A Calculus for Subjective Communication

Marino Miculan[1,2], Matteo Paier[1]

[1]University of Udine, Italy
[2]Ca' Foscari University of Venice, Italy

Abstract

In this paper we introduce *Subjective Communication*, a new interaction model for CAS and generalizing the attribute-based communication introduced in the AbC calculus. In this model, a message is broadcasted to every process, but each process can view the very same message in different ways, depending on its attributes. To formalize this model, we introduce *SCC*, the *Subjective Communication Calculus*, for which we propose two semantics: *Direct SCC*, particularly useful when dealing with an edge computing communication paradigm, and *Indirect SCC*, more suited to a cloud-centric model. We then introduce a stateless bisimilarity for our semantics, which we prove to be a congruence.

Keywords

Concurrency theory, Process algebra, Distributed Systems

1. Introduction

Collective Adaptive Systems (CAS) are systems consisting of many interacting components ("agents") which exhibit complex behaviours depending on their attributes, objectives and actions [1]. Such systems consist of both physical and virtual entities, often geographically distributed, and are capable to react to changes in the environment they operate in. Usually, a CAS component contains three main blocks: Knowledge, that contains local and global data and information; Policies, that define how different entities should interact, combine and compute; Processes, that describe how the component progresses. Global behaviour of the system emerge from the interaction between the components [2].

In these scenarios, new robust engineering techniques and programming paradigms, with solid theoretical foundations, are sought. However, traditional interaction models based on channels or point-to-point communication, as formalised in calculi such as CCS and π-calculus, appear to be inadequate and non-scalable; hence, new formal models for CAS are currently being investigated. An important example is AbC [3], a calculus inspired by SCEL [4] and based on *attribute-based communication*. In this model, communication is broadcast-based, as in CBS [5], but the receivers are dynamically identified by means of *predicates*: only the nodes whose state satisfies a given property will actually receive the message. Thus, attribute-based communication frees the nodes from knowing each other's identities, allowing for a scalable, decoupled interaction. In fact, attribute-based communication has been integrated also in declarative programming paradigms, e.g. ECA rule-based languages for "smart" systems [6, 7].

Proceedings of the 23rd Italian Conference on Theoretical Computer Science, Rome, Italy, September 7-9, 2022
marino.miculan@uniud.it (M. Miculan); paier.matteo@spes.uniud.it (M. Paier)
0000-0003-0755-3444 (M. Miculan)

However, there is a tension between the *objective* nature of messages and the fact that the same message can be accessed differently by agents with different knowledge. This is evident in the case of messages containing restricted names (akin secret keys): an agent receives a "restricted", projected version of the message, where the unknown names have been removed.

In this paper, we solve this tension by renouncing to the idea that messages have an absolute, "platonistic" meaning whose shadows is all what different agents can observed. Instead, we embrace the empiricistic principle that a message has no meaning on its own, but can be "experienced" by an observer—and clearly, different agents can experience the same message in different ways, according to their knowledge, processes, policies. We call this model *subjective communication*, since the meaning of a message is not fixed *a priori*: when a communication occurs, the very same message is delivered to *all* nodes, but the actual value received by each of them depends on their internal state.

In order to model and reason about concurrent systems interacting via subjective communications, in this paper we introduce the *Subjective Communication Calculus* SCC. In this calculus, each agent is composed by an internal state and a process, defining its behaviour. A component can broadcast a message, which is actually a (partial) function from components to values. Every component receives such function and evaluates it on itself; thus, different agents may obtain different values from the very same function—or nothing at all. This model generalizes attribute-based communication: a message that in AbC is discarded by a node corresponds to a message whose evaluation on that node's state is undefined.

Given this definition of subjective communication, we need to specify where is the control logic that decides how to modify or filter the messages that an agent will receive. We found two possibilities, yielding two different semantics. One possibility is that the agent itself checks each message and decides how to interpret it (i.e. it evaluates the function on itself, Fig. 1a); the other is that an external entity enforces this check, and then passes the evaluated message (i.e. the return value) to the agent (Fig. 1b). These possibilities, which we *Direct Subjective Communication* (DSCC, Section 2.1) and *Indirect Subjective Communication* (ISCC, Section 2.2), respectively, correspond to two common approaches to "smart" computation, namely, *edge* vs *cloud* computing.

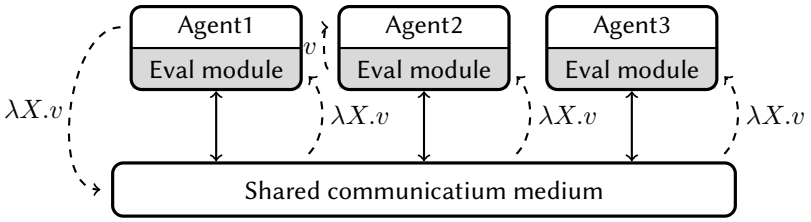
Furthermore, we give the definition of *stateless bisimilarity* over processes, and prove that it is compositional for both direct and indirect semantics.

Synopsis. In Section 2 we introduce syntax and the two semantics of SCC. An extended example is presented in Section 3. Stateless bisimilarity for SCC is introduced and studied in Section 4. Some conclusions are drawn in Section 5.

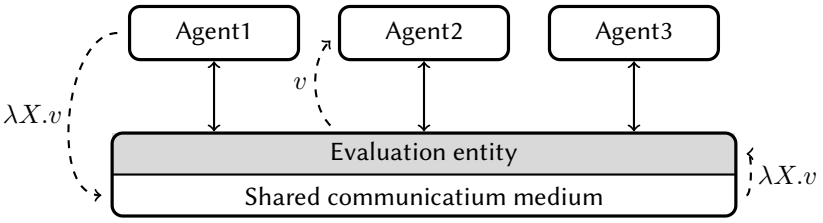
2. A calculus for Subjective Communication

In this section we introduce the syntax and two operational semantics for SCC

We define a *(collective) system* C as a collection of parallel *components* of the form $\Gamma : P$, where P is a *process* and $\Gamma : \mathcal{A} \to \mathcal{V}$ is an *environment* i.e. a map from a set $\mathcal{A}$ of *attribute identifiers* to *values* $\mathcal{V}$, representing the current internal state of the component. The special $\perp$ represents *undefined*. Processes are the building blocks of components: they define the behaviour of the component itself, whereas the environment defines the memory.



(a) The DSCC semantics.



(b) The ISCC semantics.

Figure 1: The DSCC and ISCC semantics. The dashed arrows represents the flow of information when Agent1 sends a message, and only Agent2 is interested.

We assume a set of *expressions* $e \in \mathcal{E}$ and a set of predicates $\varphi \in \Phi$ over a fixed set of term variables $\mathcal{X}$. The evaluation of an expression e w.r.t. store Γ is denoted as $[\![e]\!]_\Gamma$, which yields a value v (possibly $\perp$). We do not enforce an expression syntax to remain more general. The evaluation is defined as usual, with the requirement that it is defined only on ground expressions, i.e. expressions with no free variable; thus: if $fv(e) \neq \emptyset$ then $[\![e]\!]_\Gamma = \perp$. This means, for instance, that in an arithmetic context $[\![0 \times x]\!]_\Gamma = \perp$, instead of 0 as one may expect. The evaluation is extended to logical predicates as usual.

The complete syntax of SC is defined in terms of processes and components:

$$\begin{array}{ll} \mathcal{P} & P, Q ::= 0 \mid P|Q \mid P + Q \mid \varphi?P{:}Q \mid \langle f \rangle Q \mid (x)P \mid K(\vec{e}) \mid [l := e]P \\ \mathcal{C} & C ::= \Gamma : P \mid C \| C \mid \nu x C \end{array}$$

The behaviour of a single component is defined by a process which can:

- 0 do nothing;
- $P|Q$ interleave the execution of two processes;
- $P + Q$ non-deterministically choose a continuation;
- $\varphi?P{:}Q$ evaluate φ, and if it is a boolean value it chooses P or Q, accordingly;
- $\langle f \rangle P$ broadcast a function from *components* to *values* ($f : \mathcal{C} \to \mathcal{V}$). Each receiver process will get a value depending on itself;
- $(x)P$ wait for the reception of a value. This construct allows for the substitution, in P, of the free name x with the value received;

α-equivalence	$C \equiv D$ if C and D are α-equivalent
Commutativity of $\|$	$P\|Q \equiv Q\|P$
Associativity of $\|$	$P\|(Q\|R) \equiv (P\|Q)\|R$
Dead process	$0\|P \equiv P$
Commutativity of $\|\|$	$C\|\|D \equiv D\|\|C$
Associativity of $\|\|$	$C\|\|(D\|\|E) \equiv (C\|\|D)\|\|E$
Commutativity of $+$	$C + D \equiv D + C$
Associativity of $+$	$C + (D + E) \equiv (C + D) + E$
Idempotency of $+$	$C + C \equiv C$
Scope extrusion of ν	$(\nu x C)\|\|D \equiv \nu x(C\|\|D)$ if $x \notin \mathit{fn}(D)$
Reordering of ν	$\nu x \nu y C \equiv \nu y \nu x C$

Figure 2: Axioms of structural congruence.

$K(\vec{e})$ call a process named K, instantiating the parameters with the list $\vec{e}$. This construct enables looping of processes (using recursion);

$[l := e]P$ update its local memory, by updating the cell identified by the label l with the value of e (if not $\perp$).

Components can be assembled into *systems*:

$\Gamma : P$ is a single component, with a process P running on a store Γ.

$C\|C$ is a parallel of multiple components;

$\nu x C$ is a system with a fresh, local name, a la π-calculus.

Processes and systems are taken up-to a congruence expressing the fact that they can be denoted in different but equivalent ways. This is useful both to simplify the writing of some semantic rules (e.g. the rules for $P + Q$ and $C\|D$) and to simplify the modelling of systems. The axioms are mostly standard (see Fig. 2). The congruence axioms for the commutativity and associativity of $|$, $\|$ and $+$, for the the idempotency of $+$ and for the reordering of ν are the usual. The rule for the scope extrusion is like in π-calculus. The rule for the elimination of a dead process simplifies our terms. In particular, it states that 0 can be removed if it runs in parallel with another process.

Notice that we do not have an axiom in the form $\Gamma : 0\|C \equiv C$ because we do not allow a component to "disappear" from the system, even if its process is done: a component usually represents a physical object, which is not destroyed if the software component in it terminates.

Some common operations can be introduced as syntactic shorthands:

awareness is a blocking test, i.e. the process stops until the predicate becomes true. This is equivalent to a busy-waiting in a classical programming language:

$$\varphi?P \triangleq K(\vec{x}) \text{ where } K(\vec{x}) \triangleq \varphi?P : K \text{ and } \vec{x} = \mathit{fv}(P)$$

$$\textsc{Int}\ \frac{\Gamma : P \xrightarrow{\alpha}_D \Gamma' : P'}{\Gamma : P|Q \xrightarrow{\alpha}_D \Gamma' : P'|Q} \qquad \textsc{Choice}\ \frac{\Gamma : P \xrightarrow{\alpha}_D \Gamma' : P'}{\Gamma : P + Q \xrightarrow{\alpha}_D \Gamma' : P'}$$

$$\textsc{ExtTrue}\ \frac{[\![\varphi]\!]_\Gamma = \mathtt{tt} \quad \Gamma : P \xrightarrow{\alpha}_D \Gamma' : P'}{\Gamma : (\varphi ? P : Q) \xrightarrow{\alpha}_D \Gamma' : P'} \qquad \textsc{ExtFalse}\ \frac{[\![\varphi]\!]_\Gamma = \mathtt{ff} \quad \Gamma : Q \xrightarrow{\alpha}_D \Gamma' : Q'}{\Gamma : (\varphi ? P : Q) \xrightarrow{\alpha}_D \Gamma' : Q'}$$

$$\textsc{SkipExt}\ \frac{[\![\varphi]\!]_\Gamma = \bot}{\Gamma : (\varphi ? P : Q) \xrightarrow{?f}_D \Gamma : (\varphi ? P : Q)} \qquad \textsc{SkipZero}\ \ \Gamma : 0 \xrightarrow{?f}_D \Gamma : 0$$

$$\textsc{Output}\ \ \Gamma : \langle f \rangle P \xrightarrow{!f}_D \Gamma : P \qquad \textsc{SkipOutput}\ \ \Gamma : \langle f \rangle P \xrightarrow{?g}_D \Gamma : \langle f \rangle P$$

$$\textsc{Input}\ \frac{f(\Gamma : (x)P) = v \neq \bot}{\Gamma : (x)P \xrightarrow{?f}_D \Gamma : P[v/x]} \qquad \textsc{SkipInput}\ \frac{f(\Gamma : (x)P) = \bot}{\Gamma : (x)P \xrightarrow{?f}_D \Gamma : (x)P}$$

$$\textsc{Update}\ \frac{[\![e]\!]_\Gamma = v \neq \bot \quad \Gamma[l \mapsto v] : P \xrightarrow{\alpha}_D \Gamma' : P'}{\Gamma : [l := e]P \xrightarrow{\alpha}_D \Gamma' : P'} \qquad \textsc{SkipUpdate}\ \frac{[\![e]\!]_\Gamma = \bot}{\Gamma : [l := e]P \xrightarrow{?f}_D \Gamma : [l := e]P}$$

$$\textsc{Comm}\ \frac{C \xrightarrow{!f}_D C' \quad D \xrightarrow{?f}_D D'}{C \| D \xrightarrow{!f}_D C' \| D'} \qquad \textsc{Call}\ \frac{\Gamma : P[\vec{e}/\vec{x}] \xrightarrow{\alpha}_D C}{\Gamma : K(\vec{e}) \xrightarrow{\alpha}_D C}\ K(\vec{x}) \triangleq P$$

$$\textsc{Res}\ \frac{C \xrightarrow{\alpha}_D C'}{\nu x C \xrightarrow{[x]\alpha}_D \nu x C'} \qquad \textsc{Cong}\ \frac{C \equiv C' \quad C' \xrightarrow{\alpha}_D D' \quad D' \equiv D}{C \xrightarrow{\alpha}_D D}$$

Figure 3: Operational semantics of DSCC.

fork duplicates a process which continues as different threads:

$$!P \triangleq K(\vec{x}) \text{ where } K(\vec{x}) \triangleq P|K(\vec{x}) \text{ and } \vec{x} = fv(P)$$

case allows for choosing among multiple continuations via pattern-matching (this is useful when we need to perform different actions based on the value of a received message):

$$\begin{aligned} &\texttt{case}\ \ x\ \ \texttt{of}\ \ expr_1 \to P_1; expr_2 \to P_2; \ldots; expr_n \to P_n; \texttt{default} \to Q\ \ \texttt{end} \\ &\triangleq (x = expr_1) ? P_1 : ((x = expr_2) ? P_2 : (\ldots (x = expr_n) ? P_n : Q) \ldots) \end{aligned}$$

2.1. Semantics for Direct Subjective Communication

The operational (system) semantics of DSCC is given by a labelled transition relation $C \xrightarrow{\alpha}_D C'$, whose labels are $\alpha \in \{?f, !f \mid f : \mathcal{C} \to \mathcal{V}\}$. $?f$ is used for inputs, $!f$ is used for outputs. The rules for this semantics are in Fig. 3.

Rules prefixed with Skip are needed to have a proper broadcast communication: a component is *always* able to perform an input action, possibly ignoring the message and remaining itself. Without these rules, our systems would stop communicating if a component does not want to input something or if two components try simultaneously to perform an output. In other words, at each step a system can only perform a single communication; if two components need to send two messages, these cannot occur in parallel but will be interleaved.

Rule Int is the standard rule for handling interleaving of the actions of two processes. Rule Choice represents the nondeterministic choice between the subprocesses P and Q, i.e. that $P + Q$ can choose to continue as any of P or Q.

Rules ExtTrue and ExtFalse evaluate a predicate φ in Γ; if this returns $\mathtt{tt}$ the process $\varphi?P:Q$ proceeds as P, if it is $\mathtt{ff}$ it proceeds as Q, otherwise it does not change.

Rule SkipExt enables a component to perform an input (discarding the value) even if the evaluation of a predicate φ under the environment Γ is not possible (e.g. in the case φ is not ground). Rule SkipZero represents the trivial fact that an inactive (dead) component can always receive something in a broadcast.

Rule Output performs an output action, broadcasting a function f and continuing as P. Rule SkipOutput enables a component that wants to output something also to receive a function and remain itself. This is useful if more than a component, in a parallel system, is trying to perform an output at once; then, these components perform their outputs in turns.

Rules Input and SkipInput evaluate a received function. If the evaluation yields a value $v \neq \bot$, the substitution $[v/x]$ is done to the continuation process P. If the evaluation of the received function yields $\bot$, the component is forced to discard the input and to remain itself.

Rule Update evaluates the expressions e in Γ. If the result is a value ($\neq \bot$), it applies the attribute update to the store. Then the component performs an action with an α label if the component $\Gamma[l \mapsto v]:P$ (P under the updated environment) can do so. Rule SkipUpdate, as usual, forces a component that wants to update the store to discard any arriving inputs.

Rule Comm enables parallel systems to communicate. In particular, it states that if there is a system C that wants to perform an output in parallel with a system D that is able to perform an input (any system is able to do so, given the Skip rules), the two of them are able to continue in parallel as $C' \| D'$.

Rule Call expands the definition of a process constant, by instantiating the formal parameters (in a call-by-name fashion).

Rule Res defines the behaviour of components under name restriction. Since x is bound in $\nu x C$, it can not be visible in the label either. To this end, we use the *name binding* operation provided by Nominal Sets [8, 9]. In particular, with the notation $[x]\alpha$ we denote the label obtained by binding x in α: if $\alpha = ?f$ then $[x]\alpha = ?x.f$; if $\alpha = !f$ then $[x]\alpha = !x.f$. Notice that x is a local name, bound to the label α, so $\mathit{fn}([x]\alpha) = \mathit{fn}(\alpha) \setminus \{x\}$.

Rule Cong, finally, allows our system to be written in different, but equivalent ways, in order to be in the correct shape for performing the action.

Notice that the semantics does not allow for parallel broadcasts, even if these actually involve different subsets of the system. Without the Skip rules, a system could get stuck in two cases:

1. if two different components try to broadcast simultaneously;
2. if a component does not want to perform an input, in presence of a broadcast.

In both these cases, we would not have a rule that enables us to join the components with the parallel operator $\|$, since Comm requires the two parallel components to make, respectively, an output and an input transition.

Moreover, without the congruence axioms and Cong, we would not be able to use Comm if, e.g., we have a system $C \| D$ where C wants to input and D wants to output (we do not have the symmetric Comm rule).

The following Lemma 1 proves that the semantics behaves well, in the sense that a component executing a broadcast will always succeed in every system.

Lemma 1. *Given n parallel components, of which one wants to output, the broadcast succeeds regardless the particular component ordering. Formally, for any $C_1, \ldots, C_n$, if $C_i \xrightarrow{!f}_D C_i'$ then there exist $C_1', \ldots, C_{i-1}', C_{i+1}', \ldots C_n'$ such that*

$$C_1 \| C_2 \| C_3 \| \ldots \| C_n \xrightarrow{!f}_D C_1' \| C_2' \| C_3' \| \ldots \| C_n'$$

Proof. For the congruence axioms of commutativity and associativity of the system parallel $\|$ we have:

$$C_1 \| C_2 \| C_3 \| \ldots \| C_n \equiv (\ldots (C_i \| C_1) \| C_2) \| C_3) \| \ldots \| C_{i-1}) \| C_{i+1}) \| \ldots \| C_n)$$

Now, all of $C_1, C_2, C_3, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$ may perform an input, using either Input or a Skip rule. Then, C_i and C_1 may proceed using Comm. Since the conclusion exhibits an output label, we can again use Comm alongside C_2 and we can iterate the process until we reach C_n (with a total of $n - 1$ application of Comm). □

2.2. Semantics for Indirect Subjective Communication

The operational (system) semantics of ISCC is given by a labelled transition relation $C \xrightarrow{\alpha}_I C'$, different from the DSCC one. In particular, the associated labels are $\alpha \in \mathcal{V} \cup \{!f \mid f : \mathcal{C} \to \mathcal{V}\}$. As for DSCC these labels denotes different actions: $?v$ is used for inputs, $!f$ is used for outputs.

Notice that, contrary to the DSCC specification, here the communication is somewhat *mediated*, hence the name *Indirect Subjective Communication*. This means that a component is no longer required to get and execute a function (fundamentally a piece of code) to obtain a value; instead, the broadcasted function is collected by an external mediating entity that evaluates them on the behalf of the SC recipients. We do not discuss here how such an entity should work: we just need to know that it must have access to the stores of all agents.

An interesting thing to notice is that in this setting the agent does not need to be trusted, since it will receive only the values that it is meant to receive. This prevents e.g. sniffing of messages or unwanted behaviours by byzantine components. Suppose there is a DSCC agent that "looks into" the received function prior to evaluation: such an agent may be able to gather information that must be kept secret. With the ISCC semantics this is no longer possible, since the component will be given only the relevant result.

The rules for ISCC are in Fig. 4, and may look similar to the ones for DSCC, but of course the transition relation is different. They have, however, the same role in the two semantics, so that we can omit explaining them again here.

Rule Input, in ISCC, performs the substitution $[v/x]$ to the continuation process P, if the component is given a value v that is not $\bot$. Rule SkipInput is similar: a component, when it is given $\bot$, discards the input and remains itself.

Rule Comm, like in DSCC, enables parallel systems to communicate. Here, however, is this rule that evaluates the output function f to yield a value v. In particular, the rule states that if there is a system C that wants to perform an output in parallel with a system D that is able to perform an input, the two of them are able to continue in parallel as $C' \| D'$, provided that to D is given the value returned by the application of the output function to D itself.

$$\textsc{Int}\ \frac{\Gamma : P \xrightarrow{\alpha}_I \Gamma' : P'}{\Gamma : P|Q \xrightarrow{\alpha}_I \Gamma' : P'|Q} \qquad \textsc{Choice}\ \frac{\Gamma : P \xrightarrow{\alpha}_I \Gamma' : P'}{\Gamma : P + Q \xrightarrow{\alpha}_I \Gamma' : P'}$$

$$\textsc{ExtTrue}\ \frac{[\![\varphi]\!]_\Gamma = \mathtt{tt} \quad \Gamma : P \xrightarrow{\alpha}_I \Gamma' : P'}{\Gamma : (\varphi ? P : Q) \xrightarrow{\alpha}_I \Gamma' : P'} \qquad \textsc{ExtFalse}\ \frac{[\![\varphi]\!]_\Gamma = \mathtt{ff} \quad \Gamma : Q \xrightarrow{\alpha}_I \Gamma' : Q'}{\Gamma : (\varphi ? P : Q) \xrightarrow{\alpha}_I \Gamma' : Q'}$$

$$\textsc{SkipExt}\ \frac{[\![\varphi]\!]_\Gamma = \bot}{\Gamma : (\varphi ? P : Q) \xrightarrow{?v}_I \Gamma : (\varphi ? P : Q)} \qquad \textsc{SkipZero}\ \ \Gamma : 0 \xrightarrow{?v}_I \Gamma : 0$$

$$\textsc{Output}\ \ \Gamma : \langle f \rangle P \xrightarrow{!f}_I \Gamma : P \qquad \textsc{SkipOutput}\ \ \Gamma : \langle f \rangle P \xrightarrow{?v}_I \Gamma : \langle f \rangle P$$

$$\textsc{Input}\ \frac{v \neq \bot}{\Gamma : (x)P \xrightarrow{?v}_I \Gamma : P[v/x]} \qquad \textsc{SkipInput}\ \frac{v = \bot}{\Gamma : (x)P \xrightarrow{?v}_I \Gamma : (x)P}$$

$$\textsc{Update}\ \frac{[\![e]\!]_\Gamma = v \neq \bot \quad \Gamma[l \mapsto v] : P \xrightarrow{\alpha}_I \Gamma' : P'}{\Gamma : [l := e]P \xrightarrow{\alpha}_I \Gamma' : P'} \qquad \textsc{SkipUpdate}\ \frac{[\![e]\!]_\Gamma = \bot}{\Gamma : [l := e]P \xrightarrow{?v}_I \Gamma : [l := e]P}$$

$$\textsc{Comm}\ \frac{C \xrightarrow{!f}_I C' \quad D \xrightarrow{?v}_I D' \quad f(D) = v}{C \| D \xrightarrow{!f}_I C' \| D'} \qquad \textsc{Call}\ \frac{\Gamma : P[\vec{e}/\vec{x}] \xrightarrow{\alpha}_I C}{\Gamma : K(\vec{e}) \xrightarrow{\alpha}_I C}\ K(\vec{x}) \triangleq P$$

$$\textsc{Res}\ \frac{C \xrightarrow{\alpha}_I C'}{\nu x C \xrightarrow{[x]\alpha}_I \nu x C'} \qquad \textsc{Cong}\ \frac{C \equiv C' \quad C' \xrightarrow{\alpha}_I D' \quad D' \equiv D}{C \xrightarrow{\alpha}_I D}$$

Figure 4: Operational semantics of ISCC.

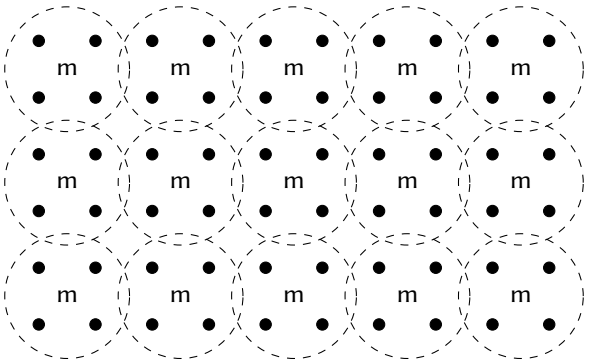


Figure 5: A smart vineyard. Each dot represents a vine (with the associated watering valve), each m represents a moisture sensor. The circles are the reach of each sensor

3. Example: a Vineyard Irrigation System

Let us consider a scenario where a winegrower wants to optimize the irrigation of some vineyards. Their main goals are: 1) to irrigate each vine with the correct amount of water; 2) to know if the soil "behaves well", i.e. if it neither causes water stagnation nor it dries too quickly; 3) to know, at every time, each watering valve status (i.e. if it is opened or closed).

Each yard is divided into zones; each zone has a moisture sensor and a certain number of vines, each with its watering valve (see Fig. 5). Additionally, each sensor and actuator knows its position (a Cartesian coordinate, e.g., $(row, column)$), fixed at deployment. The coordinates are not unique among different yards. The winegrower has a console terminal that collects the watering data and reports them in some graphical way (e.g. it shows a heat-map).

A traditional, centralised way to implement a management system for this scenario is to use a central server (possibly in the cloud) that gathers all the relevant data and remotely open and closes the valves according to some policies, in order to maintain the desired moisture level. On the other hand, there is the possibility of using a distributed, edge-computing paradigm, where the computation is moved near the source of the data. This should reduce latency and increase reliability, at the cost of increasing the computational resources required on the nodes.

With our paradigm we can implement both: DSCC is well-suited for a truly distributed system, where reliability is the main goal; ISCC is better suited when security is essential. When modelling the system we can abstract from the particular implementation: switching between the two semantics can be easily done at any point of the design.

We can model the scenario as a SC system as follows:

$$\begin{aligned} C_{yard}(name) = {} & C_{valve}(name, x_{v1}, y_{v1}) \| \ldots \| C_{valve}(name, x_{vn}, y_{vn}) \| \\ & C_{moist}(name, x_{h1}, y_{h1}) \| \ldots \| C_{moist}(name, x_{hm}, y_{hm}) \\ C_{yards} = {} & \nu n_1 C_{yard}(n_1) \| \ldots \| \nu n_k C_{yard}(n_k) \| C_{console} \end{aligned}$$

The construct $\nu k(C_1 \| C_2)$ creates a secret k (i.e., a nonce), that C_1 and C_2 can use to communicate without interfering with other components; this allows to logically separate messages that involve only some components, yielding a cleaner "communication environment".

The moisture sensor acts as a local controller for the zone, communicating a message which commands the nearby watering valves, and notifying the moisture level to the console:

$$\begin{aligned} C_{moist}(n, x, y) = {} & \Gamma_{moist} :! \langle f_{moist} \rangle 0 \\ \Gamma_{moist} = {} & \{ type \mapsto \texttt{moisture}, yard \mapsto n, x \mapsto x, y \mapsto y, \\ & reach \mapsto \texttt{2}, h \mapsto \texttt{NaN}, h_{min} \mapsto \texttt{0.4}, h_{max} \mapsto \texttt{0.5} \} \end{aligned}$$

where h bound to the output of a physical moisture sensor and f_{moist} is defined as follows:

$$f_{moist} = \lambda(type, yard, x, y).\begin{cases} \texttt{open} & \text{if } \begin{array}{r} this.h < this.h_{min} \wedge \\ type = \texttt{valve} \wedge yard = this.yard \wedge \\ \sqrt{(this.x - x)^2 + (this.y - y)^2} < this.reach \end{array} \\ \texttt{close} & \text{if } \begin{array}{r} this.h > this.h_{max} \wedge \\ type = \texttt{valve} \wedge yard = this.yard \wedge \\ \sqrt{(this.x - x)^2 + (this.y - y)^2} < this.reach \end{array} \\ \left(\begin{smallmatrix} \texttt{moist}, this.yard, \\ this.x, this.y, this.h \end{smallmatrix} \right) & \text{if } type = \texttt{console} \\ \bot & \text{otherwise} \end{cases}$$

Here we see that the same message (f_{moist}) can be interpreted in different ways:

1. a valve ($type = \texttt{valve}$) close to the sensor will evaluate it as a command to open or close;
2. the console ($type = \texttt{console}$) be evaluated as a tuple that contains all the information needed to insert in a log file the current moisture value read by the particular sensor;
3. anyone else will evaluate it as $\bot$, meaning that no value is passed to the process.

To complete our system we need to define the components for the irrigation valves and the console. The valve acts on the water dripper according to the command received from the local controller. It also updates the console with the current status of the valve (to e.g. raise an alert if a valve is stuck open).

The valve component is programmed as follows:

$$
\begin{aligned}
C_{valve}(n,x,y) &= \Gamma_{valve} :!(P_{valve} | P_{sendStatus}) \\
\Gamma_{valve} &= \{type \mapsto \texttt{valve}, yard \mapsto n, x \mapsto x, y \mapsto y, water \mapsto \texttt{ff}\} \\
P_{valve} &= (msg)\texttt{case } msg \texttt{ of} \\
&\qquad \texttt{open} \to [water := \texttt{tt}]0; \\
&\qquad \texttt{close} \to [water := \texttt{ff}]0; \\
&\quad \texttt{end} \\
P_{sendStatus} &= \langle f_{sendStatus} \rangle 0 \\
f_{sendStatus} &= \lambda(type).\begin{cases} \begin{pmatrix} \texttt{valve}, this.yard, \\ this.x, this.y, this.water \end{pmatrix} & \text{if } type = \texttt{console} \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

The console simply logs in a database the values obtained from the moisture sensors and the watering valves. This component is defined as:

$$
\begin{aligned}
C_{console} &= \Gamma_{console} :!P_{console} \\
\Gamma_{console} &= \{type \mapsto \texttt{console}\} \\
P_{console} &= (msg)\texttt{case } msg \texttt{ of} \\
&\qquad (\texttt{moist}, yard, x, y, h) \to K_{logMoisture}(yard, x, y, h); \\
&\qquad (\texttt{valve}, yard, x, y, s) \to K_{logValveStatus}(yard, x, y, s); \\
&\quad \texttt{end}
\end{aligned}
$$

where $K_{logMoisture}$ saves the moisture reading in a database and $K_{logValveStatus}$ does the same with the valve status.

4. Stateless Bisimulation

As for similar foundational calculi, SCC is intended to be also a basis for the investigation of theoretical aspects of the Subjective Communication model. As an example along this line, in this section we introduce a behavioural equivalence for processes.

In SCC, components carry (local) data (the environments or stores), which may be updated by processes. We can easily define a notion of strong bisimilarity on components and systems by applying directly the original notion of strong bisimilarity, treating attributes in the environments on a par with processes. More precisely, a *strong bisimulation* for DSCC (similary for ISCC) is a relation $R \subseteq \mathcal{C} \times \mathcal{C}$, such that for all $(C_1, C_2) \in R$:

- for all α such that $C_1 \xrightarrow{\alpha}_D C_1'$ there exists C_2' such that $C_2 \xrightarrow{\alpha}_D C_2'$ and $(C_1', C_2') \in R$;
- for all α such that $C_2 \xrightarrow{\alpha}_D C_2'$ there exists C_1' such that $C_1 \xrightarrow{\alpha}_D C_1'$ and $(C_1', C_2') \in R$.

Two systems are *strongly bisimilar* if there exists a strong bisimulation relating them.

However, this notion of bisimilarity is not very satisfactory, because systems can be strongly bisimilar even if their environments are not the same; e.g., $\Gamma_1 : 0$ and $\Gamma_2 : 0$ are strongly bisimilar for any Γ_1, Γ_2. In fact, in most applications we need to observe and compare (local) environments in a finer way; e.g., a closed valve is not equivalent to an open one. Therefore, in this paper, we restrict ourselves to comparing *processes* with respect to the same environments, in order to ensure that, when starting from the same state, they yield the same state.

Our behavioural equivalence is an adaptation of the *stateless bisimulation* given in [10]. Two process terms are stateless bisimilar if, for all identical data states, they can mimic transitions of each other and the resulting process terms are again stateless bisimilar, i.e. it compares components for all identical environments and allows all sort of changes after the transition. More formally, let $\xrightarrow{\alpha}$ be any of $\xrightarrow{\alpha}_D$, $\xrightarrow{\alpha}_I$:

Definition 1 (Stateless Bisimilarity)**.** *A symmetric relation $R_{sl} \subseteq \mathcal{P}^n \times \mathcal{P}^n$ is a* stateless bisimulation *if for all $(\vec{P}, \vec{Q}) \in R_{sl}$:*

1. *for all $\Gamma_1, \ldots, \Gamma_n$ and $P'_1, \ldots, P'_n$ such that*

$$\Gamma_1 : P_1 \| \ldots \| \Gamma_n : P_n \xrightarrow{\alpha} \Gamma'_1 : P'_1 \| \ldots \| \Gamma'_n : P'_n$$

   there exist $Q'_1, \ldots, Q'_n$ such that $(\vec{P'}, \vec{Q'}) \in R_{sl}$ and

$$\Gamma_1 : Q_1 \| \ldots \| \Gamma_n : Q_n \xrightarrow{\alpha} \Gamma'_1 : Q'_1 \| \ldots \| \Gamma'_n : Q'_n$$

2. *for all $\Delta_1, \ldots, \Delta_n$ and $Q'_1, \ldots, Q'_n$ such that*

$$\Delta_1 : Q_1 \| \ldots \| \Delta_n : Q_n \xrightarrow{\alpha} \Delta'_1 : Q'_1 \| \ldots \| \Delta'_n : Q'_n$$

   there exist $P'_1, \ldots, P'_n$ such that $(\vec{P'}, \vec{Q'}) \in R_{sl}$ and

$$\Delta_1 : P_1 \| \ldots \| \Delta_n : P_n \xrightarrow{\alpha} \Delta'_1 : P'_1 \| \ldots \| \Delta'_n : P'_n$$

Two processes P and Q are stateless bisimilar*, written $P \sim_{sl} Q$, if there exists a stateless bisimulation R_{sl} such that $(P, Q) \in R_{sl}$.*

Compositionality of stateless bisimilarity makes it particularly useful when, in a parallel system, we want to replace a component: if the new "part" is stateless bisimilar to the old one, the component (and the whole system) is guaranteed to behave the same.

Proposition 1. *Stateless bisimilarity is a congruence for both DSCC and ISCC.*

Proof. Follows from [10, Th. 14] and the fact that DSCC and ISCC are in *process-tyft format* [10]. Rule Cong is not strictly in process-tyft format, but we can omit it from the system by taking process up-to congruence, as two congruent components exhibit the same transitions. □

5. Conclusion

In this paper we have introduced the Subjective Communication Calculus SCC, a formal model for Collective Adaptive Systems with *subjective communication*. In this paradigm, messages are (partial) functions from components to values; each component may obtain a different value (or nothing at all) from the very same message, according to its specific state and perspective.

We have presented two operational semantics for SCC. On one hand, we have introduced *Direct Subjective Communication* (DSCC), where each agent receives the message function and evaluates it; this semantics is akin to "edge computing", where the burden of computation is moved towards the node of the CAS. On the other hand, in *Indirect Subjective Communication* (ISCC) the broadcasted function is evaluated by some mediating middleware; this semantics can be seen as the counterpart of usual *cloud-based* computing, where messages are collected and elaborated by some central node (possibly in the cloud). For both these semantics we have given the definition of *stateless bisimilarity*, and proved that it is a congruence. In order to showcase the expressive power of the paradigm, we have provided an example inspired to real-world application scenario, i.e., a smart irrigation system for a vineyard.

Future work. We plan to give a categorical bialgebraic semantics for SCC [11], from which we expect to derive a canonical compositional LTS and behavioural equivalence, which we plan to compare with that examined in Section 4. Since the components contain states and local names and variables, we plan to follow the approach adopted for the Fusion calculus in [12, 13]. It is interesting also to investigate behavioural equivalences where some steps are unobservable from the point of view of some participant; to this end we plan to adopt and extend the approach presented in [14]. Afterwards, we plan to consider also quantitative aspects (e.g. execution times, energy consumptions); to this end, the results of [15, 16] could be useful.

There are other interesting types of bisimilarities for semantics with data: in particular we recall the notions of *initially stateless bisimilarity* and *statebased bisimilarity*. We cannot, however, apply the general results from [10] as we have done for stateless bisimilarity since our transition system specifications are not in *sfisl format* (for initially stateless bisimilarity) or *sbsf format* (for statebased bisimilarity). It is still an open question whether initially stateless or statebased bisimilarities are congruences for our specifications.

Finally, it is interesting to develop a prototypal implementation of SCC, in order to evaluate the practical differences of the two semantics. To this end, we will leverage the experience from the implementation of SCEL, AbC and the AbU calculus [3, 4, 6]. An alternative approach is to encode SCC as a bigraphic reactive system, in order to reuse existing toolkits [17].

Acknowledgments

This work has been supported by the Italian MIUR project PRIN 2017FTXR7S *IT MATTERS (Methods and Tools for Trustworthy Smart Systems).*

References

[1] V. Andrikopoulos, A. Bucchiarone, S. Gómez Sáez, D. Karastoyanova, C. A. Mezzina, Towards modeling and execution of collective adaptive systems, in: International Conference on Service-Oriented Computing, Springer, 2013, pp. 69–81.

[2] Miller, J. H, S. E. Page, Complex adaptive systems: An introduction to computational models of social life, Princeton university press, 2009.

[3] Y. A. Alrahman, R. De Nicola, M. Loreti, On the power of attribute-based communication, in: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Springer, 2016, pp. 1–18.

[4] R. De Nicola, M. Loreti, R. Pugliese, F. Tiezzi, A formal approach to autonomic systems programming: the SCEL language, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 9 (2014) 1–29.

[5] K. V. Prasad, A calculus of broadcasting systems, Science of Computer Programming 25 (1995) 285–327.

[6] M. Miculan, M. Pasqua, A calculus for attribute-based memory updates, in: A. Cerone, P. C. Ölveczky (Eds.), Theoretical Aspects of Computing – ICTAC 2021, Springer International Publishing, Cham, 2021, pp. 366–385.

[7] M. Miculan, M. Pasqua, Distributed programming of smart systems with event-condition-action rules (short paper), in: U. Dal Lago, D. Gorla (Eds.), 23rd Italian Conference on Theoretical Computer Science (ICTCS), Proceedings, CEUR Workshop Proceedings, 2022.

[8] A. M. Pitts, Nominal logic, a first order theory of names and binding, Information and computation 186 (2003) 165–193.

[9] M. J. Gabbay, The π-calculus in FM, in: Thirty Five Years of Automating Mathematics, Springer, 2003, pp. 247–269.

[10] M. R. Mousavi, M. A. Reniers, J. F. Groote, Notions of bisimulation and congruence formats for sos with data, Information and Computation 200 (2005) 107–147.

[11] B. Klin, Bialgebras for structural operational semantics: An introduction, Theoretical Computer Science 412 (2011) 5043–5069.

[12] M. Miculan, A categorical model of the Fusion calculus, Electr. Notes Theor. Comput. Sci. 218 (2008) 275–293.

[13] M. Miculan, K. Yemane, A unifying model of variables and names, in: V. Sassone (Ed.), Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Proceedings, volume 3441 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 170–186.

[14] T. Brengos, M. Miculan, M. Peressotti, Behavioural equivalences for coalgebras with unobservable moves, J. Log. Algebraic Methods Program. 84 (2015) 826–852.

[15] M. Miculan, M. Peressotti, GSOS for non-deterministic processes with quantitative aspects, in: N. Bertrand, L. Bortolussi (Eds.), Proc. QAPL 2014, volume 154 of *EPTCS*, 2014, pp. 17–33.

[16] M. Miculan, M. Peressotti, Structural operational semantics for non-deterministic processes with quantitative aspects, Theor. Comput. Sci. 655 (2016) 135–154.

[17] G. Bacci, D. Grohmann, M. Miculan, Dbtk: A toolkit for directed bigraphs, in: Proc. CALCO, volume 5728 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 413–422.