A Calculus for Attribute-based Memory Updates*

Marino Miculan and Michele Pasqua

Dept. of Mathematics, Computer Science and Physics - University of Udine, Italy
(marino.miculan|michele.pasqua)@uniud.it

Abstract. In this paper, we present AbU a new ECA-inspired calculus with *attribute-based communication*, an interaction model recently introduced for coordinating large numbers of nodes. Attribute-based communication is similar to broadcast, but the actual receivers are selected "on the fly" by means of predicates over nodes' attributes.
After having defined syntax and formal semantics of AbU, with some examples, we give sufficient conditions on AbU systems to guarantee termination of internal steps. Then we show how to encode into AbU components written in AbC, the archetypal calculus with attribute-based communication, and we prove the correctness of such encoding.

Keywords: ECA rules · Attribute-based communication · Distributed systems · Formal methods · Autonomic computing.

1 Introduction

Event Condition Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. The fundamental construct of ECA languages are rules of the form "**on** *Event* **if** *Condition* **do** *Action*" which means: when *Event* occurs, if *Condition* is verified then execute *Action*. ECA systems receive inputs (as events) from the external environment and react by performing *internal* actions, updating the node's local memory, or *external* actions, which influence the environment itself. Due to their reactive nature, ECA languages are well-suited for programming smart systems, in particular in IoT scenarios [17, 11]. Indeed, this paradigm can be found in various commercial frameworks like IFTTT, Samsung SmartThings, Microsoft Power Automate, Zapier, etc.

In most cases, the rules are stored and executed by a central computing node, possibly in the cloud: the components of the adaptive system do not communicate directly, and the coordination is demanded to the central node/cloud service. Although simple, such a centralized architecture does not scale well to large systems, and the central node/cloud service is a Single Point Of Failure, hindering availability. Thus, in these situations we may prefer to move computation closer to the edge of the network, akin *fog computing*: the ECA rules

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

should be stored and executed directly in the nodes, in a truly distributed setting. This approach reduces data transfers between the edge and the center of the network—in fact, there can be no center at all, thus increasing scalability and resilience—but, on the other hand, it requires a distributed coordination and communication of these components.

In order to model these issues, in this paper we introduce AbU (for "Attribute-based Updates"), a new calculus aiming at merging the simplicity of ECA programming with distributed coordination mechanisms in the spirit of *attribute-based communication*. Attribute-based communication is a time-coupled, space-uncoupled interaction model recently introduced for coordinating large numbers of components and subsuming several interaction paradigms used in "smart systems", such as channels, agents, pub/sub, broadcast and multicast [5, 2, 4]. The key aspect of attribute-based communication is that the actual receivers are selected "on the fly" by means of *predicates*. Using a syntax similar to AbC [5] (the archetypal calculus for attribute-based communication), $\langle e \,@\, \Pi \rangle.P$ means "send (the value of) e to all nodes satisfying Π, then continue as P"; dually, $(x \mid \Pi).P$ means "when receiving a message x such that Π holds, continue as P".

Integrating attribute-based communication in the ECA paradigm is not obvious. One coud try to add some primitives similar to AbC's, but this would yield a disharmonious *patchwork* of different paradigms, i.e., message-passing vs. memory-based events. Instead, in AbU we choose a different path: communications are reduced to events of the same kind ECA programs already deal with, that is, memory updates. For instance, a AbU rule like the following:

$$accessT \ @(\overline{role} = \texttt{logger}) : \overline{log} \leftarrow \overline{log} + accessT$$

means "when (my local) variable $accessT$ changes, add its value to the variable log of all nodes whose variable $role$ has value `logger`". Clearly, the update of log may trigger other rules on these (remote) nodes, and so on. We call this mechanism *attribute-based memory updates*, since it can be seen as the memory-based counterpart of attribute-based (message-passing) communication.

This smooth integration of communication within the ECA paradigm makes easier to extend to the distributed setting known results and techniques. As an example, we will provide a simple syntactic check to guarantee *stabilization*, i.e., that a chain of rule executions triggered by an external event will eventually terminate. Furthermore, we will discuss how implementations can leverage well-known optimization strategies for ECA rules, like the RETE algorithm [12].

Synopsis. After a summary of related work in Section 2, in Section 3 we introduce AbU, the new ECA-inspired calculus with attribute-based memory updates. After its syntax and operational semantics, we give a simple termination criterion based on a syntactic condition. Then, in Section 4 we show how to encode AbC components in AbU, providing encoding correctness and examples. In Section 5 we discuss some issues concerning the distributed implementation of AbU. Conclusions and directions for future work are in Section 6. Full proofs of the results can be found in the companion technical report [29].

2 Related Work

To the best of our knowledge, no work in literature aims at merging the two programming paradigms taken into consideration in the present paper. An approach close in spirit to ours is that based on associative memories, that is *tuple spaces*, as in the Linda language [18] and the KLAIM calculus [20]. In fact, also tuple spaces have events (insertion or deletion of tuples) that can be notified to nodes. Furthermore, tuple spaces can be inspected via pattern matching, which can be seen as a restricted form of attribute-based lookup. Despite these analogies, tuple spaces and AbU differ on many aspects: the latter is based on ECA rules, attribute-based communication is implemented by means of remote memory updates (and hence transparent to the nodes involved in the distributed system) and the logic for predicating over attributes is more expressive than simple pattern matching.

Concerning ECA programming, [19, 14] introduce IRON, a language based on ECA rules for the IoT domain. Following other work about ECA languages, [30, 31] present verification mechanisms to check properties on IRON programs, such as termination, confluence, redundant or contradicting rules. Other work proposes approaches to verify ECA programs by using Petri Nets [27] and BDD [13]. In [16, 17], the authors present a tool-supported method for verifying and controlling the correct interactions of ECA rules. All these works do not deal with distributed systems, hence communication is not taken into account.

AbC has been introduced and studied in [5, 4, 2] as a core calculus for SCEL [22], a language à la KLAIM with collective communication primitives. Focusing on the attribute-based communication model, AbC is well-suited to model Collective Adaptive Systems (CAS) [10] from a process standpoint (as opposed to Multi-Agents Systems (MAS) that follow a logical approach [32]; we refer to [4] for more details). Various extensions of AbC has been proposed [3, 6], as well as correct implementations in Erlang [21] and Golang [1, 24]. AbC, and its parent languages, adopt a message-passing communication mechanism and a sequential, process-like, execution flow, which are orthogonal with respect to the ECA rules setting. Since the goal of the present work is to extend the ECA programming style with attributed-based communication mechanisms, we will focus on the most fundamental primitives of AbC, omitting features not strictly necessary.

Some work combining message-passing primitives and shared-memory mechanisms have been recently proposed [8, 9]. In particular, the *m&m model* of [8] allows processes to both pass messages and share memory. This approach is increasingly used in practice (e.g., in data centers), as it seems to have great impact on the performance of distributed systems. An example application is given by *Remote Direct Memory Access* (RDMA) [9], that provides processes primitives both for send/receive communication, and for direct remote memory access. This mixed approach has been recently applied also in the MAS context [7], where the local behavior of agents is based on shared variables and the global behavior is based on message-passing. These results could be very helpful for the implementation of AbU, since a message-passing with shared-memory approach perfectly fits the attribute-based memory updates setting.

3 The AbU Calculus

We present here AbU, a calculus following the Event Condition Action (ECA) paradigm, augmented with attribute-based communication. This solution embodies the programming simplicity prerogative of ECA rules, but it is expressive enough to model complex coordination scenarios, typical of distributed systems.

3.1 Syntax

A AbU *system* S is either a *node*, of the form $R\langle \Sigma, \Theta \rangle$, or a parallel composition $\mathsf{S}_1 \parallel \mathsf{S}_2$ of systems. A *state* $\Sigma \in \mathbb{X} \to \mathbb{V}$, is a map from resource (names) in $\mathbb{X}$ to values in $\mathbb{V}$, while an *execution pool* $\Theta \subseteq \bigcup_{n\in\mathbb{N}} \mathbb{U}^n$ is a set of *updates*. An update upd is a finite list of pairs $(x, v) \in \mathbb{U}$, meaning that the resource x will take the value v after the execution of the update. Each node is equipped with a non-empty finite list R of *ECA rules*, generated by the following grammar.

$$
\begin{array}{lll}
\mathsf{rule} ::= \mathsf{evt} \gg \mathsf{act}, \mathsf{task} & & \mathsf{cnd} ::= \varphi \mid @\varphi \\
\mathsf{evt} ::= x \mid \mathsf{evt}\ \mathsf{evt} & & \varphi ::= \bot \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varepsilon \bowtie \varepsilon \\
\mathsf{act} ::= \epsilon \mid x \leftarrow \varepsilon\ \mathsf{act} \mid \overline{x} \leftarrow \varepsilon\ \mathsf{act} & & \varepsilon ::= v \mid x \mid \overline{x} \mid \varepsilon \otimes \varepsilon \\
\mathsf{task} ::= \mathsf{cnd} : \mathsf{act} & & x \in \mathbb{X} \quad v \in \mathbb{V}
\end{array}
$$

A ECA rule $\mathsf{evt} \gg \mathsf{act}, \mathsf{task}$ is guarded by an *event* evt, which is a non-empty finite list of resources. When one of these resources is modified, the rule is *fired*: the *default* action act and the task are *evaluated*. Evaluation does not change the resource states immediately; instead, it yields update operations which are added to the execution pools, and applied later on.

An action is a finite (possibly empty) list of assignments of value expressions to *local* x or *remote* $\overline{x}$ resources. The default action can access and update only local resources. On the other hand, a task consists in a condition cnd and an action act. A *condition* is a boolean expression, optionally prefixed with the modifier @. If @ is not present, the task is *local*: all resources in the condition and in the action refer to the local node (thus variables of the form $\overline{x}$ cannot occur). So, the condition is evaluated locally; if it holds, the action is evaluated. Otherwise, if @ is present, then the task is *remote*: the task $@\varphi : \mathsf{act}$ reads as "for *all* external nodes where φ holds, do act". On every node where the condition holds, the action is evaluated yielding an update to be added to that node's pool. So, in remote tasks each assignment in act is on remote resources only, but still they can use values from the local node. As an example, the task $@\top : \overline{x} \leftarrow \overline{x} + x$ means "add the value of this node's x to the x of every other node".

In the syntax for boolean expressions φ and value expressions ε we let implicit comparison operators, e.g., $\bowtie \in \{<, \leq, >, \geq, =, \neq\}$, and binary operations, e.g., $\otimes \in \{+, -, *, /\}$. In expressions we can have both local and remote instance of resources, although the latter can occur only inside remote tasks.

When we have a rule of the form $\mathsf{evt} \gg \epsilon, \mathsf{task}$, namely when we have rules with empty default action, we write more concisely $\mathsf{evt}\ \mathsf{task}$ in place of $\mathsf{evt} \gg \epsilon, \mathsf{task}$.

3.2 Semantics

Given a list R of rules and a set X of resources that have been modified, we define the set of *active* rules as $\mathsf{Active}(R, X) \triangleq \{\mathsf{evt} \gg \mathsf{act}, \mathsf{task} \in R \mid \mathsf{evt} \cap X \neq \varnothing\}$, namely the rules in R that listen on resources in X and, hence, that may be fired. Given an action act, its evaluation $[\![\mathsf{act}]\!]$ in the state Σ returns an update. Formally: $[\![x_1 \leftarrow \varepsilon_1 \ldots x_n \leftarrow \varepsilon_n]\!]\Sigma \triangleq (x_1, [\![\varepsilon_1]\!]\Sigma) \ldots (x_n, [\![\varepsilon_n]\!]\Sigma)$. The evaluation semantics for value expressions ε is standard. As we will see in a moment, the semantic function $[\![\cdot]\!]$ is applied only to local actions, that do not contain instances of external resources $\overline{x}$.

The *default updates* are the updates originated from the default actions of active rules in R, namely:

$$\mathsf{DefUpds}(R, X, \Sigma) \triangleq \{[\![\mathsf{act}]\!]\Sigma \mid \exists \mathsf{evt} \gg \mathsf{act}, \mathsf{task} \in \mathsf{Active}(R, X)\}$$

The *local updates* are the updates originated from the tasks of the active rules in R that act only locally (@ is not present in the tasks' condition) and that satisfy the task's condition, namely:

$$\mathsf{LocalUpds}(R, X, \Sigma) \triangleq \{[\![\mathsf{act}_2]\!]\Sigma \mid \exists \mathsf{evt} \gg \mathsf{act}_1, \varphi : \mathsf{act}_2 \in \mathsf{Active}(R, X)\,.\,\Sigma \models \varphi\}$$

The satisfiability relation is defined as: $\Sigma \models \varphi \triangleq [\![\varphi]\!]\Sigma = \mathsf{tt}$ (the evaluation semantics for boolean expressions φ is standard as well).

When we have a task containing the modifier @, an external node is needed to evaluate the task's condition. In our semantics, when a node needs to evaluate a task involving external nodes, it partially evaluates the task (with its own state) and then it sends the partially evaluated task to all other nodes. The latter, receive the task and complete the evaluation, potentially adding updates to their pool. In particular, the partial evaluation of tasks works as follows. With $\{\!|\mathsf{task}|\!\}\Sigma$ we denote the task obtained from task with each occurrence of x in the task's condition and the right-hand sides of the assignments in task's action replaced with the value $\Sigma(x)$. After that, each instance of $\overline{x}$ in the task's action is replaced with x and each instance of x in the task's action is replaced with $\overline{x}$ (this happens, in case, only on the left-hand sides of the assignments of the task's action). For instance, $\{\!|@(x \leq \overline{x}) : \overline{y} \leftarrow x + \overline{y}|\!\}[x \mapsto 1\ y \mapsto 0] = @(1 \leq x) : y \leftarrow 1 + y$. Note that, once the task is partially evaluated and sent to other nodes, then it becomes "syntactically local" for the receiving nodes[1]. Finally, we define the *external tasks* as $\mathsf{ExtTasks}(R, X, \Sigma) \triangleq \{\!|\mathsf{task}_1|\!\}\Sigma \ldots \{\!|\mathsf{task}_n|\!\}\Sigma$ such that for each $i \in [1..n]$ there exists a rule $\mathsf{evt} \gg \mathsf{act}, \mathsf{task}_i \in \mathsf{Active}(R, X)$ such that $\mathsf{task}_i = @\varphi : \mathsf{act}$, namely the tasks of active rules in R whose condition contains @ (i.e., tasks that require an external node to be evaluated).

The (small-step) semantics of a AbU system is modeled as a labeled transition system $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_2$ whose labels are given by $\alpha ::= T \mid \rhd T \mid \blacktriangleright T$ where T is a finite list of tasks. A transition can modify the state and the execution pool of the nodes. The semantics is *distributed*, in the sense that each node's semantics

[1] This means that we can evaluate the task's action with the semantic function $[\![\cdot]\!]$.

$$(\textsc{Exec})\ \frac{\begin{array}{c}\mathsf{upd} \in \Theta \quad \mathsf{upd} = (x_1, v_1) \dots (x_k, v_k) \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \\ \Theta'' = \Theta \setminus \{\mathsf{upd}\} \quad X = \{x_i \mid i \in [1..k] \wedge \Sigma(x_i) \neq \Sigma'(x)\} \\ \Theta' = \Theta'' \cup \mathsf{DefUpds}(R, X, \Sigma) \cup \mathsf{LocalUpds}(R, X, \Sigma) \quad T = \mathsf{ExtTasks}(R, X, \Sigma)\end{array}}{R\langle \Sigma, \Theta\rangle \xrightarrow{\triangleright T} R\langle \Sigma', \Theta'\rangle}$$

$$(\textsc{Input})\ \frac{\begin{array}{c}v_1, \dots, v_k \in \mathbb{V} \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad X = \{x_1, \dots, x_k\} \\ \Theta' = \Theta \cup \mathsf{DefUpds}(R, X, \Sigma) \cup \mathsf{LocalUpds}(R, X, \Sigma) \quad T = \mathsf{ExtTasks}(R, X, \Sigma)\end{array}}{R\langle \Sigma, \Theta\rangle \xrightarrow{\blacktriangleright T} R\langle \Sigma', \Theta'\rangle}$$

$$(\textsc{Disc})\ \frac{\Theta'' = \{[\![\mathsf{act}]\!]\Sigma \mid \exists i \in [1..n]\,.\,\mathsf{task}_i = @\varphi : \mathsf{act} \wedge \Sigma \models \varphi\} \quad \Theta' = \Theta \cup \Theta''}{R\langle \Sigma, \Theta\rangle \xrightarrow{\mathsf{task}_1 \dots \mathsf{task}_n} R\langle \Sigma, \Theta'\rangle}$$

$$(\textsc{Step})\ \frac{\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{T} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ \alpha \in \{\triangleright T, \blacktriangleright T\}$$

Fig. 1. AbU semantics for nodes and systems.

does not have a global knowledge about the system. The rules are in Fig. 1. A rule (Exec) executes an update picked from the pool; while a rule (Input) models an external modification of some resources. The execution of an update, or the change of resources, may trigger some rules of the nodes. Hence, after updating a node's state, the semantics of a node launches a *discovery phase*, with the goal of finding new updates to add to the local pool (or some pools of remote nodes), given by the activation of some rules. The discovery phase is composed by two parts, the local and the external one. A node $R\langle \Sigma, \Theta\rangle$ performs a local discovery by means of the functions $\mathsf{DefUpds}$ and $\mathsf{LocalUpds}$, that add to the local pool Θ all updates originated by the activation of some rules in R. Then, by means of the function $\mathsf{ExtTasks}$, the node computes a list of tasks that may update external nodes and sends it to all nodes in the system. This is modeled with the labels $\triangleright T$, produced by the rule (Exec), and $\blacktriangleright T$, produced by the rule (Input). On the other side, when a node receives a list of tasks (executing the rule (Disc) with a label T) it evaluates them and adds to its pool the actions generated by the tasks whose condition is satisfied.

Finally, the rule (Step) completes (on all nodes in the system) a discovery phase launched by a given node. Note that, not necessarily all nodes have to modify their pool (indeed, a task's condition may not hold in an external node). At the same time, the rule synchronizes the whole discovery phase, originated by a change in the state of a node in the system. When a node executes an action originating only local updates, the rule (Step) is applied with $\mathsf{S}_2' = \mathsf{S}_2$, producing the label $\triangleright\, \varepsilon$ or the label $\blacktriangleright\, \varepsilon$ (i.e., with an empty tasks' list). The parallel composition of systems $\parallel$ is associative and commutative.

Note that, in order to start the computation for a system of nodes, an input (i.e., an external modification of the environment) is needed since, at the beginning, all pools of all nodes in the system are empty.

Wave semantics. A AbU system $\mathsf{S} = R_1\langle \Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n\langle \Sigma_n, \Theta_n\rangle$ is *stable* when no more execution steps can be performed, namely when all execution pools Θ_i, for $i \in [1..n]$, are empty. We will use $R\langle\Sigma\rangle$ as a shorthand for $R\langle\Sigma, \varnothing\rangle$. So, a system is stable when it is of the form $R_1\langle \Sigma_1\rangle \parallel \ldots \parallel R_n\langle \Sigma_n\rangle$. In the case of a stable system, only the rule (Input) can be applied, i.e., an external environment change is needed to (re)start the computation.

We can define a big-step semantics $\mathsf{S} \rightsquigarrow \mathsf{S}'$ between stable systems, dubbed *wave semantics*, in terms of the small-step semantics. Let $\twoheadrightarrow^*$ be the transitive closure of $\twoheadrightarrow$, without occurrences of labels of the form $\blacktriangleright T$, namely $\twoheadrightarrow^*$ denotes a finite sequence of internal execution steps (with the corresponding discovery phases), without interleaving input steps. The wave semantics for a system S is:

$$(\textsc{Wave})\ \frac{\mathsf{S} = R_1\langle \Sigma_1\rangle \parallel \ldots \parallel R_n\langle \Sigma_n\rangle \quad \mathsf{S} \xrightarrow{\blacktriangleright T} \mathsf{S}'' \twoheadrightarrow^* \mathsf{S}' \quad \mathsf{S}' = R_1\langle \Sigma_1'\rangle \parallel \ldots \parallel R_n\langle \Sigma_n'\rangle}{\mathsf{S} \rightsquigarrow \mathsf{S}'}$$

The idea is that a (stable) system reacts to an external stimulus by executing a series of tasks (a "wave"), until it becomes stable again, waiting for the next stimulus. Note that, in the wave semantics inputs do not interleave with internal steps: this leaves the system the time to reach stability before the next input. If we allow arbitrary input steps during the computation, possibly a system may never reach stability since the execution pools could be never emptied. This assumption has a practical interpretation: in the IoT context, usually, external changes (in sensors) take much more time than internal computation steps [15].

3.3 A Working Example

Let us consider the scenario sketched in the Introduction, where an "access" node aims at sending its local access time to all "logger" nodes in the system. In other words, this node is activated when $accessT$ changes, namely when a new user performs access. Suppose now that the node, together with the time-stamp, aims at sending the IP address of the user and the name of the accessed resource. On the other side, the logger nodes record the access time, the IP address and the resource's name. Furthermore, suppose that these nodes contain a black-list of IP addresses. This list can be updated at run-time, by external entities communicating with logger nodes, so it may be the case that different logger nodes have different black-lists. A logger node that notices an access from a black-listed IP is in charge of notifying an intrusion detection system (IDS).

The system is formalized in AbU as follows. We suppose to have two access nodes and two logger nodes. We also suppose that log is a structured type, i.e., a list of records of the form $|IP; accessT; res|$. An append to the list log is given by $\mathsf{append}\ log\ |IP; accessT; res|$, with $|IP; accessT; res|.IP$ we denote the access of the field IP, and $\mathsf{tail}[log]$ returns the last record inserted in the list log.

$$\mathsf{S}_1 \triangleq R_a\langle \Sigma_1, \varnothing\rangle = R_a\langle [IP \mapsto \varepsilon \;\; accessT \mapsto 00{:}00{:}00 \;\; res \mapsto \texttt{camera}], \varnothing\rangle$$
$$\mathsf{S}_2 \triangleq R_a\langle \Sigma_2, \varnothing\rangle = R_a\langle [IP \mapsto \varepsilon \;\; accessT \mapsto 00{:}00{:}00 \;\; res \mapsto \texttt{lock}], \varnothing\rangle$$
$$\mathsf{S}_3 \triangleq R_l\langle \Sigma_3, \varnothing\rangle = R_l\langle [role \mapsto \texttt{logger} \;\; log \mapsto \varepsilon \;\; Blist \mapsto \varepsilon \;\; IDS \mapsto \varepsilon], \varnothing\rangle$$
$$\mathsf{S}_4 \triangleq R_l\langle \Sigma_4, \varnothing\rangle = R_l\langle [role \mapsto \texttt{logger} \;\; log \mapsto \varepsilon \;\; Blist \mapsto 167.123.23.2; \;\; IDS \mapsto \varepsilon], \varnothing\rangle$$
$$R_a \triangleq accessT \;\; @(\overline{role} = \texttt{logger}) : \overline{log} \leftarrow \mathsf{append}\ \overline{log}\ |IP; accessT; res|$$
$$R_l \triangleq log\ (\mathsf{tail}[log].IP \in Blist) : IDS \leftarrow \mathsf{tail}[log]$$

At the beginning, the AbU system $\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4$ is stable, since all pools are empty. At some point, an access is made on the resource `camera`, so the rule (Input) is applied on S_1, namely $R_a\langle \Sigma_1, \varnothing\rangle \xrightarrow{\blacktriangleright T} R_a\langle \Sigma_1', \varnothing\rangle$, where $\Sigma_1' = [accessT \mapsto 15{:}07{:}00 \;\; res \mapsto \texttt{camera} \;\; IP \mapsto 167.123.23.2]$ and

$$T = @(role = \texttt{logger}) : log \leftarrow \mathsf{append}\ log\ |167.123.23.2; 15{:}07{:}00; \texttt{camera}|$$

Now, a discovery phase is performed on all other nodes. In particular, we have: $R_a\langle \Sigma_2, \varnothing\rangle \xrightarrow{T} R_a\langle \Sigma_2, \varnothing\rangle$, $R_l\langle \Sigma_3, \varnothing\rangle \xrightarrow{T} R_l\langle \Sigma_3, \Theta\rangle$, and $R_l\langle \Sigma_4, \varnothing\rangle \xrightarrow{T} R_l\langle \Sigma_4, \Theta\rangle$. Here, the pool Θ is the set $\{(log, |167.123.23.2; 15:07:00; \texttt{camera}|)\}$. Now, let $\mathsf{S}_1' = R_a\langle \Sigma_1', \varnothing\rangle$, $\mathsf{S}_3' = R_l\langle \Sigma_3', \Theta\rangle$ and $\mathsf{S}_4' = R_l\langle \Sigma_4', \Theta\rangle$. The derivation tree for the resulting system $\mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3' \parallel \mathsf{S}_4'$ is depicted in Fig. 2[top]. For space reasons, we abbreviate rules' names and we omit the premises of leaf rules.

Now, the third and the fourth nodes can apply an execution step, since their pools are not empty. Suppose the third node is chosen, namely we have $R_l\langle \Sigma_3, \Theta\rangle \xrightarrow{\triangleright\varepsilon} R_l\langle \Sigma_3', \varnothing\rangle$, by applying the rule (Exec), and $\Sigma_3' = [role \mapsto \texttt{logger}$ $log \mapsto |167.123.23.2; 15:07:00; \texttt{camera}| \;\; Blist \mapsto \varnothing \;\; IDS \mapsto \varepsilon]$. Note that, in this case, no rule is triggered by the executed update. Since there is nothing to discover, all the other nodes do not have to update their pool and the derivation tree for the resulting system $\mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4'$, where $\mathsf{S}_3'' = R_l\langle \Sigma_3', \varnothing\rangle$ is given in Fig. 2[bottom]. Finally, the fourth node can execute, namely we have that $R_l\langle \Sigma_4, \Theta\rangle \xrightarrow{\triangleright\varepsilon} R_l\langle \Sigma_4', \Theta'\rangle$, by applying the rule (Exec). Here, $\Sigma_4' = [role \mapsto$ $\texttt{logger} \;\; log \mapsto |167.123.23.2; 15:07:00; \texttt{camera}| \;\; Blist \mapsto 167.123.23.2; \;\; IDS \mapsto \varepsilon]$ and $\Theta' = \{(IDS, |167.123.23.2; 15:07:00; \texttt{camera}|)\}$. In this case, the execution of the update triggers a rule of the node but the rule is local so, also in this case, the discovery phase does not have effect. The derivation tree for this step is analogous to the derivation tree for the previous one. Finally, with a further execution on the fourth node, we obtain the system $\mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4''$, where $\mathsf{S}_4'' = R_l\langle \Sigma_4'', \varnothing\rangle$ and $\Sigma_4'' = [role \mapsto \texttt{logger} \;\; log \mapsto |167.123.23.2; 15:07:00; \texttt{camera}| \;\; Blist \mapsto 167.123.23.2; \;\; IDS \mapsto |167.123.23.2; 15:07:00; \texttt{camera}|]$. Since all pools are empty, the resulting system is stable. This means that we can perform a wave semantics step:

$$(\textsc{Wave})\ \frac{\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4 \xrightarrow{\blacktriangleright T} \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3' \parallel \mathsf{S}_4' \quad \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3' \parallel \mathsf{S}_4' \xrightarrow{\triangleright\varepsilon} \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4' \xrightarrow{\triangleright\varepsilon} \ldots \xrightarrow{\triangleright\varepsilon} \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4''}{\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4 \rightsquigarrow \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4''}$$

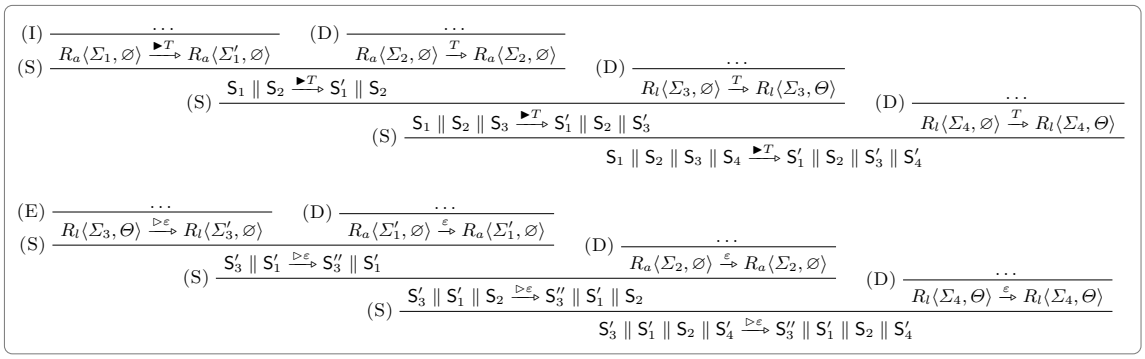

Fig. 2. Derivation trees for AbU semantic steps: (Input) [top] and (Exec) [bottom].

3.4 Termination Guarantee

The wave semantics (and, hence, a AbU system) may exhibit *internal divergence*: once an input step starts the computation, the subsequent execution steps may not reach a stable system, even if intermediate inputs are not performed.

Consider the case of the book "The Making of a fly", that reached the stellar selling price of $23,698,655.93 on Amazon, in 2001[2]. Two Amazon retailers, $profnath$ and *bordeebook*, used Amazon's automatic pricing primitives to set the price of their book's copy, depending the competitor's book price. The strategy of $profnath$ was to automatically set the price 0.99 times the *bordeebook*'s price; conversely, the strategy of *bordeebook* was to set the price 1.27 times the $profnath$'s price. Each retailer was not aware of the competitor's strategy. This scenario can be modeled with the following ECA rules:

when *bordeebook-price* changes, set $profnath$-*price* to *bordeebook-price* $* 0.99$
when $profnath$-*price* changes, set *bordeebook-price* to $profnath$-*price* $* 1.27$

It is easy to see that these rules generate a loop, leading to an uncontrolled raise of the book's price (as it happened). In order to prevent these situations, we define a simple syntactic condition on the rules that guarantees (internal) termination. In other words, each system satisfying the condition eventually becomes stable, after an initial input and without further interleaving inputs. This condition can be checked before the rules are deployed in the system.

The *output resources* of a AbU rule, namely the resources involved in the actions performed by the rule, are given by the the resources assigned in the default action and in the rule's task. The output resources of an action act are the set $\mathsf{Out}(\mathsf{act}) \triangleq \{x \mid \exists i \in \mathbb{N} \,.\, \mathsf{act}[i] = x \leftarrow \varepsilon \vee \mathsf{act}[i] = \overline{x} \leftarrow \varepsilon\}$. So, the output resources of a rule are $\mathsf{Out}(\mathsf{evt} \gg \mathsf{act}_1, \mathsf{cnd} : \mathsf{act}_2) \triangleq \mathsf{Out}(\mathsf{act}_1) \cup \mathsf{Out}(\mathsf{act}_2)$.

The *input resources* of a AbU rule are the resources that the rule listen on, namely the set $\mathsf{In}(\mathsf{evt} \gg \mathsf{act}, \mathsf{task}) \triangleq \{x \mid \exists i \in \mathbb{N} \,.\, \mathsf{evt}[i] = x\}$. Given a list R of AbU rules, its output resources $\mathsf{Out}(R)$ are the union of the output resources of

[2] https://www.michaeleisen.org/blog/?p=358.

all rules in the list. Analogously, its input resources $\mathsf{In}(R)$ are the union of the input resources of all rules in the list.

Definition 1 (ECA dependency graph). *Given a AbU system* S *such that* $\mathsf{S} = R_1\langle \Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n\langle \Sigma_n, \Theta_n\rangle$*, the* ECA dependency graph *of* S *is a directed graph* (N, E) *where the nodes* N *and the edges* E *are:*

$$N \triangleq \bigcup_{i\in[1..n]} \mathsf{In}(R_i) \cup \mathsf{Out}(R_i) \quad E \triangleq \left\{ (x_1, x_2) \;\middle|\; \begin{array}{l} \exists i \in [1..n]\, \exists j \in [1..k]\,.\, R_i = \mathsf{rule}_1 \ldots \mathsf{rule}_k \\ \wedge\, x_1 \in \mathsf{In}(\mathsf{rule}_j) \wedge x_2 \in \mathsf{Out}(\mathsf{rule}_j) \end{array} \right\}$$

The sufficient syntactic condition for the termination of the wave semantics (i.e., stabilization) consists in the acyclicity of the ECA dependency graph.

Proposition 1 (Termination of the wave semantics). *Given a AbU system* S*, if the ECA dependency graph of* S *is* acyclic*, then there exists a system* S' *such that* $\mathsf{S} \rightsquigarrow \mathsf{S}'$*.*

Therefore, a naive termination enforcing mechanism consists in computing the transitive closure E^+ of E and to check if it contains reflexive pairs, i.e., elements of the form (x, x), for a resource identifier x. If there are no reflexive elements then the graph is acyclic and the condition is fulfilled.

4 Encoding Attribute-based Communication

To showcase the generality of our calculus, in this section we encode the archetypal calculus AbC [5] in AbU. Our aim is not to prove that AbU subsumes AbC: the two calculi adopt different programming paradigms, with different peculiarities, that fit different application scenarios. Our goal here is to show that we can model within the ECA programming style the attribute-based communication.

4.1 The AbC Calculus

We focus on a minimal version of AbC [5], for which we define an operational semantics, on the line of [4]. As already pointed out, we do not aim for a full-fledged version of AbC, since the aim of this section is to encode in AbU the essence of the attribute-based communication, comprehensively expressed by the core version of AbC that we will present in the following paragraphs.

A AbC component C may be a process paired with an attribute environment, written $\Gamma : P$, or the parallel composition of two components, written $C_1 \parallel C_2$. An attribute environment Γ is a map from attribute identifiers $a \in \mathcal{A}$ to values $v \in \mathcal{V}$. Our syntax of AbC processes is as follows.

$$\begin{array}{rcl} P & ::= & \mathtt{0} \mid (x \mid \Pi).P \mid \langle e \mathbin{@} \Pi\rangle.P \mid [a := e]P \mid [\Pi]P \mid P_a + P_b \mid \mathrm{K} \\ \Pi & ::= & \mathtt{ff} \mid \mathtt{tt} \mid \Pi_1 \vee \Pi_2 \mid \Pi_1 \wedge \Pi_2 \mid \neg\Pi \mid e \bowtie e \quad \text{with } \bowtie \in \{<, \leq, >, \geq, =, \neq\} \\ e & ::= & v \mid a \mid x \mid \mathtt{this}.a \mid e \otimes e \quad \text{with } \otimes \in \{+, -, *, /\} \end{array}$$

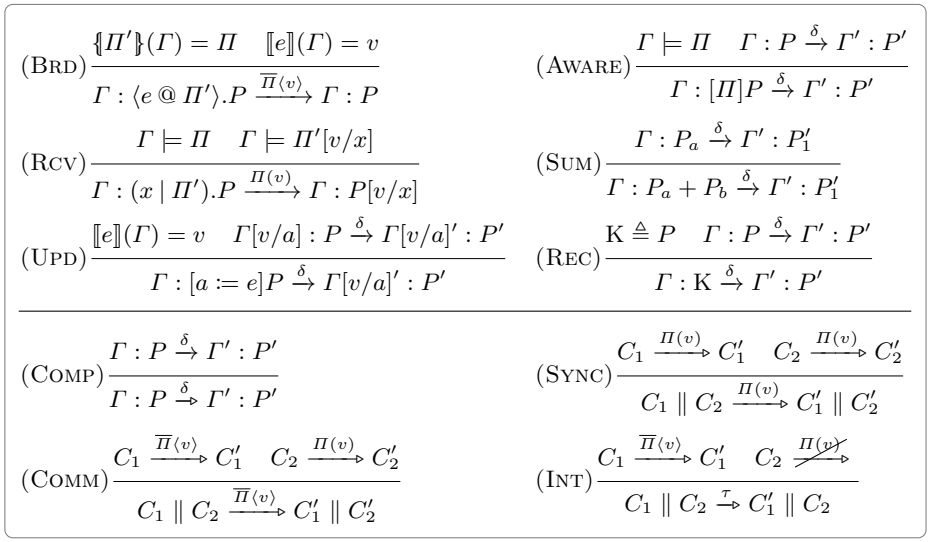

Fig. 3. AbC semantics for processes [top] and components [bottom].

In particular, the *input* $(x \mid \Pi)$ receives a message from components that satisfy the predicate Π, saving the message in the variable x. The *output* $\langle e \,@\, \Pi \rangle$ sends (the evaluation of) the expression e to all components that satisfy the predicate Π. The awareness process $[\Pi]P$ waits until Π is satisfied and then continues the execution as P. The other constructors are as in [5] (the inactive process $\mathtt{0}$, non-deterministic choice between $P_a + P_b$ and process calls K). Predicates Π and expressions e are standard. The reader can refer to [5] for more details.

We now briefly explain the semantics for AbC. $[\![e]\!](\Gamma)$ evaluates an expression e in the environment Γ and yields a value, while $[\![\Pi]\!](\Gamma)$ evaluates a predicate Π in Γ and yields $\mathtt{tt}$ or $\mathtt{ff}$. Their formal definition is straightforward, the only interesting cases are: $[\![a]\!](\Gamma) = [\![\mathtt{this}.a]\!](\Gamma) = \Gamma(a)$. When $[\![\Pi]\!](\Gamma)$ is $\mathtt{tt}$ we say that Γ satisfies Π, written $\Gamma \models \Pi$. We assume that processes do not have free variables, i.e., x is always under the scope of an input $(x \mid \Pi)$. Finally, in $\{\!|\Pi|\!\}(\Gamma)$ we substitute expressions of the form $\mathtt{this}.a$ with $\Gamma(a)$. The semantics for processes (Fig. 3[top]) and for components (Fig. 3[bottom]) is given by a labeled transition system, where a process label δ is of the form $\overline{\Pi}\langle v \rangle$ (output) or $\Pi(v)$ (input) and a component label λ can be either a process label δ or a silent action τ (i.e., a communication to a false predicate). Transitions rules in Fig. 3 are self-explanatory (symmetric rules are omitted). The parallel composition of components $\parallel$ is associative and commutative. The inactive process semantics is modeled as a communication on false, i.e., $\Gamma : \mathtt{0} \xrightarrow{\overline{\mathtt{ff}}\langle 0 \rangle} \Gamma : \mathtt{0}$.

Note that, if the rule (Comm) is applicable then Π cannot be false, since the rule (Rcv) cannot be applied with false predicates. When Π is false, (Int) is applied, representing an internal execution step of C_1. This rule applies also when C_2 is not ready (or it does not want) to communicate, allowing C_1 to progress.

4.2 Encoding AbC in AbU

Given a AbC component $\Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$, we define a AbU system $R_1\langle\Sigma_1\rangle \parallel \ldots \parallel R_n\langle\Sigma_n\rangle$ composed by n nodes, where the state Σ_i of the i^{th} node is given by the i^{th} attribute environment Γ_i (with some modifications). All nodes' pools are initially empty. In order to simulate process communication, we add to each node a special resource msg. If a node wants to communicate a message, it has to update the msg resource of all the selected communication partners. The execution of each AbC component is inherently sequential while AbU nodes follow an event-driven architecture. In order to simulate AbC's causality, we associate each generated AbU rule with a special resource, a *rule flag*, whose purpose is to enable and disable the rule. The sequential execution flow of an AbC component is reconstructed modifying the *active* flag of the rules: this simulates a "token" that rules have to hold in order to be executed. Formally, the state of the i^{th} nodes is augmented as follows:

$$\Sigma_i = \Gamma_i \cup \{(msg, 0)\} \cup \bigcup_{j\in[1..n]} \mathcal{R}^j(P_j)$$

A rule is generated for each process instance present in the AbC component to be encoded. To this end, each node is augmented with all rule flags, of all rules, given by the translation of all processes of the AbC component. Rule flags are resource of the form $P_h r_i$, with $h \in [1..n]$ and $i \geq 0$, representing the i^{th} rule generated from the component h. The function $\mathcal{R}^h$, given a process of the component h, with $h \in [1..n]$, computes the resources to add to the nodes[3].

$\mathcal{R}^h$ returns $\varnothing$ for the inactive process and for process calls, i.e., $\mathcal{R}^h(\mathsf{0}) \triangleq \mathcal{R}^h(\mathrm{K}) \triangleq \varnothing$, and nothing is added. For the other processes, it returns $\mathcal{R}^h(P) \triangleq \{(P_h r_0, \mathsf{ff})\} \cup \mathcal{R}^h(P, 0)$. The flag $P_h r_0$ is the starting point of the computation, indeed it does not represent any actual rule, and it is set to tt in order to start the computation. The function $\mathcal{R}^h(P, i)$, for $i \geq 0$, is defined inductively on the structure of P. In the base cases $P = \mathsf{0}$ and $P = \mathrm{K}$, it returns $\varnothing$ (i.e., nothing is added), otherwise it is defined as follows, where the auxiliary function Next generates a fresh index for the next rule to add.

If the process is an input $P = (x \,|\, \Pi).P'$, we add the flag for the current rule and another resource for the variable x: $\{(x, 0), (P_h r_j, \mathsf{ff})\} \cup \mathcal{R}^h(P', j)$, given $\mathsf{Next}(i) = j$. If the process is a non-deterministic choice, i.e., $P = P_a + P_b$, we add two flags, one for each branch, that will originate two different rules: $\{(P_h r_j, \mathsf{ff}), (P_h r_k, \mathsf{ff})\} \cup \mathcal{R}^h(P_a, j) \cup \mathcal{R}^h(P_b, k)$, given $\mathsf{Next}(i) = j, \mathsf{Next}(j) = k$. In all other cases, i.e., $P = [\Pi]P'$, $P = [a := e]P'$ or $P = \langle e \mathbin{@} \Pi\rangle.P'$, we add the flag for the current rule: $\{(P_h r_j, \mathsf{ff}) \cup \mathcal{R}^h(P', j)$, given $\mathsf{Next}(i) = j$.

Concerning AbU rules, we adopt the following mechanism. The i^{th} generated rule, of the component h, listens on the rule flag $P_h r_i$: when the latter becomes tt, the rule can execute. Its execution disables $P_h r_i$ (it is set to ff) and enables the next rule, setting the flag $P_h r_j$, with $j = \mathsf{Next}(i)$, to tt. In this way, the execution token can be exchanged between rules. The function $\mathcal{T}^h$, given a process of the component h, with $h \in [1..n]$, generates the rules to add to the translation.

[3] $\mathcal{R}^h$ is parametric in h, since rules are binded to the component generating them.

It relies on Next, that outputs a fresh index for the next rule to generate. We assume that Next in $\mathcal{T}^h$ is consistent with Next in $\mathcal{R}^h$, i.e., they have to produce the same sequence of indexes given a specific process. The function $\mathcal{T}^h(P, i)$, for $i \geq 0$, is defined inductively on the structure of P. In the base case $P = \mathtt{0}$, it returns ϵ (i.e., nothing is added), otherwise it is defined as follows.

If the process is a call to K, a new *call* rule is added. This rule enables the first flag (the dummy rule r_0) of the called process, defined by K.

$P = \mathrm{K}$	$\mathrm{K} \triangleq P_k$
$P_h r_i \ (P_h r_i = \top) : P_h r_i \leftarrow \bot \, P_k r_0 \leftarrow \top$	

If the process is an input x on the predicate Π, a new *receive* rule is added. The rule checks the condition given by the translation of the predicate Π. Here, Repl replaces, in a given AbU boolean expression, every instance of a specific service (x in this case) with msg. As an example, the predicate $\Pi = x < n$ is translated to $\mathsf{Repl}(\mathcal{T}(\Pi), x) = msg < n$. When the condition is satisfied, the rule saves the value msg received from the sender (in the resource x), ends the communication and enables the next rule.

$P = (x \mid \Pi).P'$	$\mathsf{Next}(i) = j$
$P_h r_i \ (P_h r_i = \top \wedge \mathsf{Repl}(\mathcal{T}(\Pi), x)) : x \leftarrow msg \ P_h r_i \leftarrow \bot \, P_h r_j \leftarrow \top \ \ \mathcal{T}^h(P', j)$	

If the process is a non-deterministic choice between P_a and P_b, two new *choice* rules are added. Both rules listen to the same flag, so the scheduler can choose non-deterministically the one to execute. The action of the first choice rule enables the next rule given by the translation of P_a, while the action of the second choice rule enables the next rule given by the translation of P_b.

$P = P_a + P_b$	$\mathsf{Next}(i) = j, \ \mathsf{Next}(j) = k$
$P_h r_i \ (P_h r_i = \top) : P_h r_i \leftarrow \bot \, P_h r_j \leftarrow \top \ \ \mathcal{T}^h(P_a, j)$	
$P_h r_i \ (P_h r_i = \top) : P_h r_i \leftarrow \bot \, P_h r_k \leftarrow \top \ \ \mathcal{T}^h(P_b, j)$	

If the process is waiting on the predicate Π (awareness), a new *awareness* rule is added, that listens on the resources contained in Π. The latter are retrieved by the function Vars that inspects the predicate Π and returns a list of resource identifiers. In particular, variables x are left untouched, while AbC expressions a and $\mathtt{this}.a$ are both translated to the resource a. The condition in the rule's task is the translation of Π. When it is satisfied, the next rule is enabled.

$P = [\Pi]P'$	$\mathsf{Next}(i) = j$
$P_h r_i \ \mathsf{Vars}(\Pi) \ (P_h r_i = \top \wedge \mathcal{T}(\Pi)) : P_h r_i \leftarrow \bot \, P_h r_j \leftarrow \top \ \ \mathcal{T}^h(P', j)$	

If the process updates the attribute a with the expression e, an *update* rule is added, assigning the translation of e to a and enabling the next rule.

$P = [a \coloneqq e]P'$	$\mathsf{Next}(i) = j$
$P_h r_i \ (P_h r_i = \top) : a \leftarrow \mathcal{T}(e) \ P_h r_i \leftarrow \bot \, P_h r_j \leftarrow \top \ \ \mathcal{T}^h(P', j)$	

If the process is an output of the expression e on the predicate Π, a new *send* rule is added. The rule checks the condition given by the translation of the predicate Π. Note that, in the AbC semantics, the predicate is partially evaluated

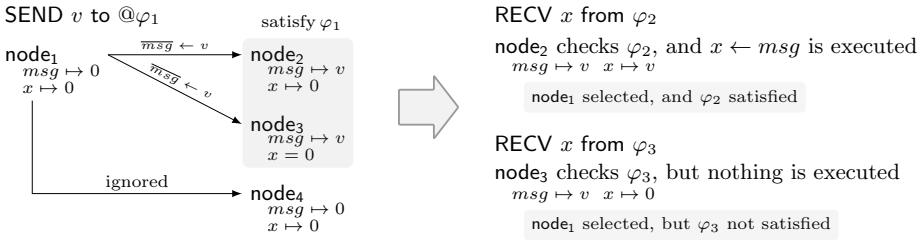

Fig. 4. Communication: a receive phase (right) after a send phase (left).

before the send, namely expressions of the form `this`.a are substituted with $\Gamma(a)$. To simulate this mechanism in AbU we use an auxiliary transformation Ext that takes a AbC predicate Π and returns its translation $\mathcal{T}(\Pi)$ where each instance (in Π) of an attribute a not prefixed by `this`. is translated to $\overline{a}$. As an example, the predicate $\Pi = \texttt{this}.n < n$ is translated to $\mathsf{Ext}(\Pi) = n < \overline{n}$. For each external node satisfying the predicate Π, the rule writes the translation of e to the external node resource msg (with $\overline{msg} \leftarrow \mathcal{T}(e)$). Outputs are non-blocking, so the rule has a default code, executed without caring about the satisfaction of the condition. It disables the current rule and enables the next one.

$$P = \langle e \mathbin{@} \Pi \rangle.P' \qquad \mathsf{Next}(i) = j$$
$$P_h r_i \gg P_h r_i \leftarrow \bot \; P_h r_j \leftarrow \top, @(P_h r_i = \top \wedge \mathsf{Ext}(\Pi)) : \overline{msg} \leftarrow \mathcal{T}(e) \;\; \mathcal{T}^h(P', j)$$

Finally, the translation of predicates $\mathcal{T}(\Pi)$ and expressions $\mathcal{T}(e)$ is recursively defined on Π and e, respectively. Its definition is straightforward, the only interesting cases are: $\mathcal{T}(\texttt{this}.a) \triangleq \mathcal{T}(a) \triangleq a$. To start the execution of the translated system, an (Input) is needed, enabling all rule flags $P_h r_0$, of all nodes.

In Fig. 4 we graphically explain how an attribute-based communication is performed in AbU, by means of attribute-based memory updates. The node node$_1$ aims to send the value v to nodes node$_2$ and node$_3$, since they satisfy $\varphi_1 = \mathsf{Ext}(\Pi_1)$. So, it updates with v the resource msg on the remote nodes node$_2$ and node$_3$. On the other side, node$_2$ and node$_3$ check if some node aims to communicate and node$_1$ is indeed selected. Since node$_1$ satisfies $\varphi_2 = \mathsf{Repl}(\mathcal{T}(\Pi_2), x)$ and does not satisfy $\varphi_3 = \mathsf{Repl}(\mathcal{T}(\Pi_3), x)$, only node$_2$ accepts the value v, saving it in the resource x, while node$_3$ ignores the communication.

In the following, we denote with $\mathcal{T}(C)$ the AbU encoding of C, where node states are defined as explained above, node pools are empty and nodes' ECA rules are generated by $\mathcal{T}$ (given the process of C).

Encoding example. Given N agents, each associated with an integer in $[1..N]$, we wish to find one holding the maximum value. This problem can be modeled in AbC by using one component type P with two attributes: s, initially set to 1, indicating that the current component is the max; and n, that stores the component's value. Formally, the process P (with $\mathrm{Max} \triangleq P$) is:

$$P = [s = 1]\,(\,\langle n \mathbin{@} n \leq \texttt{this}.n \rangle\,.\,\mathrm{Max} + (x \mid x \geq \texttt{this}.n)\,.\,[s := 0]\mathsf{0}\,)$$

P waits until s becomes 1 and then either: it sends its own value n to all other components with smaller n; or it receives (on x) a value from another component with a greater n and sets s to 0. Supposing $N = 3$, the problem is modeled in AbC with the component $C_{\max} = [s \mapsto 1\ n \mapsto 1] : P \parallel [s \mapsto 1\ n \mapsto 2] : P \parallel [s \mapsto 1\ n \mapsto 3] : P$. This AbC component translates to AbU as follows.

$$
\begin{array}{lr}
R\langle[msg{\mapsto}0\ n{\mapsto}1\ x{\mapsto}0\ s{\mapsto}0\ P_1r_0{\mapsto}\mathsf{ff}\ldots P_1r_6{\mapsto}\mathsf{ff}]\rangle & \\
R\langle[msg{\mapsto}0\ n{\mapsto}2\ x{\mapsto}0\ s{\mapsto}0\ P_1r_0{\mapsto}\mathsf{ff}\ldots P_1r_6{\mapsto}\mathsf{ff}]\rangle & \\
R\langle[msg{\mapsto}0\ n{\mapsto}3\ x{\mapsto}0\ s{\mapsto}0\ P_1r_0{\mapsto}\mathsf{ff}\ldots P_1r_6{\mapsto}\mathsf{ff}]\rangle & \\
R = P_1r_0\ (P_1r_0 = \top \wedge s = 1) : P_1r_0 \leftarrow \bot\ P_1r_1 \leftarrow \top & \text{aware rule} \\
\quad P_1r_1\ (P_1r_1 = \top) : P_1r_1 \leftarrow \bot\ P_1r_2 \leftarrow \top & \text{choice}_1 \text{ rule} \\
\quad P_1r_1\ (P_1r_1 = \top) : P_1r_1 \leftarrow \bot\ P_1r_3 \leftarrow \top & \text{choice}_2 \text{ rule} \\
\quad P_1r_2 \gg P_1r_2 \leftarrow \bot\ P_1r_4 \leftarrow \top, @(P_1r_2 = \top \wedge \overline{n} \leq n) : \overline{msg} \leftarrow n & \text{send rule} \\
\quad P_1r_4\ (P_1r_4 = \top) : P_1r_4 \leftarrow \bot\ P_1r_0 \leftarrow \top & \text{call rule} \\
\quad P_1r_3\ (P_1r_3 = \top \wedge msg \geq n) : x \leftarrow msg\ P_1r_3 \leftarrow \bot\ P_1r_5 \leftarrow \top & \text{receive rule} \\
\quad P_1r_5\ (P_1r_5 = \top) : s \leftarrow 0\ P_1r_5 \leftarrow \bot\ P_1r_6 \leftarrow \top & \text{update rule}
\end{array}
$$

4.3 Correctness of the Encoding

Since a AbU node contains auxiliary resources, in addition to those corresponding to AbC attributes, we have to establish a notion of compatibility between AbU node states and AbC attribute environments. Given a AbU node state Σ and a AbC attribute environment Γ, we say that Σ is *compatible* with Γ, written $\Sigma \succeq \Gamma$, when for each $(a, v) \in \Gamma$ there exists $(a, v) \in \Sigma$ (i.e., $\Gamma \subseteq \Sigma$). This basically means that Σ agrees, at least, on all attributes of Γ. This notion can be extended to systems and components. Given a AbC component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ and a AbU system $\mathsf{S} = R_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n\langle\Sigma_n, \Theta_n\rangle$, we say that S is *compatible* with C, written $\mathsf{S} \succeq C$, when $\Sigma_i \succeq \Gamma_i$, for each $i \in [1..n]$.

Recall that, the AbU translation $\mathcal{T}(C)$ of C yields n (one for each process) initial rule flags $P_1r_0, \ldots, P_nr_0$, initially set to ff. In order to start the computation of $\mathcal{T}(C)$, the latter have to be initialized (i.e., set to tt). In this regards, we assume an initial *input phase*, comprising n AbU (Input) steps, enabling all initial rule flags (without interleaving execution steps). Let $\twoheadrightarrow^*$ be the transitive closure of $\rightarrow\!\!\triangleright$ without occurrences of labels of the form $\triangleright\, T$. In other words, $\twoheadrightarrow^*$ denotes a finite sequence of internal input steps (with the corresponding discovery phases), without interleaving execution steps.

Now we are ready to state the correctness of the AbC encoding. The following Thm. 1 says that if a AbC component performs some computation steps, producing a residual component C', then the AbU translation of C, after an initial input phase, is able to perform an arbitrary number of computation steps, yielding a residual system attribute compatible with C'. This basically means that $\mathcal{T}(C)$ is able to "simulate" each possible execution of C.

Theorem 1 (AbC to AbU correctness). *Consider a AbC component C and its corresponding AbU encoding $\mathsf{S} = \mathcal{T}(C)$. Then, for all C' such that $C \rightarrow\!\!\triangleright^* C'$ there exists S' such that $\mathsf{S} \twoheadrightarrow^* \rightarrow\!\!\triangleright^* \mathsf{S}'$ and $\mathsf{S}' \succeq C'$.*

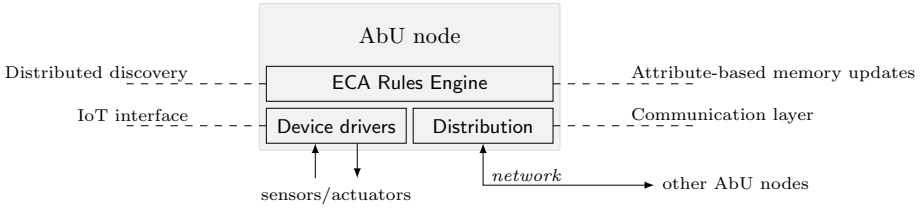

Fig. 5. High-level view of a AbU node implementation.

5 Towards a Distributed Implementation

In impementing AbU, we can basically follow two approaches. We can implement the calculus from scratch, dealing with all the problems related to a distributed infrastructure; or we can extend an existing distributed language with an abstraction layer to support ECA rules and their event-driven behavior. The latter approach can be less efficient, but more suitable for fast prototyping.

In any case, we have to deal with the intrinsic issues of distributed systems. In particular, by the CAP theorem [23] we cannot have, at the same time, consistency, availability and partition-tolerance. Hence, some compromises have to be taken, depending on the application context. For instance, in a scenario with low network traffic we can aim for correctness, implementing a robust, but slow, communication protocol. Vice versa, when nodes exchange data at a high rate (or when the network is not stable), communication should take very short time, hence we may prefer to renounce to consistency in favour of eventual consistency.

For these reasons, a flexible and modular implementation is mandatory, where modules can be implemented in different ways, depending on the application context. Hence, we present a modular architecture suitable to implement AbU nodes (see Fig. 5). A AbU node consists in a state (mapping resources to values), an execution pool (a set of updates to execute) and a list of ECA rules (modeling the node's behavior). A *ECA rules engine module* is in charge of executing the updates in the pool and to discover new rules to trigger, potentially on external nodes (distributed discovery). This module also implements the attribute-based memory updates mechanism and deals with IoT inputs (from sensors) and outputs (to actuators), which are accessed by means of a dedicated interface. A separate *Device drivers module* translates low-level IoT devices primitives to high-level signals for the rule engine and vice versa. The *Distribution module* is in charge of joining a cluster of AbU nodes and exchanging messages with them. It embodies all distributed infrastructure-related aspects, that can be tuned to meet the desired context-related requirements. Moreover, it provides the communication APIs needed by the rule engine to implement the (distributed) discovery phase (and, in turn, attributed-based memory updates). For instance, the labels $\blacktriangleright\, T$ and $\rhd\, T$ of the AbU semantics generate a broadcast communication.

In some respects, AbU is quite close to AbC, so we can borrow from one of its implementation the mechanisms that can be easily adapted to AbU. In

particular, we can exploit the GoAt [1, 24] library, in order to implement the Distribution module. GoAt is written in Golang, so we can delegate the communication layer to a Go routine, encapsulating the send and receive primitives of AbC and the cluster infrastructure, both provided by GoAt. Finally, the Device drivers module can be built on top of GOBOT [25], a mature Go library for the IoT ecosystem, with a great availability of IoT devices drivers. In Fig. 5, we show a diagram describing the structure of a AbU node (here, the Device drivers and the Distribution modules can exploit GOBOT and GoAt, respectively). At the time of writing, we are developing a prototype implementation for the AbU calculus, written in Golang and following the modular architecture sketched above. The Distribution module is now based on HashiCorp's Memberlist [26], a popular Go library for cluster membership and failures detection that uses a gossip based protocol. We plan to integrate the module with GoAt in the near future.

6 Conclusion

In this paper we have introduced AbU, a new calculus merging the simplicity of ECA programming with *attribute-based memory updates*. This new time-coupled, space-uncoupled interaction mechanism can be seen as the memory-based counterpart of attribute-based communication hinged on message-passing, and fits neatly within the ECA programming paradigm. We have shown how AbC components can be encoded in AbU systems; this result is not meant to prove that AbU subsumes AbC, but to highlight that it is possible to encode attribute-based communication within the ECA rules programming paradigm. Furthermore, we have provided a syntactic termination criterion for AbU systems, in order to assure that a AbU system does not exhibit divergent behaviors due to some cyclic interactions between nodes rules. Finally, we have discussed how the proposed calculus can be implemented, in a fully-distributed and IoT-ready setting.

Future work. The present work is the basis for several research directions. First, we plan to encode in AbU a real-world ECA language like IRON (in particular, its core version presented in [15]), similarly to what we have done for AbC. Then, we are interested in porting to AbU the verification techniques developed for IRON and other ECA languages [27, 30, 31]. Efficient distributed implementations of AbU could be obtained by extending the RETE algorithm [12] with the attribute-based memory updates mechanism. The latter can be implemented using RPCs or message-passing, taking inspiration from the implementations of AbC [1, 21, 24], as discussed in Section 5. Another interesting issue is *distributed runtime verification and monitoring*, in order to detect violations at runtime of given correctness properties, e.g., expressed in temporal logics like the μ-calculus [28]. These would be useful, for instance, to extend (and refine) the termination criterion presented in Section 3. Similarly, we can define syntactic criteria and corresponding verification mechanisms to guarantee *confluence*. Indeed, in some practical IoT scenarios, it is important to ensure that execution order does not impact the overall behavior (which is, basically, a sort of rule determinism). Finally, we can think of defining suitable behavioural equivalences for AbU systems, e.g., based on bisimulations, to compare systems with their specifications.

References

1. Abd Alrahman, Y., De Nicola, R., Garbi, G.: *GoAt*: Attribute-based interaction in Google Go. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th Int. Symp., ISoLA 2018, Limassol, Cyprus, Nov 5-9, 2018, Proceedings, Part III. LNCS, vol. 11246, pp. 288–303. Springer (2018), https://doi.org/10.1007/978-3-030-03424-5_19
2. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. pp. 1–18. Springer, Cham (2016)
3. Abd Alrahman, Y., De Nicola, R., Loreti, M.: A calculus for collective-adaptive systems and its behavioural theory. Information and Computation **268**, 104457 (2019), https://www.sciencedirect.com/science/article/pii/S0890540119300732
4. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. **192**, 102428 (2020). https://doi.org/10.1016/j.scico.2020.102428
5. Abd Alrahman, Y., De Nicola, R., Loreti, M., Tiezzi, F., Vigo, R.: A calculus for attribute-based communication. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. pp. 1840–1845. ACM, New York, NY, USA (2015), https://doi.org/10.1145/2695664.2695668
6. Abd Alrahman, Y., Garbi, G.: A distributed API for coordinating AbC programs. International Journal on Software Tools for Technology Transfer (feb 2020), https://doi.org/10.1007/s10009-020-00553-4
7. Abd Alrahman, Y., Perelli, G., Piterman, N.: Reconfigurable interaction for mas modelling. In: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems. pp. 7–15. AAMAS '20, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2020)
8. Aguilera, M.K., Ben-David, N., Calciu, I., Guerraoui, R., Petrank, E., Toueg, S.: Passing messages while sharing memory. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing. pp. 51–60. PODC '18, ACM, New York, NY, USA (2018), https://doi.org/10.1145/3212734.3212741
9. Aguilera, M.K., Ben-David, N., Guerraoui, R., Marathe, V., Zablotchi, I.: The impact of RDMA on agreement. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 409–418. PODC '19, ACM, New York, NY, USA (2019), https://doi.org/10.1145/3293611.3331601
10. Anderson, S., Bredeche, N., Eiben, A., Kampis, G., van Steen, M.: Adaptive collective systems: herding black sheep (2013)
11. Balliu, M., Merro, M., Pasqua, M., Shcherbakov, M.: Friendly fire: Cross-app interactions in IoT platforms. ACM Trans. Priv. Secur. **24**(3) (2021), https://doi.org/10.1145/3444963
12. Berstel, B.: Extending the RETE algorithm for event management. In: Proc. of 9th Int. Symp. on Temporal Representation and Reasoning. pp. 49–51. IEEE (2002)
13. Beyer, D., Stahlbauer, A.: BDD-based software verification. Int. J. Softw. Tools Tech. Transf. **16**(5), 507–518 (2014), https://doi.org/10.1007/s10009-014-0334-1
14. Cacciagrano, D.R., Culmone, R.: Formal semantics of an IoT-specific language. In: 32nd Int. Conf. on Advanced Information Networking and Applications Workshops (WAINA). pp. 579–584 (2018). https://doi.org/10.1109/WAINA.2018.00148
15. Cacciagrano, D.R., Culmone, R.: IRON: Reliable domain specific language for programming IoT devices. Internet Things **9**, 100020 (2020), https://doi.org/10.1016/j.iot.2018.09.006

16. Cano, J., Delaval, G., Rutten, E.: Coordination of ECA rules by verification and control. In: Kühn, E., Pugliese, R. (eds.) Coordination Models and Languages. pp. 33–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
17. Cano, J., Rutten, E., Delaval, G., Benazzouz, Y., Gurgen, L.: ECA rules for IoT environment: A case study in safe design. In: Proceedings of the 8th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW). pp. 116–121. IEEE Computer Society, USA (2014), https://doi.org/10.1109/SASOW.2014.32
18. Carriero, N., Gelernter, D.: The s/net's linda kernel (extended abstract). In: Proc. of the 10th ACM Symposium on Operating Systems Principles. p. 160. SOSP '85, ACM, New York, NY, USA (1985), https://doi.org/10.1145/323647.323643
19. Corradini, F., Culmone, R., Mostarda, L., Tesei, L., Raimondi, F.: A constrained ECA language supporting formal verification of WSNs. In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops. pp. 187–192 (2015). https://doi.org/10.1109/WAINA.2015.109
20. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering **24**(5), 315–330 (1998). https://doi.org/10.1109/32.685256
21. De Nicola, R., Duong, T., Loreti, M.: Provably correct implementation of the AbC calculus. Science of Computer Programming **202**, 102567 (2021), http://www.sciencedirect.com/science/article/pii/S0167642320301751
22. De Nicola, R., Latella, D., Lafuente, A.L., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL language: Design, implementation, verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems, LNCS, vol. 8998, pp. 3–71. Springer (2015), https://doi.org/10.1007/978-3-319-16310-9_1
23. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News **33**(2), 51–59 (2002), https://doi.org/10.1145/564585.564601
24. giulio-garbi.github.io: GoAt, https://giulio-garbi.github.io/goat/
25. gobot.io: GOBOT, https://gobot.io/
26. hashicorp.com: Memberlist, https://github.com/hashicorp/memberlist/
27. Jin, X., Lembachar, Y., Ciardo, G.: Symbolic verification of ECA rules. In: Moldt, D. (ed.) Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13), Milano, Italy, June 24-25, 2013. vol. 989, pp. 41–59. CEUR-WS.org (2013), http://ceur-ws.org/Vol-989/paper17.pdf
28. Miculan, M.: On the formalization of the modal μ-calculus in the Calculus of Inductive Constructions. Inf. Comput. **164**(1), 199–231 (2001), https://doi.org/10.1006/inco.2000.2902
29. Miculan, M., Pasqua, M.: A calculus for attribute-based memory updates (supplementary material) (Jul 2021), https://doi.org/10.5281/zenodo.5057165
30. Vannucchi, C., Diamanti, M., Mazzante, G., Cacciagrano, D.R., Corradini, F., Culmone, R., Gorogiannis, N., Mostarda, L., Raimondi, F.: vIRONy: A tool for analysis and verification of ECA rules in intelligent environments. In: 2017 International Conference on Intelligent Environments, IE 2017, Seoul, Korea (South), August 21-25, 2017. pp. 92–99. IEEE (2017), https://doi.org/10.1109/IE.2017.32
31. Vannucchi, C., Diamanti, M., Mazzante, G., Cacciagrano, D.R., Culmone, R., Gorogiannis, N., Mostarda, L., Raimondi, F.: Symbolic verification of event-condition-action rules in intelligent environments. J. Reliab. Intell. Environ. **3**(2), 117–130 (2017), https://doi.org/10.1007/s40860-017-0036-z
32. Wooldridge, M.: Reasoning about rational agents. Intelligent robotics and autonomous agents. The MIT Press, Cambridge, Massachussetts/London (2000)