Theoretical Computer Science 958 (2023) 113841

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

AbU: A calculus for distributed event-driven programming with attribute-based interaction ☆

Michele Pasqua [a,*], Marino Miculan [b]

[a] University of Verona, Strada le Grazie 15, Verona, 37134, Verona, Italy
[b] University of Udine, Via delle Scienze 206, Udine, 33100, Udine, Italy

ARTICLE INFO

Article history:
Received 10 January 2022
Received in revised form 14 February 2023
Accepted 21 March 2023
Available online 3 April 2023

Keywords:
ECA rules
Attribute-based communication
Distributed systems
Formal methods
Edge computing
IoT programming

ABSTRACT

In recent years, event-driven programming languages, in particular those based on Event Condition Action (ECA) rules, have emerged as a promising paradigm for implementing ubiquitous and pervasive systems. These implementations are mostly centralized, where a single server (often in the cloud) collects and processes all the inputs from the environment. In fact, placing the computation on the nodes interacting with the environment requires suitable abstractions for effective communication and coordination of (possibly large) ensembles of these distributed components — abstractions that current ECA languages are still missing.

To this end, in this paper we present AbU, a calculus for modeling and reasoning about ECA-based systems with *attribute-based communication*. The latter is an interaction model recently introduced for the coordination of (possibly large) families of nodes: communication is similar to broadcast but the actual receivers are selected on the spot, by means of *predicates* over nodes properties. Thus, the programmer can specify interactions between nodes in a declarative way, abstracting from details such as nodes identity, number, or even their existence, without the need for a central server: the computation is moved on the "edge", thus improving reliability, scalability, privacy and security.

After having defined syntax and formal semantics of AbU, we showcase its expressiveness by providing some example applications and the encoding of AbC, the archetypal calculus with attribute-based communication. Then, we focus on two key properties of reactive systems: *stabilization* (i.e., termination of internal steps) and *confluence*. For both these properties we provide formal semantic definition, sufficient syntactic conditions on AbU systems, and algorithms to statically check such conditions. Hence, AbU is both a basis for the formal analysis of event-driven architectures with attributed-based interaction, and a reference model for a full-fledged language for IoT and edge computing.

1. Introduction

The ever-growing ubiquitous and pervasive systems, like the (Industrial) Internet of Things, smart homes, smart cities, autonomous agents, etc, are characterized by many complex computational aspects, such as distributed computing, dynamic

☆ This article belongs to Section B: Logic, semantics and theory of programming, Edited by Don Sannella.

* Corresponding author.
E-mail addresses: michele.pasqua@univr.it (M. Pasqua), marino.miculan@uniud.it (M. Miculan).

https://doi.org/10.1016/j.tcs.2023.113841

network topologies, context-awareness, real-time data generated by sensors and users, strict latency bounds, etc. In order to deal with these aspects, applications for ubiquitous and pervasive systems have to efficiently handle *events*, such as sensor inputs, context changes, and changes in the internal state of components. It is therefore not surprising that *event-driven programming* has emerged as the prominent paradigm for the development of ubiquitous and pervasive applications [1,2]; indeed, this paradigm can be found in various commercial frameworks like IFTTT, Samsung SmartThings, Microsoft Power Automate, Zapier, etc. In this approach, the behavior of a system is defined by a set of rules (called also "applets", "zaps", "routines", "flows", and so on) following the well-known *Event Condition Action (ECA)* structure, originally from the field of active databases [3]:

on *Event* **if** *Condition* **do** *Action*

Intuitively, the meaning of this rule is: when *Event* occurs, if *Condition* is satisfied then execute *Action*. Thus, an ECA system can react to an event (e.g., a variable change or a signal from a sensor) by executing one or more actions, which can update the internal state of the node or act on the environment via some actuator. Of course, the effect caused by the action of a rule can trigger other rules, and so on.

However, in most current models and implementations of this paradigm, the rules are stored on, and executed by, a central computing entity, possibly hosted on some server on the cloud and accessible via the Internet. The devices of the system do not communicate and coordinate with each other but with this central node/cloud service only. Although simple, such a centralized architecture suffers of several disadvantages. First, it does not scale well: the increase of IoT and smart devices is going to produce a massive amount of data [4], and transferring, storing and processing this data in cloud data centers will overload network channels. As a consequence, cloud servers will not be able to guarantee acceptable transfer rates and response times; moreover, sending all this data back and forth on the network is a big waste of energy. Secondly, the dependence on Internet connections and of a central node/cloud service hinders availability, which is also a critical requirement for many IoT and other pervasive and autonomic applications; e.g., a smart door lock may be stuck because the server is not reachable. Third, smart and IoT devices often deal with sensible/personal data (think of, for instance, health sensors, surveillance cameras, etc) that the user would prefer to not share with some untrusted server on the cloud. Finally, the dependence of external services increases the attack surface of the smart system; e.g., an attacker can open a house front door, taking advantage of some vulnerability on the server communicating with the door, and unknown to the user.

For all these reasons, in recent years there is a growing interest in the so-called *edge computing* paradigm [4,5], which aims to move the computation away from cloud data centers towards the "edge" of the network, i.e., the smart objects which are the sources of data. This approach allows to mitigate the previous issues, as it reduces data transfers between the edge and the center of the network—in fact, there can be no center at all, thus increasing scalability, resilience, and security.

At the same time, placing the application logic on many nodes in a truly distributed and decentralized setting, introduces new issues and challenges. In particular, it requires suitable mechanisms and abstractions for communication and coordination of (possibly large) ensembles of distributed components. Traditional point-to-point communication mechanisms (like sockets, request/reply protocols or agent-like message passing) are not quite suited because they require strong coupling between parties, increasing the burden on the programmer, and are hardly scalable to large sets of components. A novel mechanism recently introduced to overcome these limitations is *attribute-based communication* [6–8], a loosely coupled message-oriented interaction model specifically designed for coordinating large numbers of components. The key aspect of attribute-based communication is that the actual receivers are selected "on the fly" by means of *predicates* over node attributes; dually, a node can "filter" incoming messages by means of predicates. Using a syntax similar to AbC [6] (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 satisfying Π, continue as P". Many interaction models used in smart systems, such as channels, agents, pub/sub, broadcast and multicast, can be readily implemented using attribute-based communication [8].

Therefore, putting together all the pieces, a promising paradigm for smart systems and edge computing could be obtained by merging the tradition of event-based programming with a new distributed coordination mechanism akin attribute-based communication. However, integrating attribute-based communication in the ECA paradigm is not obvious. Adding some send/receive primitives, similar to AbC's, would yield a disharmonious *patchwork* of different paradigms, i.e., memory-based events vs message-passing: the latter is best suited to (possibly stateless) agent-oriented models, while most ECA-based systems are meant to react to (internal or external) changes of nodes state, caused by memory updates or inputs.

For solving this conundrum, in this paper we present AbU (for "Attribute-based memory Updates"), a calculus integrating ECA programming with loosely coupled attribute-based interaction. In this model, interactions are reduced to events of the same kind ECA programs already deal with, i.e., memory updates. More precisely, an AbU system is composed by a set of agents, called *nodes*, each endowed with a local state (representing local memory, sensors, actuators, etc) and programmed with a set of ECA rules. When a rule of the form $x_1, \ldots, x_n \gg \Pi : \mathrm{act}$ is triggered on a given node due to a change in some local variable x_i, the action act can update the state of that node (like in normal ECA programming), but it can also update the state of other nodes, selected upon their states by the predicate Π. For instance, a rule like the following:

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

means "when (my local) variable *accessTime* 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*, and it can be seen as the memory-based counterpart of attribute-based communication, built upon message-passing.

Notice that, in this paradigm memory updates are directly "pushed" to remote nodes: a rule can change the state of a remote node without any corresponding "input" step on that node. Clearly, allowing for any update from anyone would be too liberal, as a node may want to keep incoming data under control. In attribute-based communication of AbC, this control is done by filtering incoming messages by means of predicates in input primitives — but in attribute-based memory updates, there are no input primitives. The solution we adopt arises from the observation that very often in state-based machines we can identify a subset of *invalid* states, which should be avoided for consistency or safety reasons; this is particularly common in IoT and smart systems, where nodes state also effects to the surrounding environment. Formally, the set of valid states of a node can be specified by means of a *state invariant*, i.e., a predicate over nodes states which should be kept valid during the whole execution. Therefore, any update (possibly from other nodes) leading to an invalid state, i.e., violating the local invariant, have to be rejected. So, invariants in attribute-based memory updates can be seen as the counterpart of message filtering in attribute-based communication.

The importance of a calculus like AbU is twofold. On one hand, it can be used as a reference model for the implementation of full-fledged programming language, or extensions of existing languages, for IoT and edge computing; as an example, a prototypal implementation in Golang of AbU is available at [9]. On the other hand, it is the basis for the investigation of important properties of event-driven architectures with attributed-based interaction, and for the development of formal methods for guaranteeing these properties. In this paper, we investigate two of these properties, namely *stabilization* and *confluence*, both very relevant in IoT and smart systems: the first guarantees that a chain of rule executions triggered by an external event will eventually terminate; the second guarantees that the effects of the rules are eventually deterministic and do not depend on the rule execution order. For both these properties we provide formal semantic definition, sufficient syntactic conditions on AbU programs, and algorithms to statically check such conditions.

Synopsis After a summary of related work in Section 2, in Section 3 we introduce the syntax and operational semantics of AbU, together with an encoding of a Turing complete computational model. In order to showcase the expressiveness of AbU, in Section 4 we provide various examples of application scenarios for AbU in the IoT. In Section 5 we show how to encode AbC components into AbU, providing encoding correctness and examples. Then, in Section 6 we define stabilization and confluence of AbU systems, discuss correctness requirements for these properties, and provide algorithms for their verification. A distributed implementation for AbU is presented in Section 7. Finally, conclusions and directions for future work are in Section 8. Full proofs of the results are collected in Appendix A.

2. Related work

To the best of our knowledge, the only previous work aiming at merging the ECA programming paradigm with attribute-based interaction is the conference version of the present paper [10]. Here, the calculus presented in [10] is enhanced with *invariants*, namely conditions that nodes have to guarantee during run-time. Furthermore, we slightly modify the semantics of the calculus, extending system labels with the committed updates. In this way, when a node performs an input or an execution step, the corresponding update committed is recorded by the semantics. This is necessary to define complex *correctness requirements* for AbU systems. In particular, in the present work we revise the termination guarantee of [10], defining *stabilization*, and we introduce the concept of *confluence* for AbU systems, that can be seen as a form of determinism in the effects of rule executions. Furthermore, we introduce two sufficient conditions for the verification of such requirements, together with soundness proofs, and we provide algorithms to verify these conditions. Finally, in the present work we have a section for AbU examples, where we revise the example presented in [10], we add a new application scenario and we discuss Turing completeness of the calculus. Full proofs of the results, not present in the conference version, are reported in the appendix of this work.

An approach close in spirit to ours is that based on associative memories, e.g., *tuple spaces* as in the Linda language [11] and the KLAIM calculus [12]. 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 interaction is implemented by means of remote memory updates (hence transparent to the nodes of the distributed system) and the logic for predicating over attributes is more expressive than simple pattern matching.

The Field Calculus [13,14] is a computational model aiming, much like AbU, to programming large scale systems of agents that should self-coordinate, in a distributed and decentralized manner. However, the two calculi differ in the programming abstractions they provide. In the Field Calculus, aggregate behavior of a system of agents (where a dynamic neighboring relation represents physical or logical proximity) is specified by a functional composition of operators that manipulate a *computational field*. All agents are equipped with the same code, but they can behave differently due to their different positions (in a particular time frame) with respect to other agents (the "field"); all agents contribute to reach a goal function, modeling a unified behavior of the system. Instead, in AbU each agent (i.e., each node) has its own code, and coordination and communication is carried out by means of an attribute-based interaction, transparent to the programmer. On the other hand, AbU does not provide a way for defining a global goal (if any) to be achieved by the agents taken as whole. Another similarity is about termination and stabilization. The Field Calculus is provided with a typing system such that well-typed

programs are guaranteed to terminate. This corresponds to (wave) stabilization of AbU systems, which is guaranteed by a different static check. Similarly, (wave) convergence for AbU corresponds to self-stabilization in the Field Calculus.

Concerning ECA programming, [15,16] introduce IRON, a language based on ECA rules for the IoT domain. Following other work about ECA languages, [17,18] 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 [19] and BDD [20]. In [21,1], a tool-supported method for verifying and controlling the correct interactions of ECA rules is presented. All these works do not deal with distributed systems, hence communication is not taken into account.

AbC has been introduced and studied in [6,8,7] as a core calculus for SCEL [22], a language à la KLAIM with collective communication primitives parametric in predicates over node attributes. Focusing on the latter primitives, i.e., on *Attribute-based Communication*, AbC is well-suited to model Collective Adaptive Systems (CAS) [23] from a process calculi standpoint, as opposed to Multi-Agents Systems (MAS) that follow a logical approach [24]; we refer to [8] for more details. Various extensions of AbC has been proposed [25,26], as well as correct implementations in Erlang [27] and Golang [28,29]. AbC, and its parent languages, adopt a message-passing communication mechanism and a sequential, process-centric, 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 interaction mechanisms, we will focus on the most fundamental primitives of AbC, omitting features not strictly necessary. We will study the connections between our work and AbC in Section 5.

Some work combining message-passing primitives and shared-memory mechanisms have been recently proposed [30, 31]. In particular, the *m&m model* of [30] 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) [31], 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 [32], 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.

Finally, recent work [33,34] on the *Reactive Data* model adopts a declarative attribute-based interaction similar to AbU's. In this model, ECA rules are given by declarative *response relations* (introduced in DCR Graphs [35]), while attribute-based interaction is obtained by using *dynamic end-points* of the relations, defined by graph query languages (e.g., XPath). The papers come with prototype implementations, which are however not distributed.

3. The AbU calculus

We present here AbU, a calculus merging a prominent event-driven programming paradigm, i.e., Event Condition Action (ECA) rules, with attribute-based interaction. This solution embodies the programming simplicity prerogative of ECA rules, but it is expressive enough to model complex coordination scenarios, typical of distributed systems. Moreover, the calculus is fully decentralized, hence perfectly suitable for edge computing. The name AbU is an acronym for Attribute-based memory Updates that is, as we will see in the following, the distributed interaction mechanism on which the calculus relies on.

3.1. Syntax

An AbU *system* S is either a *node*, of the form $R, \iota\langle\Sigma, \Theta\rangle$, or a parallel composition $\mathsf{S}_1 \parallel \mathsf{S}_2$ of systems, which is associative and commutative.[1] A *state* $\Sigma \in \mathbb{X} \to \mathbb{V}$, is a map from resources 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. Moreover, each node is equipped with: an *invariant* ι (i.e., a boolean expression) that the node has to satisfy at run-time; and a non-empty finite list R of *ECA rules*, generated by the following grammar.

$$
\begin{array}{rcll}
\mathsf{rule} & ::= & \mathsf{evt} \gg \mathsf{act}, \mathsf{task} & \text{ECA rules} \\
\mathsf{evt} & ::= & x \mid \mathsf{evt}\ \mathsf{evt} & \text{rule events} \\
\mathsf{act} & ::= & \epsilon \mid x \leftarrow \varepsilon\ \mathsf{act} \mid \overline{x} \leftarrow \varepsilon\ \mathsf{act} & \text{actions} \\
\mathsf{task} & ::= & \mathsf{cnd} :: \mathsf{act} & \text{rule tasks} \\
\mathsf{cnd} & ::= & \varphi \mid @\varphi & \text{task conditions} \\
\varphi, \iota & ::= & \mathtt{F} \mid \mathtt{T} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varepsilon \bowtie \varepsilon & \text{boolean expressions / invariants} \\
\varepsilon & ::= & v \mid x \mid \overline{x} \mid \varepsilon \otimes \varepsilon & \text{value expressions}
\end{array}
$$

[1] In particular, $\parallel$ is associative and commutative with respect to the AbU systems semantics relation $\longrightarrow$ of Fig. 1; e.g., $\mathsf{S}_1 \parallel \mathsf{S}_2$ and $\mathsf{S}_2 \parallel \mathsf{S}_1$ exhibit exactly the same execution steps.

An 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 resources state 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 processed. 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 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 $@\mathtt{T} : \overline{x} \leftarrow \overline{x} + x$ means "add the value of this node's x to the x of every other node".

In the syntax for boolean expressions φ (and invariants) 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.

Rules of the form $\mathsf{evt} \gg \epsilon, \mathsf{task}$, i.e., with empty default action, are written more concisely as $\mathsf{evt} \gg \mathsf{task}$. Finally, we implicitly extend the (binary) parallel composition operator to its n-ary version. For instance, when we write $\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3$ we actually mean $(\mathsf{S}_1 \parallel \mathsf{S}_2) \parallel \mathsf{S}_3$, which is equivalent, due to associativity, to $\mathsf{S}_1 \parallel (\mathsf{S}_2 \parallel \mathsf{S}_3)$. With a little abuse of notation, we denote with $\parallel$ the binary and the non-binary versions of the parallel composition.

3.2. Semantics

Given a list $R = \mathsf{rule}_1 \dots \mathsf{rule}_n$ of ECA rules and a set X of resources that have been modified, we define the set of *active rules* as $\mathsf{Active}(R, X) \triangleq \{\mathsf{rule}_i \mid \exists i \in [1..n] \,.\, \mathsf{rule}_i = \mathsf{evt} \gg \mathsf{act}, \mathsf{task} \wedge \mathsf{evt} \cap X \neq \varnothing\}$, 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: $[\![x_1 \leftarrow \varepsilon_1 \dots x_n \leftarrow \varepsilon_n]\!]\Sigma \triangleq (x_1, [\![\varepsilon_1]\!]\Sigma) \dots (x_n, [\![\varepsilon_n]\!]\Sigma)$. 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:

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

Similarly, 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 task condition) and whose condition is satisfied by the node state, 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 for boolean expressions (and, indeed, for invariants) is defined as: $\Sigma \models \varphi \triangleq [\![\varphi]\!]\Sigma = \mathtt{tt}$ (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 condition.[2] 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 a resource x in the task condition and the right-hand side of assignments in the task action replaced with the value $\Sigma(x)$. After that, each instance of $\overline{x}$ in the task action is replaced with x and the modifier @ is dropped. 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, it becomes "syntactically local" for the receiving nodes, i.e., its action can be evaluated with the semantic function $[\![\cdot]\!]$.

Finally, we define the *external tasks* as:

$$\mathsf{ExtTasks}(R, X, \Sigma) \triangleq \{\!|\mathsf{task}_1|\!\}\Sigma \dots \{\!|\mathsf{task}_n|\!\}\Sigma$$

given 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).

Operational (small-step) semantics The small-step operational semantics of an AbU system is modeled as a *labeled transition system (LTS)*. In particular, $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_2$ means that the AbU system S_1 evolves to the AbU system S_2, producing the LTS label (or simply, label) α. Here, LTS labels are given by:

$$\alpha ::= T \mid \mathsf{upd} \rhd T \mid \mathsf{upd} \blacktriangleright T$$

[2] When such node does not exist, the update is simply discarded.

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

$$(\textsc{Exec-Fail})\ \frac{\mathsf{upd} \in \Theta \quad \mathsf{upd} = (x_1, v_1) \dots (x_k, v_k) \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad \Sigma' \not\models \iota \quad \Theta' = \Theta \setminus \{\mathsf{upd}\}}{R, \iota\langle \Sigma, \Theta\rangle \xrightarrow{\mathsf{upd} \rhd \epsilon} R, \iota\langle \Sigma, \Theta'\rangle}$$

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

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

$$(\textsc{StepL})\ \frac{\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{T} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ \alpha \in \{\mathsf{upd} \rhd T, \mathsf{upd} \blacktriangleright T\} \qquad (\textsc{StepR})\ \frac{\mathsf{S}_1 \xrightarrow{T} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ \alpha \in \{\mathsf{upd} \rhd T, \mathsf{upd} \blacktriangleright T\}$$

Fig. 1. AbU semantics for nodes and systems.

where T is a finite (possibly empty) list of tasks and upd an update. As we will see in a moment, the labels T identify a *discovery* phase, the labels $\mathsf{upd} \rhd T$ an *update execution* and the labels $\mathsf{upd} \blacktriangleright T$ an *external input*. A transition can modify the state and the execution pool of the nodes of the AbU system. The semantics is *distributed*, in the sense that each node semantics does not have a global knowledge about the system. The transition 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 modification of resources in general, may trigger some other ECA rules in the system. Hence, after updating a node 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 ECA rules. The discovery phase is composed by two parts, the local and the remote one. A node $R, \iota\langle \Sigma, \Theta\rangle$ performs a local discovery by means of the functions DefUpds and 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 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 $\mathsf{upd} \rhd T$, produced by the rule (Exec), and $\mathsf{upd} \blacktriangleright T$, 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. 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 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 of 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 $\mathsf{upd} \rhd \epsilon$ or the label $\mathsf{upd} \blacktriangleright \epsilon$ (i.e., with an empty tasks list). The latter, is matched by a label $T = \epsilon$, that all nodes can generate by applying the rule (Disc).

The semantics also checks the fulfillment of invariants, at run-time. Indeed, when a node tries to perform an update that would break an invariant, the rule (Exec) is not applicable. Instead, the rule (Exec-Fail) is performed, that ignores the update (which is removed from the pool but it is not executed); this fact is observable by the label $\mathsf{upd} \rhd \epsilon$. Note that, we do not distinguish the case when an update breaks the invariant from the case when the update is idempotent (i.e., when resources hold the same values that the update tries to assign).

Remark 1. Invariants have been introduced to enforce node safety properties at run-time, namely not reaching erroneous states (e.g., a thermostat can allow its temperature to be set only to values within a given range). In fact, by means of AbU node invariants we can implement some form of *access control*. For instance, a node can accept updates originated only by *white-listed* nodes in this way: every update will save in an attribute *last-writer* the *id* of the originating node; the invariant will require *last-writer* to be always in a fixed set (the white-list). We will see in Section 5 that an application of invariants will be the encoding of selective inputs of AbC.

In this respect, we can define the set of all possible AbU execution states that satisfy a given invariant, dubbed *legal execution state*.

Definition 1 *(Legal execution states).* Given an AbU invariant ι, we define the set of *legal execution states* for ι as $\mathsf{legal}^\iota \triangleq \{\Sigma \in \mathbb{X} \to \mathbb{V} \mid \Sigma \models \iota\}$.

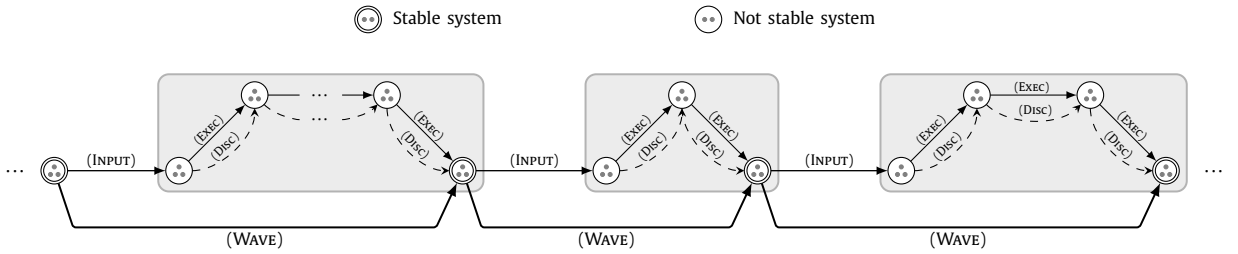


Fig. 2. A graphical representation of the AbU execution model. An AbU system is represented by the local state and pool of all nodes (here represented by dots). A stable system moves to a not stable one due to an (Input); then it executes only (Exec) steps, until it possibly reaches a new stable system. The transitions between stable systems are represented by a (Wave).

Then, an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ is said *admissible* when its execution state is legal for the corresponding invariants. Formally: $\mathsf{admissible}(\mathsf{S}) \triangleq \forall i \in [1..n]\,.\,\Sigma_i \in \mathsf{legal}^{\iota_i}$.

Note that, admissible systems are defined in terms of system invariants only, and invariants do not change during execution. It is easy to note that the semantics in Fig. 1 guarantees that the execution steps of an AbU system involve admissible systems only.

Proposition 1. *Let* S *be an admissible AbU system. If* $\mathsf{S} \xrightarrow{\mathsf{upd}_1 \rhd T_1} \mathsf{S}^1 \xrightarrow{\mathsf{upd}_2 \rhd T_2} \ldots \xrightarrow{\mathsf{upd}_k \rhd T_k} \mathsf{S}^k$ *then each* S^i*, with* $i \in [1..k]$*, is admissible. In other words,* $\mathsf{admissible}(\mathsf{S})$ *implies that* $\mathsf{admissible}(\mathsf{S}^i)$*, for all* $i \in [1..k]$.

Wave (big-step) semantics Initially, the pools of all nodes of an AbU system are empty, namely the system is *stable*, and to start the computation an input (i.e., an external modification of the environment) is needed.

Definition 2 *(Stable system).* An AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ 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 often use $R, \iota\langle\Sigma\rangle$ as a shorthand for $R, \iota\langle\Sigma, \varnothing\rangle$. Hence, a system is stable when it is of the form $R_1, \iota_1\langle\Sigma_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n\rangle$. In the case of stable systems, only the rule (Input) can be applied, i.e., an external environment change is needed to (re)start the computation.

We can define a sort of big-step semantics $\mathsf{S} \overset{\mathsf{upd}}{\rightsquigarrow} \mathsf{S}'$ between stable systems, dubbed *wave semantics*, in terms of the small-step semantics. Let $\longrightarrow^*$ be the transitive closure of $\longrightarrow$, without occurrences of labels of the form $\mathsf{upd} \blacktriangleright T$, namely $\longrightarrow^*$ 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:

$$\text{(Wave)}\ \frac{\mathsf{S} = R_1, \iota_1\langle\Sigma_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n\rangle \quad \mathsf{S} \xrightarrow{\mathsf{upd}\blacktriangleright T} \mathsf{S}'' \longrightarrow^* \mathsf{S}' \quad \mathsf{S}' = R_1, \iota_1\langle\Sigma_1'\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n'\rangle}{\mathsf{S} \overset{\mathsf{upd}}{\rightsquigarrow} \mathsf{S}'}$$

The idea is that a stable system reacts to an external stimulus by executing a series of tasks, which propagate across the nodes like 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. Moreover, it could be the case that, after a particular update, an AbU system never becomes stable, since the semantics allows systems to perform infinitely many execution steps. In Fig. 2 we have a graphical representation of the AbU execution model.

Remark 2. 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 is justified by the fact that, in the IoT context, usually, external changes (in sensors) take much more time than internal computation steps [36], or, equivalently, hardware is chosen fast enough to keep the pace of the environment changes. As an example, LoRaWAN connected IoT sensors can transmit their data with intervals of the order of minutes. In this scenario, the devices receive inputs with a very low rate. Therefore, we can assume that AbU nodes have been provided with enough computational power to complete their execution steps before the next input: the faster the inputs happen, the more powerful the nodes have to be.

We point out that environment changes possibly happening during an execution phase are not lost; instead, these inputs are queued and processed in FIFO order when the execution phase terminates. Nevertheless, we plan to relax the constraint assuming the mutual exclusion between input and execution phases as a future extension of AbU.

Remark 3. Since the semantics in Fig. 1 is labeled, one can think of AbU as a calculus with message-passing. Nevertheless, LTS labels are not actual messages exchanged by AbU nodes, in the sense that such messages are not part of the AbU syntax (has it happens for message-passing process calculi). Indeed, nodes code is not aware of LTS labels and it cannot

manipulate them. Clearly, in a distributed network of agents, communication must be implemented by means of messages exchange, but this is an implementation detail. Our point is that AbU does not require messages exchange at *design level*, namely programmers are not forced to use a message-passing mechanism to coordinate nodes (even if the underlying communication may be implemented by means of a protocol based on message-passing).

3.3. On the expressiveness of AbU

During the presentation of AbU we deliberately left implicit the full definition of boolean and value expressions. Indeed, the latter deeply affect the expressive power of the calculus. It is easy to provide AbU with expressions that make the calculus Turing complete. For instance, adding some simple constructs to value expressions we can easily encode *semi-Thue systems*, non-deterministic rewriting systems over strings which are Turing complete [37]. In brief, a semi-Thue system is composed by a (finite) alphabet of symbol and a (finite) set of rewriting rules of the form $u \to v$, where u and v are strings (i.e., finite sequences of symbols) over the alphabet. Computation is carried out transforming strings: we can transform a string s_1 into a string s_2 if $s_1 = xuy$, $s_2 = xvy$ and there exists a rewriting rule $u \to v$ in the semi-Thue system. Note that, the substitution is non-deterministic, in the sense that if s_1 contains multiple occurrences of u, the occurrence to modify is selected non-deterministically.

Suppose to have the following binary operators on value expressions: $\#[s]$ computing the number of elements of the string value s, e.g., $\#[\text{'str'}] = 3$; $s_1 :: s_2$ concatenating the two string values s_1 and s_2, e.g., 'str' :: 'ing' = 'string'; $s[\text{-} : i]$ computing the string value composed of the first i elements of the string value s, e.g., 'string'[- : 3] = 'str'; $s[i : \text{-}]$ computing the string value composed of the last i elements of the string value s, e.g., 'string'[3 : -] = 'ing'; and $s_1 \stackrel{?}{|} s_2$ returning the first index in the string value s_1 of one occurrence of the string value s_2 in s_1, chosen non-deterministically,[3] e.g., 'abuabu' $\stackrel{?}{|}$ 'bu' can return, non-deterministically, either 2 or 5. When s_2 does not occur in s_1, then $s_1 \stackrel{?}{|} s_2$ returns -1. By means of these operators we can implement *non-deterministic string substitutions*. Consider the following AbU rule:

$$x \gg (x \stackrel{?}{|} y \neq -1) : x \leftarrow x[\text{-} : x \stackrel{?}{|} y - 1] :: z :: x[\#[x] - (x \stackrel{?}{|} y + \#[y]) : \text{-}] \tag{1}$$

The rule substitutes one occurrence of the string value pointed by y in the string value pointed by x, chosen non-deterministically, with the string value pointed by z. For instance, consider the state $[x \mapsto$ 'abuabu' $y \mapsto$ 'bu' $z \mapsto$ 'bc'$]$. When the rule (1) is fired, and its update executed, we have that x can contain, non-deterministically, either 'abcabu' or 'abuabc'. Note that, since the evaluation of a rule action is atomic, we assume that all instances of $x \stackrel{?}{|} y$ in a given action compute the same value (i.e., an index of x is non-deterministically chosen, but it is the same for all occurrences of $x \stackrel{?}{|} y$ in the action). By using the rule (1), we can easily encode into AbU a generic semi-Thue system, we just have to encode into AbU its rewriting relation. In particular, for each rewriting rule $u \to v$ we generate an AbU rule of the form:

$$input \gg (input \stackrel{?}{|} \text{'u'} \neq -1) : input \leftarrow input[\text{-} : input \stackrel{?}{|} \text{'u'} - 1] :: \text{'v'} :: input[\#[input] - (input \stackrel{?}{|} \text{'u'} + \#[\text{'u'}]) : \text{-}]$$

where 'u' and 'v' are the string values of AbU corresponding to the strings u and v of the semi-Thue system, respectively. The AbU resource *input* is initialized with the input string that would be passed to the semi-Thue system to execute. Each time the resource *input* is modified, in the pool of the AbU node all updates of the rules are added, then the AbU scheduler will chose non-deterministically which substitution to apply, as exactly happens for a semi-Thue system. Note that, a semi-Thue system has, by definition, a finite number of rewriting rules, hence we can fully encode any semi-Thue system into AbU. Hence, AbU equipped with non-deterministic string substitutions is Turing complete.

Conversely, if we limit the value domains in expressions to have finite cardinality, then the language loses Turing completeness. Indeed, we have a finite number of rules, a finite number of nodes and a finite number of resource (in AbU we cannot dynamically allocate memory, namely fresh resources). If we also have a finite set of possible values for resources, then we can encode each AbU system into a NFA, hence losing Turing completeness.

Proposition 2. *If the value domain of resources in AbU expressions has finite cardinality, then each AbU system corresponds to a NFA.*

Proof *(Sketch).* The encoding of an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ into a NFA is straightforward: states of the automaton are given by all possible configurations of the $2n$-tuple $(\Sigma_1, \Theta_1, \ldots, \Sigma_n, \Theta_n)$, composed by execution states and execution pools of all nodes in the system, while transitions are given by the ECA rules in the nodes of the system.

Each execution state Σ_i can range over a finite set, whose cardinality is given by its attributes and the corresponding sets of possible values (which are finite by hypothesis). Each pool Θ_i is a subset of the set of all possible updates, and these are finite because each update is a finite list of assignments whose right hand values are finite, and are generated by ECA rules which are finite as well. Therefore, the set of states of the automaton defined above is finite, yielding a NFA. □

Corollary 3. *If the value domain of resources in AbU expressions has finite cardinality, then the AbU calculus is not Turing complete.*

[3] Note that, we do not strictly need non-deterministic substitutions, since also *deterministic* semi-Thue systems are Turing complete [38]. In this case, we may define $x \stackrel{?}{|} y$ to return the first index of the left-most occurrence of y in x [38].

4. AbU in practice

We now provide some examples of IoT scenarios modeled with AbU, to help the reader to get familiar with the AbU syntax and to understand its semantics. These examples showcase the simplicity of AbU in modeling complex scenarios.

4.1. Intrusion detection system

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. The access node is activated when *accessT* changes, namely when a user performs access; the node aims at sending the IP address of the user and the name of the accessed resource, together with a time-stamp. On the other side, the logger nodes record the access time, the IP address and the resource 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 the intrusion detection system (IDS).

The system is formalized with 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 $\langle IP, accessT, res\rangle$. An append to the list *log* is given by $\mathsf{append}\ log\ \langle IP, accessT, res\rangle$; while with $\langle IP, accessT, res\rangle.IP$ we denote the access of the field IP, and with $\mathsf{tail}\ log$ we denote the last record inserted in the list *log*. Given a list of (black-listed) IP addresses *Blist*, we denote with $IP \in Blist$ the fact that *IP* is an element of the list. The four AbU nodes are the following, where $R_a\langle\Sigma_1\rangle$ and $R_a\langle\Sigma_2\rangle$ are the access nodes, while $R_l\langle\Sigma_3\rangle$ and $R_l\langle\Sigma_4\rangle$ are the logger nodes.

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

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`, and it is recorded by first access node. Hence, the rule (Input) can be applied by S_1, namely $R_a\langle\Sigma_1\rangle \xrightarrow{\mathsf{upd}_1 \blacktriangleright T} R_a\langle\Sigma_1'\rangle$, where:

$$\mathsf{upd}_1 = (accessT, 15{:}07{:}00)(res, \texttt{camera})(IP, 167.123.23.2)$$
$$\Sigma_1' = [accessT\mapsto 15{:}07{:}00\ \ res\mapsto \texttt{camera}\ \ IP\mapsto 167.123.23.2]$$
$$T = (role = \texttt{logger}) : log \leftarrow \mathsf{append}\ log\ \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle$$

Now, a discovery phase is performed on all other nodes. In particular, we have that $R_a\langle\Sigma_2\rangle \xrightarrow{T} R_a\langle\Sigma_2\rangle$, $R_l\langle\Sigma_3\rangle \xrightarrow{T} R_l\langle\Sigma_3, \Theta\rangle$ and $R_l\langle\Sigma_4\rangle \xrightarrow{T} R_l\langle\Sigma_4, \Theta\rangle$, with $\Theta = \{(log, \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle)\}$. Now, let $\mathsf{S}_1' = R_a\langle\Sigma_1'\rangle$, $\mathsf{S}_3' = R_l\langle\Sigma_3', \Theta\rangle$ 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. 3[top].

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 $R_l\langle\Sigma_3, \Theta\rangle \xrightarrow{\mathsf{upd}_2 \rhd \epsilon} R_l\langle\Sigma_3'\rangle$, by applying the rule (Exec), with $\Sigma_3' = [role\mapsto \texttt{logger}\ \ log\mapsto \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle\ \ Blist\mapsto\epsilon\ \ IDS\mapsto\epsilon]$ and $\mathsf{upd}_2 = (log, \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle)$. 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 $\mathsf{S}_3'' = R_l\langle\Sigma_3'\rangle$ is given in Fig. 3[bottom]. Finally, the fourth node can execute, namely $R_l\langle\Sigma_4, \Theta\rangle \xrightarrow{\mathsf{upd}_2 \rhd \epsilon} R_l\langle\Sigma_4', \Theta'\rangle$, by applying the rule (Exec). Here, $\Sigma_4' = [role\mapsto \texttt{logger}\ \ log\mapsto \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle\ \ Blist\mapsto 167.123.23.2\ \ IDS\mapsto\epsilon]$ and $\Theta' = \{(IDS, \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle)\}$. 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 previous one. 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 $\mathsf{S}_4'' = R_l\langle\Sigma_4''\rangle$ and $\Sigma_4'' = [role\mapsto \texttt{logger}\ \ log\mapsto \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle\ \ Blist\mapsto 167.123.23.2\ \ IDS\mapsto \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle]$. Since all pools are empty, this system is stable. This means that we can perform a wave semantics step (where $\mathsf{upd}_3 = (IDS, \langle 167.123.23.2, 15{:}07{:}00, \texttt{camera}\rangle)$):

$$(\textsc{Wave})\ \frac{\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4 \xrightarrow{\mathsf{upd}_1 \blacktriangleright T} \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3' \parallel \mathsf{S}_4' \xrightarrow{\mathsf{upd}_2 \rhd \epsilon} \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4' \xrightarrow{\mathsf{upd}_2 \rhd \epsilon} \dots \xrightarrow{\mathsf{upd}_3 \rhd \epsilon} \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4''}{\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4 \rightsquigarrow \mathsf{S}_1' \parallel \mathsf{S}_2 \parallel \mathsf{S}_3'' \parallel \mathsf{S}_4''}$$

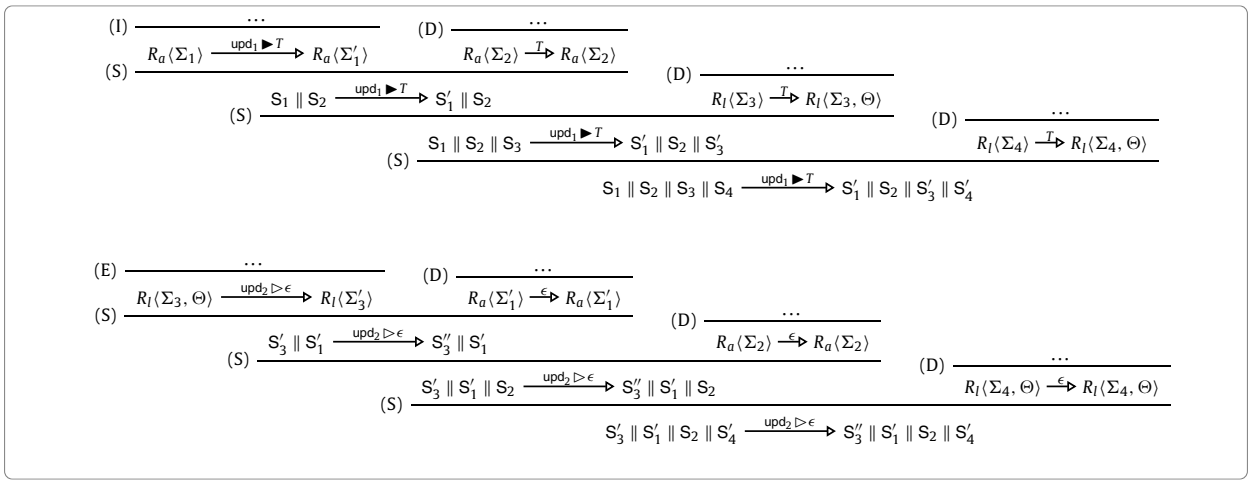


Fig. 3. Derivation trees for AbU semantic steps of Subsection 4.1: (INPUT) [top] and (EXEC) [bottom]. For space reasons, we abbreviate semantic rule names and omit the premises of leaf semantic rules.

4.2. Swarm of robots

Consider now a scenario where a swarm of drones is in charge of taking specific measurements, randomly picked in a large uninhabited area. Each drone is equipped with a battery that periodically needs to be recharged by returning to a docking station. It may happen that a drone runs out of energy before returning to the charging spot. In this case, the low-battery drone asks for help from its neighbors. If a drone has some energy to share and it is close enough to the requester, it will enter the "rescue" mode. A drone in "rescue" mode will reach the drone in distress, sharing with it some energy. This phase is not modeled in the example for space reasons. We can model this scenario in AbU as follows.

Suppose to have four drones. For each drone we have an AbU node with a resource *battery*, indicating the battery level of the drone; a resource *position*, indicating where is located the drone; a resource *mode*, indicating in which operative state is the drone; and a resource *helpPos*, indicating the position of a drone that needs help. Formally, the AbU system modeling the drone-swarm scenario is $R\langle\Sigma_1\rangle \parallel R\langle\Sigma_2\rangle \parallel R\langle\Sigma_3\rangle \parallel R\langle\Sigma_4\rangle$, where R contains, among the others, the following two rules:

$$\mathit{battery} \gg @(\mathit{battery} < 5 \wedge \overline{\mathit{battery}} > 80) : \overline{\mathit{helpPos}} \leftarrow \mathit{position}$$
$$\mathit{helpPos} \gg (|\mathit{position} - \mathit{helpPos}| < 7.0) : \mathit{mode} \leftarrow \texttt{rescue}$$

Now suppose that the execution states of the drones are the following:

$$\Sigma_1 = [\mathit{battery} \mapsto 4 \;\; \mathit{position} \mapsto 2.0 \;\; \mathit{mode} \mapsto \texttt{normal} \;\; \mathit{helpPos} \mapsto 0.0]$$
$$\Sigma_2 = [\mathit{battery} \mapsto 81 \;\; \mathit{position} \mapsto 15.0 \;\; \mathit{mode} \mapsto \texttt{normal} \;\; \mathit{helpPos} \mapsto 0.0]$$
$$\Sigma_3 = [\mathit{battery} \mapsto 97 \;\; \mathit{position} \mapsto 6.0 \;\; \mathit{mode} \mapsto \texttt{normal} \;\; \mathit{helpPos} \mapsto 0.0]$$
$$\Sigma_4 = [\mathit{battery} \mapsto 65 \;\; \mathit{position} \mapsto 8.0 \;\; \mathit{mode} \mapsto \texttt{normal} \;\; \mathit{helpPos} \mapsto 0.0]$$

The first rule says that when the current drone battery level is low (i.e., when $\mathit{battery} < 5$), then the current drone have to send to all neighbors (using @) that have some energy to share (i.e., that have $\overline{\mathit{battery}} > 80$) its position, performing a remote update (i.e., $\overline{\mathit{helpPos}} \leftarrow \mathit{position}$). In the example, the first node can fire the rule, since its battery level is low. Then, it pre-evaluates the task condition, yielding $(4 < 5 \wedge \mathit{battery} > 80)$, which is sent to the other nodes, together with the pre-evaluation of the task action, i.e., $\mathit{helpPos} \leftarrow 2.0$. Among all receivers, only the second and the third nodes are interested in the communication, since they are the only nodes with battery level greater than 80. So, they both add to their pool the update $(\mathit{helpPos}, 2.0)$. This ends the discovery phase originated by the first node.

The second rule, instead, is fired when a drone receives a help request (i.e., when its resource *helpPos* changes) and basically checks if the current drone position is close to the requester node position (i.e., when $|\mathit{position} - \mathit{helpPos}| < 7.0$). If it is the case, the current drone enters the rescue mode performing a local update ($\mathit{mode} \leftarrow \texttt{rescue}$). In the example, when the second and the third nodes execute the update $(\mathit{helpPos}, 2.0)$, the task of the rule may be executed. For the second node this does not happen, since $|15.0 - 2.0| < 7.0$ is not true (the node is too far from the first node). Instead, $|6.0 - 2.0| < 7.0$ is true and the third node can execute the rule task, adding to its pool the update $(\mathit{mode}, \texttt{rescue})$.

Fig. 4. A vineyard irrigation system divided in fifteen zones. Each zone has a moisture sensor m_i, with $i \in [1..15]$, and it covers four vines. Each vine in a zone has a dedicated water valve v_i, with $i \in [1..4]$.

4.3. A vineyard irrigation system

Finally, let us consider a scenario where a winegrower wants to optimize the irrigation system of a vineyard. Optimization here means that the system should irrigate each vine with the correct amount of water, maintaining the moisture level of the soil within a given range. The system is composed by a set of water valves, each one attached to a vine. The vineyard is divided in zones, as depicted in Fig. 4, each one equipped with a moisture sensor. The vines falling into a given zone are watered depending on the moisture level of the corresponding sensor: when the zone soil is too dry, the water valves of the vines in the zone are opened; dually, the valves are closed when the zone soil is sufficiently moist.

The system also provides to the winegrower a console, to monitor (and log) the status of the water valves (open/close). Furthermore, from the console it is possible to change the moisture range of the sensors. We can model this scenario in AbU as follows.

As depicted in Fig. 4, suppose to have sixty vines, divided in fifteen zones. This means that the irrigation system comprises fifteen moisture sensors and sixty water valves. Note that, even if in Fig. 4 sensors (m_i) and valves (v_i) have a name, in AbU the nodes are typically anonymous.

Indeed, we have fifteen (unnamed) AbU nodes representing the moisture sensors, each one equipped with a resource *position*, indicating where is located the sensor; a resource *range* indicating the size of area covered by the sensor; a resource *moisture*, indicating the (average) moisture level of the soil; and two resources *minMoist* and *maxMoist*, indicating the minimum and the maximum moisture levels, respectively. Furthermore, we have sixty (unnamed) AbU nodes representing the water valves, each one equipped with a resource *position*, indicating where is located the valve; and a resource *valve*, indicating if the valve is open or close. We also have an AbU node representing the console, equipped with a resource *log*, to record the actions performed on the valves in the system; and two resources *deltaMin* and *deltaMax*, indicating the variation of the lower and upper moisture level bounds to be applied to sensors, respectively. Finally all nodes have a resource *node*, to indicate if a node is a moisture sensor (`sensor`), a water valve (`valve`) or a console (`console`).

Formally, the AbU system modeling the vineyard irrigation system is $R_s, \iota_v\langle\Sigma_{s,1}\rangle \parallel \ldots \parallel R_s, \iota_s\langle\Sigma_{s,15}\rangle \parallel R_v\langle\Sigma_{v,1}\rangle \parallel \ldots \parallel R_v\langle\Sigma_{v,60}\rangle \parallel R_c\langle\Sigma_c\rangle$, where $\Sigma_{s,i}$, with $i \in [1..15]$, are the execution states of moisture sensor nodes; $\Sigma_{v,i}$, with $i \in [1..60]$, are the execution states of water valve nodes; and Σ_c is the execution state of the console node.

$$\Sigma_{s,1} = [\mathit{node} \mapsto \texttt{sensor}\ \mathit{position} \mapsto 2.0\ \mathit{scope} \mapsto 9.0\ \mathit{moisture} \mapsto 3.1\ \mathit{minMoist} \mapsto 2.5\ \mathit{maxMoist} \mapsto 5.3]$$

$$\ldots$$

$$\Sigma_{s,15} = [\mathit{node} \mapsto \texttt{sensor}\ \mathit{position} \mapsto 32.0\ \mathit{scope} \mapsto 9.0\ \mathit{moisture} \mapsto 3.7\ \mathit{minMoist} \mapsto 2.7\ \mathit{maxMoist} \mapsto 5.8]$$

$$\Sigma_{v,1} = [\mathit{node} \mapsto \texttt{valve}\ \mathit{position} \mapsto 1.0\ \mathit{valve} \mapsto \texttt{close}]$$

$$\ldots$$

$$\Sigma_{v,60} = [\mathit{node} \mapsto \texttt{valve}\ \mathit{position} \mapsto 33.0\ \mathit{valve} \mapsto \texttt{close}]$$

$$\Sigma_c = [\mathit{node} \mapsto \texttt{console}\ \mathit{log} \mapsto \epsilon\ \mathit{deltaMin} \mapsto 0.0\ \mathit{deltaMax} \mapsto 0.0]$$

The system is controlled by means of the following AbU rules, where R_s are those equipped on moisture sensor nodes; R_v are those equipped on water valve nodes; and R_c are those equipped to the console node.

$$R_s \triangleq moisture \gg @(moisture > maxMoist \wedge |position - \overline{position}| < scope) : \overline{valve} \leftarrow \texttt{close}$$
$$moisture \gg @(moisture < minMoist \wedge |position - \overline{position}| < scope) : \overline{valve} \leftarrow \texttt{open}$$
$$R_v \triangleq valve \gg @(\overline{node} = \texttt{console}) : \overline{log} \leftarrow \mathsf{append}\ \overline{log}\ \langle position, valve\rangle$$
$$R_c \triangleq button \gg @(\overline{node} = \texttt{zone}) : \overline{minMoist} \leftarrow \overline{minMoist} + deltaMin\ \ \overline{maxMoist} \leftarrow \overline{maxMoist} + deltaMax$$

The first rule in R_s basically closes a water valve when a moisture sensor detects a change in the soil moist, and the new moisture level is above the maximum allowed level. The selected valves to close are those falling in the area covered by the sensor, i.e., valves with position $\overline{position}$ such that $|position - \overline{position}| < scope$. The second rule in R_s is the dual, that opens a water valve when the new moisture level is below the minimum allowed level.

The rule in R_v notifies the console, when a water valve changes status, by selecting all nodes with $\overline{node} = \texttt{console}$. The console resource *log* is appended with current (water valve) node position and the new status of the valve. Finally, the rule in R_c modifies the minimum and maximum moisture level boundaries of all sensors, by *deltaMin* and *deltaMax*, respectively. The latter values can be set by the winegrower at any time, then the rule is fired when a button on the console is pressed.

Finally, to avoid misconfigurations due to improper usage of the console, sensor nodes have an invariant guaranteeing that lower and upper moisture level bounds always form a valid numeric interval:

$$\iota_s \triangleq (minMoist + 1.0) < maxMoist$$

Indeed suppose that a sensor node has $minMoist \mapsto 2.5$ and $maxMoist \mapsto 5.3$, and suppose that the console sends, by mistake, un update with $deltaMin \mapsto 3.0$ and $deltaMax \mapsto 0.0$. If applied, the update would yield $[5.5, 5.3]$, that is not a valid interval, possibly corrupting the sensor intended behavior. This scenario is prevented by the invariant ι_s. Indeed, such update would fire the semantic rule (Exec-Fail), discarding the update, instead of the semantic rule (Exec).

Remark 4. Note that, nodes in AbU are anonymous, and they are programmed by means of predicates. For instance, we can select groups of nodes by changing (at any time) the sensors area (i.e., the zones) or the sensors moisture level boundaries, without knowing the actual water valves position. This is particularly useful in the context of large IoT systems, where the huge number of nodes makes unfeasible to identify a precise node and where the nodes topology frequently changes (nodes may change their position or can be lost and must be replaced). Nevertheless, we can still adopt a classic single node programming strategy in AbU, by just keeping each node name in a dedicated resource and defining a predicate selecting the nodes with a particular name.

5. Encoding attribute-based communication into AbU

To showcase the generality of our calculus, in this section we encode the archetypal calculus AbC [6] into 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 attribute-based communication within the ECA programming paradigm.

5.1. The AbC calculus

We focus on a minimal version of AbC [6], for which we define an operational semantics, as in [8]. As already pointed out, we do not target a full-fledged version of AbC, since the aim of this section is to encode into AbU the essence of the attribute-based communication, comprehensively expressed by the fragment of AbC we focus on.

An AbC *component* C can be a process paired with an attribute environment, written $\Gamma : P$, or the parallel composition of components, written $C_1 \parallel C_2$. An attribute environment Γ maps attributes $a \in \mathbb{A}$ to values $v \in \mathbb{V}$. Our syntax of AbC processes is as follows.

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

In particular, the *input* $(x \mid \Pi)$ receives a message that satisfies the predicate Π, savint it in the variable x. The *output* $\langle e \mathbin{@} \Pi\rangle$ sends (the value 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 constructs are exactly as in [6]: the *inactive process* $\texttt{0}$; the *non-deterministic choice* between $P_a + P_b$; and the *process call* K. Predicates Π and expressions e are standard. We refer the reader to [6] for more details.

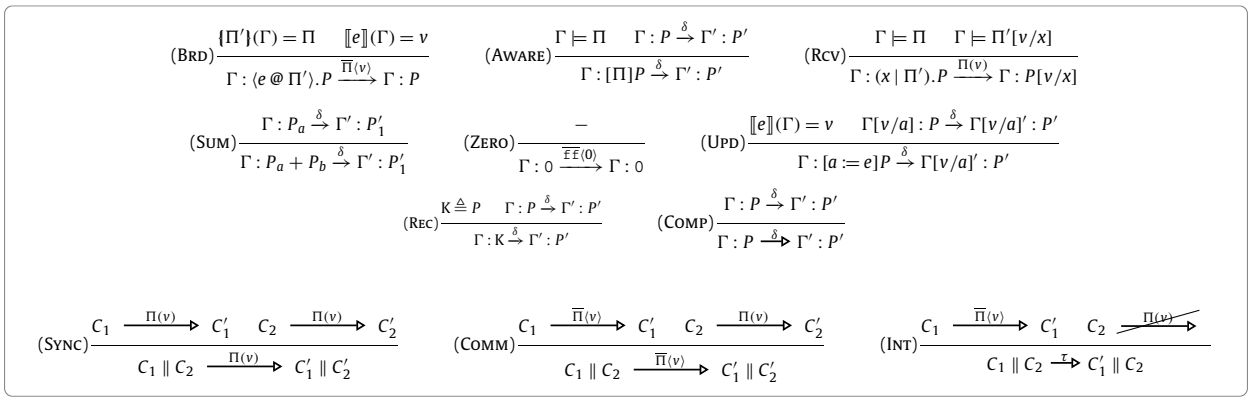


Fig. 5. AbC semantics for processes [top] and components [bottom] (symmetric rules are omitted).

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 tt or ff. Their formal definition is straightforward, the only interesting cases are: $[\![a]\!](\Gamma) = [\![\mathtt{this}.a]\!](\Gamma) = \Gamma(a)$. When $[\![\Pi]\!](\Gamma)$ is 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. 5[top]) and for components (Fig. 5[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). Transition rules in Fig. 5 are self-explanatory (symmetric rules are omitted).

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 also applies when C_2 is not ready (or it does not want) to communicate, allowing C_1 to progress.

5.2. Encoding AbC into AbU

Given a AbC component $\Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$, we define an AbU system $R_1, \iota_1\langle\Sigma_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n\rangle$ 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 that we will see in a moment). The execution pools of all nodes 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 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:

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

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 Res^h, given a process of the component h, with $h \in [1..n]$, computes the resources to add to the nodes. Here, Res^h is parametric in h since rules are binded to the component generating them. In particular, Res^h returns $\varnothing$ for the inactive process and for process calls, i.e., $\mathsf{Res}^h(0) \triangleq \mathsf{Res}^h(\mathrm{K}) \triangleq \varnothing$, and nothing is added. For the other processes, it returns $\mathsf{Res}^h(P) \triangleq \{(P_h r_0, \mathrm{ff})\} \cup \mathsf{Res}^h(P, 0)$. 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 $\mathsf{Res}^h(P, i)$, for $i \geq 0$, is defined inductively on the structure of P. In the base cases $P = 0$ 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. Let $\mathsf{Next}(i) = j$ and $\mathsf{Next}(j) = k$, then:

$$\mathsf{Res}^h(P, i) \triangleq \begin{cases} \{(x, 0), (P_h r_j, \mathrm{ff})\} \cup \mathsf{Res}^h(P', j) & \text{if } P = (x \mid \Pi).P' \\ \{(P_h r_j, \mathrm{ff}), (P_h r_k, \mathrm{ff})\} \cup \mathsf{Res}^h(P_a, j) \cup \mathsf{Res}^h(P_b, k) & \text{if } P = P_a + P_b \\ \{(P_h r_j, \mathrm{ff}) \cup \mathsf{Res}^h(P', j) & \text{if } P = [\Pi]P' \text{ or } P = [a := e]P' \\ & \text{or } P = \langle e \,@\, \Pi \rangle.P' \end{cases}$$

In particular, if the process is an input, we add the flag for the current rule and another resource for the variable x. If the process is a non-deterministic choice, we add two flags, one for each branch, that will originate two different rules. In all other cases, we just add the flag for the current rule.

Concerning the translation of 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 $\mathtt{tt}$, the rule can execute. Its execution disables $P_h r_i$ (it is set to $\mathtt{ff}$) and enables the next rule, setting the flag $P_h r_j$, with $j = \mathsf{Next}(i)$, to $\mathtt{tt}$. In this way, the execution token can be exchanged between rules. The function Enc^h, given a process of the component h, with $h \in [1..n]$, generates the rules to add to the translation. It relies on Next, that outputs a fresh index for the next rule to generate. We assume that Next in Enc^h is consistent with Next in Res^h, i.e., they have to produce the same sequence of indexes given a specific process. The function $\mathsf{Enc}^h(P, i)$, 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. In other words, given $K \triangleq P_k$:

$$\mathsf{Enc}^h(K, i) \triangleq P_h r_i \gg (P_h r_i = \mathtt{T}) : P_h r_i \leftarrow \mathtt{F}\ P_k r_0 \leftarrow \mathtt{T}$$

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 variable (x in this case) with *msg*. As an example, the predicate $\Pi = x < n$ is translated to $\mathsf{Repl}(\mathsf{Enc}(\Pi), x) = msg < n$. 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. In other words, given $j = \mathsf{Next}(i)$:

$$\mathsf{Enc}^h((x \mid \Pi).P', i) \triangleq P_h r_i \gg (P_h r_i = \mathtt{T} \wedge \mathsf{Repl}(\mathsf{Enc}(\Pi), x)) : x \leftarrow msg\ P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T}\ \ \mathsf{Enc}^h(P', j)$$

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. In other words, given $j = \mathsf{Next}(i)$ and $k = \mathsf{Next}(j)$:

$$\begin{aligned}\mathsf{Enc}^h(P_a + P_b, i) \triangleq\ & P_h r_i \gg (P_h r_i = \mathtt{T}) : P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T}\ \ \mathsf{Enc}^h(P_a, j) \\ & P_h r_i \gg (P_h r_i = \mathtt{T}) : P_h r_i \leftarrow \mathtt{F}\ P_h r_k \leftarrow \mathtt{T}\ \ \mathsf{Enc}^h(P_b, j)\end{aligned}$$

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 `this.a` are both translated to the resource a. The condition in the rule task is the translation of Π. When it is satisfied, the next rule is enabled. In other words, given $j = \mathsf{Next}(i)$:

$$\mathsf{Enc}^h([\Pi]P', i) \triangleq P_h r_i\ \mathsf{Vars}(\Pi) \gg (P_h r_i = \mathtt{T} \wedge \mathsf{Enc}(\Pi)) : P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T}\ \ \mathsf{Enc}^h(P', j)$$

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. In other words, given $j = \mathsf{Next}(i)$:

$$\mathsf{Enc}^h([a := e]P', i) \triangleq P_h r_i \gg (P_h r_i = \mathtt{T}) : a \leftarrow \mathsf{Enc}(e)\ P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T}\ \ \mathsf{Enc}^h(P', j)$$

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 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 $\mathsf{Enc}(\Pi)$ where each instance (in Π) of an attribute a not prefixed by `this.` is translated to $\overline{a}$. As an example, the predicate $\Pi = \mathtt{this}.n < n$ 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 $\overline{msg} \leftarrow \mathsf{Enc}(e)$). 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. In other words, given $j = \mathsf{Next}(i)$:

$$\mathsf{Enc}^h(\langle e\ @\ \Pi\rangle.P', i) \triangleq P_h r_i \gg P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T}, @(P_h r_i = \mathtt{T} \wedge \mathsf{Ext}(\Pi)) : \overline{msg} \leftarrow \mathsf{Enc}(e)\ \ \mathsf{Enc}^h(P', j)$$

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

In the following, we denote with $\mathsf{Enc}(C)$ the AbU encoding of C, where node states are defined as explained above, node pools are empty and nodes ECA rules are generated by Enc (for each process of C).

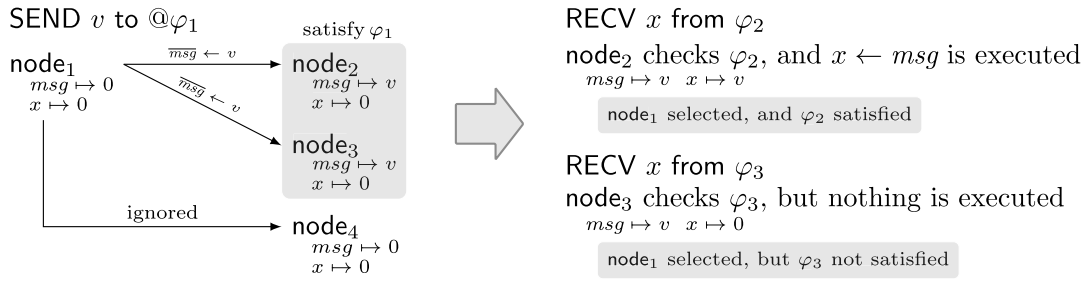


Fig. 6. Attribute-based communication in AbU, a receive phase (right) after a send phase (left).

In Fig. 6 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 $\varphi_2 = \mathsf{Repl}(\mathsf{Enc}(\Pi_2), x)$ and does not satisfy $\varphi_3 = \mathsf{Repl}(\mathsf{Enc}(\Pi_3), x)$, only node_2 accepts the value v, saving it in the resource x, while node_3 ignores the communication.

Encoding example We now show an example encoding of an AbC component into AbU. The example is taken from [27] (Subsection 2.1). 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 value. Formally, the process P (with $\mathrm{Max} \triangleq P$) is:

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

Basically, 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.

$$\begin{aligned}
R = {} & P_1r_0 \gg (P_1r_0 = \mathtt{T} \wedge s = 1) : P_1r_0 \leftarrow \mathtt{F}\ P_1r_1 \leftarrow \mathtt{T} && \text{aware rule}\\
& P_1r_1 \gg (P_1r_1 = \mathtt{T}) : P_1r_1 \leftarrow \mathtt{F}\ P_1r_2 \leftarrow \mathtt{T} && \text{choice}_1 \text{ rule}\\
& P_1r_1 \gg (P_1r_1 = \mathtt{T}) : P_1r_1 \leftarrow \mathtt{F}\ P_1r_3 \leftarrow \mathtt{T} && \text{choice}_2 \text{ rule}\\
& P_1r_2 \gg P_1r_2 \leftarrow \mathtt{F}\ P_1r_4 \leftarrow \mathtt{T}, @(P_1r_2 = \mathtt{T} \wedge \overline{n} \leq n) : \overline{msg} \leftarrow n && \text{send rule}\\
& P_1r_4 \gg (P_1r_4 = \mathtt{T}) : P_1r_4 \leftarrow \mathtt{F}\ P_1r_0 \leftarrow \mathtt{T} && \text{call rule}\\
& P_1r_3 \gg (P_1r_3 = \mathtt{T} \wedge msg \geq n) : x \leftarrow msg\ P_1r_3 \leftarrow \mathtt{F}\ P_1r_5 \leftarrow \mathtt{T} && \text{receive rule}\\
& P_1r_5 \gg (P_1r_5 = \mathtt{T}) : s \leftarrow 0\ P_1r_5 \leftarrow \mathtt{F}\ P_1r_6 \leftarrow \mathtt{T} && \text{update rule}
\end{aligned}$$

$$R\langle[msg{\mapsto}0\ n{\mapsto}1\ x{\mapsto}0\ s{\mapsto}0\ P_1r_0{\mapsto}\mathtt{ff}\ P_1r_1{\mapsto}\mathtt{ff}\ P_1r_2{\mapsto}\mathtt{ff}\ P_1r_3{\mapsto}\mathtt{ff}\ P_1r_4{\mapsto}\mathtt{ff}\ P_1r_5{\mapsto}\mathtt{ff}\ P_1r_6{\mapsto}\mathtt{ff}]\rangle$$
$$R\langle[msg{\mapsto}0\ n{\mapsto}2\ x{\mapsto}0\ s{\mapsto}0\ P_1r_0{\mapsto}\mathtt{ff}\ P_1r_1{\mapsto}\mathtt{ff}\ P_1r_2{\mapsto}\mathtt{ff}\ P_1r_3{\mapsto}\mathtt{ff}\ P_1r_4{\mapsto}\mathtt{ff}\ P_1r_5{\mapsto}\mathtt{ff}\ P_1r_6{\mapsto}\mathtt{ff}]\rangle$$
$$R\langle[msg{\mapsto}0\ n{\mapsto}3\ x{\mapsto}0\ s{\mapsto}0\ P_1r_0{\mapsto}\mathtt{ff}\ P_1r_1{\mapsto}\mathtt{ff}\ P_1r_2{\mapsto}\mathtt{ff}\ P_1r_3{\mapsto}\mathtt{ff}\ P_1r_4{\mapsto}\mathtt{ff}\ P_1r_5{\mapsto}\mathtt{ff}\ P_1r_6{\mapsto}\mathtt{ff}]\rangle$$

5.3. Correctness of the encoding

We finally provide a correctness result for the previously defined encoding. In particular, we prove that translated AbU systems preserve the semantics of the original AbC components. Since an 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 an AbU node state Σ and an 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., when $\Gamma \subseteq \Sigma$). This basically means that Σ agrees, at least, on all attributes of Γ. This notion can be extended to systems and components.

Definition 3 *(AbU to AbC compatibility).* Given an AbC component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ and an 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 $i \in [1..n]$.

The AbU translation $\mathsf{Enc}(C)$ of C yields n (one for each process) initial rule flags $P_1 r_0, \ldots, P_n r_0$, initially set to ff. In order to start the computation of $\mathsf{Enc}(C)$, the latter have to be initialized (i.e., set to tt). In this regard, we assume an initial *input phase*, comprising n AbU (Input) steps, enabling all initial rule flags (without interleaving execution steps). Let $\longrightarrow\!\!\!\!\triangleright^*$ be the transitive closure of $\longrightarrow\!\!\!\!\triangleright$ without occurrences of labels of the form $\mathsf{upd} \triangleright T$. In other words, $\longrightarrow\!\!\!\!\triangleright^*$ 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 Theorem 4 says that if an 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 $\mathsf{Enc}(C)$ is able to "simulate" each possible execution of C.

Theorem 4 *(AbC to AbU correctness). For any AbC component C, let $\mathsf{S} = \mathsf{Enc}(C)$; then for all C' such that $C \longrightarrow\!\!\!\!\triangleright^* C'$ there exists an AbU system S' such that $\mathsf{S} \longrightarrow\!\!\!\!\triangleright^* \longrightarrow\!\!\!\!\triangleright^* \mathsf{S}'$ and $\mathsf{S}' \succeq C'$.*

Proof. The proof is quite complex and it requires some preliminary results. In order to simplify the reading, we moved the full proof to Appendix A.1.

6. Correctness requirements for AbU systems

Correctness requirements aim at preventing the nodes of an AbU system to exhibit unintended behaviors. For instance, the wave semantics (and, hence, an 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 no other inputs are performed.

As a motivating example, consider the curious case of the book "The Making of a fly", that reached the stellar selling price of $23,698,655.93 on Amazon, in 2001.[4] Two Amazon retailers, *profnath* and *bordeebook*, used Amazon automatic pricing primitives to set the price of their book copy, depending on the competitor book price. The strategy of *profnath* was to automatically set the price 0.99 times the *bordeebook* price; conversely, the strategy of *bordeebook* was to set the price 1.27 times the *profnath* price. Obviously, each retailer was not aware of the competitor's strategy. This scenario can be modeled by means of the following (informal) ECA rules:

- when *bordeebook-price* changes, set *profnath-price* as *bordeebook-price* decreased by 1%
- when *profnath-price* changes, set *bordeebook-price* as *profnath-price* increased by 27%

that translate to AbU as:

$$\mathit{bordeebook\text{-}price} \gg (\mathtt{T}) : \mathit{profnath\text{-}price} \leftarrow \mathit{bordeebook\text{-}price} * 0.99$$
$$\mathit{profnath\text{-}price} \gg (\mathtt{T}) : \mathit{bordeebook\text{-}price} \leftarrow \mathit{profnath\text{-}price} * 1.27$$

It is easy to see that these rules generate a loop, leading to an uncontrolled raise of the book price (as it happened). In order to prevent these situations, we may define a 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.

Another quite import requirement for AbU systems is confluence, that can be seen as a sort of scheduler-independence. In other words, confluence says that no matter which is the order of execution of the updates in the pools (given by the AbU scheduler), the system eventually produces a unique result. This can be useful in contexts where the final result of rules application must be predictable.

In the following, we first formally define stabilization and confluence requirements, then we provide syntactic sufficient condition that guarantee the satisfaction of these requirements by a given AbU system.

6.1. Stabilization and confluence

Before formally defining the correctness requirements, we have to introduce some notions and notations. Given the set $\mathbb{N}$ of natural numbers, with cardinality $|\mathbb{N}| = \omega$, we denote with $\mathbb{N}^*$ and $\mathbb{N}^\omega$ the set of *finite* sequences and *infinite* sequences, respectively, over naturals. We denote with $\mathbb{N}^\infty \triangleq \mathbb{N}^* \cup \mathbb{N}^\omega$ the set of (finite and infinite) sequences over naturals and with ϵ the empty sequence. Given a sequence $\overline{n} \in \mathbb{N}^\infty$, we define the set of its (finite) *prefixes* as:

$$\mathsf{pref}(\overline{n}) \triangleq \left\{ \overline{n'} \in \mathbb{N}^* \;\middle|\; \exists \overline{n''} \in \mathbb{N}^\infty \cup \{\epsilon\} \,.\, \overline{n'}\,\overline{n''} = \overline{n} \right\}$$

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

An infinite sequence $\overline{n} \in \mathbb{N}^\omega$ approximates (to the limit) a set of finite sequences $X \subseteq \mathbb{N}^*$ when $\overline{n}$ has an infinite number of prefixes in common with X. In this case, $\overline{n}$ is exactly the *Eilenberg limit* [39,40] of X, and it is formally defined as:

$$\mathsf{E\text{-}limit}(X) \triangleq \{\overline{n} \in \mathbb{N}^\omega \mid |\mathsf{pref}(\overline{n}) \cap X| = \omega\}$$

As an example, the Eilenberg limit of the (infinite) set $\bigcup_{i\in\mathbb{N}}\{(57)^i\} = \{\epsilon, 57, 5757, 575757, \ldots\}$ is the infinite sequence $(57)^\omega$. Note that, $\mathsf{E\text{-}limit}(X) = \varnothing$ when X is finite.

Given a set of sets $\mathfrak{X}$, we denote with $\mathsf{max}(\mathfrak{X}) \triangleq \{X \in \mathfrak{X} \mid \forall X' \in \mathfrak{X}\,.\, X \subseteq X' \Rightarrow X = X'\}$ the set of maximal elements of $\mathfrak{X}$. Then, given a set X, we define the set of its maximal subsets as:

$$\wp^{\mathsf{max}}(X) \triangleq \{\mathfrak{X} \subseteq \wp(X) \mid \mathsf{max}(\mathfrak{X}) = \mathfrak{X} \wedge \bigcup \mathfrak{X} = X\}$$

where the symbol $\wp$ denotes the powerset operator. In other words, each element of $\wp^{\mathsf{max}}(X)$ is closed under maximal elements and saturates X.

As we will see in a moment, to reason about complex AbU system behaviors we need a detailed notion of system trace, that comprises execution states, execution pools *and* LTS labels. We want to stress the fact that the usage of labels in the semantics and in the definition of system trace does not imply that AbU is based on messages exchange. LTS labels are used by the AbU semantics only to record nodes state variation, they do not carry any actual semantic information regarding nodes. In other words, what we actually observe of the AbU semantics is not the messages (LTS labels) exchanged but, rather, the variation of nodes state.

We now formally define the internal termination requirement, that we call *stabilization*. We denote with ExecLabels the set of all possible execution labels, namely AbU labels of the form $\mathsf{upd} \triangleright T$, for some upd and T. Recall that, $\mathsf{S} \longrightarrow^* \mathsf{S}'$ means that that S' can be obtained from S by applying execution steps only, namely in $\longrightarrow^*$ we can only have labels belonging to ExecLabels. Note that, when $\mathsf{S} \longrightarrow^* \mathsf{S}'$ then there exists $k \in \mathbb{N}$ such that $\mathsf{S} \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} \mathsf{S}'$. The latter is a *finite computation* (of length k) originated by S. A finite computation may lead to a stable system (i.e., all pools in S' are empty) or to a not stable system (i.e., we have at least one non-empty pool in S'). Stabilization states that after some execution steps a system becomes stable.

Definition 4 *(Weak stabilization).* An AbU system S is *weakly stabilizing* when there exists an AbU system S' such that S' is stable and $\mathsf{S} \longrightarrow^* \mathsf{S}'$.

Weak stabilization does not exclude that S may originate both computations leading to a stable system and computations not leading to any stable system. Indeed, we can have an infinite sequence of applications of the AbU semantics involving execution steps only. In other words, S may originate an *infinite computation* $\mathsf{S} \xrightarrow{\alpha_0} \mathsf{S}^0 \xrightarrow{\alpha_1} \ldots$ (with $\alpha_i \in$ ExecLabels and $i \in \mathbb{N}$) that does not lead to any stable system.

Definition 5 *(Stabilization).* An AbU system S is *stabilizing* when it does not yield infinite computations, namely S cannot originate infinite sequences $\mathsf{S} \xrightarrow{\alpha_0} \mathsf{S}^0 \xrightarrow{\alpha_1} \ldots$ (with $\alpha_i \in$ ExecLabels and $i \in \mathbb{N}$).

In other words, a weakly stabilizing system *may* become stable, i.e., there exists a scheduling that yields a stable system; while a stabilizing system *must* become stable, i.e., all possible schedulings yield, eventually, a stable system. The reader familiar with Term Rewriting Systems can note that stable AbU systems are conceptually analogous to terms in *normal form* [41]. Indeed, a term is in normal form when no rewriting rules can be applied to it; analogously, an AbU system is stable when no (Exec) rules can be applied to it. Similarly, weak stabilization is analogous to *weak normalization* [41] (all terms admit a normal form) and stabilization is analogous to *strong normalization* [41] (all rewriting sequences are finite).

We now define the second requirement for AbU systems, i.e., *confluence*, that can be seen as a sort of scheduler-independence. In other words, confluence says that no matter which is the order of execution of the updates in the pools (given by the AbU scheduler), the system eventually produces a unique result.

Definition 6 *(Confluence).* An AbU system S is *confluent* when for all AbU systems S_1 and S_2 such that $\mathsf{S} \longrightarrow^* \mathsf{S}_1$ and $\mathsf{S} \longrightarrow^* \mathsf{S}_2$ there exists an AbU system S' such that $\mathsf{S}_1 \longrightarrow^* \mathsf{S}'$ and $\mathsf{S}_2 \longrightarrow^* \mathsf{S}'$.

In other words, if $\mathsf{S} \longrightarrow^* \mathsf{S}_1$ and $\mathsf{S} \longrightarrow^* \mathsf{S}_2$, for different systems S_1 and S_2, we have two different scheduling that we can apply from S. Nevertheless, if S is confluent, both scheduling eventually yield a unique system S'. Note that, confluence does not imply that the system stabilizes, indeed, we can have confluent systems that generates infinite computations.

Definition 7 *(Convergence).* An AbU system S is *convergent* when it is confluent *and* stabilizing.

Convergence means that a system eventually stabilizes, yielding an unique result, no matter what scheduling have been chosen. Continuing the previous parallelism, AbU confluence and convergence are analogous to the *Church-Rosser property*[5] and *convergence* [41] of Term Rewriting Systems, respectively. Indeed, the latter holds when a Term Rewriting System is both strongly normalizing and Church-Rosser, like AbU convergence holds when a system is both stabilizing and confluent.

Definition 8 *(Wave weak stabilization, stabilization, confluence and convergence).* A stable AbU system S is *wave weakly stabilizing*, *wave stabilizing*, *wave confluent* or *wave convergent* when for any input upd we have that $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$ implies S' is a weakly stabilizing, stabilizing, confluent or convergent AbU system, respectively.

Wave weak stabilization means that given an input, then the system *may* become stable again, after executing that input. In other words, if $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}''$ then there exists S' such that $\mathsf{S} \overset{\mathsf{upd}}{\rightsquigarrow} \mathsf{S}'$. Wave stabilization means that given an input, then the system *must* become stable again, after executing that input. In other words, if $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$ then all execution path starting from S' eventually yields a stable system. Wave confluence means that given an input, then system eventually reaches a unique system after executing that input. In other words, if $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$ then S' eventually reaches a unique system (possibly not stable). Finally, wave convergence means that given an input, then system eventually stabilizes to an unique system after executing that input.

As we have seen in Subsection 3.3, AbU is in general Turing complete, hence checking requirements as stabilization and confluence is an undecidable problem. For instance, if we were able to decide stabilization for AbU with non-deterministic substitutions we were also able to decide termination of Turing machines. Similarly, if we were able to decide confluence for AbU with non-deterministic substitutions we were also able to decide confluence of rewriting systems, which is known to be undecidable in general [42]. Nevertheless, we can still aim for verification procedures for stabilization and confluence that are sound but possibly not complete, i.e., they may fail to recognize stabilizing or confluent systems as such. In the following we will present two sound verification mechanisms for checking stabilization and confluence of AbU systems.

6.2. Verifying stabilization

We now show how stabilization can be statically checked, for a given AbU system, namely we provide a verification mechanism that can be used before the deployment of the system. It basically consists in checking if there are chains of AbU rules in the system that form cycles.

The *output resources* of an AbU rule, namely the resources involved in the actions performed by the rule, are given by the resources assigned in the default action and in the rule 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 an 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 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. More formally:

$$\mathsf{Out}(\mathsf{rule}_1 \ldots \mathsf{rule}_n) \triangleq \bigcup_{i \in [1..n]} \mathsf{Out}(\mathsf{rule}_i) \quad \text{and} \quad \mathsf{In}(\mathsf{rule}_1 \ldots \mathsf{rule}_n) \triangleq \bigcup_{i \in [1..n]} \mathsf{In}(\mathsf{rule}_i)$$

Definition 9 *(ECA dependency graph).* Given an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$, the *ECA dependency graph* of S is a directed graph (V, E) where vertices (or nodes) V and edges E are:

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

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

Theorem 5 *(Soundness for stabilization). Given a stable AbU system* S*, if the ECA dependency graph of* S *is acyclic, then* S *is wave stabilizing.*

Proof. The proof is quite long and involved and it requires some preliminary results. For sake of readability, we moved the full proof to Appendix A.2.

A naive termination verification 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 x. If there are no reflexive elements then the graph

[5] Very often called confluence even in the case of Term Rewriting Systems.

```
Algorithm hasCycles(V, E)
1   visited := [notVisited]
2   cycle := False
3   for v ∈ V and cycle = False do
4       if visited[v] = notVisited then
5           S := [ ]
6           S.push(v)
7           visited[v] := inStack
8           cycle := hasCyclesDFS(V, E, S, visited)
        end
    end
9   return cycle
```

```
Procedure hasCyclesDFS(V, E, S, visited)
1   cycle := False
2   for v ∈ adjacent(V, S.top) and cycle = False do
3       if visited[v] = inStack then
4           cycle := True
        else
5           if visited[v] = notVisited then
6               S.push(v)
7               visited[v] := inStack
8               cycle := hasCyclesDFS(V, E, S, visited)
            end
        end
    end
9   return cycle
```

Algorithm 1: Detect cycles in an ECA dependency graph using a DFS search.

is acyclic and the condition is fulfilled. A more efficient solution consists in Algorithm 1. It is a DFS-based algorithm to check cycles in directed graphs. It maintains a stack of visited nodes in the DFS to check back edges. If a back edge in the DFS-traversing of the graph is found, then the graph is not acyclic (Algorithm 1 returns **False**). The time complexity of Algorithm 1 is mainly due to the DFS search, hence it is $O(|V| + |E|)$, in the worst case. Notice that, in V each variable is counted once even if it is present in more than one node; hence, $|V|$ counts the number of different variables appearing in the whole system. On the other hand, $|E|$ is bound by the total number of rules of the system. Nevertheless, between each pair of variables there can be at most one edge; so different rules connecting the same pairs of resources are counted once.

All three AbU systems presented in Section 4 are wave stabilizing, and our verification mechanism is able to correctly state that the systems satisfy the requirement. Indeed, all ECA dependency graphs in the examples are acyclic.

Remark 5. Note that, in some cases, the proposed approach based on the ECA dependency graph may be too restrictive. For instance, many distributed algorithms (e.g., the Bracha-Toueg crash consensus algorithm [43] or the Franklin's election algorithm [44]) are based on *rounds*, thus requiring some synchronization between nodes. AbU is asynchronous by design, but round computations can be simulated by using indexed variables: each resource x can be an array of variables $x_1\, x_2 \ldots$ and at round $i+1$, events and updates read from i-th elements of the array (i.e., x_i) and update the $i+1$-th. For instance, an update $x \leftarrow x * 2$ at round $i+1$ would be translated to $x_{i+1} \leftarrow x_i * 2$. This round-based mechanism introduces a cyclic dependency on resources, thus failing the proposed stabilization check. This is, in principle, not necessarily bad since rounds-based algorithms may diverge, if not properly designed, and convergence can be proved in different ways (as for [43,44]). Nevertheless, the static condition we have proposed, based on the acyclicity of the ECA dependency graph, is a sufficient but not necessary condition for stabilization. As future work, we plan to develop more precise solutions, i.e., allowing also for some programs having a cyclic ECA dependency graph but that are not actually diverging (as in the case of [43,44]).

6.3. Verifying confluence

In order to verify confluence, the information contained in the ECA dependency graph is not sufficient. Indeed, we need to track not only which resource depends on another, but the ECA rule introducing that dependency as well. We can extend the ECA dependency graph with a label on arcs containing the rule that justifies the dependency. An arc of the form (x, rule, y) means: modifying x we may modify y as well, by applying the rule rule. In the following, we denote an AbU rule of an AbU system by means of a numeric index: $r^{\langle i,j \rangle}$ stands for the j^{th} rule of the i^{th} node of the system.

Definition 10 *(Labeled ECA dependency graph).* Given an AbU system $\mathsf{S} = R_1, \iota_1\langle \Sigma_1, \Theta_1 \rangle \parallel \ldots \parallel R_n, \iota_n\langle \Sigma_n, \Theta_n \rangle$, the *labeled ECA dependency graph* of S is a directed graph $(V, {}^{\ell}E)$ where vertices (or nodes) V and labeled edges ${}^{\ell}E$ are:

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

A *finite walk* in the labeled ECA dependency graph between the resources x and y is a sequence of rules that connect the two resources (a sort of transitive dependency) and it is given by the labels of the graph. Roughly speaking, a finite walk in the labeled ECA dependency graph represents a possible schedule of updates execution. More formally, given a labeled ECA dependency graph $(V, {}^{\ell}E)$, two arcs (x, r, y) and (x', r', y') are *joinable*, written $(x, r, y) \frown (x', r', y')$, when $y = x'$. We then define the finite walks between two resources x and y as:

$$\mathsf{finite\text{-}walks}(x, y) \triangleq \{ r_1 \ldots r_n \mid \exists z_1, \ldots, z_{n-1} \in V\, .\, (x, r_1, z_1) \frown \ldots \frown (z_{n-1}, r_n, y) \}$$

Note that, all sequences in the set $\mathsf{finite\text{-}walks}(x, y)$ are finite, but the set may have an infinite number of elements (meaning that x and y are involved in a cycle). In order to precisely verify confluence, we have to separate finite computations from computations involved in cycles, that not necessarily break confluence. In this respect, since a finite walk is a sequence of indexes (i.e., natural numbers), we can use the Eilenberg-limit in order to represent an infinite set of finite walks by means of a finite set of infinite walks. In particular, we define the set of *limit walks* between the resource x and the resource y as:

$$\mathsf{limit\text{-}walks}(x, y) \triangleq \mathsf{E\text{-}limit}(\mathsf{finite\text{-}walks}(x, y))$$

In this way, we can distinguish finite computations from infinite ones, involved in cycles. Hence, we can formally define the *walks* that link two resources, consisting in the limit walks, together with the finite walks that are not approximated by any limit walk. Formally, the walks between the resources x and y are:

$$\mathsf{walks}(x, y) \triangleq \mathsf{limit\text{-}walks}(x, y) \cup \mathsf{finite\text{-}walks}(x, y) \setminus \left\{ \overline{n} \in \mathsf{pref}(\overline{n'}) \;\middle|\; \overline{n'} \in \mathsf{limit\text{-}walks}(x, y) \right\}$$

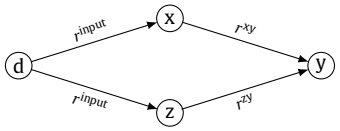
The sufficient syntactic condition for the confluence of the AbU semantics consists in checking the cardinality of the walks in the labeled ECA dependency graph between all pairs of nodes. Indeed, if $|\mathsf{walks}(x, y)| > 1$, we have different sequences of rules (i.e., scheduling) that may lead to a modification of y when we modify x, hence possibly breaking confluence. Unfortunately, confluence depends also on the input that has been performed, not only on the syntactic structure of the AbU rules. Consider the following two rules:

$$x \gg (\mathtt{T}) : y \leftarrow 1 \quad \text{and} \quad z \gg (\mathtt{T}) : y \leftarrow 2 \tag{2}$$

that seems to be confluent (and, indeed, we have at most 1 walk between each pair of resources). Actually we may have different results if an input changes x and z *simultaneously*. Indeed, the final value for y can be either 1 or 2, depending on the scheduling. For this reason, we need to augment the labeled ECA dependency graph with dummy vertices and arcs, simulating simultaneous inputs.

Let $\mathsf{input}^{\mathsf{S}} \triangleq \bigcup_{i \in [1..n]} \mathsf{In}(R_i)$ be the input resources of the system $\mathsf{S} = R_1, \iota_1\langle \Sigma_1, \Theta_1 \rangle \parallel \ldots \parallel R_n, \iota_n\langle \Sigma_n, \Theta_n \rangle$. We define an *input sampling family* as an element $\mathcal{F} \in \wp^{\mathsf{max}}(\mathsf{input}^{\mathsf{S}})$. The latter corresponds to the possible combination of inputs (of S) that can be simultaneously changed by the environment. Chosen a sampling family $\mathcal{F} = \{S_1, \ldots, S_k\}$, which is a finite set, we add to the labeled ECA dependency graph of S all the arcs $(s_i, r^{\mathsf{input}}, x)$ such that: $i \in [1..k]$ and $x \in S_i$. Here s_i, for $i \in [1..k]$, are dummy fresh resources not present in S, and r^{input} represents a dummy rule not present in S. The best case corresponds to the sampling family such that $|\mathcal{F}| = |\mathsf{input}^{\mathsf{S}}|$, namely when $\mathcal{F}$ contains only singletons. In this case, we have that we can change only one resource at each input step.[6] Conversely, the worst case corresponds to the sampling family such that $|\mathcal{F}| = 1$, namely when $\mathcal{F}$ contains only the set $\mathsf{input}^{\mathsf{S}}$. In this case, we have that we can simultaneously change all (input) resources at each input step.

Continuing the example with the AbU rules in Equation (2), suppose to have a sampling family $\mathcal{F} = \{\{x, z\}\}$, namely x and y can be simultaneously changed by input steps. The augmented labeled ECA dependency graph of the system can be depicted as:



r^{xy}: index of $x \gg (\mathtt{T}) : y \leftarrow 1$
r^{zy}: index of $z \gg (\mathtt{T}) : y \leftarrow 2$

with $\mathsf{walks}(d, y) = 2$, hence we can find the confluence failure. Indeed, with the augmented labeled ECA dependency graph, the cardinality of the walks between each pair of resources provides us a verification mechanism for confluence.

Theorem 6 *(Soundness for confluence). Given a stable AbU system* S*, if for each pair of resources* x *and* y *in the (augmented) labeled ECA dependency graph of* S *we have that* $|\mathsf{walks}(x, y)| \leq 1$*, then* S *is wave confluent.*

Proof. The proof is quite long and involved and it requires some preliminary results. For sake of readability, we moved the full proof to Appendix A.2.

A confluence verification mechanism consists in computing the cardinality of the sets $\mathsf{walks}(x, y)$, for each pair of resources x and y. Algorithm 2 checks if the number of walks between each pair of resources in the labeled ECA dependency graph is greater than 1 (in that case it returns **False**). The algorithm first initializes the adjacent matrix *adj* of the graph (line **1**) and then computes the successive powers of the adjacent matrix (lines **2**..**6**), in order to find multiple finite walks.

[6] Indeed, the added dummy resources and rules are not useful and therefore can be removed.

```
Algorithm hasMultipleWalks(V, ℓE)
1   adj := initWalksCount(|V|, ℓE)
2   walks := adj
3   for i from 1 to |V| − 1 do
4       if multipleWalks(|V|, walks) then
5           return False
        end
6       walks := matrixMult(walks, adj)
    end
7   if walks = 0 then
8       return True
    end
9   for i from 1 to |V| do
10      if selectColumn(walks, i) ≠ 0 then
11          if sumRow(adj[i]) > 1 then
12              return False
            end
        end
    end
13  return True

Procedure initWalksCount(n, ℓE)
1   for i from 1 to n do
2       for j from 1 to n do
3           walks[i][j] := 0
        end
    end
4   for (v_i, r, v_j) ∈ ℓE do
5       walks[i][j] := walks[i][j] + 1
    end
6   return walks

Procedure multipleWalks(n, walks)
1   for i from 1 to n do
2       for j from 1 to n do
3           if walks[i][j] > 1 then
4               return True
            end
        end
    end
5   return False
```

Algorithm 2: Detect multiple walks between all pairs of resources in a labeled ECA dependency graph.

Indeed, in each element (i, j) of adj^n we have the number of finite walks of length n between the nodes v_i and v_j. Hence, computing $adj^{|V|-1}$ we obtain all the possible finite walks, since we cannot have walks longer than, or equal to, the number of nodes, if we exclude cycles. At each iteration, in line **4** we check if we already have multiple walks between the resources: if it is the case then the algorithm says that the system is not confluent. Once we have checked the finite walks, we need to check eventual multiple limit walks (lines **7**..**8**). If no cycles are detected, then the system is confluent. When $adj^{|V|}$ is not zero, then we have a walk of length $|V|$ involving at most $|V|$ nodes, hence we have a cycle. In this case, we check the out-degree of the nodes in the (initial) adjacent matrix involved in cycles (lines **9**..**12**). To select such nodes, we look at $adj^{|V|}$, selecting its columns that have at least one non-zero element. Indeed, if a column i has a non-zero element then the corresponding node v_i is the last step of a walk with length $|V|$, hence it is involved in (or it is reachable from) a cycle. For the selected nodes, we compute the out-degree, making the sum of the corresponding row in the initial adjacent matrix *adj*. If for all nodes the out-degree is less than 2 we have confluence.

The time complexity of the procedure `multipleWalks` is $O(|V|^2)$, in the worst case. Matrix multiplication can be implemented in $O(|V|^3)$, hence the total time complexity of the loop at lines **3**..**6** is $O(|V|^4)$, in the worst case. The loop at lines **9**..**12** has time complexity $O(|V|^2)$, in the worst case. Finally, the time complexity of the procedure `initWalksCount` is $O(|V|^2 + |\ell E|)$, in the worst case. Indeed, the computation of the direct dependencies between resources in the procedure `initWalksCount` requires to access all elements of ℓE, that may have a cardinality less, equal or greater than the cardinality of V. Hence, the time complexity of the whole Algorithm 2 is $O(|V|^4 + |\ell E|)$, in the worst case.

All three AbU systems presented in Section 4 are wave confluent, but our verification mechanism is able to correctly state that the systems satisfy the requirement only for the first two. Indeed, for the vineyard example, we have two (finite) walks between the resources *moisture* and *valve* (one originated by the rule in R_s that opens the valve and another originated by the rule in R_s that closes the valve). Nevertheless, these two rules cannot yield non-confluent behaviors, since their conditions are disjoint (the two conditions cannot be true at the same time). In this case we have a false positive, since the verification mechanism flags as not correct a system that actually fulfills the requirement. We plan to improve our verification mechanism with more "semantic" checks as a future work.

7. Towards a distributed implementation

In this section, we discuss how the proposed calculus AbU can be implemented. 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 [45] 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 favor 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, that is summarized in Fig. 7.

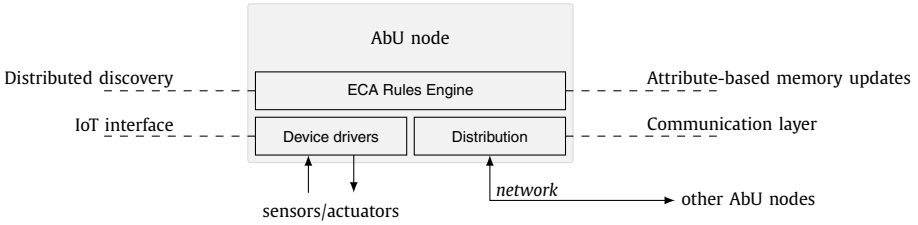


Fig. 7. High-level view of a AbU node implementation.

An 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 behavior). An *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 $\mathsf{upd} \blacktriangleright T$ and $\mathsf{upd} \triangleright T$ 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 [28,29] 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 [46], a mature Go library for the IoT ecosystem, with a great availability of IoT devices drivers. Indeed, in Fig. 7 the Device Drivers and the Distribution modules can exploit GOBOT and GoAt, respectively.

Remark 6. Filling the gap between the theoretical model provided by AbU and an actual implementation of the calculus is not a trivial task. Indeed, some crucial implementation choices should be made. For instance, the implementation should be space-uncoupled: apart from an operation to allow a new node to join an AbU system, the implementation shall provide complete location transparency. In particular, a user shall not (need to) know the address of any node in the system and shall program interactions using AbU primitives only. This requires the development of a suitable distribution module enforcing such requirement; e.g., membership and of member failure detection can be implemented by using a SWIM-like [47] gossip-based protocol. As another example, in the operational semantics of AbU, the discovery phase is executed in a single atomic step. This implies that the partially evaluated tasks must be delivered, to the other nodes, with an atomic "all or none" operation, similarly to causally ordered reliable broadcast. We can enforce such requirement by means of *distributed transactions*, viewing an AbU system as a distributed data store and performing the discovery phase using a two-phase commit protocol [48]. We refer to [49] for more details about the design and implementation of a DSL based on AbU.

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 [50], 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.

8. Conclusion

In this paper we have presented AbU, a calculus merging the simplicity of ECA programming with *attribute-based memory updates*, This new loosely-coupled 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. Therefore, AbU aims to be a formal programming paradigm for smart systems, IoT scenarios and edge computing. To showcase the expressiveness and generality of this calculus, we have modeled with AbU some application situations typical of IoT and smart systems, discussed the Turing completeness of the calculus, and encoded AbC, the paradigmatic calculus for attribute-based communication, into AbU. 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.

Then, we have used AbU for studying *stabilization* and *confluence*, two properties very relevant for IoT and smart devices: the first guarantees that a chain of rule executions triggered by an external event will eventually terminate; the second guarantees that the effects of the rules are eventually deterministic and do not depend on the rule execution order. For both these properties we have provided formal definition, sound syntactic verification criteria, and effective algorithms to statically check such criteria.

Summarizing, AbU, as a formal model, is the basis for investigating important properties of event-driven architectures with attributed-based interaction, and for the development of formal methods for guaranteeing these properties.

Moreover, due to its decentralized nature, AbU helps IoT systems in being more scalable, dropping the dependency on a central controlling service and allowing inter-node (direct) communication. This also mitigates availability issues, since no Internet connection and no external controlling nodes, prune to failures, are needed in local executions. In addition, not having the necessity to interact with external unknown/untrusted parties (e.g., the central controlling service on the cloud) results in a more secure setting, by design. Indeed, actual IoT systems should implement complex mechanisms to enforce privacy and security (e.g., by using an encrypted communication), since data is continuously sent on the cloud. Unfortunately, very often these mechanisms are weak, or not present at all, opening to potential attacks. In AbU this problem is mitigated, since no communication to external/untrusted parties is mandatory. Nevertheless, in contexts where an external communication is required, also in the case of AbU suitable security mechanisms should be deployed.

Finally, AbU can be used as a reference model for the implementation of full-fledged programming language, or extensions of existing languages, for IoT and edge computing. Finally, the new interaction model we have introduced in this paper, namely attribute-based memory updates, can be of interest also in other settings and applications, beyond IoT and edge computing.

Future work The present work is the basis for several research directions. First, we can use AbU for defining suitable behavioral equivalences, e.g., based on *bisimulations*, to compare systems with their specifications and to specify other properties, such as non-interference. Some results in this direction for verifying *safety* and *security* requirements, have been presented in [51].

The smooth integration of attribute-based communication within the ECA paradigm makes easier to extend to the distributed setting many known results and techniques from the literature. In particular, we are interested in porting to AbU the verification techniques developed for ECA languages, such as IRON, a real-world ECA language for IoT [19,17,18]. This will also go in par with refining the verification techniques for stabilization and confluence proposed in Section 6, which are sound but sometimes too restrictive; e.g., programs which stabilize even if their ECA dependency graph is cyclic, are ruled out by these conditions.

Concerning implementations, a prototype in Golang of AbU is under development [9,49], as discussed in Section 7. Efficient distributed implementations of AbU could be obtained using RPCs or message-passing, taking inspiration from the implementations of AbC [28,27,29].

Another important subject is *distributed run-time verification and monitoring*; this would allow us to detect at run-time the violations of given properties, e.g., expressed in temporal logics like the modal μ-calculus [52]. These would be useful, for instance, to extend (and refine) the criteria presented in Section 6.

Most IoT systems rely on a notion of time to operate. We can reason about AbU system time aspects externally, e.g., by counting the number of inter-node communications or the internal execution steps. However, an AbU semantics embedding a notion of time, possibly stochastic, would result in a more effective and elegant way of reasoning about time aspects of AbU systems. As a future work, we plan to apply the general approach presented in [53,54] to our calculus.

Finally, we plan to investigate the connection between the Field Calculus [13,14] and AbU, in particular whether it is possible to encode the former in our calculus. This would require to define a notion of global goal function (if any) for an AbU system, that is carried out by AbU system nodes taken as whole.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgement

This work was partially supported by Italian MIUR project PRIN 2017FTXR7S *IT MATTERS (Methods and Tools for Trustworthy Smart Systems)* and MUR project NRRP PE00000014 *SERICS (SEcurity and RIghts In the CyberSpace)* funded by the EU—NGEU.

Appendix A. Proofs

A.1. Proofs of Section 5

Proof of Theorem 4. In order to prove the theorem, we need some supporting lemmas and some preliminary notions. Given an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$, we denote with $\mathsf{S}[R'_k, \iota'_k\langle\Sigma'_k, \Theta'_k\rangle \gg R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle]$, for $k \in [1..n]$, the system S where $R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle$ is replaced by $R'_k, \iota'_k\langle\Sigma'_k, \Theta'_k\rangle$. We use the same notation for replacing AbC components,

namely we denote with $C[\Gamma'_k : P'_k \gg \Gamma_k : P_k]$, for $k \in [1..n]$, the component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ where $\Gamma_k : P_k$ is replaced by $\Gamma'_k : P'_k$. Given an AbC component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$, the translation functions Res^h and Enc^h, with $h \in [1..n]$, retrieve all the auxiliary resources and the AbU rules, respectively, needed to model the behavior of the processes P_h (see Section 5 for the definition of such functions). In particular, the function Res^h retrieves, among other resources, all the *rule flags* of the process P_h. The functions Res^h and Enc^h are inductively defined on the structure of each P_h. Indeed, there is a one-to-one correspondence between the AbC process instances in P_h and the generated AbU rules (see again Section 5). The latter, have an unique rule flag, of the form $P_h r_i$, with $h \in [1..n]$ and $i \geq 0$. Hence, for each *residual* process P'_h of P_h, i.e., a process we can obtain applying zero or more times the rewriting semantics of AbC, we can associate a *firing rule* $\mathsf{firing}(P'_h)$, namely the rule flag that must be active, in order to execute the process P'_h. Note that, P_h is a residual of P_h itself. The notion of residual can be extended to components: C' is a residual of C if we can obtain C' applying zero or more times the rewriting semantics of AbC to C. We can then define the firing rules set $\mathsf{firing}(C')$ of the residual component $C' = \Gamma'_1 : P'_1 \parallel \ldots \parallel \Gamma'_n : P'_n$ as $\{\mathsf{firing}(P'_h) \mid h \in [1..n]\}$. Note that, $\mathsf{firing}(C) = \{\mathsf{firing}(P_1), \ldots, \mathsf{firing}(P_n)\} = \{P_1 r_0, \ldots, P_n r_0\}$, by definition.

Definition 11 *(Firing AbU system).* An AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ is *firing* for the AbC component $C' = \Gamma'_1 : P'_1 \parallel \ldots \parallel \Gamma'_n : P'_n$, residual of C, when all the following hold, for each $k \in [1..n]$:

- $\mathsf{firing}(P'_k) = P_h r_i$, for some $h \in [1..n]$ and $i \geq 0$
- $\Sigma_k(P_h r_i) = \mathsf{tt}$
- $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k) \subseteq \Theta_k$
- $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = \mathsf{task}_1 \ldots \mathsf{task}_m$
- $\forall j \in [1..n] \setminus \{k\} \,.\, \{[\![\mathsf{act}]\!]\Sigma_j \mid \exists l \in [1..m] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma_j \models \varphi\} \subseteq \Theta_j$

Proposition 7. *We can formulate the following observations, assuming a component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ of AbC and an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$, such that $R_i = \mathsf{Enc}^i(P_i)$ for $i \in [1..n]$.*

- A *Let $\Gamma : P$ be a residual of $\Gamma_i : P_i$, with $i \in [1..n]$, and e an expression. If $\Sigma_i \succeq \Gamma : P$ and $[\![e]\!](\Gamma_i) = v$, for some v, then $[\![\mathsf{Enc}(e)]\!](\Sigma_i) = v$.*
- B *Let $\Gamma : P$ be a residual of $\Gamma_i : P_i$, with $i \in [1..n]$, and Π a predicate. If $\Sigma_i \succeq \Gamma : P$, $\Gamma \models \Pi[v/x]$, for some v, and $\Sigma_i(msg) = v$, then $\Sigma_i \models \mathsf{Repl}(\mathsf{Enc}(\Pi), x)$.*
- C *Let $C' = \Gamma'_1 : P'_1 \parallel \ldots \parallel \Gamma'_n : P'_n$ be residual of C, $\Gamma'_1 : P'_1 \xrightarrow{\delta} \Gamma''_1 : P''_1$, for some AbC process label δ, and $\mathsf{S} \succeq C'$. If $\mathsf{S} \longrightarrow^* \mathsf{S}'$, for some $\mathsf{S}' = R_1, \iota_1\langle\Sigma'_1, \Theta'_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma'_n, \Theta'_n\rangle$, by executing the AbU rule translation of P'_1, then no updates on attributes of Γ'_i on the AbU node i, for $i \in [2..n]$, are performed and no rule flags on the AbU node i, with $i \in [2..n]$, are modified. In other words: $\Sigma_i(a) = \Sigma'_i(a)$, for all attributes a of Γ'_i, with $i \in [2..n]$; and $\Sigma_i(P_k r_j) = \Sigma'_i(P_k r_j)$, for all $i \in [2..n]$, $k \in [1..n]$ and $j \geq 0$.*
- D *Let $\Gamma'_i : P'_i$ and $\Gamma'_j : P'_j$ be residuals of $\Gamma_i : P_i$ and $\Gamma_j : P_j$, respectively, with $i, j \in [1..n]$, and Π a predicate. If $\Sigma_i \succeq \Gamma'_i : P'_i$, $\Sigma_j \succeq \Gamma'_j : P'_j$, $\{\!|\Pi|\!\}(\Gamma'_i) = \Pi'$ and $\Gamma'_j \models \Pi'$, then $\Sigma_j \models \{\!|\mathsf{Ext}(\mathsf{Enc}(\Pi))|\!\}\Sigma_i$.*

Proof. Observations A and B are trivial, they hold by definition of the translation function Enc. For the observation C, we can note, by the definition of Enc, that each action in the translated rule originated from P'_1 is local, hence all modifications are on the current node 1. This means that no updates on attributes and rule flags are made on other nodes i, with $i \in [2..n]$. The only rules updating external nodes are the *send* rules, with the action $\overline{msg} \leftarrow \mathsf{Enc}(e)$, which modifies a state in a node i such that $i \in [2..n]$. But, msg is an auxiliary resource introduced by the translation, it is not an attribute of any Γ_i, with $i \in [2..n]$, nor a rule flag of any i, with $i \in [2..n]$. Finally, for the observation D, we have that $\{\!|\Pi|\!\}(\Gamma'_i)$ substitutes the instances $\mathtt{this}.a$ with $\Gamma'_i(a) = v$ in Π, and lets untouched the instances a. Instead, $\mathsf{Ext}(\Pi)$ substitutes the instances of a not prefixed by $\mathtt{this}.$ (in Π) with $\overline{a}$ (and then apply the translation Enc). It is easy to note that in $\Gamma'_j \models \Pi'$ each original instance of $\mathtt{this}.a$ in Π now takes the value $\Gamma'_i(a)$ and each instance of a takes the value $\Gamma'_j(a)$. Similarly, in $\Sigma_j \models \{\!|\mathsf{Ext}(\mathsf{Enc}(\Pi))|\!\}\Sigma_i$ each original instance of $\mathtt{this}.a$ in Π takes the value $\Sigma_i(a)$ and each instance of a takes the value $\Sigma_j(a)$. But, by the observation A, we have that $\Gamma'_i(a) = \Sigma_i(a)$ and $\Gamma'_j(a) = \Sigma_j(a)$.

Lemma 8. *Consider an AbC component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ and an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$, such that $R_i = \mathsf{Enc}^i(P_i)$, for $i \in [1..n]$. Let $C' = \Gamma'_1 : P'_1 \parallel \ldots \parallel \Gamma'_n : P'_n$ be a residual of C such that $\mathsf{S} \succeq C'$ and S is firing for C'. Let $\Gamma : P$ be a component of C', namely $\Gamma : P = \Gamma'_k : P'_k$, for $k \in [1..n]$. Then, for all Γ', P' such that $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$, for some δ, there exists an AbU system $\mathsf{S}' = R_1, \iota_1\langle\Sigma'_1, \Theta'_1\rangle \parallel \ldots \parallel R_n, \iota_1\langle\Sigma'_n, \Theta'_n\rangle$ such that $\mathsf{S} \longrightarrow^* \mathsf{S}'$, $\mathsf{S}' \succeq C''$ and S' is firing for C'', with $C'' = C'[\Gamma : P \gg \Gamma' : P']$.*

Proof. The only rule applicable is (Comp), namely $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$ is obtained from $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$. The proof is by induction on the derivation tree for $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$. Depending on the last rule used in the derivation, we have the following cases.

Case (Zero). Then: $\delta = \overline{\mathtt{ff}}\langle 0\rangle$; $P' = P = 0$; and $\Gamma' = \Gamma$. Setting $\mathsf{S}' = \mathsf{S}$, we have the conclusion, since $C' = C$.

Case (BRD). Then: $\delta = \overline{\Pi}\langle v \rangle$, for some Π and v; $P = \langle e \mathbin{@} \Pi' \rangle.P''$, for some P'', Π' such that $\llparenthesis \Pi' \rrparenthesis(\Gamma) = \Pi$ and e such that $[\![e]\!](\Gamma) = v$; $P' = P''$; and $\Gamma' = \Gamma$. Let $P_h r_i = \mathsf{firing}(P)$ be the firing rule of P and $j = \mathsf{Next}(i)$. The translation of P generates the following AbU rule:

$$P_h r_i \gg P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T}, @(P_h r_i = \mathtt{T} \wedge \mathsf{Ext}(\Pi)) : \overline{msg} \leftarrow \mathsf{Enc}(e) \tag{A.1}$$

We assumed that $\Gamma : P$ is the AbC component $\Gamma'_k : P'_k$ of C', so we are on the node k of the AbU translation, i.e., $R_k, \iota_k\langle \Sigma_k, \Theta_k \rangle$. Since S is firing for C', we have that Θ_k contains the update $\mathsf{upd} = (P_h r_i, \mathtt{ff})(P_h r_j, \mathtt{tt})$, by definition of $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k)$. Since Rule (A.1) is not local we have that $\mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k) = \varnothing$. For the same reason, we have that $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = T$, with $T = (\mathsf{Ext}(\Pi)) : msg \leftarrow v$, given $[\![\mathsf{Enc}(e)]\!]\Sigma_k = v$ (justified by Proposition 7A). Furthermore, for all $w \in [1..n] \setminus \{k\}$ we have that Θ_w is a superset of $\{(msg, v) \mid \Sigma_w \models \mathsf{Ext}(\Pi)\}$. When the AbU semantics evolves with the rule (STEPL) (or, equivalently, with the symmetric rule (STEPR)) applied to the system S, performing an execution step (EXEC), the update upd is committed, obtaining $\Sigma'_k = \Sigma_k[\mathtt{ff}/P_h r_i\ \mathtt{tt}/P_h r_j]$. The discovery phase launched during (STEPL) computes $\Theta'_k = \Theta_k \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_k, \{P_h r_i, P_h r_j\}, \Sigma'_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_i, P_h r_j\}, \Sigma'_k)$. By definition of Enc and since $\mathsf{Next}(i) = j$, we have that $\mathsf{firing}(P'') = P_h r_j$. Hence, we have that Θ'_k is a superset of $\mathsf{DefUpds}(R_k, \{P_h r_j\}, \Sigma'_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_j\}, \Sigma'_k)$. The last two requirements of Definition 11 are satisfied by the definition of (STEPL), that guarantees to perform the discovery of $T' = \mathsf{ExtTasks}(R_k, \{P_h r_i, P_h r_j\}, \Sigma_k)$ by applying the rule (DISC) with T' on all nodes $w \in [1..n] \setminus \{k\}$. This means that $\mathsf{S}' = \mathsf{S}[R_k, \iota_k\langle \Sigma'_k, \Theta'_k \rangle \gg R_k, \iota_k\langle \Sigma_k, \Theta_k \rangle]$ is firing for C''. Furthermore, no attributes of Γ are modified (only auxiliary resources added by the translation are involved in the update). Hence, we also have that $\mathsf{S}' \succeq C''$, concluding the proof for this case.

Case (RCV). Then: $\delta = \Pi(v)$, for some Π such that $\Gamma \models \Pi$ and v; $P = (x \mid \Pi').P''$, for some P'' and Π' such that $\Gamma \models \Pi'[v/x]$; $P' = P''[v/x]$; and $\Gamma' = \Gamma$. Let $P_h r_i = \mathsf{firing}(P)$ be the firing rule of P and $j = \mathsf{Next}(i)$. The translation of P generates the following AbU rule:

$$P_h r_i \gg (P_h r_i = \mathtt{T} \wedge \mathsf{Repl}(\mathsf{Enc}(\Pi), x)) : x \leftarrow msg\ P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T} \tag{A.2}$$

We assumed that $\Gamma : P$ is the AbC component $\Gamma'_k : P'_k$ of C', so we are on the node k of the AbU translation, i.e., $R_k, \iota_k\langle \Sigma_k, \Theta_k \rangle$. Since S is firing for C', we have that Θ_k contains the update $\mathsf{upd} = (x, v)(P_h r_i, \mathtt{ff})(P_h r_j, \mathtt{tt})$, by definition of $\mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k)$, given $\Sigma_k(msg) = v$. Note that, the satisfiability of the condition $\mathsf{Repl}(\mathsf{Enc}(\Pi), x))$ is guaranteed by Proposition 7B. Since Rule (A.2) is local and it does not have default actions, we have that $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k) = \varnothing$ and $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = \varepsilon$. When the AbU semantics evolves with the rule (STEPL) (or, equivalently, with the symmetric rule (STEPR)) applied to the system S, performing an execution step (EXEC), the update upd is committed, obtaining $\Sigma'_k = \Sigma_k[v/x\ \mathtt{ff}/P_h r_i\ \mathtt{tt}/P_h r_j]$. The discovery phase launched during (STEPL) computes $\Theta'_k = \Theta_k \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_k, \{x, P_h r_i, P_h r_j\}, \Sigma'_k) \cup \mathsf{LocalUpds}(R_k, \{x, P_h r_i, P_h r_j\}, \Sigma'_k)$. By definition of Enc and since $\mathsf{Next}(i) = j$, we have that $\mathsf{firing}(P'') = P_h r_j$. Hence, we have that Θ'_k is a superset of $\mathsf{DefUpds}(R_k, \{P_h r_j\}, \Sigma'_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_j\}, \Sigma'_k)$. The last two requirements of Definition 11 are satisfied by the definition of (STEPL), that guarantees to perform the discovery of $T' = \mathsf{ExtTasks}(R_k, \{x, P_h r_i, P_h r_j\}, \Sigma_k)$ by applying the rule (DISC) with T' on all nodes $w \in [1..n] \setminus \{k\}$. This means that $\mathsf{S}' = \mathsf{S}[R_k, \iota_k\langle \Sigma'_k, \Theta'_k \rangle \gg R_k, \iota_k\langle \Sigma_k, \Theta_k \rangle]$ is firing for C'' (the substitution $[v/x]$ in P'' is recorded by updating the resource x with the value v). Furthermore, no attributes of Γ are modified (only auxiliary resources added by the translation are involved in the update). Hence, we also have that $\mathsf{S}' \succeq C''$, concluding the proof for this case.

Case (UPD). Then: $P = [a := e]P''$, for some P'' and e; $\Gamma' = \Gamma[v/a]'$, given $[\![e]\!](\Gamma) = v$ and $\Gamma[v/a] : P'' \xrightarrow{\delta} \Gamma[v/a]' : P'$. Let $P_h r_i = \mathsf{firing}(P)$ be the firing rule of P and $j = \mathsf{Next}(i)$. The translation of P generates the following AbU rule:

$$P_h r_i \gg (P_h r_i = \mathtt{T}) : a \leftarrow \mathsf{Enc}(e)\ P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T} \tag{A.3}$$

We assumed that $\Gamma : P$ is the AbC component $\Gamma'_k : P'_k$ of C', so we are on the node k of the AbU translation, i.e., $R_k, \iota_k\langle \Sigma_k, \Theta_k \rangle$. Since S is firing for C', we have that Θ_k contains the update $\mathsf{upd} = (a, v)(P_h r_i, \mathtt{ff})(P_h r_j, \mathtt{tt})$, by definition of $\mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k)$. Here, the value v in (a, v) is justified by Proposition 7A. Since Rule (A.3) is local and it does not have default actions, we have that $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k) = \varnothing$ and $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = \varepsilon$. When the AbU semantics evolves with the rule (STEPL) (or, equivalently, with the symmetric rule (STEPR)) applied to the system S, performing an execution step (EXEC), the update upd is committed, obtaining $\Sigma''_k = \Sigma_k[v/a\ \mathtt{ff}/P_h r_i\ \mathtt{tt}/P_h r_j]$. The discovery phase launched during (STEPL) computes $\Theta''_k = \Theta_k \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_k, \{a, P_h r_i, P_h r_j\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{a, P_h r_i, P_h r_j\}, \Sigma''_k)$. By definition of Enc and since $\mathsf{Next}(i) = j$, we have that $\mathsf{firing}(P'') = P_h r_j$. Hence, we have that Θ''_k is a superset of $\mathsf{DefUpds}(R_k, \{P_h r_j\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_j\}, \Sigma''_k)$. The last two requirements of Definition 11 are satisfied by the definition of (STEPL), that guarantees to perform the discovery of $T = \mathsf{ExtTasks}(R_k, \{a, P_h r_i, P_h r_j\}, \Sigma_k)$ by applying the rule (DISC) with T on all nodes $w \in [1..n] \setminus \{k\}$. This means that $\mathsf{S}'' = \mathsf{S}[R_k, \iota_k\langle \Sigma''_k, \Theta''_k \rangle \gg R_k, \iota_k\langle \Sigma_k, \Theta_k \rangle]$ is firing for $C''' = C'[\Gamma : P \gg \Gamma[v/a] : P'']$. Furthermore, the only attribute modified in Γ is a, which is updated with the value v by the AbU semantics in Σ''_k. Hence, we also have that $\mathsf{S}'' \succeq C'''$. Since the derivation tree for $\Gamma[v/a] : P'' \xrightarrow{\delta} \Gamma[v/a]' : P'$ is smaller than the derivation tree for $\Gamma : P \xrightarrow{\delta} \Gamma[v/a]' : P'$, by inductive hypothesis we have that there exists S' such

that $\mathsf{S}' \succeq C''$ and S' is firing for C'', noting that $C'' = C'''[\Gamma[v/a] : P'' \gg \Gamma[v/a]' : P']$. This concludes the proof for the case.

Case (Aware)**.** Then: $P = [\Pi]P''$, for some P'' and Π such that $\Gamma \models \Pi$, given $\Gamma : P'' \xrightarrow{\delta} \Gamma' : P'$. Let $P_h r_i = \mathsf{firing}(P)$ be the firing rule of P and $j = \mathsf{Next}(i)$. The translation of P generates the following AbU rule:

$$P_h r_i \ \mathsf{Vars}(\Pi) \gg (P_h r_i = \mathtt{T} \wedge \mathsf{Enc}(\Pi)) : P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T} \tag{A.4}$$

We assumed that $\Gamma : P$ is the AbC component $\Gamma'_k : P'_k$ of C', so we are on the node k of the AbU translation, i.e., $R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle$. Since S is firing for C', we have that Θ_k contains the update $\mathsf{upd} = (P_h r_i, \mathtt{ff})(P_h r_j, \mathtt{tt})$, by definition of $\mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k)$. Since Rule (A.4) is local and it does not have default actions, we have that $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k) = \varnothing$ and $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = \varepsilon$. When the AbU semantics evolves with the rule (StepL) (or, equivalently, with the symmetric rule (StepR)) applied to the system S, performing an execution step (Exec), the update upd is committed, obtaining $\Sigma''_k = \Sigma_k[\mathtt{ff}/P_h r_i\ \mathtt{tt}/P_h r_j]$. The discovery phase launched during (StepL) computes $\Theta''_k = \Theta_k \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_k, \{P_h r_i, P_h r_j\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_i, P_h r_j\}, \Sigma''_k)$. By definition of Enc, and $\mathsf{Next}(i) = j$, we have that $\mathsf{firing}(P'') = P_h r_j$. Hence, we have that Θ''_k is a superset of $\mathsf{DefUpds}(R_k, \{P_h r_j\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_j\}, \Sigma''_k)$. The last two requirements of Definition 11 are satisfied by the definition of (StepL), guaranteeing to perform the discovery of $T = \mathsf{ExtTasks}(R_k, \{P_h r_i, P_h r_j\}, \Sigma_k)$ by applying the rule (Disc) with T on all nodes $w \in [1..n] \setminus \{k\}$. This means that $\mathsf{S}'' = \mathsf{S}[R_k, \iota_k\langle\Sigma''_k, \Theta''_k\rangle \gg R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle]$ is firing for $C''' = C'[\Gamma : P \gg \Gamma : P'']$. Furthermore, no attributes of Γ are modified (only auxiliary resources added by the translation are involved in the update). Hence, we also have that $\mathsf{S}'' \succeq C'''$. Since the derivation tree for $\Gamma : P'' \xrightarrow{\delta} \Gamma' : P'$ is smaller than the derivation tree for $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$, by inductive hypothesis we have that there exists S' such that $\mathsf{S}' \succeq C''$ and S' is firing for C'', noting that $C'' = C'''[\Gamma : P'' \gg \Gamma' : P']$. This concludes the proof for the case.

Case (Sum)**.** Then: $P = P_a + P_b$, for some P_a and P_b, given $\Gamma : P_a \xrightarrow{\delta} \Gamma' : P'$. The symmetric case is analogous. Let $P_h r_i = \mathsf{firing}(P)$ be the firing rule of P, $j = \mathsf{Next}(i)$ and $w = \mathsf{Next}(j)$. The translation of P generates the following AbU rules:

$$P_h r_i \gg (P_h r_i = \mathtt{T}) : P_h r_i \leftarrow \mathtt{F}\ P_h r_j \leftarrow \mathtt{T} \tag{A.5}$$

$$P_h r_i \gg (P_h r_i = \mathtt{T}) : P_h r_i \leftarrow \mathtt{F}\ P_h r_k \leftarrow \mathtt{T} \tag{A.6}$$

We assumed that $\Gamma : P$ is the AbC component $\Gamma'_k : P'_k$ of C', so we are on the node k of the AbU translation, i.e., $R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle$. Since S is firing for C', we have that Θ_k contains the update $\mathsf{upd} = (P_h r_i, \mathtt{ff})(P_h r_j, \mathtt{tt})$, by definition of $\mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k)$, and assuming that AbC and AbU schedulers make the same decisions (i.e., P_a and Rule (A.5) are chosen). Since Rule (A.5) and Rule (A.6) are local and they do not have default actions, we have that $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k) = \varnothing$ and $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = \varepsilon$. When the AbU semantics evolves with the rule (StepL) (or, equivalently, with the symmetric rule (StepR)) applied to the system S, performing an execution step (Exec), the update upd is committed, obtaining $\Sigma''_k = \Sigma_k[\mathtt{ff}/P_h r_i\ \mathtt{tt}/P_h r_j]$. The discovery phase launched during (StepL) computes $\Theta''_k = \Theta_k \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_k, \{P_h r_i, P_h r_j\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_i, P_h r_j\}, \Sigma''_k)$. By definition of Enc and since $\mathsf{Next}(i) = j$, we have that $\mathsf{firing}(P'') = P_h r_j$. Hence, we have that Θ''_k is a superset of $\mathsf{DefUpds}(R_k, \{P_h r_j\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_j\}, \Sigma''_k)$. The last two requirements of Definition 11 are satisfied by definition of the semantic rule (StepL), that guarantees to perform the discovery of $T = \mathsf{ExtTasks}(R_k, \{P_h r_i, P_h r_j\}, \Sigma_k)$ by applying the rule (Disc) with T on all nodes $w \in [1..n] \setminus \{k\}$. This means that $\mathsf{S}'' = \mathsf{S}[R_k, \iota_k\langle\Sigma''_k, \Theta''_k\rangle \gg R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle]$ is firing for $C''' = C'[\Gamma : P \gg \Gamma : P_a]$. Furthermore, no attributes of Γ are modified (only auxiliary resources added by the translation are involved in the update). Hence, we also have that $\mathsf{S}'' \succeq C'''$. Since the derivation tree for $\Gamma : P_a \xrightarrow{\delta} \Gamma' : P'$ is smaller than the derivation tree for $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$, by inductive hypothesis we have that there exists S' such that $\mathsf{S}' \succeq C''$ and S' is firing for C'', noting that $C'' = C'''[\Gamma : P_a \gg \Gamma' : P']$. This concludes the proof for the case (the symmetric counterpart where the scheduler chooses P_b is analogous).

Case (Rec)**.** Then: $P = \mathsf{K}$, given $\mathsf{K} \triangleq P_w$, with $w \in [1..n]$, and $\Gamma : P_w \xrightarrow{\delta} \Gamma' : P'$. Let $P_h r_i = \mathsf{firing}(P)$ be the firing rule of P. The translation of P generates the following AbU rule:

$$P_h r_i \gg (P_h r_i = \mathtt{T}) : P_h r_i \leftarrow \mathtt{F}\ P_k r_0 \leftarrow \mathtt{T} \tag{A.7}$$

We assumed that $\Gamma : P$ is the AbC component $\Gamma'_k : P'_k$ of C', so we are on the node k of the AbU translation, i.e., $R_k, \iota_k\langle\Sigma_k, \Theta_k\rangle$. Since S is firing for C', we have that Θ_k contains the update $\mathsf{upd} = (P_h r_i, \mathtt{ff})(P_w r_0, \mathtt{tt})$, by definition of $\mathsf{LocalUpds}(R_k, \{P_h r_i\}, \Sigma_k)$. Since Rule (A.7) is local and it does not have default actions, we have that $\mathsf{DefUpds}(R_k, \{P_h r_i\}, \Sigma_k) = \varnothing$ and $\mathsf{ExtTasks}(R_k, \{P_h r_i\}, \Sigma_k) = \varepsilon$. When the AbU semantics evolves with the rule (StepL) (or, equivalently, with the symmetric rule (StepR)) applied to the system S, performing an execution step (Exec), the update upd is committed, obtaining $\Sigma''_k = \Sigma_k[\mathtt{ff}/P_h r_i\ \mathtt{tt}/P_w r_0]$. The discovery phase launched during (StepL) computes $\Theta''_k = \Theta_k \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_k, \{P_h r_i, P_w r_0\}, \Sigma''_k) \cup \mathsf{LocalUpds}(R_k, \{P_h r_i, P_w r_0\}, \Sigma''_k)$. By definition of Enc, we have that $\mathsf{firing}(P_w) = P_w r_0$. Hence, we have that Θ''_k is a superset of $\mathsf{DefUpds}(R_k, \{P_w r_0\}, \Sigma''_k) \cup$

$\mathsf{LocalUpds}(R_k, \{P_w r_0\}, \Sigma''_k)$. The last two requirements of Definition 11 are satisfied by the definition of (StepL), that guarantees to perform the discovery of $T = \mathsf{ExtTasks}(R_k, \{P_h r_i, P_w r_0\}, \Sigma_k)$ by applying the rule (Disc) with T on all nodes $w \in [1..n] \setminus \{k\}$. This means that $\mathsf{S}'' = \mathsf{S}[R_k, \iota_k\langle \Sigma''_k, \Theta''_k\rangle \gg R_k, \iota_k\langle \Sigma_k, \Theta_k\rangle]$ is firing for $C''' = C'[\Gamma : P \gg \Gamma : P_w]$. Furthermore, no attributes of Γ are modified (only auxiliary resources added by the translation are involved in the update). Hence, we also have that $\mathsf{S}'' \succeq C'''$. Since the derivation tree for $\Gamma : P_w \xrightarrow{\delta} \Gamma' : P'$ is smaller than the derivation tree for $\Gamma : P \xrightarrow{\delta} \Gamma' : P'$, by inductive hypothesis we have that there exists S' such that $\mathsf{S}' \succeq C''$ and S' is firing for C'', noting that $C'' = C'''[\Gamma : P_w \gg \Gamma' : P']$. This concludes the proof for the case.

Lemma 9. *Consider an AbC component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ and an AbU system $\mathsf{S} = R_1, \iota_1\langle \Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle \Sigma_n, \Theta_n\rangle$, such that $R_i = \mathsf{Enc}^i(P_i)$, for $i \in [1..n]$. Let C'' be a residual of C. If $\mathsf{S} \succeq C''$ and S is firing for C'', then for all C' such that $C'' \xrightarrow{\Pi(\nu)} C'$, for some Π and ν, there exists an AbU system S' such that $\mathsf{S} \longrightarrow^* \mathsf{S}'$, $\mathsf{S}' \succeq C'$ and S' is firing for C'.*

Proof. Since C'' is a residual of C, it is of the form $\Gamma''_1 : P''_1 \parallel \ldots \parallel \Gamma''_n : P''_n$, with $\Gamma''_i : P''_i$ residuals of $\Gamma_i : P_i$, for $i \in [1..n]$. Then, $C' = \Gamma'_1 : P'_1 \parallel \ldots \parallel \Gamma'_n : P'_n$, given $\Gamma''_i : P''_i \xrightarrow{\Pi(\nu)} \Gamma'_i : P'_i$, for each $i \in [1..n]$. By Lemma 8, we have that there exists S^1 such that $\mathsf{S} \longrightarrow^* \mathsf{S}^1$, $\mathsf{S}^1 \succeq C^1$ and S^1 is firing for C^1, where $C^1 = C''[\Gamma''_1 : P''_1 \gg \Gamma'_1 : P'_1]$. Applying again Lemma 8, we have that there exists S^2 such that $\mathsf{S}^1 \longrightarrow^* \mathsf{S}^2$, $\mathsf{S}^2 \succeq C^2$ and S^2 is firing for C^2, where $C^2 = C^1[\Gamma''_2 : P''_2 \gg \Gamma'_2 : P'_2]$. We can repeat this reasoning $n-2$ times and prove that there exists S^n such that $\mathsf{S}^{n-1} \longrightarrow^* \mathsf{S}^n$, $\mathsf{S}^n \succeq C^n$ and S^n is firing for C^n, where $C^n = C^{n-1}[\Gamma''_n : P''_n \gg \Gamma'_n : P'_n]$. Note that, for each step i, with $i \in [1..n]$, we have that no modifications of the state of the nodes j, with $j \in [1..n] \setminus \{i\}$, are made on attributes and rule flags (by Proposition 7C). At this point we can note that $C^n = C'$, hence we have that $\mathsf{S}' \succeq C'$ and S' is firing for C', taking $\mathsf{S}' = \mathsf{S}^n$.

Lemma 10. *Consider an AbC component $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$ and an AbU system $\mathsf{S} = R_1, \iota_1\langle \Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle \Sigma_n, \Theta_n\rangle$, such that $R_i = \mathsf{Enc}^i(P_i)$, for $i \in [1..n]$. Let C'' be a residual of C. If $\mathsf{S} \succeq C''$ and S is firing for C'', then for all C' such that $C'' \xrightarrow{\lambda} C'$ there exists an AbU system S' such that $\mathsf{S} \longrightarrow^* \mathsf{S}'$, $\mathsf{S}' \succeq C'$ and S' is firing for C'.*

Proof. Since C'' is a residual of C, it is of the form $\Gamma''_1 : P''_1 \parallel \ldots \parallel \Gamma''_n : P''_n$, with $\Gamma''_i : P''_i$ residuals of $\Gamma_i : P_i$, for $i \in [1..n]$. Let $C_r = \Gamma''_2 : P''_2 \parallel \ldots \parallel \Gamma''_n : P''_n$, i.e., $C'' = \Gamma''_1 : P''_1 \parallel C_r$. The proof is by case analysis on λ.

Case $\lambda = \tau$. Then, $C' = \Gamma'_1 : P'_1 \parallel C_r$, given $\Gamma''_1 : P''_1 \xrightarrow{\overline{\Pi}\langle\nu\rangle} \Gamma'_1 : P'_1$, for some Π and ν, and $C_r \xrightarrow{\Pi(\nu)}$, i.e., $\Gamma''_i : P''_i \xrightarrow{\Pi(\nu)}$ for each $i \in [2..n]$. By Lemma 8, we have that there exists S' such that $\mathsf{S} \longrightarrow^* \mathsf{S}'$, $\mathsf{S}' \succeq C^1$ and S' is firing for C^1, where $C^1 = C''[\Gamma''_1 : P''_1 \gg \Gamma'_1 : P'_1]$. Since no modifications of the state of the node i, with $i \in [2..n]$, are made on attributes and rule flags (by Proposition 7C), we have that $\mathsf{S}' \succeq C_r$ and S' is firing for C_r. Since C_r is unmodified, we can conclude that $\mathsf{S}' \succeq C'$ and S' is firing for C'.

Case $\lambda = \overline{\Pi}\langle\nu\rangle$. Then, $C' = \Gamma'_1 : P'_1 \parallel C'_r$, given $\Gamma''_1 : P''_1 \xrightarrow{\overline{\Pi}\langle\nu\rangle} \Gamma'_1 : P'_1$, for some Π and ν, $C'_r = \Gamma'_2 : P'_2 \parallel \ldots \parallel \Gamma'_n : P'_n$ and $C_r \xrightarrow{\Pi(\nu)} C'_r$, i.e., $\Gamma''_i : P''_i \xrightarrow{\Pi(\nu)} \Gamma'_i : P'_i$ for each $i \in [2..n]$. By Lemma 8, there exists S^1 such that $\mathsf{S} \longrightarrow^* \mathsf{S}^1$, $\mathsf{S}^1 \succeq C^1$ and S^1 is firing for C^1, where $C^1 = C''[\Gamma''_1 : P''_1 \gg \Gamma'_1 : P'_1]$. Since no modifications of the state of the nodes i, with $i \in [2..n]$, are made on attributes and rule flags (by Proposition 7C), we have that $\mathsf{S}^1 \succeq C_r$ and S^1 is firing for C_r. Then, applying Lemma 9 on C_r, we have that there exists S' such that $\mathsf{S}' \succeq C'_r$ and S' is firing for C'_r. So, we can conclude that $\mathsf{S}' \succeq C'$ and S' is firing for C', noting that $C' = C^1[C_r \gg C'_r]$.

Case $\lambda = \Pi(\nu)$. Then, $C' = \Gamma'_1 : P'_1 \parallel C'_r$, given $\Gamma''_1 : P''_1 \xrightarrow{\Pi(\nu)} \Gamma'_1 : P'_1$, for some Π and ν, $C'_r = \Gamma'_2 : P'_2 \parallel \ldots \parallel \Gamma'_n : P'_n$ and $C_r \xrightarrow{\Pi(\nu)} C'_r$, i.e., $\Gamma''_i : P''_i \xrightarrow{\Pi(\nu)} \Gamma'_i : P'_i$ for each $i \in [1..n]$. By Lemma 8, there exists S^1 such that $\mathsf{S} \longrightarrow^* \mathsf{S}^1$, $\mathsf{S}^1 \succeq C^1$ and S^1 is firing for C^1, where $C^1 = C''[\Gamma''_1 : P''_1 \gg \Gamma'_1 : P'_1]$. Since no modifications of the state of the nodes i, with $i \in [2..n]$, are made on attributes and rule flags (by Proposition 7C), we have that $\mathsf{S}^1 \succeq C_r$ and S^1 is firing for C_r. Then, applying Lemma 9 on C_r, we have that there exists S' such that $\mathsf{S}' \succeq C'_r$ and S' is firing for C'_r. So, we can conclude that $\mathsf{S}' \succeq C'$ and S' is firing for C', noting that $C' = C^1[C_r \gg C'_r]$.

Lemma 11. *Consider an AbC component C and its corresponding AbU encoding $\mathsf{S} = \mathsf{Enc}(C)$. Then there exists S' such that $\mathsf{S} \longrightarrow^* \mathsf{S}'$, S' is firing for C and $\mathsf{S}' \succeq C$.*

Proof. Suppose that C is composed by n components, i.e., $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$. Consider its AbU translation $\mathsf{S} = R_1, \iota_1\langle \Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle \Sigma_n, \Theta_n\rangle = \mathsf{Enc}(C)$. By definition, we have that $R_i = \mathsf{Enc}^i(P_i)$, for $i \in [1..n]$. The translation yields n (one for each component) initial rule flags $P_1 r_0, \ldots, P_n r_0$, initially set to ff (i.e., $\Sigma_i(P_i r_0) = \mathsf{ff}$, for $i \in [1..n]$). Suppose to perform an input (Input) on the node 1, setting $P_1 r_0$ to tt, namely $\Sigma'_1 = \Sigma_1[\mathsf{tt}/P_1 r_0]$ and $\Theta_1 = \mathsf{DefUpds}(R_1, \{P_1 r_0\}, \Sigma'_1) \cup \mathsf{LocalUpds}(R_1, \{P_1 r_0\}, \Sigma'_1)$. The input is wrapped into the AbU rule (StepL) (or, equivalently, into the symmetric rule (StepR)), that guarantees to perform the discovery of $T^1 = \mathsf{task}_1 \ldots \mathsf{task}_w = \mathsf{ExtTasks}(R_1, \{P_1 r_0\}, \Sigma'_1)$, by applying the rule (Disc) with T^1 on all nodes $j \in [2..n]$. This potentially initialize the pools of other nodes, namely for all $j \in [2..n]$ we have

that $\Theta_j = \{[\![\mathsf{act}]\!]\Sigma_j \mid \exists l \in [1..w] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma_j \models \varphi\}$. Hence we have $\mathsf{S} \xrightarrow{\mathsf{upd}^1 \blacktriangleright T^1} \mathsf{S}_1$, where $\mathsf{upd}^1 = (P_1 r_0, \mathsf{tt})$ and $\mathsf{S}_1 = R_1, \iota_1\langle\Sigma_1', \Theta_1\rangle \parallel R_2, \iota_k\langle\Sigma_2, \Theta_2\rangle \parallel \ldots \parallel R_n, \mathsf{id}\langle\Sigma_n, \Theta_n\rangle$. Now, we can perform another input (Input) on the node 2, setting $P_2 r_0$ to tt, namely $\Sigma_2' = \Sigma_2[\mathsf{tt}/P_2 r_0]$ and $\Theta_2' = \Theta_2 \cup \mathsf{DefUpds}(R_2, \{P_2 r_0\}, \Sigma_2') \cup \mathsf{LocalUpds}(R_2, \{P_2 r_0\}, \Sigma_2')$. The input is wrapped into the AbU rule (StepL) (or, equivalently, into the symmetric rule (StepR)), that guarantees to perform the discovery of $T^2 = \mathsf{task}_1 \ldots \mathsf{task}_m = \mathsf{ExtTasks}(R_2, \{P_2 r_0\}, \Sigma_2')$, by applying the rule (Disc) with T^2 on all nodes $j \in [1..n] \setminus \{2\}$. This potentially enlarges the pools of other nodes, namely for all $j \in [1..n] \setminus \{2\}$ we have that $\Theta_j' = \Theta_j \cup \{[\![\mathsf{act}]\!]\Sigma_j \mid \exists l \in [1..m] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma_j \models \varphi\}$. Hence we have $\mathsf{S}^1 \xrightarrow{\mathsf{upd}^2 \blacktriangleright T^2} \mathsf{S}^2$, where $\mathsf{upd}^2 = (P_2 r_0, \mathsf{tt})$ and $\mathsf{S}^2 = R_1, \iota_1\langle\Sigma_1', \Theta_1'\rangle \parallel R_2, \iota_2\langle\Sigma_2', \Theta_2'\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n'\rangle$. We can repeat this reasoning $n-2$ times, with $n-2$ subsequent input steps, obtaining the system S^n such that $\mathsf{S}^{n-1} \xrightarrow{\mathsf{upd}^n \blacktriangleright T^n} \mathsf{S}^n$. Since we are not performing execution steps (Exec), the pools of the nodes can only be enlarged. Hence, it is easy to note that all requirements of Definition 11 are satisfied, implying that S^n is firing for C. Furthermore, no attributes of C are modified, so $\mathsf{S}^n \succeq C$. Finally, setting $\mathsf{S}' = \mathsf{S}^n$ and $\longrightarrow\!\blacktriangleright^* = \xrightarrow{\mathsf{upd}^1 \blacktriangleright T^1} \ldots \xrightarrow{\mathsf{upd}^n \blacktriangleright T^n}$, we can conclude that that $\mathsf{S} \longrightarrow\!\blacktriangleright^* \mathsf{S}'$, S' is firing for C and $\mathsf{S}' \succeq C$.

Theorem 4. *(AbC to AbU correctness).*

Proof. Suppose that C is composed by n components, i.e., $C = \Gamma_1 : P_1 \parallel \ldots \parallel \Gamma_n : P_n$. Consider its AbU translation $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle = \mathsf{Enc}(C)$. By definition, we have that $R_i = \mathsf{Enc}^i(P_i)$, for $i \in [1..n]$. By Lemma 11, we have that there exists S'' such that $\mathsf{S} \longrightarrow\!\blacktriangleright^* \mathsf{S}''$, S'' is firing for C and $\mathsf{S}'' \succeq C$. This, in particular, means that each P_i of C is ready to execute. Now, suppose that $C \longrightarrow^* C'$ comprises m semantic steps, namely $C \xrightarrow{\lambda_1} C^1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_m} C'$, for some component labels $\lambda_1, \lambda_2, \ldots, \lambda_m$. Applying Lemma 10, we have that there exists S^1 such that $\mathsf{S}'' \longrightarrow\!\blacktriangleright^* \mathsf{S}^1$, $\mathsf{S}^1 \succeq C^1$ and S^1 is firing for C^1. We can repeat this reasoning $m-1$ times and prove that there exists S^m such that $\mathsf{S}^{m-1} \longrightarrow\!\blacktriangleright^* \mathsf{S}^m$, $\mathsf{S}^m \succeq C^m$ and S^m is firing for C^m (which coincides with C'). By composition of all (StepL) (or (StepR)) rules (given by multiple application of Lemma 10) we can conclude that $\mathsf{S}'' \longrightarrow\!\blacktriangleright^* \mathsf{S}'$ such that $\mathsf{S}' \succeq C'$, for $\mathsf{S}' = \mathsf{S}^m$.

A.2. Proofs of Section 6

Proof of Theorem 5. In order to prove the theorem, we need some supporting lemmas and some preliminary notions. If the ECA dependency graph of S is acyclic then we can define a topological sort of its vertices, i.e., a linear ordering of the vertices such that for every arc (x, y) of the graph, x comes before y in the ordering. Given $|V| = n$, we denote with $v_1 \ldots v_n$ such topological sort, and with $\mathsf{indexOf}(x)$ the index in the ordering that correspond to the resource x, namely $\mathsf{indexOf}(x) = i$, with $i \in [1..n]$, iff $v_i = x$.

When we modify a resource, by means of an (Input) or (Exec) rule, we may trigger some AbU rules of the system, potentially leading to new updates to add to the pools. The ECA dependency graph provides an over-approximation of the resources involved in such updates.

Definition 12 *(Potential updates).* Consider an AbU system with acyclic ECA dependency graph. Let $v_1 \ldots v_n$ a topological sort of its nodes. The set of *potential updates* originated by a resource x consists of all resources that may be updated when x is modified, and it is defined as:

$$\mathsf{mayBeUpdated}(x) \triangleq \bigcup_{j \in [\mathsf{indexOf}(x)+1..n]} \{x' \in \mathbb{X} \mid \mathsf{indexOf}(x') = j\}$$

It is easy to note that, $\mathsf{mayBeUpdated}(x)$ decreases in the number of elements as $\mathsf{indexOf}(x)$ increases. In particular, $x \notin \mathsf{mayBeUpdated}(x)$ and $\mathsf{mayBeUpdated}(x) = \varnothing$ when $\mathsf{indexOf}(x) = n$. Similarly, we can define the updates that are actually originated when some resources are modified at runtime.

Definition 13 *(Discovered updates).* Given the systems $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ and $\mathsf{S}' = R_1, \iota_1\langle\Sigma_1', \Theta_1'\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n', \Theta_n'\rangle$ such that $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$ or $\mathsf{S} \xrightarrow{\mathsf{upd} \triangleright T} \mathsf{S}'$, we define the *discovered updates* $\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}')$ as the new updates inserted in the pools, given by the actuation of the update upd. Formally:

$$\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}') \triangleq \bigcup_{i \in [1..n]} \Theta_i' \setminus \Theta_i$$

Given an update $\mathsf{upd} = (x_1, v_1) \ldots (x_n, v_n)$, we define its resources as $\mathsf{Vars}(\mathsf{upd}) \triangleq \{x_1, \ldots, x_n\}$. For a pool Θ (or a generic set of updates), its resources are defined as $\mathsf{Vars}(\Theta) \triangleq \bigcup_{\mathsf{upd} \in \Theta} \mathsf{Vars}(\mathsf{upd})$. Similarly, given an AbU rule $\mathsf{rule} = \mathsf{evt} \gg \mathsf{act}, \mathsf{task}$ we define its resources as $\mathsf{Vars}(\mathsf{rule}) \triangleq \mathsf{Vars}(\mathsf{act}) \cup \mathsf{Vars}(\mathsf{task})$, where $\mathsf{Vars}(\mathsf{act})$ for an action act, is the set of resources in the left-hand side of the assignments in act and $\mathsf{Vars}(\mathsf{task})$, for a task $\mathsf{task} = \mathsf{cnd} : \mathsf{act}$, is $\mathsf{Vars}(\mathsf{act})$. Given a set of rules X, its resources are defined as $\mathsf{Vars}(X) \triangleq \bigcup_{\mathsf{rule} \in X} \mathsf{Vars}(\mathsf{rule})$.

Given list of AbU rules $R = \mathsf{rule}_1 \dots \mathsf{rule}_n$ and a set of resources X, we define the *rules that may modify* the resources in X as: $\mathsf{mayModifyRules}(R, X) \triangleq \{\mathsf{rule}_i \mid i \in [1..n] \wedge \mathsf{Vars}(\mathsf{rule}_i) \cap X \neq \varnothing\}$.

Lemma 12. *Let* $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ *be an AbU system with acyclic ECA dependency graph. Consider a list of AbU rule* R_i*, with* $i \in [1..n]$*, and a set of resources* X*, then we have:*

$$\mathsf{Vars}(\mathsf{Active}(R_i, X)) \subseteq \bigcup_{x \in X} \mathsf{mayBeUpdated}(x)$$

Proof. By definition, $\mathsf{Vars}(\mathsf{rule}) = \mathsf{Out}(\mathsf{rule})$ and $\mathsf{rule} \in \mathsf{Vars}(\mathsf{Active}(R_i, X))$ iff $\mathsf{In}(\mathsf{rule}) \cap X$, for any AbU rule rule. By definition of the ECA dependency graph, for any $\mathsf{rule} \in \mathsf{Vars}(\mathsf{Active}(R_i, X))$ we have that for any $x \in \mathsf{In}(\mathsf{rule})$ there exists $y \in \mathsf{Out}(\mathsf{rule})$ such that (x, y) is an arc of the ECA dependency graph. This means that for any $y \in \mathsf{Vars}(\mathsf{Active}(R_i, X))$ there exists $x \in X$ such that (x, y) is an arc of the ECA dependency graph. Since the graph is acyclic, we have that y follows x in the topological sort of the ECA dependency graph, hence $y \in \mathsf{mayBeUpdated}(x)$. This applies for any $y \in \mathsf{Vars}(\mathsf{Active}(R_i, X))$ and $x \in X$, hence $\mathsf{Vars}(\mathsf{Active}(R_i, X)) \subseteq \bigcup_{x \in X} \mathsf{mayBeUpdated}(x)$.

Lemma 13. *Let* S *be an AbU system with acyclic ECA dependency graph. Let* $v_1 \dots v_n$ *be a topological sort of the vertices of the graph. If* $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$ *or* $\mathsf{S} \xrightarrow{\mathsf{upd} \triangleright T} \mathsf{S}'$ *then:*

$$\mathsf{Vars}(\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}')) \subseteq \bigcup_{x \in \mathsf{Vars}(\mathsf{upd})} \mathsf{mayBeUpdated}(x)$$

Proof. Consider the case $\mathsf{S} \xrightarrow{\mathsf{upd} \triangleright T} \mathsf{S}'$, the other case is analogous. The only difference is that when the new pools are computed we do not have to remove upd from the pools, since it already not present. Suppose $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$, $\mathsf{S}' = R_1, \iota_1\langle\Sigma_1', \Theta_1'\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n', \Theta_n'\rangle$, $T = \mathsf{task}_1 \dots \mathsf{task}_m$ and $\mathsf{upd} = (x_1, v_1) \dots (x_k, v_k)$. By definition of the AbU semantics, we have that one node, say $R_i, \iota_i\langle\Sigma_i, \Theta_i\rangle$ with $i \in [1..n]$, performs an (Exec) step while the other nodes $R_j, \iota_j\langle\Sigma_j, \Theta_j\rangle$ with $j \in [1..n] \setminus \{i\}$, perform a discovery step (Disc). By definition of the semantic rule (Exec), we have that $\Theta_i' = \Theta_i \setminus \{\mathsf{upd}\} \cup \mathsf{DefUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i) \cup \mathsf{LocalUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i)$. By definition of the semantic rule (Disc), for each $j \in [1..n] \setminus \{i\}$ we have $\Theta_j' = \Theta_j \cup \{[\![\mathsf{act}]\!]\Sigma \mid \exists l \in [1..m] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma \models \varphi\}$. Hence, $\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}')$ is the set:

$$\mathsf{DefUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i) \cup \mathsf{LocalUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i) \cup$$
$$\bigcup_{j \in [1..n] \setminus \{i\}} \left\{ [\![\mathsf{act}]\!]\Sigma_j \;\middle|\; \exists l \in [1..m] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma_j \models \varphi \right\}$$

Note that, $\bigcup_{h \in [1..m]} \mathsf{Vars}(\mathsf{task}_h)$ is a subset of $\mathsf{Vars}(\mathsf{Active}(R_i, \{x_1, \dots, x_k\}))$, namely the updates that can be discovered in the nodes j, with $j \in [1..n] \setminus \{i\}$ are originated by the active AbU rules of the node i. Moreover, by definition of Vars, $\mathsf{Vars}(\bigcup_{j \in [1..n] \setminus \{i\}} \{[\![\mathsf{act}]\!]\Sigma_j \mid \exists l \in [1..m] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma_j \models \varphi\}) \subseteq \bigcup_{h \in [1..m]} \mathsf{Vars}(\mathsf{task}_h)$. Since $\{x_1, \dots, x_k\} = \mathsf{Vars}(\mathsf{upd})$, we can apply Lemma 12 and conclude that $\mathsf{Vars}(\mathsf{Active}(R_i, \{x_1, \dots, x_k\}))$ is contained in $\bigcup_{x \in \mathsf{Vars}(\mathsf{upd})} \mathsf{mayBeUpdated}(x)$. Hence we have:

$$\mathsf{Vars}(\textstyle\bigcup_{j \in [1..n] \setminus \{i\}} \{[\![\mathsf{act}]\!]\Sigma_j \mid \exists l \in [1..m] \,.\, \mathsf{task}_l = \varphi : \mathsf{act} \wedge \Sigma_j \models \varphi\}) \subseteq \textstyle\bigcup_{x \in \mathsf{Vars}(\mathsf{upd})} \mathsf{mayBeUpdated}(x)$$

Similarly, by definition of DefUpds and LocalUpds, we have that $\mathsf{Vars}(\mathsf{DefUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i))$ and $\mathsf{Vars}(\mathsf{LocalUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i))$ are contained in $\mathsf{Vars}(\mathsf{Active}(R_i, \{x_1, \dots, x_k\}))$, namely the updates that can be discovered in the node i are originated by the active AbU rule of the same node i. Hence, by applying again Lemma 12, we have:

$$\mathsf{Vars}(\mathsf{DefUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i)) \cup \mathsf{Vars}(\mathsf{LocalUpds}(R_i, \{x_1, \dots, x_k\}, \Sigma_i)) \subseteq \textstyle\bigcup_{x \in \mathsf{Vars}(\mathsf{upd})} \mathsf{mayBeUpdated}(x)$$

This is sufficient to prove that $\mathsf{Vars}(\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}')) \subseteq \bigcup_{x \in \mathsf{Vars}(\mathsf{upd})} \mathsf{mayBeUpdated}(x)$.

Theorem 5. *(Soundness for stabilization).*

Proof. We have to prove that for all input upd, if $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$ then S' is stabilizing. Consider an arbitrary input upd, we have that $\mathsf{S} \xrightarrow{\mathsf{upd} \blacktriangleright T} \mathsf{S}'$, for some S'. Since the ECA dependency graph of S is acyclic, we have that $\mathsf{Vars}(\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}')) \subseteq \bigcup_{x \in \mathsf{Vars}(\mathsf{upd})} \mathsf{mayBeUpdated}(x)$, by applying Lemma 13. This also means that $\mathsf{Vars}(\mathsf{upd}) \cap \mathsf{Vars}(\mathsf{discovered}(\mathsf{S}, \mathsf{upd}, \mathsf{S}')) = \varnothing$, namely we cannot discover new updates involving resources that have been currently modified. Similarly, if $\mathsf{S}' \xrightarrow{\mathsf{upd}' \blacktriangleright T} \mathsf{S}''$, for some S'', by applying again Lemma 13, we have that $\mathsf{Vars}(\mathsf{discovered}(\mathsf{S}', \mathsf{upd}', \mathsf{S}'')) \subseteq \bigcup_{x \in \mathsf{Vars}(\mathsf{upd}')} \mathsf{mayBeUpdated}(x)$ and $\mathsf{Vars}(\mathsf{upd}') \cap \mathsf{Vars}(\mathsf{discovered}(\mathsf{S}', \mathsf{upd}', \mathsf{S}'')) = \varnothing$. Since the resources of a system are finite, at each execution step we remove an update and, due to Lemma 13, we cannot discover already updated resources (i.e., we cannot add

a new update concerning the resources modified by the removed update). This means that the discovered updates set will eventually be empty. Since the initial pools of S are empty, we can conclude that eventually all pools in the system will become empty again, after executing the input upd. In other words, there exists $k \in \mathbb{N}$ such that $\mathsf{S}' \xrightarrow{\mathsf{upd}^0 \triangleright T^0} \mathsf{S}^0 \xrightarrow{\mathsf{upd}^1 \triangleright T^1} \dots \xrightarrow{\mathsf{upd}^{k-1} \triangleright T^{k-1}} \mathsf{S}^{k-1} \xrightarrow{\mathsf{upd}^k \triangleright T^k} \mathsf{S}^k$, $\mathsf{Vars}(\mathsf{discovered}(\mathsf{S}^{k-1}, \mathsf{upd}^k, \mathsf{S}^k)) = \varnothing$ and $\Theta_i^k = \varnothing$, for all $i \in [1..n]$, given $\mathsf{S}^k = R_1, \iota_1\langle\Sigma_1^k, \Theta_1^k\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n^k, \Theta_n^k\rangle$. Hence, S' is stabilizing. Since the initial input upd has been chosen arbitrarily, we can conclude that S is wave stabilizing.

Proof of Theorem 6. In order to prove the theorem, we need some supporting lemmas and some preliminary notions. Given a label $\alpha = \mathsf{upd} \triangleright T$, we denote with $\mathsf{lastof}(x, \alpha)$ the value of the last assignment to the resource x in upd. Formally, given $\mathsf{upd} = (x_1, v_1) \dots (x_k, v_k)$, we have that $\mathsf{lastof}(x, \mathsf{upd} \triangleright T) \triangleq v_i$ iff $x_i = x$ and for all $j > i$ we have that $x_j \neq x$ (the function is undefined when x is not present in upd). Given an update $\mathsf{upd} = (x_1, v_1) \dots (x_k, v_k)$, we define as $\mathsf{Vars}(\mathsf{upd}) \triangleq \{x_1, \dots, x_n\}$ the set of its resources. Similarly, we define the set of resources of a label $\alpha = \mathsf{upd} \triangleright T$ as $\mathsf{Vars}(\alpha) \triangleq \mathsf{Vars}(\mathsf{upd})$.

Definition 14 *(AbU systems diversity).* Two AbU systems $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ and $\mathsf{S}' = R_1, \iota_1\langle\Sigma_1', \Theta_1'\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n', \Theta_n'\rangle$ differ in the resource set $X \subseteq \mathbb{X}$, written $\mathsf{S} \neq^X \mathsf{S}'$, when there exists $i \in [1..n]$ and $x \in X$ such that $\Sigma_i(x) \neq \Sigma_i'(x)$.

In the sake of simplicity, when X is a singleton $\{x\}$, for some resource x, we write $\neq^x$ in place of $\neq^{\{x\}}$.

The following proposition says that if an AbU system S evolves in two systems S_1 and S_2 that differ in the value of a resource y, then during the execution of S two updates must be discovered (or they are already present in the pools of S), that assign y with two different values.

Proposition 14. *Let* $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ *be an AbU system. Suppose* $\mathsf{S} \xrightarrow{\alpha_0} \mathsf{S}_0^\alpha \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_k} \mathsf{S}_k^\alpha$ *and* $\mathsf{S} \xrightarrow{\beta_0} \mathsf{S}_0^\beta \xrightarrow{\beta_1} \dots \xrightarrow{\beta_h} \mathsf{S}_h^\beta$*, with* $k, h \in \mathbb{N}$*, such that:*

- $\mathsf{S}_k^\alpha = R_1, \iota_1\langle\Sigma_1^\alpha, \Theta_1^\alpha\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n^\alpha, \Theta_n^\alpha\rangle$
- $\mathsf{S}_h^\beta = R_1, \iota_1\langle\Sigma_1^\beta, \Theta_1^\beta\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n^\beta, \Theta_n^\beta\rangle$
- $\mathsf{S}_h^\alpha \neq^y \mathsf{S}_h^\beta$*, namely there exists* $i \in [1..n]$ *such that* $\Sigma_i^\alpha(y) \neq \Sigma_i^\beta(y)$

Then, there exist $k' \leq k$ *and* $h' \leq h$*, such that* $\mathsf{lastof}(y, \alpha_{k'}) = \Sigma_i^\alpha(y)$ *and* $\mathsf{lastof}(y, \alpha_{h'}) = \Sigma_i^\alpha(y)$.

Proof. Straightforward.

Lemma 15. *If* $\mathsf{S} \xrightarrow{(x,v)\blacktriangleright T} \mathsf{S}'$ *and* $\mathsf{S}' \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_l} \mathsf{S}^l$ *then for all* $l' \leq l$ *there exists a walk* $w_{l'}$ *in the labeled ECA dependency graph of* S *connecting* x *with* y*, given* $y \in \mathsf{Vars}(\alpha_{l'})$.

Proof. The proof is by induction on l. Suppose $l = 1$. Since S is stable (i.e., all its pools are empty), we have that the update in α_1 has been discovered during the input step $(x, v)\blacktriangleright T$. Then we have that there exists an AbU rule rule such that $\mathsf{Vars}(\alpha_1) = \mathsf{Out}(\mathsf{rule})$ and $x \in \mathsf{In}(\mathsf{rule})$. This means that (x, rule, y), with $y \in \mathsf{Out}(\mathsf{rule})$, is an arc of the labeled ECA dependency graph of S. Hence, we have a walk of length 1 connecting x with y, given $y \in \mathsf{Vars}(\alpha_1)$. Suppose now $l = k > 1$. By inductive hypothesis we have that $\mathsf{S} \xrightarrow{(x,v)\blacktriangleright T} \mathsf{S}'$, $\mathsf{S}' \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{k-1}} \mathsf{S}^{k-1}$ and for all $k'' \leq k-1$ there exists a walk $w_{k''}$ in the labeled ECA dependency graph of S connecting x with y, given $y \in \mathsf{Vars}(\alpha_{k''})$. Since S is stable (i.e., all its pools are empty), we have that the update in α_k has been discovered during the input step $(x, v)\blacktriangleright T$ or during one of the execution steps $\alpha_{k''}$. In the first case, we have a walk of length 1 connecting x with y, given $y \in \mathsf{Vars}(\alpha_k)$, as it happens in the case $l = 1$. Otherwise, we can use the inductive hypothesis. So, suppose that the update in α_k has been discovered during one of the execution steps $\alpha_{k''}$. This means that there exists an AbU rule rule such that $\mathsf{Vars}(\alpha_k) = \mathsf{Out}(\mathsf{rule})$ and $\mathsf{Vars}(\alpha_{k''}) \cap \mathsf{In}(\mathsf{rule}) \neq \varnothing$. This means that (y, rule, y'), with $y \in \mathsf{Vars}(\alpha_{k''}) \cap \mathsf{In}(\mathsf{rule})$ and $y' \in \mathsf{Out}(\mathsf{rule})$, is an arc of the labeled ECA dependency graph of S. By inductive hypothesis, we have that there exists a walk $w_{k''}$ connecting x with y, given $y \in \mathsf{Vars}(\alpha_{k''})$. Hence, $w_k = w_{k''}\mathsf{rule}$ is a walk connecting x with y', given $y' \in \mathsf{Vars}(\alpha_k)$.

Lemma 16. *If* $\mathsf{S} \xrightarrow{(x,v)\blacktriangleright T} \mathsf{S}'$ *and* $\mathsf{S}' \longrightarrow^* \xrightarrow{(x_1,v_1)\dots(x_n,v_n)\triangleright T'} \mathsf{S}''$*, then for all* x_i*, with* $i \in [1..n]$*, there exists a walk in the labeled ECA dependency graph of* S *connecting* x *with* x_i*. In other words,* $\mathsf{walks}(x, x_i) \neq \varnothing$*, for all* $i \in [1..n]$.

Proof. Applying Lemma 15, we have that there exists a walk in the labeled ECA dependency graph of S connecting x with y, given $y \in \mathsf{Vars}((x_1, v_1) \dots (x_n, v_n) \triangleright T')$. Since $\mathsf{Vars}((x_1, v_1) \dots (x_n, v_n) \triangleright T') = \{x_1, \dots, x_n\}$, we have that there exists a walk connecting x with x_i for all x_i, with $i \in [1..n]$. Hence, $\mathsf{walks}(x, x_i) \neq \varnothing$, for all $i \in [1..n]$.

Theorem 6. *(Soundness for confluence).*

Proof. Without loss of generality, suppose that the input sampling family of S consists in singletons only, namely $|\mathcal{F}| = |\mathsf{input}^{\mathsf{S}}|$. In this case we have that at each input step we can modify only one resource. The case when we can modify more resources simultaneously is analogous, we just have to augment the labeled ECA dependency graph as explained in Subsection 6.3, and the proof continues as follows.

We have to prove that for all (single) input upd, if $\mathsf{S} \xrightarrow{\mathsf{upd}\blacktriangleright T} \mathsf{S}'$ then S' is confluent. Consider an arbitrary (single) input $\mathsf{upd} = (x, v)$, for some resource x and value v, we have that $\mathsf{S} \xrightarrow{(x,v)\blacktriangleright T} \mathsf{S}'$, for some T. Suppose, by contradiction, that S' is not confluent. This means that there exist S_1, S_2 and $X \subseteq \mathbb{X}$ such that:

$$\mathsf{S}' \longrightarrow^* \mathsf{S}_1 \wedge \mathsf{S}' \longrightarrow^* \mathsf{S}_2 \wedge \forall \mathsf{S}'_1, \mathsf{S}'_2 \,.\, (\mathsf{S}_1 \longrightarrow^* \mathsf{S}'_1 \wedge \mathsf{S}_2 \longrightarrow^* \mathsf{S}'_2) \Rightarrow \mathsf{S}'_1 \neq^X \mathsf{S}'_2$$

Or, equivalently, that there exist S_1, S_2 and $X \subseteq \mathbb{X}$ such that: $\mathsf{S}' \longrightarrow^* \mathsf{S}_1 \wedge \mathsf{S}' \longrightarrow^* \mathsf{S}_2$ implies $\mathsf{S}_1 \neq^X \mathsf{S}_2$. Without loss of generality, suppose that X is a singleton, i.e., $X = \{y\}$, for some resource y. Since $\mathsf{S}_1 \neq^y \mathsf{S}_2$, we can apply Proposition 14. Therefore, there exist $k, h \in \mathbb{N}$ such that $\mathsf{S}' \xrightarrow{\alpha_0} \mathsf{S}_0^\alpha \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_k} \mathsf{S}_k^\alpha \longrightarrow^* \mathsf{S}_1$, $\mathsf{S} \xrightarrow{\beta_0} \mathsf{S}_0^\beta \xrightarrow{\beta_1} \dots \xrightarrow{\beta_h} \mathsf{S}_h^\beta \longrightarrow^* \mathsf{S}_2$ and $\mathsf{lastof}(y, \alpha_k) = v_1 \neq v_2 = \mathsf{lastof}(y, \alpha_h)$, for some values v_1 and v_2. Since $\mathsf{S} \xrightarrow{(x,v)\blacktriangleright T} \mathsf{S}'$ and $\mathsf{S}' \longrightarrow^* \xrightarrow{\alpha_k} \mathsf{S}_k^\alpha$, we can apply Lemma 16, concluding that there exists a walk w_1 in the labeled ECA dependency graph of S connecting x and y. Analogously, since $\mathsf{S} \xrightarrow{(x,v)\blacktriangleright T} \mathsf{S}'$ and $\mathsf{S}' \longrightarrow^* \xrightarrow{\beta_h} \mathsf{S}_h^\beta$, we can apply again Lemma 16, concluding that there exists a walk w_2 in the labeled ECA dependency graph of S connecting x and y. The fact that $w_1 \neq w_2$ follows from the fact that α_k and β_h update y with different values (hence are generated by different AbU rule) and from the fact that the AbU rule generating α_k is the last element of w_1 and the AbU rule generating β_h is the last element of w_2. Hence, we have that $|\mathsf{walks}(x, y)| > 1$. But this cannot happen, by hypothesis.

Therefore, we can conclude that S' must be confluent. Finally, since the initial input upd has been chosen arbitrarily, we can conclude that S is wave confluent.

References

[1] J. Cano, E. Rutten, G. Delaval, Y. Benazzouz, L. Gurgen, 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), IEEE Computer Society, USA, 2014, pp. 116–121.

[2] M. Balliu, M. Merro, M. Pasqua, M. Shcherbakov, Friendly fire: cross-app interactions in IoT platforms, ACM Trans. Priv. Secur. 24 (3) (2021), https://doi.org/10.1145/3444963.

[3] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, R. Jauhari, The HiPAC project: combining active databases and timing constraints, SIGMOD Rec. 17 (1) (1988) 51–70, https://doi.org/10.1145/44203.44208.

[4] IDC, Data age 2025, https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html, 2018.

[5] B. Gill, D. Smith, The edge completes the cloud: a Gartner trend insight, report, 2018, https://emtemp.gcom.cloud/ngw/globalassets/en/doc/documents/3889058-the-edge-completes-the-cloud-a-gartner-trend-insight-report.pdf.

[6] Y. Abd Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, R. Vigo, A calculus for attribute-based communication, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2015, pp. 1840–1845.

[7] Y. Abd Alrahman, R. De Nicola, M. Loreti, On the power of attribute-based communication, in: E. Albert, I. Lanese (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, Springer, Cham, 2016, pp. 1–18.

[8] Y. Abd Alrahman, R. De Nicola, M. Loreti, Programming interactions in collective adaptive systems by relying on attribute-based communication, Sci. Comput. Program. 192 (2020) 102428, https://doi.org/10.1016/j.scico.2020.102428.

[9] The AbU Language, GoAbU, https://github.com/abu-lang/goabu, 2023.

[10] M. Miculan, M. Pasqua, A calculus for attribute-based memory updates, in: A. Cerone, P. Ölveczky (Eds.), Proc. ICTAC, in: Lecture Notes in Computer Science, vol. 12819, Springer, 2021, pp. 366–385.

[11] N. Carriero, D. Gelernter, The s/net's linda kernel (extended abstract), in: Proc. of the 10th ACM Symposium on Operating Systems Principles, SOSP '85, ACM, New York, NY, USA, 1985, p. 160.

[12] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: a kernel language for agents interaction and mobility, IEEE Trans. Softw. Eng. 24 (5) (1998) 315–330, https://doi.org/10.1109/32.685256.

[13] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From field-based coordination to aggregate computing, in: G.D.M. Serugendo, M. Loreti (Eds.), Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings, in: Lecture Notes in Computer Science, vol. 10852, Springer, 2018, pp. 252–279.

[14] G. Audrito, M. Viroli, F. Damiani, D. Pianini, J. Beal, A higher-order calculus of computational fields, ACM Trans. Comput. Log. 20 (1) (jan 2019), https://doi.org/10.1145/3285956.

[15] F. Corradini, R. Culmone, L. Mostarda, L. Tesei, F. Raimondi, A constrained ECA language supporting formal verification of WSNs, in: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, 2015, pp. 187–192.

[16] D.R. Cacciagrano, R. Culmone, Formal semantics of an IoT-specific language, in: 32nd Int. Conf. on Advanced Information Networking and Applications Workshops (WAINA), 2018, pp. 579–584.

[17] C. Vannucchi, M. Diamanti, G. Mazzante, D.R. Cacciagrano, F. Corradini, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, 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, IEEE, 2017, pp. 92–99.

[18] C. Vannucchi, M. Diamanti, G. Mazzante, D.R. Cacciagrano, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, Symbolic verification of event-condition-action rules in intelligent environments, J. Reliab. Intell. Environ. 3 (2) (2017) 117–130, https://doi.org/10.1007/s40860-017-0036-z.

[19] X. Jin, Y. Lembachar, G. Ciardo, Symbolic verification of ECA rules, in: D. Moldt (Ed.), Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13), Milano, Italy, June 24–25, 2013, vol. 989, CEUR-WS.org, 2013, pp. 41–59.

[20] D. Beyer, A. Stahlbauer, BDD-based software verification, Int. J. Softw. Tools Technol. Transf. 16 (5) (2014) 507–518, https://doi.org/10.1007/s10009-014-0334-1.

[21] J. Cano, G. Delaval, E. Rutten, Coordination of ECA rules by verification and control, in: E. Kühn, R. Pugliese (Eds.), Coordination Models and Languages, Springer, Berlin, Heidelberg, 2014, pp. 33–48.
[22] R. De Nicola, D. Latella, A.L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, A. Vandin, The SCEL language: design, implementation, verification, in: M. Wirsing, M. Hölzl, N. Koch, P. Mayer (Eds.), Software Engineering for Collective Autonomic Systems, in: LNCS, vol. 8998, Springer, 2015, pp. 3–71.
[23] S. Anderson, N. Bredeche, A. Eiben, G. Kampis, M. van Steen, Adaptive Collective Systems: Herding Black Sheep, Bookprints, 2013.
[24] M. Wooldridge, Reasoning About Rational Agents, Intelligent Robotics and Autonomous Agents, The MIT Press, Cambridge, Massachusetts/London, 2000.
[25] Y. Abd Alrahman, R. De Nicola, M. Loreti, A calculus for collective-adaptive systems and its behavioural theory, Inf. Comput. 268 (2019) 104457, https://doi.org/10.1016/j.ic.2019.104457.
[26] Y. Abd Alrahman, G. Garbi, A distributed API for coordinating AbC programs, Int. J. Softw. Tools Technol. Transf. (feb 2020), https://doi.org/10.1007/s10009-020-00553-4.
[27] R. De Nicola, T. Duong, M. Loreti, Provably correct implementation of the AbC calculus, Sci. Comput. Program. 202 (2021) 102567, https://doi.org/10.1016/j.scico.2020.102567.
[28] Y. Abd Alrahman, R. De Nicola, G. Garbi, GoAt: attribute-based interaction in Google Go, in: T. Margaria, B. Steffen (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, in: LNCS, vol. 11246, Springer, 2018, pp. 288–303.
[29] giulio-garbi.github.io, GoAt, https://giulio-garbi.github.io/goat/, 2018.
[30] M.K. Aguilera, N. Ben-David, I. Calciu, R. Guerraoui, E. Petrank, S. Toueg, Passing messages while sharing memory, in: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC '18, ACM, New York, NY, USA, 2018, pp. 51–60.
[31] M.K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, I. Zablotchi, The impact of RDMA on agreement, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC '19, ACM, New York, NY, USA, 2019, pp. 409–418.
[32] Y. Abd Alrahman, G. Perelli, N. Piterman, Reconfigurable interaction for mas modelling, in: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '20, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2020, pp. 7–15.
[33] J. Costa Seco, S. Debois, T. Hildebrandt, T. Slaats, RESEDA: declaring live event-driven computations as REactive SEmi-structured DAta, in: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), 2018, pp. 75–84.
[34] L. Galrinho, J. Costa Seco, S. Debois, T. Hildebrandt, H. Norman, T. Slaats, ReGraDa: reactive graph data, in: F. Damiani, O. Dardha (Eds.), Coordination Models and Languages, Springer International Publishing, Cham, 2021, pp. 188–205.
[35] T. Hildebrandt, M. Marquard, R.R. Mukkamala, T. Slaats, Dynamic condition response graphs for trustworthy adaptive case management, in: Y.T. Demey, H. Panetto (Eds.), On the Move to Meaningful Internet Systems: OTM 2013 Workshops, Springer, Berlin, Heidelberg, 2013, pp. 166–171.
[36] D.R. Cacciagrano, R. Culmone, IRON: reliable domain specific language for programming IoT devices, Int. Things 9 (2020) 100020, https://doi.org/10.1016/j.iot.2018.09.006.
[37] D.I. Cohen, Introduction to Computer Theory, 2nd edition, John Wiley & Sons, Inc., USA, 1996.
[38] V. Halava, Deterministic semi-thue systems and variants of post correspondence problem, in: J. Karhumäki, A. Lepistö, L.Q. Zamboni (Eds.), Combinatorics on Words - 9th International Conference, WORDS 2013, Turku, Finland, September 16-20. Proceedings, in: Lecture Notes in Computer Science, vol. 8079, Springer, 2013, pp. 4–13.
[39] S. Eilenberg, Automata, Languages, and Machines, Academic Press, Inc., Orlando, FL, USA, 1974.
[40] M. Pasqua, I. Mastroeni, On topologies for (hyper)properties, in: D. Della Monica, A. Murano, S. Rubin, L. Sauro (Eds.), Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic, Naples, Italy, September 26-28, 2017, in: CEUR Workshop Proceedings, vol. 1949, CEUR-WS.org, 2017, pp. 150–161.
[41] J.W. Klop, Term Rewriting Systems, Oxford University Press, Inc., USA, 1993, pp. 1–116, Ch. 1.
[42] Terese, Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55, Cambridge University Press, 2003.
[43] G. Bracha, S. Toueg, Asynchronous consensus and broadcast protocols, J. ACM 32 (4) (1985) 824–840, https://doi.org/10.1145/4221.214134.
[44] R. Franklin, On an improved algorithm for decentralized extrema finding in circular configurations of processors, Commun. ACM 25 (5) (1982) 336–337, https://doi.org/10.1145/358506.358517.
[45] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News 33 (2) (2002) 51–59, https://doi.org/10.1145/564585.564601.
[46] gobot.io, GOBOT, https://gobot.io/, 2021.
[47] A. Das, I. Gupta, A. Motivala, SWIM: scalable weakly-consistent infection-style process group membership protocol, in: Proceedings International Conference on Dependable Systems and Networks, 2002, pp. 303–312.
[48] J. Gray, Notes on data base operating systems, in: Operating Systems, an Advanced Course, Springer-Verlag, Berlin, Heidelberg, 1978, pp. 393–481.
[49] M. Pasqua, M. Comuzzo, M. Miculan, The AbU language: IoT distributed programming made easy, IEEE Access 10 (2022) 132763–132776.
[50] hashicorp.com, Memberlist, https://github.com/hashicorp/memberlist/, 2022.
[51] M. Pasqua, M. Miculan, On the security and safety of AbU systems, in: R. Calinescu, C.S. Pasareanu (Eds.), Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Proceedings, in: Lecture Notes in Computer Science, vol. 13085, Springer, 2021, pp. 178–198.
[52] M. Miculan, On the formalization of the modal μ-calculus in the Calculus of Inductive Constructions, Inf. Comput. 164 (1) (2001) 199–231, https://doi.org/10.1006/inco.2000.2902.
[53] M. Miculan, M. Peressotti, GSOS for non-deterministic processes with quantitative aspects, in: Proc. QAPL, in: EPTCS, vol. 154, 2014, pp. 17–33.
[54] M. Miculan, M. Peressotti, Structural operational semantics for non-deterministic processes with quantitative aspects, Theor. Comput. Sci. 655 (2016) 135–154.