcal{K}_{\mathbb{N}} \models 3 + 2 \preceq 1 + 2$; and $\mathcal{K}_{\mathbb{N}} \models 3 \div 2 \succeq 1 + 2$. Moreover, if $c_1 = \mathtt{tt} \wedge 3 \preceq \mathfrak{a}$ and $c_2 = \mathfrak{a} + 0 \succeq 5 \div 2$ then $c_1 \approx c_2$.

The process $\mathbf{0}$ does nothing. The process $\ell\langle !e\rangle(?c).P$ offers the *solid link* ℓ along with the value denoted by the expression e and *checks* whether the constraint c holds. After this interaction, the process behaves as P. The non-deterministic process $p + q$ can either behave as p or q. Interleaved parallel composition is denoted as $P \mid Q$. The process $(\nu\, a)P$ behaves as P but it cannot exhibit any action a. Hence, we can say that a is local (or private) in P. As usual, $(\nu\, a)P$ binds the free occurrences of a in P.

The call $A\langle a_1, ..., a_n; e_1, ..., e_m\rangle$ behaves as the process $P[a_1/b_1, ..., a_n/b_n][e_1/x_1, \ldots, e_m/x_m]$ if the constant A is defined as $A(b_1, ..., b_n; x_1, ..., x_m) \triangleq P$. As expected, the actual parameters substitute the formal parameters of the definition. Besides binding the variables x_i, the above definition binds the names b_i. We shall use $fn(P)$ and $bn(P)$ to denote, respectively, the free and bound names in P.

We shall impose some restrictions on process that are commonplace in the literature (see e.g., [12]).

Definition 3.6 (Valid Processes) *Processes are taken up to alpha-conversion (renaming of bound names). We assume that in a process definition* $A(\boldsymbol{b};\boldsymbol{x}) \triangleq P$, $fn(P) \subseteq \boldsymbol{b}$ *and* $fv(P) \subseteq \boldsymbol{x}$. *Moreover, in order to guarantee the transition system to be finitely branching, we assume that recursive calls in* P *must be guarded inside a prefix* $\ell\langle !e\rangle(?c).Q$. *Finally, we assume that in all processes, the occurrence of a data-variable* x *(different from* $\mathfrak{a}$*) is always bound by a process definition.*

For instance, the processes $\ell\langle !2\rangle(?x > 5).Q$ and $\ell\langle !2 + x\rangle(?\mathfrak{a} > 5).Q$ are not valid since x occurs free (and not bound by a process definition). Moreover, the process definition $A() \stackrel{\texttt{def}}{=} A\langle\rangle | P$ is not valid since the call $A\langle\rangle$ is not guarded (thus making the transition system infinitely branching).

Notation 3.7 *For the sake of readability, we shall use the following shortcuts. We shall omit a trailing* $\mathbf{0}$*, e.g. by writing* ${}^{a}\backslash_{b}$ *instead of* ${}^{a}\backslash_{b}.\mathbf{0}$. *We shall also write* $\ell\langle !e\rangle$ *instead of* $\ell\langle !e\rangle(?\,\mathtt{tt})$. *Recall that* $v \otimes \top = v$ *for all* $v \in \mathcal{D}$. *Hence, we shall write* $\ell(?c)$ *instead of* $\ell\langle !\top\rangle(?c)$. *Finally, we shall simply write* ℓ *instead of* $\ell\langle !\top\rangle(?\,\mathtt{tt})$. *Note that any CNA process is also a CCNA process whose shared value (*$\top$*) is irrelevant for the final interaction and whose constraint always holds (*$\mathtt{tt}$*).*

The following example shows how constraints can be used to limit the number of participants in a multiparty interaction. For the moment, the discussion about the behavior of process remains informal until we define the semantics in the next section.

Example 3.8 *The process* $P = {}^{a}\backslash_{b}$ *is able to interact with any other process, say* $Q = {}^{c}\backslash_{d}$ *and synchronize producing the link chain* ${}^{a}\backslash_{b}^{\Box}\backslash_{\Box}^{c}\backslash_{d}$. *Other outputs can be also expected from that interaction, as e.g.* ${}^{c}\backslash_{d}^{\Box}\backslash_{\Box}^{a}\backslash_{b}$. *In fact, a 3-party synchronization is also possible with yet another process* $R = {}^{b}\backslash_{c}$, *thus building the chain* ${}^{a}\backslash_{b}^{b}\backslash_{c}^{c}\backslash_{d}$. *More generally,* P *can interact with an unbounded number of other processes by building suitable (valid) link chains. This means that interactions in CNA are open since the number of participants is not fixed a priori. This is a quite expressive feature of the calculus but it makes also difficult to reason about processes. (See [7] and [8] for symbolic techniques to deal with this problem). Consider the structure* $\mathcal{K}_{\mathbb{N}}$ *and the processes below*

$$P' = {}^{a}\backslash_{b}\langle !1\rangle(?\mathfrak{a} \leq 2) \qquad Q' = {}^{c}\backslash_{d}\langle !1\rangle(?\mathfrak{a} \leq 2) \qquad R' = {}^{b}\backslash_{c}\langle !1\rangle(?\mathfrak{a} \leq 2)$$

Hence, P' *can interact with at most one of the other two processes (*Q' *or* R'*) since, in each interaction, the value* 1 *is accumulated and such value must be less than* 2. *This is a very intuitive (and declarative) mechanism for counting and restricting the participants in an interaction.* □

Derived construct (tuples). Let us introduce an idiomatic construct that will be useful. Due to Lemma 3.3, we can assume that agents offer links with tuples of

$$\frac{}{\ell\langle !e\rangle(?c).P \xrightarrow{\ell\langle !e\rangle(?c)} P}\ Act \qquad \frac{p \xrightarrow{\mu} P'}{p+q \xrightarrow{\mu} P'}\ Lsum \qquad \frac{q \xrightarrow{\mu} Q'}{p+q \xrightarrow{\mu} Q'}\ Rsum$$

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ Lpar \qquad \frac{Q \xrightarrow{\mu} Q'}{P \mid Q \xrightarrow{\mu} P \mid Q'}\ Rpar \qquad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\phi} Q'}{P \mid Q \xrightarrow{\mu \bullet \phi} P' \mid Q'}\ Com$$

$$\frac{P[\boldsymbol{a}/\boldsymbol{b}][\boldsymbol{v}/\boldsymbol{x}] \xrightarrow{\mu} P' \qquad A(\boldsymbol{b};\boldsymbol{x}) \triangleq P}{A\langle \boldsymbol{a};\boldsymbol{v}\rangle \xrightarrow{\mu} P'}\ Ide \qquad \frac{P \xrightarrow{\mu} P'}{(\nu a)P \xrightarrow{(\nu a)\mu} (\nu a)P'}\ Res$$

Fig. 1. Semantics. All the rules have the proviso that the labels in the transitions are valid (Def. 3.10).

elements as, e.g., $(e_1, \ldots, e_n)$. Such tuples are elements of the CLM $\mathcal{K}_1 \times \cdots \times \mathcal{K}_n$. Hence, we shall use the more convenient notation

$$\ell\langle !e_1, \ldots, e_n\rangle(?c_1 \wedge \cdots \wedge c_n)$$

where each constraint c_i may use the special symbol $\mathfrak{a}_i$ (denoting the accumulated values in the i-th position of the tuple). Note that these tuples and constraints can be easily rewritten in the syntax of Definition 3.4 by using the construction in Lemma 3.3. Since each CLM $\mathcal{K}_i$ represents a different value/measure to be accumulated, it is not legal to combine values of different CLMs in the same expressions. For instance, expressions as, e.g., $\mathfrak{a}_1 \div \mathfrak{a}_2$ are not legal. Similarly for constraints. Next notational convention allows us to give alias for the "accumulating" variables $\mathfrak{a}_i$ to make specifications cleaner.

Notation 3.9 (Tuples and variables) *For a given specification, we can determine that tuples used in interactions are of the form $\langle x_1, ..., x_n\rangle$ where the aliases x_i cannot be bound by a process definitions. Hence, we write $\ell\langle !x_1 = e_1, \ldots, x_n = e_n\rangle(?c_1 \wedge \cdots \wedge c_n)$ to mean $\ell\langle !e_1, \ldots, e_n\rangle(?\,(c_1 \wedge \cdots \wedge c_n)\,[\mathfrak{a}_1/x_1, ..., \mathfrak{a}_n/x_n])$. Moreover, if $v_i = \top$, we further omit "$x_i = \top$". For instance, if we determine that tuples are of the form $\langle cost, speed\rangle$, the expression $\ell\langle !speed = 1\rangle(?cost \leq 3 \wedge speed \geq 10)$ means $\ell\langle !\top, 1\rangle(?\mathfrak{a}_1 \leq 3 \wedge \mathfrak{a}_2 \geq 10)$.*

3.2 Semantics

In this section we introduce the operational semantics for CCNA. A novelty in this semantics is the use of a *late* approach where the number of participants is inferred only in the communication rule in contrast to the *early* approach adopted in [4] (where the rule *Act* needs to "guess" the size of the interaction) . This distinction will be clarified in brief.

Before defining the semantics, we shall lift the definition of merge on link chains (Definition 2.3) to consider also *valued-constrained-chains* (VCC) of the form $s\langle !e\rangle(?c)$ where s is a link chain, e is a ground expression (no free variables) and c is a constraint where $fv(c) \subseteq \{\mathfrak{a}\}$. For that, we allow link-chains to be enlarged (injecting virtual links) or contracted via the relation $\bowtie$ defined below.

Definition 3.10 (Valued-constrained-chains and operations) *We let* $\bowtie$ *be the least equivalence relation over link chains closed under the axioms (whenever both sides are valid link chains):*

$$s \,{}^{\Box}\backslash_{\Box} \bowtie s \qquad s_1 \,{}^{\Box}\backslash^{\Box}_{\Box}\backslash_{\Box}\, s_2 \bowtie s_1 \,{}^{\Box}\backslash_{\Box}\, s_2 \qquad {}^{\Box}\backslash_{\Box}\, s \bowtie s \qquad s_1 \,{}^{\alpha}\backslash^{a}_{a}\backslash_{\beta}\, s_2 \bowtie s_1 {}^{\alpha}\backslash^{\Box}_{a}\backslash^{a}_{\Box}\backslash_{\beta}\, s_2$$

We merge VCCs as follow: $s\langle !e\rangle(?c) \bullet s'\langle !e'\rangle(?c') = (w \bullet w')\langle !e \otimes e'\rangle(?c \wedge c')$ *with* $s \bowtie w$ *and* $s' \bowtie w'$*. We define* $(\nu a)(s\langle !e\rangle(?c))$ *as* $((\nu a)s)\langle !e\rangle(?c)$*. We say that a VCC* $s\langle !e\rangle(?c)$ *is valid iff* s *is a valid link chain and* $\mathcal{K} \models c[e/\mathfrak{a}]$*.*

We shall use μ, ϕ to range over VCCs. Note that the values are merged using the $\otimes$ operator of the CLM. Note also that, in order to check the validity of a VCC, the symbol $\mathfrak{a}$ is replaced with the current *accumulated* value e. Since we are assuming that there are no free data-variables in a process (see Definition 3.6), once the data-variables of a definition have been replaced with concrete values, $c[e/\mathfrak{a}]$ is indeed a ground constraint. Finally, since there are no occurrences of names in values and constraints, the restriction operation on VCCs acts only on the link chains (Definition 2.4) and hence, e and c remain the same.

The structural operational semantics is given by the labeled transition system $(\mathcal{P}, VCC, \longrightarrow)$ where the set $\mathcal{P}$ of states is the set of CCNA processes, labels are valid valued-constrained-chains (VCC) and the transition relation $\longrightarrow$ is the minimal transition relation generated by the rules in Figure 1.

The prefix process $\ell\langle !e\rangle(?c).P$ simply offers the link ℓ with value/expression e and checks whether c is valid, i.e., $\ell\langle !e\rangle(?c)$ must be a valid VCC (Rule *Act*). If p is able to exhibit a transition to P' with label μ, then $p + q \xrightarrow{\mu} P'$ (Rule *Lsum*). Similarly for q (Rule *Rsum*). If P can exhibit a transition, it can also exhibit the same transition when running in parallel with Q (Rules *Lpar* and *Rpar*). In *Res*, if P offers the action μ then $(\nu a)P$ offers $(\nu a)\mu$ (if it is valid). Rule *Ide* simply replaces the formal parameter with the actual parameters.

The synchronization mechanism (Rule *Com*) works by merging two VCCs. When doing that, note that the link chains can be enlarged (Definition 3.10) and hence, the links of one chain can be placed in an admissible position of the other chain. Note that the decision about the length of the resulting chain is postponed until the use of the rule *Com*. This is a different approach from the one considered in [4] (for CNA processes) where the size of the interaction must be inferred in the *Act* rule (by enlarging in this rule ℓ with $\bowtie$). We also note that contrary to CCS, the Rule *Com* in CCNA (and CNA) can appear several times in the proof tree of a transition since the merging operator can always inject more virtual links to allow other agents to participate as shown in the following example.

Example 3.11 *Consider the CLM* $\mathcal{K}_{\mathbb{N}}$ *and the processes:*

$$P = {}^{\tau}\backslash_{a}\langle !2\rangle(?\mathfrak{a} \leq 10).P' \qquad Q = {}^{a}\backslash_{b}\langle !3\rangle(?\mathfrak{a} \leq 12).Q' \qquad R = {}^{b}\backslash_{\tau}\langle !5\rangle(?\mathfrak{a} \geq 4).R'$$

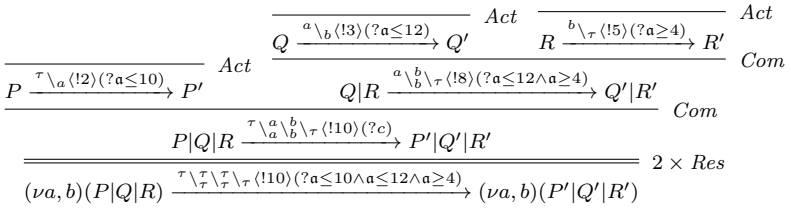


Fig. 2. Derivation in Example 3.11.

representing three agents interested in building a house. Each of them has a cost for her services (e.g., P charges \$2). Moreover, P is not willing to participate in a project that costs more than \$10 and R does not participate in "small" projects with a cost below \$4. P requires someone providing a service matching its output link a. Q offers a and expects in exchange b and R provides b. The three agents can indeed engage in the project as the derivation in Figure 3.2 shows. Note that $(\nu a, b)^{\tau}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_{\tau} = {}^{\tau}\backslash^{\tau}_{\tau}\backslash^{\tau}_{\tau}\backslash_{\tau}$. Also, in all the steps of this derivation the labels of the transitions are valid VCCs. Rule Com is used twice in the derivation, thus resulting in a 3-party interaction.

If we define P_2 as ${}^{\tau}\backslash_{a}\langle!2\rangle(?\mathfrak{a} < 10).P_2'$ the process $(\nu a, b)(P_2|Q|R)$ does not have any transition since $s\langle!10\rangle(?\mathfrak{a} < 10 \wedge \mathfrak{a} \leq 12 \wedge \mathfrak{a} \geq 4)$ is not a valid VCC. Moreover the names a, b are restricted and none of the processes in $P_2|Q|R$ can evolve independently using the rules Lpar and Rpar (e.g., a is not matched in ${}^{\tau}\backslash_{a}$ and then, $(\nu a)^{\tau}\backslash_{a}$ is not valid). In words, P_2 refuses a 3-party interaction with Q and R since the project will cost more than she expects. □

Example 3.12 (Conditionals) *Given an atomic constraint $c = e_1 \star e_2$, let $\widehat{c}$ denote the atomic constraint $e_1 \widehat{\star} e_2$ where $\widehat{\star}$ substitutes $=$ with $\neq$, $\preceq$ with $\succ$, $\prec$ with $\succeq$, etc. It is straightforward to see that: (1) $\widehat{\widehat{c}} = c$; and (2) given a ground atomic constraint c, $\mathcal{K} \models c$ iff $\mathcal{K} \not\models \widehat{c}$. We can define a conditional construct to select the continuation of a process depending on the entailment of a constraint. More precisely, if $c = c_1 \wedge \cdots \wedge c_n$ where each c_i is an atomic constraint, we can write $\ell\langle!e\rangle(\texttt{if } ?c \texttt{ then } P \texttt{ else } Q)$ to denote the process $\ell\langle!e\rangle(?c).P + \ell\langle!e\rangle(?\widehat{c_1}).Q + \cdots + \ell\langle!e\rangle(?\widehat{c_n}).Q$. Note that P is executed whenever all c_i hold (and hence c holds) and Q is executed if there is a c_i that does not hold in the underlying CLM.* □

Definition 3.13 (Computations) *Let W^* be the set of finite and infinite sequences of valid VCCs. Let P be a process and $\sigma = s_1.s_2... \in W^*$ be an infinite sequence. We say that σ is a computation of P if $P = P_0 \xrightarrow{s_1} P_1 \xrightarrow{s_2} P_2 \xrightarrow{s_3} \cdots$. If P cannot afford any transition, we shall write $P \not\longrightarrow$ and we call P a* dead-lock. *If $\sigma = s_1.s_2...s_n \in W^*$ is a finite sequence, we say that σ is a (finite) computation of P if $P = P_0 \xrightarrow{s_1} P_1 \cdots P_{n-1} \xrightarrow{s_n} P_n \not\longrightarrow$. In both cases we shall write $P \xrightarrow{\sigma}$. We shall use $\mathcal{O}(P) \subseteq W^*$ to denote the set $\{\sigma \mid P \xrightarrow{\sigma}\}$.*

3.3 Network bisimulation

Now we define a behavioral equivalence on CCNA processes that we show to be a congruence. In the tradition of CNA, we do not distinguish between processes that exhibit different internal transitions. This is reflected in the following extension of the equivalence relation $\rhd\lhd$ originally proposed in [4].

Definition 3.14 (Equivalence $\rhd\lhd$) *We let $\rhd\lhd$ be the least equivalence relation over link chains closed under the following inference rules:*

$$\frac{s \blacktriangleright\!\blacktriangleleft s'}{s \rhd\lhd s'} \qquad\qquad s_1 \ {}^{\alpha}\backslash^{\tau}_{\tau}\backslash_{\beta} \ s_2 \rhd\lhd s_1 \ {}^{\alpha}\backslash_{\beta} \ s_2$$

We lift such relation to VCC as follows: $s\langle !e\rangle(?c) \rhd\lhd s'\langle !e\rangle(?c')$ iff $s \rhd\lhd s'$ and $c \approx c'$.

Definition 3.15 (Network Bisimulation) *A* network bisimulation $\mathbf{R}$ *is a binary relation over CCNA processes such that, if $P\ \mathbf{R}\ Q$ then:*

- *if $P \xrightarrow{\mu} P'$, then $\exists\ \phi$, Q' such that $\mu \rhd\lhd \phi$, $Q \xrightarrow{\phi} Q'$, and $P'\ \mathbf{R}\ Q'$;*
- *if $Q \xrightarrow{\mu} Q'$, then $\exists\ \phi$, P' such that $\mu \rhd\lhd \phi$, $P \xrightarrow{\phi} P'$, and $P'\ \mathbf{R}\ Q'$.*

P and Q are network bisimilar*, notation $P \sim Q$, if there exists a network bisimulation $\mathbf{R}$ s.t. $P\ \mathbf{R}\ Q$.*

Following standard techniques (see e.g., [12,23]), we can show that $\sim$ is an equivalence relation and it is the largest network bisimulation relation.

Let us give some illustrative examples with the structure $\mathcal{K}_{\mathbb{N}}$. For any P, $\mathbf{0} \sim \ell\langle !3\rangle(?\mathfrak{a} \leq 2).P$ since $\mathcal{K}_{\mathbb{N}} \not\models 3 \leq 2$. Moreover, merging constraints cannot make $\mathfrak{a} \leq 2$ valid (due to intensiveness of $\otimes$ w.r.t. $\preceq$, i.e., $\mathfrak{a}$ will be always greater than 3).

The processes $P = \ell\langle !3\rangle(?\mathfrak{a} \leq 5)$ and $Q = \ell\langle !3\rangle(?\mathfrak{a} \leq 7)$ cannot be considered equivalent: Q can, for instance, synchronize with $R = \ell'\langle !4\rangle$ while P cannot. Now consider $P = \ell\langle !2\rangle(?\mathfrak{a} \leq 2)$ and $Q = \ell\langle !4\rangle(?\mathfrak{a} \leq 4)$. Note that both processes can synchronize with a process of the form $R = \ell'\langle !0\rangle(?c)$. However, if $c = \mathfrak{a} \leq 2$, P and R can synchronize but Q and R cannot.

Next we show that $\sim$ is a congruence relation and then, we can replace "equals by equals" in any context. For that, let $\mathcal{C}[\cdot]$ denote a process expression with a single occurrence of a hole $[\cdot]$. Moreover, if P is a process, $\mathcal{C}[P]$ denotes a process expression resulting from the substitution of the hole $[\cdot]$ with P.

Theorem 3.16 (Congruence) *If $P \sim Q$ then, for any context $\mathcal{C}[\cdot]$, $\mathcal{C}[P] \sim \mathcal{C}[Q]$.*

The above theorem is proved by exhibiting appropriate network bisimulations for any case/context. The complete proof is in the appendix.

4 Applications: fairness and constrained interactions

In this section we give three compelling examples showing how to declaratively control multiparty interactions by means of constraints. The first example is the

canonical problem of the dinning philosophers. In this case, by adding constraints, we are able to specify a deadlock free and fair solution for the problem. The second example models a network transportation system where constraints may represent costs or temporal restrictions. In our last example, constraints are used to model service level agreements in a negotiation protocol.

4.1 The dinning philosophers

The classical example of the dining philosophers (DP) has been introduced to study interactions between concurrent entities that want to share some resources. The problem relates n philosophers sitting around a table, where each one has its own dish, and they can only eat or think. When they, independently, decide to eat, they need two forks. On the table, there is only one fork between two dishes, i.e., exactly n forks.

It is well known there is no symmetric and deadlock-free specification of this system using only binary interactions [12] as in, e.g., CCS. Let us illustrate the problem considering only two philosophers. The philosophers are specified as the CCS process $P_i = f_i.f_{(i+1)mod\ 2}.\overline{eat_i}.P_i'$ where P_0 first grabs the fork 0, then he grabs the fork 1 to later start eating (similarly for P_1). If we run in parallel P_0 and P_1 along with the processes specifying the two forks ($F_i = \overline{f_i}.F_i'$), the system can reach a deadlock when P_0 takes the fork 0 and P_1 takes the fork 1.

By using a multiparty synchronization calculus, the DP problem has a simple and very natural deadlock free specification (see [7,8] for a solution using CNA and [12] for a solution using Multi-CCS). In that case, in an atomic (or multiparty) interaction, a philosopher takes at the same time both forks, thus avoiding the deadlock situation described above. However, the solutions in Multi-CCS and CNA may exhibit unfair computations where, e.g., a given philosopher eats or thinks all the time (and the others cannot progress).

Definition 4.1 (Fairness [24]) *Let π be an infinite computation, $\pi = P_0 \xrightarrow{s_1} P_1 \xrightarrow{s_2} P_2 \xrightarrow{s_3} \cdots$. A VCC μ is relentlessly enabled in π if $\forall\ \pi''$, π' s.t. $\pi = \pi''\pi'$, π' contains a process P_i that can afford a transition labeled with μ. Moreover, π is strongly fair if each relentlessly enabled VCC μ on π' occurs in π'.*

In words, a computation π is strongly fair if an action (VCC) that is relentlessly enabled in π, occurs infinitely often in π.

Here we focus on a fair solution for the DP problem: due to deadlock-freeness, every computation is infinite and, by fairness, in every computation each philosopher eats infinitely many times. For that, we use constraints to neatly implement a sort of ticket service, thus guaranteeing that philosophers must alternate the use of the forks. From now on, we fix the CLM to be the structure $\mathcal{K}_{\mathbb{N}}$.

Below we describe our first attempt to solve the problem. Unfortunately, the specification is deadlock-free but it is not fair. We shall use $DP(n)$ to denote the instance of the DP problem with n philosophers and i_n^+ to denote $(i+1)mod\ n$.

Example 4.2 (Dinning Philosophers) *Let $n \geq 2$ be the number of philosophers (and forks) in the problem and define $Fork_i$ with $i \in [0, n)$ as follows:*

$$Fork_i(l,r) \triangleq {}^{\tau}\backslash_{\tau}(?l = 0 \wedge r = 0).Fork_i\langle N, N\rangle$$
$$+ {}^{\tau}\backslash_{upL_i}(?l > 0).{}^{\tau}\backslash_{dw_i}.Fork_i\langle l\text{-}1, r\rangle \quad + \quad {}^{upR_i}\backslash_{\tau}(?r > 0).{}^{dw_i}\backslash_{\tau}.Fork_i\langle l, r\text{-}1\rangle$$

In this specification, N is a global parameter/constant of the model indicating how many times we allow a philosopher to consecutively use the forks. The parameter l of the definition is the maximum number of times the philosopher on the left can use the i-th fork. Similarly, for the parameter r. Every time the i-th fork is used by the philosopher on its left (resp. right), the parameter l (resp. r) is decremented (using $\div$) by 1. The process $Fork_i(l,r)$ can reset its parameters to the initial values only when both l and r are equal to zero. The values of the parameters are checked by the constraints associated to the prefix ${}^{\tau}\backslash_{upL_i}$ (resp. ${}^{upR_i}\backslash_{\tau}$) that allow the philosopher on the left (resp. right) to grab the fork.

The specification of the philosophers is as follows:

$$Phil_i() \triangleq {}^{\tau}\backslash_{tk_i}.Phil_i\langle\rangle + {}^{upL_i}\backslash_{upR_{i^+_n}}.{}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i^+_n}}.Phil_i\langle\rangle$$

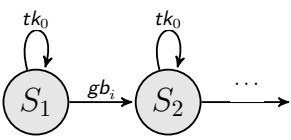
Hence, the process $Phil_i\langle\rangle$ can either think or establish a 3-party synchronization with the forks on his right and on his left. In that case, he can eat to later release both forks in another 3-party synchronization. In fact, the release of the forks can be done independently and there is no need for a multiparty synchronization.

The whole system restricts all the channel names but eat_i which is a visible action:

$$DP = (\nu\, \widetilde{upL}_i, \widetilde{upR}_i, \widetilde{dw}_i)(Phil_0 | \ldots | Phil_{n-1} | Fork_0(N,N) | \ldots | Fork_{n-1}(N,N))$$

Here $\widetilde{upL}_i, \widetilde{upR}_i, \widetilde{dw}_i$ stand for the sets of channel names used in $Phil_i$ and $Fork_i$ with $i \in [0, n\text{-}1)$. □

The transition system generated by the process DP (that can be computed with our tool, see Section 5) is indeed deadlock free. Moreover, once P_i has used the forks N times, he has to wait until his neighbors eat also N times to be able to eat again. This means that P_i cannot take control of the forks forever and, at some point, he has to wait for the others. In other words, there are no computations where, e.g., P_i eats infinitely many times and P_j can never grab the forks.



In this model, however, we cannot prove that P_i can eat infinitely many times. The problem is the *thinking* action: there is an infinite computation where, e.g., P_0 is always thinking and nobody is eating. Such computation corresponds to the loop on state S_1 in the abstract version of the transition system in the figure on the left. In this loop, the action gb_i, representing P_i grabbing the forks,

is always enabled. This situation can be interpreted as "P_i has the potential of grabbing the forks but the scheduler never gives him the chance to do it".

The fair system can be obtained by controlling also the thinking action. Similar to the solution in [12], we can enforce that philosophers must eat after thinking, thus alternating between thinking and eating states. This is the purpose of moving to the state $Phil'_i$ after exhibiting the think action in the model below.

Example 4.3 (Fair alternating system) *Consider the processes definitions Fork and DP in Example 4.2 where the definition of $Phil_i$ is modified as follows:*

$$Phil_i() \triangleq {}^{\tau}\backslash_{tk_i}.Phil'_i\langle\rangle + {}^{upL_i}\backslash_{upR_{i_n^+}}.{}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_i\langle\rangle$$

$$Phil'_i() \triangleq {}^{upL_i}\backslash_{upR_{i_n^+}}.{}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_i\langle\rangle$$

For illustration, consider a possible transition where the philosopher n-1 takes both forks to later eat and release the forks:

$$DP \xrightarrow{(\nu\,\widetilde{n})\,{}^{\tau}\backslash^{upL_{n-1}}_{upL_{n-1}}\backslash^{upR_0}_{upR_0}\backslash_{\tau}(?1>0\,\wedge\,?1>0)} DP_1 \xrightarrow{{}^{\tau}\backslash_{eat_{n-1}}} DP_2 \xrightarrow{(\nu\,\widetilde{n})\,{}^{\tau}\backslash^{dw_{n-1}}_{dw_{n-1}}\backslash^{dw_0}_{dw_0}\backslash_{\tau}} DP_3$$

where $\widetilde{n} = \widetilde{upL_i}, \widetilde{upR_i}, \widetilde{dw_i}$; DP_1 is as DP but $Phil_0$ is replaced with ${}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_0\langle\rangle$; DP_2 is as DP but $Phil_0$ is replaced with ${}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_0\langle\rangle$; and, finally,

$$DP_3 = (\nu\,\widetilde{upL_i}, \widetilde{upR_i}, \widetilde{dw_i})(Phil_0|\ldots|Phil_{n-1}|Fork_0(N\text{-}1,N)|\ldots|Fork_i(N,N)|\ldots|Fork_{n-1}(N,N\text{-}1))\,.$$

Note the new state of the forks 0 and n-1, namely, $Fork_0(N\text{-}1,N)$ and $Fork_{n\text{-}1}(N,N\text{-}1)$. □

We can show that the process DP in the above example produces an alternating execution between the philosophers. For concreteness, consider only two philosophers and let DP_W be as $DP(2)$ where we ignore the *think* actions, i.e., we consider only the process $Phil'_i()$ calling to itself, instead of calling the process $Phil_i\langle\rangle$. We can define the following specification:

$$Spec() \stackrel{\texttt{def}}{=} {}^{\tau}\backslash_{eat_0}.{}^{\tau}\backslash_{eat_1}.Spec\langle\rangle \quad + \quad {}^{\tau}\backslash_{eat_1}.{}^{\tau}\backslash_{eat_0}.Spec\langle\rangle$$

stating that philosophers must alternate when $N = 1$. We can then prove the equivalence $D_W \sim Spec\langle\rangle$.

Let us now show that fairness holds even considering the *thinking* action. Without loss of generality, let $N = 1$. For readability, let us give a more abstract representation of the state of the forks as a ring of tuples of the form $\mathcal{S}_n = \langle a_0, a_1\rangle\langle a_1, a_2\rangle\langle a_2, a_3\rangle\ldots\langle a_{n-1}, a_0\rangle$ where $a_i \in \{0, 1, ?\}$. Such ring is subject to the following transition rules:

- **Grab**: $\ldots\langle x, 1\rangle\langle 1, y\rangle\cdots \Rightarrow \ldots\langle x, ?\rangle\langle ?, y\rangle\ldots$

  Grab-0: $\langle 1, x\rangle\ldots\langle y, 1\rangle \Rightarrow \langle ?, x\rangle\ldots\langle y, ?\rangle$
- **Grab-0-end**: $\langle ?, x\rangle\ldots\langle y, ?\rangle \Rightarrow \langle 0, x\rangle\ldots\langle y, 0\rangle$

  Grab-end: $\ldots\langle x, ?\rangle\langle ?, y\rangle\cdots \Rightarrow \ldots\langle x, 0\rangle\langle 0, y\rangle\ldots$
- **Reset**: $\ldots\langle 0, 0\rangle\cdots \Rightarrow \ldots\langle 1, 1\rangle\ldots.$

The state of the fork i is represented by the i-th tuple $\langle l, r\rangle$ where l=1 (resp. l=0)

means that the left philosopher may (resp. cannot) take this fork and $l =?$ is the intermediate state where the philosopher is eating to later release the forks (thus abstracting the steps DP_1 and DP_2 in Example 4.3). The transition **Grab** (and **Grab-0** for $Phil_0$) abstracts the operational step from DP to DP_1 and **Grab-end** (and **Grab-0-end** for $Phil_0$) the transition from DP_1 to DP_3. Moreover, due to the definition of *Fork* in Example 4.2, from the state $\langle 0, 0\rangle$ the only possible transition for the fork is to reset its parameters leading to the state $\langle 1, 1\rangle$.

Next lemma proves that, for all philosopher P_i, *always is the case that P_i will eventually eat.* More precisely,

Lemma 4.1 (fairness) *Let $n \geq 2$, $N \geq 1$ and $DP(n)$ be the n-dining philosopher system formalized in Example 4.3. Then, $\mathcal{O}(DP) \neq \emptyset$ and, for all $\sigma \in \mathcal{O}(DP)$:*

(i) ***Deadlock-freeness**: σ is an infinite sequence ;*

(ii) ***Fairness**: for all $i \in [0, n)$, the label ${}^{\tau}\backslash_{eat_i}$ appears infinitely often in σ.*

Proof. Let us give a sketch of the proof (more details in the appendix and an automatic proof for a specific instance of n in the next section). For deadlock-freeness, consider the state $\mathcal{S}_n = \langle a_0, a_1\rangle\langle a_1, a_2\rangle\langle a_2, a_3\rangle \dots \langle a_{n-1}, a_0\rangle$ where $a_i \in \{0, 1, ?\}$. If there is an a_i s.t. $a_i \in \{1, ?\}$, then, it is always possible to apply the rules **Grab** or **Grab-end** to make a transition. Otherwise (i.e., if $a_i = 0$ for all $i \in 0..n-1$) then, it is always possible to apply the **Reset** rule. The proof of fairness follows by contradiction. Let σ be a computation of $DP(n)$ such that $0 < m \leq n$ philosophers will never perform the eating action. This means that in σ, the state includes tuples of the form $\cdots\langle i, 1\rangle\langle 1, j\rangle\cdots$ where the 1's remain the same (i.e., these values never become 0 due to an application of **Grab**). We can show that σ must be finite and, indeed, there will be a maximum number of transitions before getting a deadlock (thus a contradiction since σ must be infinite). □

A direct consequence of Lemma 4.1 is given by the following corollary.

Corollary 4.1 (Starvation freedom) *Let $n \geq 2$, $N \geq 1$ and $DP(n)$ be a n-dining philosopher system formalized as in Example 4.3. In every (infinite) computation, the **Grab-end** transition (the action after eating) occurs infinitely often for each adjacent tuple.*

Note that in Example 4.3 we can specify different constants N_i for the parameters of the process $Fork_i\langle l_i, r_i\rangle$, with the restriction that $r_i = l_{i_n^+}$, i.e. the number of times that two consecutive forks are used by their common adjacent philosopher must be the same. This is useful when we are modeling systems in which there are different priorities for the use of the resources. Also in situations where the network of agents is not completely balanced (since some of them may work faster than others).

In the next example, we show that values and constraints can be useful to specify, declaratively, the internal state of processes. For that, we consider the case where philosophers may decide to remain thinking for a while and then, they decide to eat. In this scenario, it is important for the system that philosophers moving to the thinking state do not block the activities of the others. We thus assume

that philosophers take the decision (of thinking or eating again) and such decision must be communicated to the two adjacent forks. In turn, the forks may perform synchronizations with philosophers that have already consumed all their N usages whenever the adjacent philosopher is in the thinking state.

Example 4.4 (Constraints as states) *Consider the following specification for the philosophers:*

$$Phil_i() \stackrel{\texttt{def}}{=} {}^{idlL_i}\backslash_{idlR_{i_n^+}}.Phil\text{-}Idle_i\langle\rangle + {}^{upL_i}\backslash_{upR_{i_n^+}}.Phil\text{-}Eat_i\langle\rangle$$

$$Phil\text{-}Eat() \stackrel{\texttt{def}}{=} {}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_i\langle\rangle$$

$$Phil\text{-}Idle_i() \stackrel{\texttt{def}}{=} {}^{tau}\backslash_{tk_i}.Phil\text{-}Idle_i\langle\rangle + {}^{wL_i}\backslash_{wR_i}.{}^{upL_i}\backslash_{upR_{i_n^+}}.Phil\text{-}Eat_i\langle\rangle$$

In $Phil_i$ we see two new sets of names and links: $idlL_i$ and $idlR_i$ (resp. wL_i and wR_i) are used to synchronize with both forks and communicate the fact that the philosopher goes to the thinking state (resp. starts eating again). The model for the forks is as follows:

$$\begin{aligned} Fork_i(l,r,idll,idlr) \stackrel{\texttt{def}}{=} {}& {}^{idlR_i}\backslash_{\tau}.Fork_i\langle l,r,idll,1\rangle + {}^{\tau}\backslash_{idlL_i}.Fork_i\langle l,r,1,idlr\rangle + \\ & {}^{wR_i}\backslash_{\tau}.Fork_i\langle l,r,idll,0\rangle + {}^{\tau}\backslash_{wL_i}.Fork_i\langle l,r,0,idlr\rangle + \\ & {}^{\tau}\backslash_{upL_i}(?l=0\wedge idlr=1).{}^{\tau}\backslash_{dw_i}.Fork_i\langle l,r,idll,idlr\rangle + \\ & {}^{upR_i}\backslash_{\tau}(?r=0\wedge idll=1).{}^{dw_i}\backslash_{\tau}.Fork_i\langle l,r,idll,idlr\rangle + \\ & \textit{the 3 choices in Example 4.2} \end{aligned}$$

The process $DP(n)$ is defined as usual, adding the new names in the set of restricted names.

In $Fork_i$, besides l and r, we also have parameters to determine the current state of the left and right philosophers. The first line in the definition allows for communicating the decision of going to the thinking/idle state (for the right and left philosopher). Similarly, the second line is used to communicate the intention of start eating again. Most importantly, the third line allows for a synchronization with the left philosopher even if $l = 0$. In that case, the right philosopher must be in the idle state. Similarly for the forth line. In this system we cannot prove fairness as in Lemma 4.1 (since there are unfair always-*thinking* computations). We can assume (externally) a fairness condition ruling out such computations or specify a more controlled version of *Phil-Idle*$_i$ that, for instance, "counts" and controls the number of *thinking* actions.

Before closing this section, let us note that all the solutions presented here satisfy the usual requirements for this problem: *fully distribution* (there is no central agent coordinating the activities of the philosophers) and *symmetry* (all philosophers and forks are identical). The control of the agents defined here rely completely on their internal state and all of them are symmetrically defined as $Phil_i$ and $Fork_i$. If we dispense with symmetry, there is a simple solution for the problem in CCS where P_0 grabs first the fork on his left (F_0) and P_1 grabs first the fork on his right (again, F_0). Hence, there is no a deadlock in this asymmetric specification as the one described

in the beginning of this section. As pointed out in [12], the solution for the problem in Multi-CCS (as well as ours in CCNA) is fully distributed in an abstract level: there is no a central shared memory. However, it is not possible to have a truly distributed deterministic implementation of this kind of multiparty synchronization mechanisms [15].

4.2 The network transportation system

The following example is a simplified version of the network transportation system presented in [7] where each transportation system has a specific cost. Passengers may specify a threshold for the value they are willing to pay for a trip starting and ending at the required stations. For simplicity, we are not considering the model of the stations. Here we model a complete trip of the passenger as a single transition that also records all the data concerning the trip (i.e. the sum of the costs of the used means of transport).

Example 4.5 (Network Transportation System) *Consider the following definitions:*

$$P = {}^{\tau}\backslash_{s_1} \mid {}^{s_3}\backslash_{\tau}(?\mathfrak{a} \leq 5) \qquad MoT(s1, s2, c) \triangleq {}^{s1}\backslash_{s2}\langle !c\rangle.MoT$$

$$M1 = MoT(s1, s2, 3) \qquad M2 = MoT(s2, s3, 2)$$

$$T1 = MoT(s1, s3, 7)$$

$$System = (\nu \boldsymbol{s})(P \mid M1 \mid M2 \mid T)$$

Here, P is willing to go from s_1 to s_3. She offers its links for free but she constraints synchronizations to have cost at most 5. On the other side, $M1$ does not impose any constraint but it forces the final agreement to have cost at least 3. In this system, there is only one possible transition, namely,

$$System \xrightarrow{(\nu \boldsymbol{s})\, {}^{\tau}\backslash^{s1}_{s1}\backslash^{s2}_{s2}\backslash^{s3}_{s3}\backslash_{\tau}\langle !5\rangle(?5\leq 5)} (\nu \boldsymbol{s})(M1 \mid M2 \mid T)$$

where P has to synchronize with both $M1$ and $M2$ (and pay 5). Note that P cannot take the train T since the chain ${}^{\tau}\backslash^{s1}_{s1}\backslash^{s3}_{s3}\backslash_{\tau}\langle !7\rangle(?7 \leq 5)$ is not valid (Def. 3.10).

We can model the situation in which the passengers have two kinds of constraints: cost and time. In this case, values are tuples (Notation 3.9) of the form $\langle cost, time\rangle$. Each means of transport offers services at a given cost and speed and passengers may pose constraints on those values. Furthermore, adding a third element to the tuple, we can also restrict the number of connections a passenger is willing to accept (see Example 3.8).

4.3 Service Level Agreements (SLA)

We propose an extended model for the Service Level Agreements (SLA) protocol specified in [9]. In this kind of protocols, before the effective provisioning of a service, the involved parties should agree on a set of parameters, such as the cost the client should pay or the service quality the provider is willing to offer.

Example 4.6 (SLA Protocol) *Here we consider a client C asking a web hosting provider P the use of a service. P, in turn, can offer the service once it receives the availability of the bandwidth from a third party T. Hence we shall consider tuples of the form $\langle cost, bw\rangle$ (see Notation 3.9). The client is modeled as*

$$C \triangleq {}^{\tau}\backslash_{s}(?cost < Max_C \wedge bw > Min_B).C$$

where Max_C (maximal cost) and Min_B (minimal bandwidth) are constants for the model. We may have several $T's$ offering different options for the provider, for instance:

$$T_1 \triangleq {}^{th}\backslash_{\tau}\langle !25, 100\rangle.T_1 \qquad T_2 \triangleq {}^{th}\backslash_{\tau}\langle !17, 70\rangle.T_2 + {}^{th}\backslash_{\tau}\langle !32, 130\rangle.T_2\,.$$

Here, T_1 has only one option of service (at cost 25) while T_2 offers two possibilities of bandwidth (70 and 130) at different costs. The provider P charges an extra fee depending on the bandwidth availability that he has received from T:

$$P \triangleq {}^{s}\backslash_{th}\langle !2, 0\rangle(?cost < 60).P \quad + {}^{s}\backslash_{th}\langle !3, 0\rangle(?60 \leq cost < 100).P \quad + {}^{s}\backslash_{th}\langle !5, 0\rangle(?cost \geq 100).P$$

The system is $SLA \triangleq (\nu\, s, th)(P \mid C \mid T_1 \mid T_2)$ and a possible transition is

$$SLA \xrightarrow{(\nu\, s,th)\, {}^{\tau}\backslash_{s}^{s}\backslash_{th}^{th}\backslash_{\tau}\langle !20\rangle(?c)} SLA$$

where $c = 20 < Max_C \wedge 70 > Min_B \wedge 60 \leq 70 < 100$. What we are observing is a synchronization between P, C and the first option of T_2 (and thus, the final cost is 20). □

5 Concluding Remarks

On top of the tool described in [8], we have implemented a rewriting logic [16] specification of the semantics proposed here in the Maude System. The tool is available at `https://gitlab.com/carlos_olarte/SiLVer`. We built a suitable signature for the syntax of CCNA processes and specified the operational rules as rewriting rules. We profit from the symbolic techniques proposed in [8] to efficiently check when two or more links can be combined into a valid link chain. Using this tool, we can check for instance that for a particular instance of n, all the systems $DP(n)$ discussed in Section 4.1 are deadlock free. For that, it is just a matter to ask whether there is a reachable state without transition (i.e., a normal form, "$\Rightarrow !$"): `search [1] DP(2) =>! S:State .`. The answer is `No solution.` telling us that such state does not exist, thus proving deadlock freeness for $DP(2)$. More interestingly, we can

verify the fairness condition in Lemma 4.1. For that, we use the model checker in Maude and ask if the property $\Box\Diamond\{^{\tau}\backslash_{eat_0}\}$ is valid. Here $\Box$ and $\Diamond$ are the linear time temporal logic (LTL) modalities "always" and "eventually". The answer is `true` for the system in Example 4.3 and `false` (with a suitable counterexample) for the other models. We can also verify *safety* for all the systems. For that, we can ask if there is a state reachable from $DP(2)$ where both $^{\tau}\backslash_{eat_0}$ and $^{\tau}\backslash_{eat_1}$ are enabled. The answer is `No solution.`. The tool also offers facilities to traverse the transition system, generate traces and produce a DOT file (graph description language) with the resulting transition system.

We are currently working on an extension of the Symbolic Link Modal Logic proposed in [8] to offer mechanisms to specify properties involving constraints. This should allow us to state properties such as "the process P cannot exhibit a n-party synchronization with $n > N$" or "the server will never admit more connections that its bandwidth-limit allows". Coupling this logic with the already implemented infrastructure for model checking in Maude will provide more (automatic) verification techniques to reason about CCNA specifications. It would be also interesting to explore a wider range of behavioral equivalences including weak-network-bisimulation and also stronger versions of network-bisimulations (where, e.g, $\rhd\lhd$-related link chains are not identified). Efficient decision procedures for those equivalences have not been explored yet.

In the examples presented here, for the sake of uniformity, we have used only one CLM ($\mathcal{K}_{\mathbb{N}}$). There are many choices for it (see e.g., [11,3]). For instance, consider the structures $\mathcal{K}_P = \langle [0,1], \leq, \times\rangle$ and $\mathcal{K}_F = \langle [0,1], \leq, min\rangle$. In the first one, the subscript "P" is for *probability* and agents accumulate values in the real interval $[0,1]$ by multiplying them. Hence, the accumulated value gets closer to 0 when more agents are involved in an interaction. In the second structure the "F" stands for *fuzzy*, where values are accumulated by choosing the minimal value. In this case, agents can define a threshold for interaction based on their preferences expressed as values in the interval $[0,1]$. As pointed out in Lemma 3.3, it is possible to combine such structures to obtain richer ones. Some examples using those structures are in the tool's web page.

Multiparty calculi with different synchronization mechanisms have been proposed, e.g., in CSP [13] and full Lotos [22]. These calculi offer parallel operators that exhibit a set of action names (or channel names), and all the parallel processes offering that action (or an action along that channel) can synchronize by executing it. In [20], a binary form of input allows for a three-way communication. The reader may also refer to [14] where it is shown that CCS^n (or n-join CCS), an extension of CCS that allows prefixes to synchronize with at most n outputs, is strictly more expressive than CCS^{n-1}. The multiparty calculus most related to CNA is in [19], where links are named and are distinct from the usual input/output actions: there is one sender and one receiver (the output includes the final receiver name). Finally, we mention the cc-π calculus [9] that combines the name-passing discipline of the π-calculus with constraints in the style of concurrent constraint programming (see a survey in [21]). This calculus does not offer multiparty synchronization and its

semantics is necessarily more involved due to the name-passing discipline of the π-calculus. As showed here, constraints in CCNA allows for a declarative control of processes in a very natural way.

References

[1] Abadi, M. and A. D. Gordon, *A calculus for cryptographic protocols: The spi calculus*, Inf. Comput. **148** (1999), pp. 1–70.
URL https://doi.org/10.1006/inco.1998.2740

[2] Bistarelli, S. and F. Gadducci, *Enhancing constraints manipulation in semiring-based formalisms*, in: G. Brewka, S. Coradeschi, A. Perini and P. Traverso, editors, *ECAI 2006*, Frontiers in Artificial Intelligence and Applications **141** (2006), pp. 63–67.
URL http://www.booksonline.iospress.nl/Content/View.aspx?piid=1647

[3] Bistarelli, S., U. Montanari and F. Rossi, *Semiring-based contstraint logic programming: syntax and semantics*, ACM Trans. Program. Lang. Syst. **23** (2001), pp. 1–29.
URL https://doi.org/10.1145/383721.383725

[4] Bodei, C., L. Brodo and R. Bruni, *Open multiparty interaction*, in: N. Martí-Oliet and M. Palomino, editors, *WADT 2012, Revised Selected Papers*, Lecture Notes in Computer Science **7841** (2012), pp. 1–23.
URL http://dx.doi.org/10.1007/978-3-642-37635-1_1

[5] Bodei, C., L. Brodo and R. Bruni, *A formal approach to open multiparty interactions*, Theor. Comput. Sci. **763** (2019), pp. 38–65.
URL https://doi.org/10.1016/j.tcs.2019.01.033

[6] Bodei, C., L. Brodo, R. Bruni and D. Chiarugi, *A flat process calculus for nested membrane interactions*, Sci. Ann. Comp. Sci. **24** (2014), pp. 91–136.
URL http://dx.doi.org/10.7561/SACS.2014.1.91

[7] Brodo, L. and C. Olarte, *Symbolic semantics for multiparty interactions in the link-calculus*, in: B. Steffen, C. Baier, M. van den Brand, J. Eder, M. Hinchey and T. Margaria, editors, *SOFSEM 2017*, LNCS **10139** (2017), pp. 62–75.
URL https://doi.org/10.1007/978-3-319-51963-0_6

[8] Brodo, L. and C. Olarte, *Verification techniques for a network algebra*, Fundam. Inform. **172** (2020), pp. 1–38.
URL https://doi.org/10.3233/FI-2020-1890

[9] Buscemi, M. G. and U. Montanari, *Cc-pi: A constraint-based language for specifying service level agreements*, in: R. De Nicola, editor, *ESOP 2007*, LNCS **4421** (2007), pp. 18–32.
URL https://doi.org/10.1007/978-3-540-71316-6_3

[10] Cardelli, L., *Brane calculi*, in: V. Danos and V. Schächter, editors, *CMSB 2004*, LNCS **3082** (2004), pp. 257–278.
URL https://doi.org/10.1007/978-3-540-25974-9_24

[11] Gadducci, F., F. Santini, L. F. Pino and F. D. Valencia, *Observational and behavioural equivalences for soft concurrent constraint programming*, J. Log. Algebr. Meth. Program. **92** (2017), pp. 45–63.
URL https://doi.org/10.1016/j.jlamp.2017.06.001

[12] Gorrieri, R. and C. Versari, "Introduction to Concurrency Theory - Transition Systems and CCS," Texts in Theoretical Computer Science. An EATCS Series, Springer, 2015.
URL https://doi.org/10.1007/978-3-319-21491-7

[13] Hoare, C. A. R., "Communications Sequential Processes," Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[14] Laneve, C. and A. Vitale, *The expressive power of synchronizations*, in: *LICS 2010* (2010), pp. 382–391.
URL https://doi.org/10.1109/LICS.2010.15

[15] Lehmann, D. J. and M. O. Rabin, *On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem*, in: J. White, R. J. Lipton and P. C. Goldberg, editors, *POPL* (1981), pp. 133–138.
URL http://doi.acm.org/10.1145/567532.567547

[16] Meseguer, J., *Twenty years of rewriting logic*, J. Log. Algebr. Program. **81** (2012), pp. 721–781.
URL https://doi.org/10.1016/j.jlap.2012.06.003

[17] Milner, R., "Communication and Concurrency," International Series in Computer Science, Prentice Hall, 1989, sU Fisher Research 511/24.

[18] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, I and II*, Inf. Comput. **100** (1992), pp. 1–40.
URL https://doi.org/10.1016/0890-5401(92)90008-4

[19] Montanari, U. and M. Sammartino, *Network conscious pi-calculus: a concurrent semantics*, in: *MFPS 2012*, Electronic Notes in Theoretical Computer Science 286 (2012), pp. 291–306.

[20] Nestmann, U., *On the expressive power of joint input*, Electronic Notes in Theoretical Computer Science **16(2)** (1998).

[21] Olarte, C., C. Rueda and F. D. Valencia, *Models and emerging trends of concurrent constraint programming*, Constraints An Int. J. **18** (2013), pp. 535–578.
URL https://doi.org/10.1007/s10601-013-9145-3

[22] Peter H. J. van Eijk, M. D., Chris A. Vissers, "The Formal Description Technique LOTOS," North-Holland, 1989.

[23] Sangiorgi, D., "Introduction to Bisimulation and Coinduction," Cambridge University Press, 2011.

[24] van Glabbeek, R. and P. Höfner, *Progress, justness, and fairness*, ACM Comput. Surv. **52** (2019), pp. 69:1–69:38.
URL https://doi.org/10.1145/3329125

Appendix

A Congruence (Theorem 3.16)

In this section we prove that $P \sim Q$ implies that, for any context $\mathcal{C}[\cdot]$, $\mathcal{C}[P] \sim \mathcal{C}[Q]$. Recall that the (co-inductive) technique for showing that two processes are (network) bisimilar consists in exhibiting a network bisimulation $\mathcal{R}$ containing the two processes (see, e.g., [23]). Since $P \sim Q$, we know that there exists a network bisimulation $\mathcal{R}$ containing the pair (P, Q). For each context $\mathcal{C}[\cdot]$, we show a suitable $\mathcal{R}'$ s.t. $(\mathcal{C}[P], \mathcal{C}[Q]) \in \mathcal{R}'$.

- Case $\mu.[\cdot]$. The needed relation is $\mathcal{R}' = \{(\mu.P, \mu.Q) \mid \mu \in VCC\} \cup \mathcal{R}$. Clearly, $\mu.P$ can only perform μ and proceed as P. Similarly for $\mu.Q$. Since $(P, Q) \in \mathcal{R}$, $\mathcal{R}'$ is indeed a network bisimulation.
- Case $[\cdot] + R$. Note that P and Q must be sequential processes. Let $\mathcal{R}' = \mathcal{R} \cup \{(P + R, Q + R) \mid R \in \mathcal{P}\} \cup \mathcal{I}$ where $\mathcal{I}$ is the identity relation on $\mathcal{P}$ ($\mathcal{P}$ is the set of CCNA processes). There are two possible transitions for $P + R$. (1) If $P + R \xrightarrow{\mu} P'$ then, it must be the case that $P \xrightarrow{\mu} P'$. Hence, there exists Q' and $\mu' \bowtie \mu$ s.t. $Q \xrightarrow{\mu'} Q'$. We conclude by noticing that $Q + R \xrightarrow{\mu'} Q'$ and $(P', Q') \in \mathcal{R}'$. (2) If R moves, we observe $P + R \xrightarrow{\mu} R'$. Then, $Q + R \xrightarrow{\mu} R'$ and clearly $(R', R') \in \mathcal{R}'$. The case $R + [\cdot]$ follows similarly.
- Case $[\cdot]|R$. The needed network bisimulation is $\mathcal{R}' = \{(P|R, Q|R) \mid (P, Q) \in \mathcal{R} \text{ and } R \in \mathcal{P}\}$. The process $P|R$ exhibits 3 kind of transitions. $P|R$ moves to $P'|R$ with label μ using the rule *Lpar*. Hence, $P \xrightarrow{\mu} P'$ and there exists Q' and $\mu' \bowtie \mu$ s.t. $Q \xrightarrow{\mu'} Q'$ and $(P', Q') \in \mathcal{R}$. By using *Lpar*, $Q|R \xrightarrow{\mu'} Q'|R$ and clearly $(P'|R, Q'|R) \in \mathcal{R}'$ as needed. The case when R moves (using *Rpar*) is trivial. If P and R synchronizes (using *Com*), it must be the case that $P \xrightarrow{\mu} P'$,

$R \xrightarrow{\psi} R'$ and the label of the transition is $\mu \bullet \psi$. We also know that there exists Q' and $\xi \rhd\lhd \mu$ s.t. $Q \xrightarrow{\xi} Q'$ and $(P', Q') \in \mathcal{R}$. We can suitable enlarge (via $\blacktriangleright\!\!\blacktriangleleft$ and $\rhd\lhd$) the link chains in ψ and ξ to make $\psi \bullet \xi$ valid and $Q|R \xrightarrow{\xi\bullet\psi} Q'|R'$ as needed. The case $R|[\cdot]$ is similar.

- Case $(\nu a)[\cdot]$. The needed relation is $\mathcal{R}' = \{((\nu a)P, (\nu a)Q) \mid (P, Q) \in \mathcal{R} \text{ and } a \in VCC\}$. If $(\nu a)P \xrightarrow{(\nu a)\mu} (\nu a)P'$ it must be the case that $P \xrightarrow{\mu} P'$. Hence there exists Q' and $\mu' \rhd\lhd \mu$ s.t. $Q \xrightarrow{\mu'} Q'$ and $(P', Q') \in \mathcal{R}$. We conclude by noticing that $(\nu a)Q \xrightarrow{(\nu a)\mu'} (\nu a)Q'$ (for that, we can easily show that if $(\nu a)\mu$ is valid, then $(\nu a)\mu'$ is also valid for $\mu' \rhd\lhd \mu$.) and hence, $((\nu a)P', (\nu a)Q') \in \mathcal{R}'$ as needed. □

B Proof of fairness (Lemma 4.1)

For the sake of readability, we shall change marginally the notation of the reachable states (processes) from $DP(\mathrm{n})$. Note that the set of labels (ignoring the constraint `tt`) of the transition system generate from DP is $L = \{{}^{\tau}\backslash_{eat_i}, {}^{\tau}\backslash_{tk_i} \mid i \in [0, n)\} \cup \{{}^{\tau}\backslash_{\tau}{}^{\tau}\backslash_{\tau}{}^{\tau}\backslash_{\tau}\} \cup \{{}^{\tau}\backslash_{\tau}\}$. The first component corresponds to the (visible) eating and thinking actions; the second component to the 3-party interaction for grabbing and releasing the forks (see transitions in Example 4.3); and ${}^{\tau}\backslash_{\tau}$ to the reset action (first line in the definition of Fork in Example 4.2). Let us use eat_i, tk_i, $grab_i$, $release_i$ and $reset_i$ to denote such actions. For conciseness, we fix $N = 1$.

The process $Fork_i(l, r)$ and its sucessor states will be represented as the tuple $\langle l, r\rangle$. For that, note that a fork can be in one of the following states (see Example 4.2):

- $Fork_i(1, r)$ (resp. $Fork_i(l, 1)$) where it can synchronize with the philosopher on the right (resp. on the left). We shall write these states, respectively, as $\langle 1, r\rangle$ and $\langle l, 1\rangle$.
- ${}^{\tau}\backslash_{dw_i}.Fork_i\langle 0, r\rangle$ and ${}^{dw_i}\backslash_{\tau}.Fork_i\langle l, 0\rangle$. The first (resp. second) is the result after a synchronization with the philosopher on the left (resp. right). Let us denote those states as the tuples $\langle ?, r\rangle$ and $\langle l, ?\rangle$.
- $Fork_i(0, 0)$ (notation $\langle 0, 0\rangle$) where the only possible transition is a reset action leading to $\langle 1, 1\rangle$.

We can also simplify the notation to represent the philosophers. For that, note that they can be in one of the following states (see processes and transitions in Example 4.3):

- $Phil_i$, where he can grab the forks or think. We shall use GT_i to denote that state.
- After thinking, the resulting process is $Phil_i'$ whose only possible action is to grab the forks. We shall denote that state as G_i.
- After grabbing the forks, the new state is ${}^{\tau}\backslash_{eat_i}.{}^{dw_i}\backslash_{dw_{i_n^+}}.Phil_i\langle\rangle$ where the only possible action is to eat. We shall use E_i to denote that state.

- After exhibiting the eat_i action, the new state is ${}^{dw_i}\backslash_{dw_{i_n^+}}.Phil$, from now on denoted as R_i.
- After releasing the forks, we are back in the state GT_i.

Hence, any resulting process from $DP(n)$ can be succinctly represented as $P_0 \cdots P_i \cdots P_{n-1}\ \langle l_0, r_1\rangle\langle l_1, r_2\rangle\langle l_2, r_3\rangle \cdots\ \langle l_{n-1}, r_0\rangle$ where each P_i is either GT_i, G_i, E_i, or R_i.

Some valid transitions of this system are:

(i) $P_0 \quad \cdots GT_i \quad \cdots P_{n-1} \quad \langle l_0, r_1\rangle \cdots \langle l_{i-1}, r_i\rangle\langle l_i, r_{i+1}\rangle \cdots \langle l_{n-1}, r_0\rangle \xrightarrow{tk_i}$
$P_0 \ \cdots G_i \ \cdots P_{n-1} \ \cdots$ (Phil i thinks)

(ii) $\cdots G_i \cdots \ \cdots \langle l_{i-1}, 1\rangle\langle 1, r_{i+1}\rangle \cdots \xrightarrow{grab_i} \cdots E_i \ \cdots \ \cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots$ (grabbing the forks)

(iii) $\cdots E_i \ \cdots \ \cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots \xrightarrow{eat_i} \cdots R_i \ \cdots \ \cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots$ (eating)

(iv) $\cdots R_i \ \cdots\cdots \langle l_{i-1}, ?\rangle\langle ?, r_{i+1}\rangle \cdots \xrightarrow{release_i} \cdots GT_i \ \cdots \ \cdots \langle l_{i-1}, 0\rangle\langle 0, r_{i+1}\rangle \cdots$ (release the forks).

(v) $\cdots \ \cdots \langle 0, 0\rangle \cdots \xrightarrow{reset_i} \cdots \ \cdots \langle 1, 1\rangle \cdots$ (reset the fork i)

After the transition (iv), the only available action for the i-th philosopher is to think (only once). In this state, he can only grab the forks again once his neighbors eat and the forks perform the reset action. More precisely, the configuration $\ldots\langle 1, 0\rangle\langle 0, 1\rangle\ldots$ means that the philosopher in the middle has eaten and, after thinking, he remains blocked until his neighbors eat and release the forks.

Proof. Lemma 4.1. We prove the two points (deadlock-freeness and fairness) separately.

(i) Consider the state $\mathcal{S}_n = P_0 \cdots P_{n-1}\ \langle a_0, a_1\rangle\langle a_1, a_2\rangle\langle a_2, a_3\rangle \ldots \langle a_{n\text{-}1}, a_0\rangle$ where $a_i \in \{0, 1, ?\}$. If there is an a_i s.t. $a_i \in \{1, ?\}$, then, it is always possible to exhibit a grab ($a_i = 1$), eat or release ($a_i =?$) transition. Otherwise (i.e., if $a_i = 0$ for all $i \in 0..n\text{-}1$) then, it is always possible to exhibit a reset transition.

(ii) The proof is by contradiction. Let $DP(n)$ be a n-dining philosopher system such that there exists $i \in [0, n)$ and $Phil_i$ that performs eat_i finitely often. Without loss of generality (due to the circularity of the configuration) assume that $i = 0$. Consider the suffix of the computation σ where $Phil_0$ has already performed all the eating actions, i.e., in the rest of the (infinite) computation, we do not observe eat_0. Hence, this philosopher is either in the state GT_0 and, after thinking, in state G_0. We shall show that this computation cannot be infinite (thus a contradiction).

Once the neighbor $Phil_1$ eats and releases the forks, we are in the following situation $DP' = G_0\ GT_1 \cdots \langle a_0, 0\rangle\langle 0, a_2\rangle \ldots$.

In this state, $Phil_1$ cannot eat again (he can only think). If $Phil_2$ eats, a_2 becomes 0 and a reset on $Fork_1$ is possible: $DP' \longrightarrow^*$

$G_0\ GT_1 \cdots \langle a_0, 0\rangle\langle 0, 0\rangle \ldots \longrightarrow^* \stackrel{reset_1}{\longrightarrow} G_0\ GT_1 \cdots \langle a_0, 0\rangle\langle 1, 1\rangle \ldots$

If $a_0 = 1$, $Fork_0$ cannot reset until $Phil_0$ eats (which is not possible by hypothesis). This reasoning goes on for all the $n-1$ philosophers that are willing to eat. Once all of them have eaten, and all the forks that could have reset have already performed that action, the configuration is:

$$DP'' = G_0 \cdots G_{n-1}\langle 1, 0\rangle\langle 1, 0\rangle \ldots \langle 1, 0\rangle\langle 0, 1\rangle\langle 0, 1\rangle \ldots \langle 0, 1\rangle\langle 0, 1\rangle$$

Hence, if $Phil_0$ does not eat, at some point, all the philosophers will be blocked. In fact, counting only the eating and reset actions, the system can perform at most $\sum_{i=1}^{n\text{-}1} i = \frac{n\times(n-1)}{2}$ transitions:

$\langle 1, 1\rangle\langle 1, 1\rangle\langle 1, 1\rangle \ldots \langle 1, 1\rangle\langle 1, 1\rangle\langle 1, 1\rangle$	at most n-1 eat actions if $Phil_0$ does not eat
$\langle 1, 0\rangle\langle 0, 0\rangle\langle 0, 0\rangle \ldots \langle 0, 0\rangle\langle 0, 0\rangle\langle 0, 1\rangle$	at most n-2 reset actions
$\langle 1, 0\rangle\langle 1, 1\rangle\langle 1, 1\rangle \ldots \langle 1, 1\rangle\langle 1, 1\rangle\langle 0, 1\rangle$	at most n-3 eat actions
$\langle 1, 0\rangle\langle 1, 0\rangle\langle 0, 0\rangle \ldots \langle 0, 0\rangle\langle 0, 1\rangle\langle 0, 1\rangle$	at most n-4 reset actions
...	
$\langle 1, 0\rangle\langle 1, 0\rangle \ldots \langle 1, 0\rangle\langle 0, 1\rangle\langle 0, 1\rangle \ldots \langle 0, 1\rangle\langle 0, 1\rangle$	no transition if $Phil_0$ does not eat

If $Phil_0$ does not eat, $Fork_0$ and $Fork_{n\text{-}1}$ cannot reset. Hence, from row 2 on, the state of these forks will be, respectively, $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$. At this point, $Fork_1$ and $Fork_{n\text{-}2}$ are able to reset, and this justifies the configuration in row 3. At row 4, $Phil_1$ and $Phil_{n\text{-}2}$ cannot eat since the needed forks are not available and they cannot reset. Hence, assuming that $Phil_0$ does not eat breaks the possibility of restarting all the forks and a deadlock is reached.

□