International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-026-00856-y

FOUNDATION FOR MASTERING CHANGE

ReoCAS 2024

Attribute-based memory updates with priorities for collective adaptive systems

Michele Pasqua[1] · Marino Miculan[2]

Accepted: 5 March 2026
© The Author(s) 2026

Abstract

Event-driven programming provides a natural fit for the reactive nature of pervasive systems like the Internet of Things (IoT) and Collective Adaptive Systems (CASs). *Attribute-based memory Updates* (AbU) is a calculus based on Event-Condition-Action (ECA) rules, well-suited for modeling such decentralized systems. This paper introduces a novel extension of AbU by incorporating *ECA rule priorities*. We show how this extension facilitates the natural expression of prioritized behaviors and enables the implementation of distributed data structures like Conflict-free Replicated Data Types (CRDTs). Furthermore, by leveraging the local invariants of AbU nodes and priorities we address the problem of enforcing *global invariants* in order to enhance the reliability and predictability of CASs. This is achieved through a syntactic transformation that projects global invariants into local ones and introduces high-priority synchronization rules, so that system-level properties can be guaranteed without relying on a central authority.

Keywords Smart systems · Distributed verification · ECA rules · Global invariants · CRDT

1 Introduction

The implementation of smart pervasive systems, exemplified by the *Internet of Things* (IoT), smart homes, and autonomous agents, poses significant challenges due to their intrinsic characteristics: distributed computing, dynamic network topologies, context awareness, and the need for real-time data processing. To effectively manage this complexity, the *event-driven programming* paradigm has gained prominence. Its reactive nature aligns well with pervasive systems, which continuously interact with their surroundings through sensor events and actuator commands [1, 2]. In particular, *Event-Condition-Action* (ECA) languages represent an intuitive, yet powerful, paradigm for programming reactive systems. The fundamental construct of ECA languages consists of rules of the form

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

which means: when *Event* occurs, if *Condition* is verified then execute *Action*. An ECA system is composed by one or more *nodes*, which receive inputs from the external environment and react by performing *internal* actions, updating the nodes' local memory, or *external* actions, which influence the environment itself. This paradigm can be found, often under the name of *trigger-action platforms* (TAP), in various commercial IoT frameworks like IFTTT, Samsung SmartThings, Microsoft Power Automate, Zapier, etc.

The underlying infrastructure of most ECA rule-based smart systems, such as those previously mentioned, is often centralized, relying on servers in the *Cloud*. However, the continuous growth in the scale of these systems and the ever more important security concerns strongly suggest a shift towards decentralized architectures, moving away from dependence on Cloud servers and constant Internet connectivity. Consequently, systems like IoT solutions are increasingly aligned with the computational paradigm of a *Collective Adaptive System* (CAS), which is inherently decentralized. In CASs, a large number of autonomous agents interact to achieve individual or collective goals, with the overall system behavior emerging as a result of this collective intelligence.

The recent introduction of *Attribute-based memory Updates* (AbU) [3, 4] provides support for this paradigm shift by offering a communication mechanism specifically designed

✉ M. Pasqua
michele.pasqua@univr.it

✉ M. Miculan
marino.miculan@uniud.it

[1] Department of Computer Science, University of Verona, Strada le Grazie 15, Verona, 37134, Italy

[2] Department of Mathematics, Computer Science and Physics, University of Udine, Via delle Scienze 206, Udine, 33100, Italy

Published online: 25 March 2026

for reactive and decentralized programming. This model enables nodes, such as IoT devices, to directly communicate and self-coordinate in a truly decentralized manner, eliminating the need for a central entity. Furthermore, AbU preserves the programming simplicity of ECA rules by adapting the *Attribute-based Communication* model from [5, 6] to a declarative, event-driven framework. Specifically, in AbU, an event occurring on one node can trigger updates to the states of multiple *remote* nodes. These target nodes are selected dynamically based on the conditions specified in ECA rules. Therefore, AbU enables the propagation of effects to collections of nodes simultaneously, abstracting away their individual identities (and even their presence), aligning with the principles of a pure CAS. For instance, the following rule:

$$login > @(\overline{role} = \text{`logger'}) : \overline{log} \leftarrow \overline{log} \cdot time$$

means "when the (local) variable *login* changes, on every node whose *role* is 'logger' append my current (local) *time* to the (remote) variable *log*".

The declarative nature of ECA rules can make it challenging to predict the order of the updates they trigger. When multiple rules are activated simultaneously, the sequence in which their updates are executed is non-deterministic. This non-determinism aligns with the characteristics of distributed systems and the heterogeneity of nodes within CAS environments, but in certain scenarios, programmers need finer-grained control over the scheduling of these updates. A natural approach to introduce this control in ECA rule-based programming, without compromising its declarative essence, is to enable programmers to assign *priorities* to rules. Consequently, if several rules are triggered concurrently, their execution order adheres to the assigned priority levels.

In this paper, we introduce an extension of AbU that incorporates *rule priorities*. We demonstrate that this extension, backward compatible with previous version [3, 4], enables a natural description of behaviors governed by priority policies. Furthermore, it can be effectively employed to implement distributed data structures such as *Conflict-free Replicated Data Types* (CRDTs) [7], which are specifically designed to guarantee eventual consistency despite concurrent and independent modifications across different nodes.

A particularly compelling application of AbU with priorities lies in its potential to enforce *global, system-wide invariants*, such as correctness guarantees encompassing functional or safety properties of the entire system. Notably, AbU nodes inherently support *local invariants* [8], which are properties of individual nodes automatically enforced by the AbU semantics; however, implementing global invariants has historically posed significant challenges. A centralized approach, where a central node gathers data from all others to verify the global property, suffers from scalability limitations and is ill-suited for truly distributed environments like CASs. We demonstrate that system-level invariants can be enforced by strategically leveraging AbU local invariants and rules with priorities, without the need for a central authority. This can be achieved by *projecting* the global invariant into a set of local invariants and *augmenting* each node with appropriate high-priority rules designed for the synchronization of the attributes relevant to the global invariant.

This paper makes the following contributions.

- We present an extension of the AbU calculus that incorporates the concept of rule priorities. This extension is backward compatible with existing versions of AbU.
- We show that priority-enhanced AbU allows for the natural description of behaviors governed by priority policies, and for implementing distributed data structures like CRDTs.
- We provide a novel approach to enforcing global, system-wide invariants in decentralized systems, by leveraging local invariants of AbU nodes and high-priority rules to synchronize the attributes relevant to the global invariant.

Synopsis Section 2 introduces an extension of AbU [3], an ECA-inspired calculus with attribute-based memory updates; this extension incorporates node invariants and priorities among ECA rules. Section 3 then demonstrates the applicability of AbU through illustrative examples within the domains of CASs and distributed systems. Following this, Sect. 4 focuses on defining system-level invariants in AbU and provides a syntactic transformation that automatically adds node-level invariants to AbU code, ensuring the enforcing of a system-level invariant. Finally, Sect. 5 discusses relevant prior research, and Sect. 6 concludes the paper by summarizing its contributions and outlining directions for future work.

2 Attribute-based memory updates with priority

AbU [3, 4] is a calculus merging the programming simplicity of ECA rules with *attribute-based memory updates*, a powerful distributed communication mechanism inspired by attribute-based communication [9]. AbU's communication mechanism allows a node to update at once the states of many nodes, which are selected by means of their attributes without the need for central coordination and a shared knowledge about network topology. The central idea of AbU is that rules activation may produce update events on arbitrary nodes in the network, either locally on the node on which the rule is executed, or on remote nodes. Such updates a propagated in the network and dispatched to the nodes selected upon their attributes. In this section, we will introduce the AbU calculus of [3, 4] enhanced with a notion of *priority* between updates. That is, update events come with a priority level affecting the actual order of their application.

Fig. 1 Grammar of ECA rules in the AbU calculus with updates priority and node invariants

$$\begin{aligned} \mathsf{rule} &::= \mathsf{evt} \gg^{\ell} \mathsf{task} \\ \mathsf{evt} &::= \mathrm{x} \mid \mathsf{evt}\ \mathsf{evt} \\ \mathsf{task} &::= \mathsf{cnd} : \mathsf{act} \\ \mathsf{act} &::= \epsilon \mid \mathrm{x} \leftarrow \varepsilon\ \mathsf{act} \mid \overline{\mathrm{x}} \leftarrow \varepsilon\ \mathsf{act} \end{aligned} \qquad \begin{aligned} \mathsf{cnd} &::= \varphi \mid @\varphi \\ \varphi &::= \mathtt{ff} \mid \mathtt{tt} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varepsilon \bowtie \varepsilon \mid (\varphi) \\ \varepsilon &::= v \mid \mathrm{x} \mid \overline{\mathrm{x}} \mid \varepsilon \otimes \varepsilon \\ & \mathrm{x} \in \mathbb{X} \quad v \in \mathbb{V} \quad \ell \in \mathcal{P} \end{aligned}$$

2.1 AbU with priority: syntax

In AbU nodes are programmed by using Event-Condition-Action (ECA) rules, a powerful yet intuitive coding style particularly adopted in the IoT. Indeed, AbU turns out to be well suited for the IoT and, more generally, in collective adaptive systems. The calculus has been extended in [8] with *node invariants*, namely predicates on each node's state which must be always satisfied during execution. This is useful to avoid erroneous or dangerous states, like inconsistent or out-of-range values, and forbidden trajectories in planning. In practical scenarios, ECA rules can have different priority, thus a different ordering of execution. For instance, rules concerning safety-critical components can have a privileged execution over rules logging events. For this reason, we introduce here *priority levels* among ECA rules.

An AbU system S is basically a list of *nodes* which execute in parallel:

$$\mathsf{S} ::= R, \iota\langle\Sigma, \Theta\rangle \mid \mathsf{S} \parallel \mathsf{S}$$

Each *node* $R, \iota\langle\Sigma, \Theta\rangle$ consists of the following components:

- a list R of ECA rules;
- an invariant ι, namely a boolean expression that the node must satisfy at runtime;
- a *state* $\Sigma \in \mathbb{X} \to \mathbb{V}$, mapping resources $\mathrm{x} \in \mathbb{X}$ to values $v \in \mathbb{V}$;
- and an *execution pool with priority* $\Theta \subseteq \mathbb{U} \times \mathcal{P}$.

Here, $\mathbb{U} \triangleq (\mathbb{X} \times \mathbb{V})^*$ is the set of all possible *updates*, that is, finite lists of pairs of the form $((\mathrm{x}_1, v_1) \ldots (\mathrm{x}_m, v_m))$. Each list $\mathsf{upd} \in \mathbb{U}$ represents a simultaneous, multiple update waiting to be applied to the state. Instead, $(\mathcal{P}, \preccurlyeq)$ is the set of priority levels, partially ordered by the relation $\preccurlyeq \subseteq \mathcal{P} \times \mathcal{P}$. Hence, a pool with priority is a set of updates each with a priority level.

An AbU ECA rule is defined by the grammar in Fig. 1. Each ECA rule, represented as $\mathsf{evt} \gg^{\ell} \mathsf{task}$, consists of an *event* evt and a *task* task, with an associated *priority level* ℓ. The event is a list of resources the rule "listens" to. When one of these resources is modified, the rule is triggered, and its task is evaluated. This evaluation does not immediately change resource states; instead, it generates update operations that are added to execution pools and applied later.

A task is composed of a *condition* cnd and an *action* act. A condition is a boolean expression, which can be optionally prefixed with the modifier $@$. If the $@$ modifier is absent, the task is considered *local*, and its condition is checked only on the current node. If the condition is met, the action is evaluated. Conversely, if the task is *remote* (prefixed with $@$), its condition is checked on every node in the system. The action is then evaluated on all nodes where the condition holds.

The action is a list of assignments that set values to either *local resources* (x) or *remote resources* ($\overline{\mathrm{x}}$). The evaluation of an action yields an update, which is added to the current node's pool for local tasks or to the pools of the remote nodes for remote tasks. Each update is tagged with the priority level ℓ of the rule that generated it.

Finally, we model a *node invariant* ι as a system of linear inequalities over expressions ε, where expressions can only range over the resources of the node the invariants belong to. That is, invariants are defined by the following grammar.

$$\begin{aligned} \iota &::= \mathsf{ineq} \mid \neg\iota \mid \iota \wedge \iota \mid (\iota) \\ \mathsf{ineq} &::= \varepsilon < v \mid \varepsilon \leq v \mid \varepsilon \neq v \qquad \text{with } v \in \mathbb{V} \end{aligned}$$

Node invariants are limited to local resources and cannot contain remote resource lookups ($\overline{\mathrm{x}}$).

In the following, we assume to have boolean, numeric (integers and decimals) and string values in $\mathbb{V}$. Expressions can utilize standard arithmetic operations for numeric values, and basic string operations (length, substring, and concatenation). Comparison operators like $<$ and $\leq$ are defined exclusively for numeric values, while $\neq$ is applicable to all data types. We can express more complex inequalities by simple syntactic manipulation; for instance, $\varepsilon_1 < \varepsilon_2$ can be rewritten as $\varepsilon_1 - \varepsilon_2 < 0$. Similarly, equality can be represented as the negation of an inequality, such as $\varepsilon_1 = \varepsilon_2$ being equivalent to $\neg(\varepsilon_1 - \varepsilon_2 \neq 0)$.

Nodes with tautological invariants, such as *true* or $0 < 1$, are simply written as $R\langle\Sigma, \Theta\rangle$.

2.2 AbU with priority: semantics

The AbU (small-step) semantics is modeled as a labeled transition system $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_2$ whose labels α are given by the grammar:

$$\alpha ::= T \mid (\mathsf{upd}, \ell) \triangleright T \mid \mathsf{upd} \blacktriangleright T$$

Here, upd is an update, ℓ is a priority level and $T = (\mathsf{task}_1, \ell_1) \ldots (\mathsf{task}_n, \ell_n)$ is a finite list of tasks each with

$$(\textsc{Exec})\ \frac{\begin{array}{c}\mathsf{pick}(\Theta) = (\mathsf{upd}, \ell) \quad \mathsf{upd} = (\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad \Sigma' \models \iota \\ X = \{\mathrm{x}_i \mid i \in [1..k] \wedge \Sigma(\mathrm{x}_i) \neq \Sigma'(\mathrm{x}_i)\} \quad \mathsf{LocalUpds}(R, X, \Sigma') = \Theta'' \\ \Theta' = (\Theta \setminus \{(\mathsf{upd}, \ell)\}) \cup \Theta'' \quad \mathsf{ExtTasks}(R, X, \Sigma') = T\end{array}}{R, \iota\langle \Sigma, \Theta \rangle \xrightarrow{(\mathsf{upd}, \ell) \triangleright T} R, \iota\langle \Sigma', \Theta' \rangle}$$

$$(\textsc{Exec-F})\ \frac{\begin{array}{c}\mathsf{pick}(\Theta) = (\mathsf{upd}, \ell) \quad \mathsf{upd} = (\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \\ \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad \Sigma' \not\models \iota \quad \Theta' = \Theta \setminus \{(\mathsf{upd}, \ell)\}\end{array}}{R, \iota\langle \Sigma, \Theta \rangle \xrightarrow{(\mathsf{upd}, \ell) \triangleright \epsilon} R, \iota\langle \Sigma, \Theta' \rangle}$$

$$(\textsc{Input})\ \frac{\begin{array}{c}v_1, \dots, v_k \in \mathbb{V} \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad X = \{\mathrm{x}_i \mid i \in [1..k] \wedge \Sigma(\mathrm{x}_i) \neq v_i\} \\ \mathsf{LocalUpds}(R, X, \Sigma') = \Theta'' \quad \Theta' = \Theta \cup \Theta'' \quad \mathsf{ExtTasks}(R, X, \Sigma') = T\end{array}}{R, \iota\langle \Sigma, \Theta \rangle \xrightarrow{(\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \blacktriangleright T} R, \iota\langle \Sigma', \Theta' \rangle}$$

$$(\textsc{Disc})\ \frac{\Theta'' = \{(\llbracket \mathsf{act} \rrbracket \Sigma, \ell_i) \mid \exists i \in [1..n] \,.\, \mathsf{task}_i = \varphi : \mathsf{act} \wedge \Sigma \models \varphi\} \quad \Theta' = \Theta \cup \Theta''}{R, \iota\langle \Sigma, \Theta \rangle \xrightarrow{(\mathsf{task}_1, \ell_1) \dots (\mathsf{task}_n, \ell_n)} R, \iota\langle \Sigma, \Theta' \rangle}$$

$$(\textsc{Step-L})\ \frac{\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{T} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ \alpha \in \left\{\begin{array}{l}(\mathsf{upd}, \ell) \triangleright T, \\ \mathsf{upd} \blacktriangleright T\end{array}\right\} \qquad (\textsc{Step-R})\ \frac{\mathsf{S}_1 \xrightarrow{T} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ \alpha \in \left\{\begin{array}{l}(\mathsf{upd}, \ell) \triangleright T, \\ \mathsf{upd} \blacktriangleright T\end{array}\right\}$$

Fig. 2 LTS Semantics of the AbU calculus with updates priority and node invariants

$$\mathsf{LocalUpds}(R, X, \Sigma) \triangleq \{(\llbracket \mathsf{act} \rrbracket \Sigma, \ell) \mid \exists \mathsf{evt} \gtrdot^{\ell} \varphi : \mathsf{act} \in R \,.\, \mathsf{evt} \cap X \neq \varnothing \wedge \Sigma \models \varphi\}$$

$$\mathsf{ExtTasks}(R, X, \Sigma) \triangleq (\{\!\!\{\mathsf{task}_1\}\!\!\}\Sigma, \ell) \dots (\{\!\!\{\mathsf{task}_n\}\!\!\}\Sigma, \ell) \text{ such that } \exists \mathsf{evt} \gtrdot^{\ell} \mathsf{task}_i \in R \,.\, \mathsf{evt} \cap X \neq \varnothing \wedge \mathsf{task}_i = @\varphi : \mathsf{act}$$

Fig. 3 Discovery functions for the AbU calculus semantics with updates priority and node invariants

a priority level. What changes from the standard AbU semantics [3, 4] is that updates execution also records the corresponding priority level and the updated resources. A transition can modify the state and the execution pool of the nodes but, at the same time, each node does not have a global knowledge about the system. Figure 2 depicts the transition semantic rules of AbU. Rule (Exec) executes an update picked from the pool; while rule (Input) models an external modification of some resources. The execution of an update, or the external change of resources, may trigger some rules of the nodes. Hence, after updating a node state, the node launches a *discovery phase*, for finding updates to add to the local or remote pools, given by the activation of some rules.

The discovery phase is composed by two parts, the *local* and the *external* one. A node $R, \iota\langle \Sigma, \Theta \rangle$ performs a local discovery by means of the functions LocalUpds, that add to the local pool Θ all updates originated by the activation of some rules in R, together with the corresponding priority level. Then, by means of the function ExtTasks, the node computes a list of tasks that may affect other nodes and sends it to all nodes in the system, together with the corresponding priority level.

The definitions of LocalUpds and ExtTasks are presented in Fig. 3. In these functions, the term $\{\!\!\{\mathsf{task}\}\!\!\}\Sigma$ refers to the *pre-evaluation* of task against a state Σ. This operation involves a two-step process: first, every occurrence of a resource variable x in the condition and the right-hand side of assignments in the task's action is substituted with its corresponding value from $\Sigma(\mathrm{x})$; subsequently, each instance of $\overline{\mathrm{x}}$ in the action is replaced with x, and the modifier @ is removed.

External task spreading is modeled by labels $(\mathsf{upd}, \ell) \triangleright T$ and $\mathsf{upd} \blacktriangleright T$, produced by rules (Exec) and (Input) respectively. 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 rules (StepL) and (StepR) (to enforce symmetry) complete and synchronize (on all nodes in the system) a discovery phase originated by a state change of a node in the system.

In the AbU calculus, updates are consumed asynchronously from the local pools, aligning with the asynchronous nature of each node's execution. However, the subsequent communication phase between nodes, triggered by a successful update, is modeled as a synchronous, transactional operation. This synchronous communication is essential for the distributed discovery phase, where all nodes must agree on the discovery action to collectively perform. Therefore, while individual nodes operate independently and asynchronously, the inter-node communication for state synchronization is a concerted, synchronous process.

The update to pick from the pool in the (Exec) rule is demanded to the node scheduler, which has been deliberately left undefined in [3, 4] and demanded to actual implementations. This is to decouple the theoretical calculus

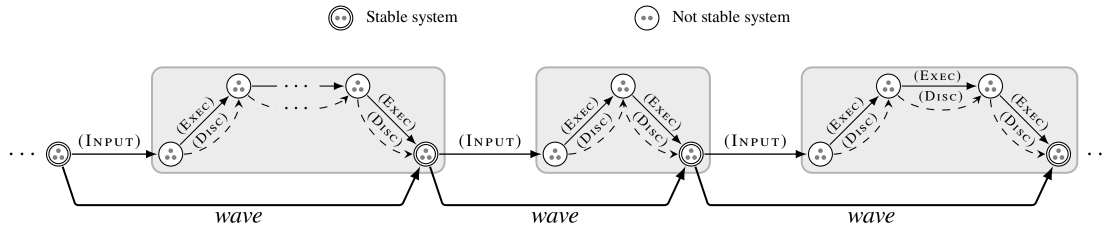

Fig. 4 A graphical representation of the AbU execution model. An AbU system is represented by the local state and pool of all nodes (here represented by dots). A stable system moves to a not stable one due to an (Input); then it executes only (Exec) steps, until it possibly reaches a new stable system. The transitions between stable systems are represented by a *wave*

from the implementations, that may require context-specific schedulers. Still, when priority levels come into play, some constraints on the scheduler must be imposed. Specifically, the AbU scheduler must rely on an *update selection function* $\mathsf{pick} : \wp(\mathbb{U} \times \mathcal{P}) \to \mathbb{U} \times \mathcal{P}$ in charge of picking one update with higher priority from the pool of available updates. Higher priority here means that the selected update is guaranteed to be attached with the highest possible priority level. Formally, we require that the update selection function is *priority-respectful*.

Definition 1 (Priority-respectful)
An update selection function $\mathsf{pick} : \wp(\mathbb{U} \times \mathcal{P}) \to \mathbb{U} \times \mathcal{P}$ is *priority-respectful* if, for every pool $\Theta \subseteq \mathbb{U} \times \mathcal{P}$:

$$\mathsf{pick}(\Theta) = (\mathsf{upd}, \ell) \Rightarrow \forall (\mathsf{upd}', \ell') \in \Theta \,.\, \ell \not\preccurlyeq \ell' \vee \ell = \ell'$$

A priority-respectful selection function pick can be implemented by choosing an update with the highest priority level from the pool. Since the priority levels form a partial order, a "highest" priority may not be unique or even exist. In this case, we can select any update whose priority level is a least upper bound (supremum) of the priority levels of all updates within the pool, with respect to $\preccurlyeq$. If such a unique least upper bound does not exist within the pool itself, we can fall back to selecting any update whose priority level is a maximal element with respect to $\preccurlyeq$.

Finally, the semantics also checks the fulfillment of node invariants. Indeed, the rule (Exec) is applied only if the state modified by the update still satisfies the invariant (i.e., $\Sigma' \models \iota$); otherwise, rule (Exec-F) is applied, discarding from the pool the update that would lead to a "bad" state.

A form of big-step semantics, called *wave* in [4], can be defined for AbU, modeling the behavior of a system when an external stimulus is provided. A *wave* comprises all LTS steps occurring in between two *stable systems*, that is, having all nodes' pool empty (hence with no update to perform). When a stable system receives an external stimulus by using the LTS rule (Input), some ECA rules may be triggered, possibly adding updates to the pools. Such updates are consumed by nodes (possibly triggering other ECA rules and, hence, adding new updates to the pools) until all pools are again all empty. This is depicted in Fig. 4, where we provide a graphical explanation of the AbU execution model (taken from [4]).

Note that, despite the choice of which node should perform an update is non-deterministic, when a node actually performs an update, the remaining nodes wait for the completion of the discovery phase, which propagates the effects of the update through transactional communication. This transactional execution model of course introduces communication overhead, but this is unavoidable in a truly decentralized setting. We refer to [10] for some implementation details.

Classic AbU with no priority [3, 4] is a special case of the calculus just introduced. Indeed, when considering a singleton priority set, e.g., $(\{0\}, \{(0,0)\})$, we have that the semantics in Fig. 2 boils down to the original AbU semantics from [3, 4]. Having a single priority level tantamount to having no priority at all, since all updates will be picked by the AbU selection function with the same precedence order. In the following, when considering a singleton priority set $(\{0\}, \{(0,0)\})$, we restrain the original AbU syntax by omitting the priority level in the rule syntax, i.e., we will write $\mathsf{evt} \gg \mathsf{task}$ in place of $\mathsf{evt} \gg^0 \mathsf{task}$.

3 Applications of AbU

We now show two application scenarios for AbU in the context of Collective Adaptive Systems. In particular, the first (Sect. 3.1) leverages the introduced rule priorities to model precedence between the tasks executed by the system's nodes, while the second (Sect. 3.2) leverages local node invariants to perform runtime consistency checks on nodes' resources. Finally (Sect. 3.3), we show how an example of Conflict-free Replicated Data Types (CRDTs) [7] can be implemented in AbU.

3.1 Terrestrial rover swarm

Consider a scenario where a swarm of terrestrial rovers is in charge of taking specific measurements, randomly picked

in a large uninhabited area. Each rover is equipped with a battery that periodically needs to be recharged by returning to a docking station. It may happen that a rover runs out of energy before returning to the charging spot. In this case, the low-battery rover asks for help from its neighbors. If a rover has some energy to share and it is close enough to the requester, it will enter the rescue mode which starts a rover-to-rover charging protocol. The latter has precedence over the other tasks of the rover (e.g., taking a measurement), meaning that the rover will enter the rescue mode even if other, less important tasks have been previously scheduled. We can model this scenario in AbU as follows (the energy transfer phase is omitted here for space reasons).

Suppose to have four rovers. For each rover we have an AbU node with a resource battery, indicating the battery level; a resource position, indicating the position; a resource mode, indicating in which operative state is the rover; a resource helpPos, indicating the position of a rover that needs help; and a resource takeMeasurement indicating whether the rover has to take a new measurement or not. Formally, the AbU system modeling the rover-swarm scenario is

$$R\langle\Sigma_1,\Theta_1\rangle \parallel R\langle\Sigma_2,\Theta_1\rangle \parallel R\langle\Sigma_3,\Theta_3\rangle \parallel R\langle\Sigma_4,\Theta_4\rangle$$

where R contains the following three rules:

$$\begin{aligned}&\mathrm{battery} \gg^{1} @(\mathrm{battery} < 5 \wedge \overline{\mathrm{battery}} > 80) : \\ &\quad \overline{\mathrm{helpPos}} \leftarrow \mathrm{position}\end{aligned} \tag{1}$$

$$\begin{aligned}&\mathrm{helpPos} \gg^{1} (|\mathrm{position} - \mathrm{helpPos}| < 7.0) : \\ &\quad \mathrm{mode} \leftarrow \text{‘}\mathit{rescue}\text{’}\end{aligned} \tag{2}$$

$$\begin{aligned}&\mathrm{position} \gg^{0} (\mathrm{mode} = \text{‘}\mathit{measure}\text{’} \wedge \mathsf{rand}) : \\ &\quad \mathrm{takeMeasurement} \leftarrow \mathtt{tt}\end{aligned} \tag{3}$$

Rule (1) is used by out-of-battery rovers to send a “help request” to its neighbors, while rule (2) leads a rover to enter the rescue mode. Rule (3) is used to take a new measurement: when a rover in measure mode changes position, it will take a measurement depending on a random event rand. The first two rules have greater importance than the third one; this is modeled by the priority set $(\{0,1\},\{(0,0),(0,1),(1,1)\})$, giving precedence to rules tagged with 1 over rules tagged with 0.

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

$$\begin{aligned}\Sigma_1 &= [\,\mathrm{battery}\mapsto 4 \;\; \mathrm{position}\mapsto 2.0 \\ &\qquad \mathrm{mode}\mapsto \text{‘}\mathit{measure}\text{’} \;\; \mathrm{helpPos}\mapsto 0.0\,] \\ \Sigma_2 &= [\,\mathrm{battery}\mapsto 81 \;\; \mathrm{position}\mapsto 15.0 \\ &\qquad \mathrm{mode}\mapsto \text{‘}\mathit{measure}\text{’} \;\; \mathrm{helpPos}\mapsto 0.0\,] \\ \Sigma_3 &= [\,\mathrm{battery}\mapsto 97 \;\; \mathrm{position}\mapsto 6.0 \\ &\qquad \mathrm{mode}\mapsto \text{‘}\mathit{measure}\text{’} \;\; \mathrm{helpPos}\mapsto 0.0\,] \\ \Sigma_4 &= [\,\mathrm{battery}\mapsto 65 \;\; \mathrm{position}\mapsto 8.0 \\ &\qquad \mathrm{mode}\mapsto \text{‘}\mathit{measure}\text{’} \;\; \mathrm{helpPos}\mapsto 0.0\,]\end{aligned}$$

Also, suppose that all rovers have the following pools: $\Theta_1 = \Theta_2 = \Theta_3 = \Theta_4 = \{((\mathrm{takeMeasurement}, \mathtt{tt}), 0)\}$. The update in the pools has been obtained by a previous application of rule (3) on all rovers.

Rule (1) says that when a rover’s battery level is low (i.e., battery < 5), then the rover has to send to all neighbors with some energy to share (i.e., $\overline{\mathrm{battery}} > 80$) its position, performing a remote update: $\overline{\mathrm{helpPos}} \leftarrow \mathrm{position}$. In the example, the first rover can fire rule (1), since its battery level is low. Then, it pre-evaluates the task condition, yielding $4 < 5 \wedge \mathrm{battery} > 80$, which is sent to the other rovers, together with the pre-evaluation of the task action ($\mathrm{helpPos} \leftarrow 2.0$) and the priority level 1. Only the second and the third rovers are interested in the communication, since they only have battery level greater than 80. So they both add to their pool the update $((\mathrm{helpPos}, 2.0), 1)$. This ends the discovery phase originated by the first rover. Now, the pool of the second and third rovers contain the updates $((\mathrm{helpPos}, 2.0), 1)$ and $((\mathrm{takeMeasurement}, \mathtt{tt}), 0)$. The first update has higher priority (i.e., 1) than the second one (i.e., 0), hence the rover’s scheduler will pick the former to be executed next. This allows the second and the third rovers to apply rule (2).

Rule (2) is fired when a rover receives a help request (i.e., when its resource helpPos changes) and checks if the current rover position is close to the requester position (i.e., $|\mathrm{position} - \mathrm{helpPos}| < 7.0$). If it is the case, the current rover enters the rescue mode performing a local update $\mathrm{mode} \leftarrow \text{‘}\mathit{rescue}\text{’}$. The latter is executed before any other lower priority (i.e., 0) update in the pool, since it has higher priority (i.e., 1). In the example, when the second and the third rovers execute the update $((\mathrm{helpPos}, 2.0), 1)$, the task of rule (2) may be executed. For the second rover this does not happen, since $|15.0 - 2.0| < 7.0$ does not hold (the rover is too far from the distressed one). Instead, $|6.0 - 2.0| < 7.0$ holds and the third rover can execute the rule task, adding to its pool the update $((\mathrm{mode}, \text{‘}\mathit{rescue}\text{’}), 1)$. Now, the pool of the third rover contains the updates $((\mathrm{mode}, \text{‘}\mathit{rescue}\text{’}), 1)$ and $((\mathrm{takeMeasurement}, \mathtt{tt}), 0)$. Since, the first update has higher priority (i.e., 1) than the second update (i.e., 0), the rover’s scheduler will pick such update giving precedence to entering ‘rescue-mode’ over taking a measurement.

3.2 Smart HVAC system

In this example, we provide an AbU implementation of a Heating, Ventilation and Air Conditioning (HVAC) system, that makes use of local invariants on single nodes. In this scenario we have three devices connected through a network:

the HVAC control system, a temperature sensor, and a humidity sensor. To distinguish the devices, a logical resource node is used, which takes the values *'system'*, *'tempSens'* and *'humSens'* on the HVAC control system, the temperature sensor and the humidity sensor, respectively.

We model such scenario in AbU as follows. The execution state for the HVAC control system is:

$$\Sigma_s = [\,\text{heating}\mapsto\texttt{ff}\ \ \text{conditioning}\mapsto\texttt{ff}\ \ \text{temperature}\mapsto 0\ \ \text{humidity}\mapsto 0\ \ \text{airButton}\mapsto\texttt{ff}\ \ \text{node}\mapsto\textit{'system'}\,]$$

while its rules R_s are:

$$\text{temperature} \gg (\text{temperature} < 18) : \text{heating} \leftarrow \texttt{tt} \tag{4}$$

$$\text{temperature} \gg (\text{temperature} > 27) : \text{heating} \leftarrow \texttt{ff} \tag{5}$$

$$\text{airButton} \gg (\text{airButton} = \texttt{tt}) : \text{conditioning} \leftarrow \texttt{ff} \tag{6}$$

$$\text{humidity temperature} \gg (2 + 0.5 * \text{temperature} < \text{humidity} \wedge 38 - \text{temperature} < \text{humidity}) : \text{conditioning} \leftarrow \texttt{tt} \tag{7}$$

In this example, rules are not tagged with priority levels since a singleton priority set, indicating equal precedence between rules, is considered.

The HVAC control system activates heating and air conditioning according to the values of temperature and humidity, received by the sensors. In particular, when the temperature is lower than 18 $°C$ (i.e., temperature < 18), rule (4) activates the heating with the update: heating $\leftarrow \texttt{tt}$. Instead, when the temperature is greater than 27 $°C$ (i.e., temperature > 27), then rule (5) deactivates the heating with the update: heating $\leftarrow \texttt{ff}$. The air conditioning is turned on (with the update conditioning $\leftarrow \texttt{tt}$), by means of rule (7), when the humidity exceeds the upper bound of Givoni's *comfort zone* [11].

Execution states for temperature and humidity sensors are:

$$\Sigma_t = [\,\text{temperature}\mapsto 19\ \ \text{node}\mapsto\textit{'tempSens'}\,]$$

$$\Sigma_h = [\,\text{humidity}\mapsto 40\ \ \text{node}\mapsto\textit{'humSens'}\,]$$

While their ECA rules are:

$$R_t \triangleq \text{temperature} \gg @(\text{node} = \textit{'system'}) : \overline{\text{temperature}} \leftarrow \text{temperature} \tag{8}$$

$$R_h \triangleq \text{humidity} \gg @(\text{node} = \textit{'system'}) : \overline{\text{humidity}} \leftarrow \text{humidity} \tag{9}$$

Rule (8) on the temperature sensor device is only responsible of signaling changes to the resource temperature to the HVAC control system, by selecting all devices that have the resource node equals to *'system'*; rule (9) does the same for humidity on the humidity sensor device.

The HVAC control system is also bestowed with a physical button for manually stopping the air conditioning. Indeed, rule (6) stops the air conditioning (with the update conditioning $\leftarrow \texttt{ff}$) when the button is pressed (i.e., airButton $= \texttt{tt}$).

Finally, by means of the invariant

$$\iota_s = \neg(\text{conditioning} \wedge \text{heating})$$

on the control system device we specify that heating and air conditioning can never be activated at the same time. The complete AbU system is

$$R_s, \iota_s\langle\Sigma_s, \Theta_s\rangle \parallel R_t\langle\Sigma_t, \Theta_t\rangle \parallel R_h\langle\Sigma_h, \Theta_h\rangle$$

with $\Theta_s = \Theta_t = \Theta_h = \varnothing$.

Note that, the same problem can be modeled by means of a single device, embedding the two sensors and the control system. We can model this scenario in AbU with a single device comprising all resources introduced so far and transforming remote rules into local ones. This highlights the flexibility of AbU, that is able to model both distributed and centralized ensembles of devices.

3.3 CRTD in AbU

Conflict-free Replicated Data Types (CRDTs) [7] are replicated data structures designed for distributed systems that guarantee eventual consistency despite concurrent and independent changes across nodes. Unlike traditional approaches relying on synchronization or conflict resolution, CRDTs achieve convergence by design, ensuring that all replicas will eventually reach the same state regardless of the order of operations. This is accomplished through specific data structure and operation design that inherently avoids conflicts.

CRDTs are broadly categorized into two families: *state-based* (also known as *convergent replicated data types*) and *operation-based* (also known as *commutative replicated data types*). In state-based CRDTs, each node maintains a complete local state. Synchronization occurs by periodically exchanging these states between nodes (often upon local updates). Convergence is ensured by a commutative, associative, and idempotent *merge function*. This function combines two states to produce a new, consistent state, making the CRDT resilient to message reordering and duplication. A grow-only counter exemplifies this: each node stores a counter, and the merge function takes the maximum value, guaranteeing that all nodes eventually reach the highest observed count.

Operation-based CRDTs propagate individual update operations, instead of sharing entire states. When a local update happens, an operation describing the change is generated and broadcast to other nodes. Convergence relies on the commutativity and associativity of these operations, ensuring that applying them in any order yields the same final state. Unlike state-based CRDTs, operation idempotency is not strictly required, as reliable broadcast mechanisms can handle duplicate delivery. A simple increment/decrement counter illustrates this: each increment or decrement operation (e.g., $+10$, -5) is broadcast, and if delivered without duplicates, all nodes will converge to the same final count.

In [7], it has been proved that state-based and operation-based CRDTs are theoretically equivalent, as each can emulate the other.

Operation-based CRDT in AbU AbU allows to seamlessly implement operation-based CRDTs. Indeed, idempotence of the operation is not needed at all, as any additional requirement on communication, since AbU guarantees by design the absence of duplicated updates.

Consider the following example modeling an increment/decrement counter where AbU nodes share an integer-valued resource num. Suppose to have the system

$$\mathsf{S} = R\langle \Sigma_1, \Theta_1 \rangle \parallel R\langle \Sigma_2, \Theta_2 \rangle$$

where R contains the following two rules:

$$\begin{aligned} \mathrm{delta} \gg^1 @(\mathtt{tt}) : \mathrm{num} &\leftarrow \mathrm{num} + \mathrm{delta} \\ \overline{\mathrm{sync}} &\leftarrow \mathrm{delta} \end{aligned} \tag{10}$$

$$\mathrm{sync} \gg^1 (\mathtt{tt}) : \mathrm{num} \leftarrow \mathrm{num} + \mathrm{sync} \tag{11}$$

Specifically, delta and sync are integer-valued resources indicating the increment or decrement of num applied by nodes (delta is used locally, while sync is used for synchronization). When a node needs to modify the counter, a rule performing an update of the resource delta is triggered. We assume such a rule having priority lower than 1. Indeed, we need to require that rules involved in CRDT resources update and synchronization have high priority level, thus precedence over other rules. This ensures that CRDT resources update and synchronization occur in batch, without being interleaved with non-CRDT related updates possibly making CRDT resource updates inconsistent. As we will se in the next Section, we can always add a priority level to enforce such a behavior.

Now suppose that the execution states and the pools of the two nodes are the following:

$$\Sigma_1 = \Sigma_2 = [\,\mathrm{num} \mapsto 10 \;\; \mathrm{delta} \mapsto 0 \;\; \mathrm{sync} \mapsto 0\,]$$

$$\Theta_1 = \Theta_2 = \varnothing$$

Upon committing an update that changes the resource delta to -7 at the first node (e.g., as a result of the activation of a rule on the first node having priority lower than 1), the update is immediately propagated to the second node. Consequently, the first node's state and pool become:

$$\Sigma_1 = [\,\mathrm{num} \mapsto 10, \mathrm{delta} \mapsto -7, \mathrm{sync} \mapsto 0\,]$$

$$\Theta_1 = \{((\mathrm{num}, 3), 1)\}$$

The discovery phase then adds the update $((\mathrm{sync}, -7), 1)$ to the pool Θ_2 of the second node, leaving its execution state unchanged.

At this stage, both nodes can perform an (Exec). If the second node proceeds, the update $((\mathrm{sync}, -7), 1)$ is removed from Θ_2, and its state becomes: $\Sigma_2 = [\,\mathrm{num} \mapsto 10 \;\; \mathrm{delta} \mapsto 0 \;\; \mathrm{sync} \mapsto -7\,]$. Consequently, the update $((\mathrm{num}, 3), 1)$ is added to Θ_2 due to the triggering of rule (11) by the change in sync. Now, both nodes can apply the remaining updates, eventually leading to execution states

$$\Sigma_1 = [\,\mathrm{num} \mapsto 3 \;\; \mathrm{delta} \mapsto -7 \;\; \mathrm{sync} \mapsto 0\,]$$

$$\Sigma_2 = [\,\mathrm{num} \mapsto 3 \;\; \mathrm{delta} \mapsto 0 \;\; \mathrm{sync} \mapsto -7\,]$$

where the shared resource num has the same value and the pools Θ_1 and Θ_2 are empty.

In the case Θ_1 or Θ_2 were not initially empty, or if some other rule is triggered during computation adding updates to the pools, the above updates involving the CRDT resource are executed first, since they have higher priority.

4 Distributed verification of system-level invariants

AbU with priority allows us to implement *distributed* verification of *system-level invariants*, that is, invariants of a whole AbU system. These invariants stipulate a requirement that multiple nodes have to collectively fulfill, and typically require a centralized enforcing mechanism. As we will see shortly, having a centralized entity enforcing system-level invariants is not mandatory when AbU updates come with priority levels.

Global invariants, or system-level invariants, have the same syntactic structure of local invariants, the only difference is that in global invariants I expressions can range over resources of all nodes in the system (thus, not restricted to the resources of a specific node). Moreover, in global invariants resources may be indexed with node identifiers, to distinguish resources (potentially having the same name) belonging to different nodes. If not indexed, global invariant resources are considered on all possible nodes in the system, keeping node anonymity typical of AbU. In some scenarios it may be necessary to refer to specific resources on specific nodes, so we added that possibility in global invariants. The

indexing is just syntactic sugar: when nodes are not anonymous we can rename their resources with unique identifiers.

Suppose to modify the HVAC example of Sect. 3.2, in order to remove the control system node. Heating and conditioning controllers are moved to the temperature and humidity sensor nodes. The execution state for the temperature and the humidity sensors become:

$$\Sigma_t = [\,\text{temperature} \mapsto 19 \;\; \text{heating} \mapsto \texttt{ff}\,]$$
$$\Sigma_h = [\,\text{humidity} \mapsto 40 \;\; \text{conditioning} \mapsto \texttt{ff} \;\; \text{airButton} \mapsto \texttt{ff}\,]$$

The rules R_t for the temperature sensor node are:

$$\text{temperature} \gg (\texttt{tt}) : \overline{\text{temperature}} \leftarrow \text{temperature} \tag{12}$$

$$\text{temperature} \gg (\text{temperature} < 18) : \text{heating} \leftarrow \texttt{tt} \tag{13}$$

$$\text{temperature} \gg (\text{temperature} > 27) : \text{heating} \leftarrow \texttt{ff} \tag{14}$$

The rules R_h for the humidity sensor node are:

$$\text{airButton} \gg (\text{airButton} = \texttt{tt}) : \text{conditioning} \leftarrow \texttt{ff} \tag{15}$$

$$\text{humidity temperature} \gg (2 + 0.5 * \text{temperature} < \text{humidity} \wedge 38 - \text{temperature} < \text{humidity}) : \text{conditioning} \leftarrow \texttt{tt} \tag{16}$$

This formulation of the problem is equivalent to that in Sect. 3.2, except for the invariant: it is not guaranteed that the conditioning system and the heater cannot be on at the same time. To enforce such behavior we need a global invariant

$$I = \neg(\text{conditioning}_h \wedge \text{heating}_t)$$

meaning that the resources conditioning_h of the humidity node and the resource heating_t of the temperature node cannot be $\texttt{tt}$ at the same time. We could also have not indexed the resources conditioning and heating. Without indexes, the global invariant I holds for all nodes having conditioning and heating as resources.

4.1 From global to local invariants

Global invariants are properties, possibly involving multiple nodes, that must hold for all components of the system. Ensuring their fulfillment requires, in general, a central authority enforcing such property and, consequently, knowing the topology of the system (or, at least, keeping an inventory of all deployed nodes). A central authority is in contrast with autonomic systems, which are decentralized in nature and usually rely on peer-to-peer communication only.

By exploiting AbU with priorities, we can guarantee global invariants in CASs without the need of a central controlling authority. This is done by projecting a system-level invariant to an ensemble of node-level invariants, that is, AbU local invariants. The idea is that the fulfillment of local invariants, under specific assumptions, guarantees the fulfillment of the corresponding global invariant. This requires the replication of a global invariant on all nodes in its *scope*, that is, on all nodes having at least a resource appearing in the (global) invariant. Since AbU nodes do not have a shared knowledge about the state of external resources, we have to propagate modifications to resources in the scope of global invariants to all interested nodes. Such synchronization is achieved by adding suitable remote updates for each resource in the scope of global invariants.

Algorithm 1 describes how an AbU system S can be modified in order to fulfill a global invariant I by means of an ensemble of local invariants, added to the nodes in S. In particular, the algorithm assumes as input a global invariant I in the conjunctive normal form, that is, of the form $\bigwedge_{i=1}^{m} \iota_i$ where each ι_i are either of the form ineq or $\neg\iota$. The outer loops at lines **1..2** try to add each conjunct of I to each node of the system S. This happens only when at least one resource in the scope of the conjunct belongs to a node (condition at line **3**). The line **4** add such conjunct ι_j to the local invariant $\hat{\iota}_i$ of the ith node of S. The inner loop at line **5** then adds all resources in the scope of the added conjunct not already belong to the modified node to the node's state Σ_i (line **6**). The added resources are initialized with a random value of the correct type. Finally, the inner loop at line **7** adds to the modified node the ECA rules need for synchronization. In particular, each resource x in the scope of the conjunct that already belongs to the modified node (condition at line **7**) can potentially be (locally) updated by the node. Such modification should be reported to the other nodes involved in the fulfillment of the conjunct (and, hence, of the global invariant). This is done by performing a remote update of the local resource x by adding a new ECA rule to rule list of the ith node in S (line **8**). The added rule evt $\gg^{\mathsf{H}}$ task here is a rule having higher priority, since synchronization updates must be considered before any other update. Here, we consider a fresh priority level H not belonging to the priority set $(\mathcal{P}, \preccurlyeq)$ defined for S. The priority set then becomes $(\mathcal{P} \cup \{\mathsf{H}\}, \preccurlyeq \cup \{(\ell, \mathsf{H}) \mid \ell \in \mathcal{P}\})$, indicating that H has higher priority than any other element of the original set. The AbU semantics scheduler will first execute all updates in the pool having priority H, that is, synchronization updates, and then the updates having lower priority coming from the original system's rules.

By applying Algorithm 1 to the smart HVAC system introduced at the beginning of the section, we obtain the following

```
Algorithm DecentralizeInvariant(S, I)
    /* the AbU system S is of the form R_1, î_1⟨Σ_1, Θ_1⟩ ‖ ... ‖ R_n, î_n⟨Σ_n, Θ_n⟩      */
    /* the global invariant I is of the form ι_1 ∧ ··· ∧ ι_m                               */
    /* H is a fresh priority level not belonging to the priority set (𝒫, ≼) of S          */
    /* vars(Σ) denotes the resource of the state Σ                                         */
    /* vars(ι) denotes the resource identifiers occurring in the invariant ι               */
1   for i from 1 to n do
2       for j from 1 to m do
3           if vars(ι_j) ∩ vars(Σ_i) ≠ ∅ then
4               î_i := î_i ∧ ι_j
5               for all x in vars(ι_j) \ vars(Σ_i) do
6                   Σ_i := Σ_i ⊎ [x ↦ v]     // here ⊎ denotes state join and v has the same type of x
                end
7               for all x in vars(ι_j) ∩ vars(Σ_i) do
8                   R_i := R_i :: x ⋗^H @(tt) : x̄ ← x     // here :: denotes list concat
                end
            end
        end
    end
9   return S
```

Algorithm 1. Enhancing an AbU system with local invariants derived from a given global invariant

AbU system. The resource conditioning is added to the state of the temperature node, and the resource heating is added to the state of the humidity node. That is:

$$\Sigma_t = [\,\text{temperature} \mapsto 19\ \ \text{heating} \mapsto \texttt{ff}\ \ \text{conditioning} \mapsto \texttt{ff}\,]$$

$$\Sigma_h = [\,\text{humidity} \mapsto 40\ \ \text{conditioning} \mapsto \texttt{ff}\ \ \text{airButton} \mapsto \texttt{ff}\ \ \text{heating} \mapsto \texttt{ff}\,]$$

Then, synchronization ECA rules are added: one propagating the modifications of the resource heating from the temperature node to external nodes; and another propagating the modifications of the resource conditioning from the humidity node to external nodes. That is, the rule

$$\text{heating} \gtrdot^{\mathsf{H}} @(\texttt{tt}) : \overline{\text{heating}} \leftarrow \text{heating} \tag{17}$$

is added to R_t and the rule

$$\text{conditioning} \gtrdot^{\mathsf{H}} @(\texttt{tt}) : \overline{\text{conditioning}} \leftarrow \text{conditioning} \tag{18}$$

is added to R_h. These rules have higher priority than the rules already present in the nodes.

Finally, to both temperature and humidity nodes the invariant $\neg(\text{conditioning} \wedge \text{heating})$ is added. In this example, the global invariant and the local invariants coincide, but this is not always the case. Indeed, if the global invariant would have stipulated a constraint on resources not affecting the humidity and temperature nodes (e.g., $I = \neg(\text{conditioning} \wedge \text{heating}) \wedge \text{brightness} < 15$), then the local invariants on those nodes and the global invariant would have been different.

4.2 Soundness of local invariants

Assuming a priority ordering of updates delivery and execution, we can prove that the enforcing of local invariants, as defined by Algorithm 1, is sufficient to guarantee the satisfaction of the corresponding global invariant. In other words, given a global invariant I for the system S, the system

$$\mathsf{S}_\mathsf{D} \triangleq \texttt{DecentralizeInvariant}(\mathsf{S}, I)$$

is guaranteed to not violate I, for all its possible executions. This is done by enforcing all local invariants in S_D. We assume that the system S adopts a priority set $(\mathcal{P}, \preccurlyeq)$ not containing the element H, which is added to the set to be adopted by S_D. That is, the system S_D adopts the priority set $(\mathcal{P}_\mathsf{H}, \preccurlyeq_\mathsf{H})$ where $\mathcal{P}_\mathsf{H} \triangleq \mathcal{P} \cup \{\mathsf{H}\}$ and $\preccurlyeq_\mathsf{H} \triangleq \preccurlyeq \cup \{(\ell, \mathsf{H}) \mid \ell \in \mathcal{P}\}$.

In the following, we assume that inputs from the environment cannot make a system to violate an invariant. Indeed, external stimuli are uncontrollable events that cannot be suppressed, thus impossible to enforce within the language.

Theorem 1 (Local Invariants Soundness)
Let S_D be an AbU system obtained from the AbU system S by decentralizing the invariant I as per Algorithm 1. If S_D satisfies I, then for all R reachable from S_D, R satisfies I.

Proof

Let us consider a sequence of transitions $\mathsf{S_D} \xrightarrow{\alpha_1} \mathsf{R}_1 \xrightarrow{\alpha_2} \mathsf{R}_2 \ldots$ (possibly infinite). If α_1 does not contain a H priority update (i.e., α_1 is of the form $(\mathsf{upd}, \ell) \triangleright T$ with $\ell \neq \mathsf{H}$), then the update executed at the first step does not involve any variable of the invariant I, because the only H priority updates are those generated by the rules added by Algorithm 1. Hence, I is still valid for R_1, and we can repeat the argument starting from R_1.

Let us consider the case when α_1 contains a H priority update, i.e., α_1 is of the form $(\mathsf{upd}, \mathsf{H}) \triangleright T$; this means that some variable of I has been modified at the first step. Let x one of these variables, and let us consider any node $R, \iota \wedge \iota_\mathsf{D}\langle\Sigma, \Theta\rangle$ in R_1 where ι_D is the part of invariant added by Algorithm 1. Clearly, the pool Θ contains $((\mathrm{x}, v), \mathsf{H})$, to update the local copy of x with the new value. This update can be executed by (Exec) because it does not violate neither the local invariant ι (since x has been freshly added to the store Σ) nor ι_D (otherwise the update would have violated the invariant on the originating node at $\mathsf{S_D}$ already). Therefore, the update (x, v) is executed before any update in Θ having priority $\ell \preccurlyeq \mathsf{H}$ due to the respectfulness of pick (Definition 1), yielding to state R_1, where the invariant still holds. The same argument can be repeated until all high priority updates are executed; this leads to some state R_n, where there are no further synchronization updates to apply, and the invariant I still holds. We can repeat the argument starting from R_n, proving the thesis. ◻

The system $\mathsf{S_D}$ must preserve the semantics of the original system S, up-to the added synchronization updates. Indeed, the ECA rules added by Algorithm 1 do not affect the resources of the original system. Of course, since $\mathsf{S_D}$ enforces locally the global invariant I, only the actions performed by S that do not violate the global invariant are preserved by $\mathsf{S_D}$. In other words, $\mathsf{S_D}$ must perform all and only the actions that S perform, excluding synchronizations (i.e., H priority updates).

Formally, let $\mathcal{L}$ be the set of all AbU system labels. Consider the following projection function $\cdot\backslash_\mathsf{H} \in \mathcal{L} \to \mathcal{L}$ on AbU system labels $\alpha \in \mathcal{L}$

$$\alpha\backslash_\mathsf{H} \triangleq \begin{cases} T\backslash_\mathsf{H} & \text{if } \alpha = T \\ \epsilon & \text{if } \alpha = (\mathsf{upd}, \mathsf{H}) \triangleright T \\ (\mathsf{upd}, \ell) \triangleright (T\backslash_\mathsf{H}) & \text{if } \alpha = (\mathsf{upd}, \ell \neq \mathsf{H}) \triangleright T \\ \mathsf{upd} \blacktriangleright (T\backslash_\mathsf{H}) & \text{if } \alpha = \mathsf{upd} \blacktriangleright T \end{cases}$$

where $T\backslash_\mathsf{H}$ is recursively defined on a task list $T = \mathsf{task}_1 \ldots \mathsf{task}_n$ as follows (ϵ is the empty list):

$$\epsilon\backslash_\mathsf{H} \triangleq \epsilon$$

$$((\mathsf{task}, \ell)T)\backslash_\mathsf{H} \triangleq \begin{cases} (\mathsf{task}, \ell)(T\backslash_\mathsf{H}) & \text{if } \ell \neq \mathsf{H} \\ T\backslash_\mathsf{H} & \text{if } \ell = \mathsf{H} \end{cases}$$

Now let $\mathcal{L}_\mathsf{H} \subseteq \mathcal{L}$ be the set of all H priority labels, that is, labels involving at least a H priority update or task. Formally, $\mathcal{L}_\mathsf{H} \triangleq \{\alpha \in \mathcal{L} \mid \alpha\backslash_\mathsf{H} \neq \alpha\}$. We denote with $\twoheadrightarrow^*$ the transitive closure of the relation $\bigcup\{\xrightarrow{\alpha} \mid \alpha \in \mathcal{L} \setminus \mathcal{L}_\mathsf{H}\}$ and with $\twoheadrightarrow^*_\mathsf{H}$ the transitive closure of the relation $\bigcup\{\xrightarrow{\alpha} \mid \alpha \in \mathcal{L}\}$. Then, $\xrightarrow{\alpha}{}^* \triangleq \twoheadrightarrow^* \xrightarrow{\alpha} \twoheadrightarrow^*$ means that we can perform an arbitrary, possibly empty, sequence of labels except the H priority ones, but at least one observable label $\alpha \in \mathcal{L}$ must be performed. Similarly, $\xrightarrow{\alpha}{}^*_\mathsf{H} \triangleq \twoheadrightarrow^*_\mathsf{H} \xrightarrow{\alpha} \twoheadrightarrow^*_\mathsf{H}$ means that we can perform an arbitrary, possibly empty, sequence of labels comprising the H priority ones, but at least one observable label $\alpha \in \mathcal{L}$ must be performed.

In the following theorem, statement *1* of stipulates that $\mathsf{S_D}$ can perform any action that S can do without violating the invariant I. Instead, statement *2* stipulates that $\mathsf{S_D}$ does not perform any action that S cannot do, except for H priority updates (i.e., synchronizations).

Theorem 2 (Semantics Preservation)
Let $\mathsf{S_D}$ be an AbU system obtained from the AbU system S by decentralizing the invariant I as per Algorithm 1. If S satisfies I, then the following hold:

1. *for all S' satisfying I, if $\mathsf{S} \xrightarrow{\alpha} \mathsf{S}'$ then there exists R such that $\mathsf{S_D} \xrightarrow{\beta}{}^*_\mathsf{H} \mathsf{R}$, for some β such that $\alpha = \beta\backslash_\mathsf{H}$, and $\mathsf{R} = (\mathsf{S}')_\mathsf{D}$;*
2. *for all R, if $\mathsf{S_D} \xrightarrow{\alpha}{}^*_\mathsf{H} \mathsf{R}$ then there exists S' satisfying I such that $\mathsf{S} \xrightarrow{\beta}{}^* \mathsf{S}'$, with $\beta = \alpha\backslash_\mathsf{H}$, and $(\mathsf{S}')_\mathsf{D} = \mathsf{R}$.*

Proof

Let us prove *1* first. The proof is by case analysis on the label α such that $\mathsf{S} \xrightarrow{\alpha} \mathsf{S}'$. Since S does not contain H priority rules, $\alpha \notin \mathcal{L}_\mathsf{H}$.

Case $\alpha = (\mathsf{upd}, \ell) \triangleright T$. Since $\alpha \notin \mathcal{L}_\mathsf{H}$, we have that $\ell \neq \mathsf{H}$. Since the rules in S are a subset of those in $\mathsf{S_D}$, and Algorithm 1 does not change pools, we have that also $\mathsf{S_D}$ can execute the update (upd, ℓ). Hence, $\mathsf{S_D} \xrightarrow{\beta}{}^*_\mathsf{H} \mathsf{R}$, where $\beta = (\mathsf{upd}, \ell) \triangleright T'$, for some T'. Since in $\mathsf{S_D}$ we have all the rules in S plus H priority rules only, in T' we have all the tasks in T, eventually plus some H priority tasks. Hence, $\alpha = \beta\backslash_\mathsf{H}$. Moreover, since S and S' have the same rules, Algorithm 1 will add the same rules and resources when applied to S'. What can change are the pools in R, that may differ from $(\mathsf{S}')_\mathsf{D}$ by containing some H priority updates. But, since the relation $\xrightarrow{\alpha}{}^*_\mathsf{H}$ can perform an arbitrary number of actions, we can assume that eventual H priority updates can be consumed from the pools in R. Thus, $\mathsf{R} = (\mathsf{S}')_\mathsf{D}$.

Case $\alpha = T$ or $\alpha = \mathsf{upd} \blacktriangleright T$. Since the discovery and inputs can be always performed by any system, we have that also $\mathsf{S_D}$ can perform α. Then, the proof proceeds as in the previous case.

Let us now prove *2*. The proof is by case analysis on the label α such that $\mathsf{S_D} \xrightarrow{\alpha}{}^*_\mathsf{H} \mathsf{R}$.

Case $\alpha = (\mathsf{upd}, \mathsf{H}) \triangleright T$. Since S does not have H priority rules, the update $(\mathsf{upd}, \mathsf{H})$ cannot be performed by S. But, $\beta = \epsilon = \alpha \backslash_{\mathsf{H}}$ means that the system S can perform no action in place of α. Hence, $\mathsf{S} \twoheadrightarrow^* \mathsf{S}'$. Moreover, since S and S' have the same rules, Algorithm 1 will add the same rules and resources when applied to S'. What can change are the pools in R, that may differ from $(\mathsf{S}')_{\mathsf{D}}$ by containing some H priority updates. But, tasks in T can only be the result of an activation of H priority rules (since upd has priority H), and the latter can be consumed from the pools in R since the relation $\xrightarrow{\alpha}{}^*_{\mathsf{H}}$ can perform an arbitrary number of actions. Thus, $(\mathsf{S}')_{\mathsf{D}} = \mathsf{R}$. Since no change has been made on S', we have that S' satisfies I.

Case $\alpha = (\mathsf{upd}, \ell) \triangleright T$, with $\ell \neq \mathsf{H}$. Since the rules in S are a subset of those in S_{D}, and Algorithm 1 does not change pools, we have that also S can execute the update (upd, ℓ). Hence, $\mathsf{S} \xrightarrow{\beta}{}^* \mathsf{S}'$, where $\beta = (\mathsf{upd}, \ell) \triangleright T'$, for some T'. Since the rules in S are a subset of those in S_{D}, and in S_{D} we have additional H priority rules only, in T' we have all the tasks in T, except eventual H priority tasks. Hence, $\beta = \alpha \backslash_{\mathsf{H}}$. Moreover, since S and S' have the same rules, Algorithm 1 will add the same rules and resources when applied to S'. What can change are the pools in R, that may differ from $(\mathsf{S}')_{\mathsf{D}}$ by containing some H priority updates. But, since the relation $\xrightarrow{\alpha}{}^*_{\mathsf{H}}$ can perform an arbitrary number of actions, we can assume that eventual H priority updates can be consumed from the pools in R. Thus, $(\mathsf{S}')_{\mathsf{D}} = \mathsf{R}$. Since the update upd can be performed by S_{D}, by Theorem 1 we have that such an update does not violate I. Thus, the same update performed on S leads to S' satisfying I.

Case $\alpha = T$ or $\alpha = \mathsf{upd} \blacktriangleright T$. Since the discovery and inputs can be always performed by any system, we have that also S can perform β such that $\beta = \alpha \backslash_{\mathsf{H}}$, for a similar reasoning as the one done in the previous case. Also, S' satisfies I because T does not modify nodes' state and from the fact that inputs cannot make a system to violate an invariant. Then, the proof proceeds as in the previous case. □

4.3 Application: drone coalition

To showcase the generality of AbU global/local invariants, we will present an additional application scenario: a coalition of aerial drones is in charge of surveilling a residential area. The flight of a drone is governed by a simple strategy: a target position the drone is supposed to reach is periodically updated (e.g., at a constant refresh-rate) by a control room communicating with the drone. When the drone position or the target position change, the drone computes the movement to perform in order to fly towards its target. We model such a movement with a distance shift from the current position (the translation of this movement into actual rotor commands is demanded to the low-level physics engine of the drone).

Let us assume to have n drones; each drone $i \in [1..n]$ has the following AbU resources: a sensor gps, modeling the current GPS position of the drone (updated by the physics engine of the drone); a sensor target, modeling the target GPS position the drone is aiming at (updated remotely by the control room); and an actuator move, modeled as a GPS coordinates shift (that will instruct the physics engine on how to move the drone).

Each drone is equipped with the following rule:

$$\begin{aligned} &\mathrm{gps}\ \mathrm{target} \gg (|\mathrm{target} - \mathrm{gps}| \geq \delta) : \\ &\quad \mathrm{move} \leftarrow (\mathrm{target} - \mathrm{gps}) / |\mathrm{target} - \mathrm{gps}| \end{aligned} \tag{19}$$

The rule says that each time the drone position or the destination change, if their distance is greater than a given threshold δ (possibly zero), the drone should move one distance unit (e.g., a meter) towards the target. Given this simple setting, we can specify interesting global invariants for the drone coalition. To ease the notation, we denote with Δ_i the distance shift, computed as in (19), for the drone $i \in [1..n]$, i.e., $\Delta_i \triangleq (\mathrm{target}_i - \mathrm{gps}_i)/|\mathrm{target}_i - \mathrm{gps}_i|$.

Ground control station nearness Each drone cannot fly too far, i.e., beyond a given limit l, from a specific geographical point G; formally:

$$I_{\mathrm{gcsn}} \triangleq \bigwedge_{i \in [1..n]} |(\mathrm{gps}_i + \Delta_i) - G| < l$$

Collision avoidance Drones cannot get too close, i.e., no less than a given threshold t, to each other (to prevent collisions); formally:

$$I_{\mathrm{ca}} \triangleq \bigwedge_{i,j \in [1..n]}^{i \neq j} |(\mathrm{gps}_i + \Delta_i) - \mathrm{gps}_j| > t$$

Dispersion limit A drone cannot stray too far, i.e., beyond a given limit l, from the coalition barycenter; formally:

$$I_{\mathrm{dl}} \triangleq \bigwedge_{i \in [1..n]} \left| (\mathrm{gps}_i + \Delta_i) - \frac{\sum_{j \in [1..n]}^{j \neq i} \mathrm{gps}_j}{n-1} \right| < l$$

In the case of I_{gcsn}, Algorithm 1 will simply add to each node $i \in [1..n]$ the local invariant $|(\mathrm{gps}_i + \Delta_i) - G| < l$. No synchronization rules will be added, since no coordination between drones is needed to fulfill the (global) invariant. For I_{ca}, Algorithm 1 will add to each node i the local invariant $\bigwedge_{j \in [1..n] \setminus \{i\}} |(\mathrm{gps}_i + \Delta_i) - \mathrm{gps}_j| > t$ and the resources in the scope of the added invariant missing from node i, i.e., the resources gps_j such that $j \in [1..n] \setminus \{i\}$. Finally, the synchronization rule $\mathrm{gps}_i \gg^{\mathsf{H}} @(\mathtt{tt}) : \overline{\mathrm{gps}_i} \leftarrow \mathrm{gps}_i$ will be added. The third case I_{dl} is similar to the previous one.

5 Related work

The verification of distributed systems has been a consistent focus in the literature, ranging from foundational works by

Chandy and Lamport [12] and Babaoglu and Raynal [13] to more contemporary approaches, as surveyed by Francalanza et al. [14]. However, many of these methodologies assume a centralized setting or a shared understanding of the system's network topology, rendering them less suitable for CASs.

Within the CAS community, Aldrini [15] proposes a framework for designing and verifying trust within CASs, but it does not address the specification and enforcement of global or local invariants. Audrito et al. [16] advocate for *aggregate computing* as a paradigm well-suited for distributed runtime verification in CASs. Aggregate computing treats a network of devices as a unified computational entity, abstracting away individual device specifics to focus on collective computation. Nevertheless, this abstraction can be challenging to apply in scenarios where individual devices have distinct roles and must adhere to specific constraints, as is common in many IoT applications. Finally, Bortolussi et al. [17] introduce CARMA, a framework for modeling the dynamic and adaptive behavior of CASs, enabling performance analysis and the identification of potential implementation issues. However, CARMA does not provide mechanisms for defining or enforcing global and local invariants.

The present work significantly extends our prior contribution [18] in several key aspects, notably by providing more comprehensive examples to illustrate the expressiveness and applicability of prioritized AbU. A key addition is the detailed demonstration of how prioritized AbU can be effectively used for the implementation of distributed data structures such as Conflict-free Replicated Data Types. Moreover, the present work generalizes the calculus to an arbitrary priority set, not limited to two-levels settings as in [18].

Furthermore, this work presents a significant advancement about the enforcement of global invariants using local invariants and prioritized rules. In [18] we have only outlined the potential for this approach, but the current paper provides a rigorous syntactic transformation that automatically derives node-level invariants and necessary high-priority synchronization rules to ensure the enforcement of a system-level invariant. Crucially, we include a formal proof establishing the soundness and semantics preservation of this transformation, providing a strong theoretical foundation for our decentralized invariant enforcement methodology.

Significant effort has been spent on *distributed* Runtime Verification (RV), aiming at checking general temporal properties, like those expressible in Linear Temporal Logic (LTL), in systems whose components operate concurrently without centralized observation. Typically, a global LTL specification is decomposed into sub-formulas, assigning them to local monitors capable of partial evaluation and communication to detect satisfaction or violation collectively. For instance, Bauer and Falcone [19] introduced a decentralized LTL-monitoring algorithm where each component only observes and communicates necessary parts, significantly reducing communication overhead compared to central monitoring. Mostafa and Bonakdarpour [20] proposed a sound and complete method for asynchronous distributed systems using three-valued LTL semantics. Aceto et al. [21] explored synthesis procedures for decentralized monitors of *hyperproperties* [22]. Our work shares a similar goal with these works, but it aims to provide a native mechanism for decentralized invariant enforcement at the programming language level, instead of relying on an external verification infrastructure (e.g., via monitors).

6 Conclusions

In this paper, we have presented a prioritized extension of the AbU calculus, a communication mechanism well-suited for modeling reactive and decentralized systems like CASs. This extension, while remaining backward compatible with existing versions of AbU, introduces the crucial concept of rule priorities, offering programmers more control over the execution order of updates in scenarios with concurrent rule triggering. We have illustrated the utility of this extension through its ability to naturally describe prioritized behaviors and its applicability in implementing complex distributed data structures such as CRDTs, highlighting its potential in managing consistency in decentralized environments.

A key contribution of this work lies in a new decentralized methodology for enforcing global, system-wide invariants. By strategically leveraging the local invariants in AbU nodes and introducing appropriately prioritized synchronization rules, we have shown that it is possible to guarantee global system-level invariants without resorting to centralized control. The proposed approach of projecting global invariants into local constraints and employing prioritized rules for attribute synchronization offers a scalable and robust mechanism for guaranteeing the correctness and safety of distributed systems.

Future work A thorough analysis of the performance implications of prioritized rule execution in large-scale deployments is crucial. Indeed, in a decentralized context like ours, every state change must be propagated to the other nodes, but the invariant check does not require messages since it is local. Therefore, in the worst-case scenario, each update requires $n - 1$ messages (with n being the number of nodes in the system) to propagate synchronization updates. Nevertheless, not all nodes are involved in all invariants, hence the worst-case scenario may be not applicable in most real situations. By conducting an extensive empirical evaluation, we plan to assess the actual invariant decentralization overhead as proposed in the current paper. Developing practical tools and methodologies to assist programmers in defining, implementing, and verifying global invariants within AbU would also be a valuable contribution.

While the current work demonstrates the feasibility of decentralized invariant enforcement with a simple logic, it paves the way for the decentralization of more general properties, such as liveness properties and hyperproperties. To this end, we need to extend the logic of local invariants on the individual AbU nodes with temporal operators. Consequently, the primary challenge lies in enhancing the logical reasoning and verification capabilities of the nodes themselves (e.g., along the lines of [19, 21]), while the core projection algorithm should not be significantly more complex than the one presented in this paper.

Another line of research involves exploring the relaxation of the AbU semantics, specifically by loosening the atomicity constraint on update distribution. This relaxation could necessitate the development of alternative scheduling policies to maintain the crucial relationship between global and local invariants, as discussed in Sect. 4.

Finally, extending the semantic framework to incorporate quantitative aspects, along the lines of prior work in [23, 24], could provide valuable insights into resource usage and performance characteristics of prioritized AbU programs.

Funding information Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Cano, J., Rutten, E., Delaval, G., Benazzouz, Y., Gurgen, L.: ECA rules for IoT environment: a case study in safe design. In: Proc. 8th SASOW, pp. 116–121. IEEE (2014). https://doi.org/10.1109/SASOW.2014.32
2. Balliu, M., Merro, M., Pasqua, M., Shcherbakov, M.: Friendly fire: Cross-app interactions in IoT platforms. ACM Trans. Priv. Secur. **24**(3) (2021)
3. Miculan, M., Pasqua, M.: A calculus for attribute-based memory updates. In: Cerone, A., Ölveczky, P. (eds.) Proc. 18th International Colloquium on Theoretical Aspects of Computing (ICTAC). Lecture Notes in Computer Science, vol. 12819, pp. 366–385. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-85315-0_21
4. Pasqua, M., Miculan, M.: AbU: a calculus for distributed event-driven programming with attribute-based interaction. Theor. Comput. Sci., 1–32 (2023). https://doi.org/10.1016/j.tcs.2023.113841
5. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems, pp. 1–18. Springer, Cham (2016)
6. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. **192**, 102428 (2020). https://doi.org/10.1016/j.scico.2020.102428
7. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, pp. 386–400. Springer, Berlin (2011)
8. Pasqua, M., Miculan, M.: Behavioral equivalences for AbU: Verifying security and safety in distributed IoT systems. Theor. Comput. Sci., 1–32 (2024). https://doi.org/10.1016/j.tcs.2024.114537
9. Abd Alrahman, Y., De Nicola, R., Loreti, M., Tiezzi, F., Vigo, R.: A calculus for attribute-based communication. In: Proc. 30th SAC, pp. 1840–1845. ACM (2015)
10. Pasqua, M., Comuzzo, M., Miculan, M.: The AbU language: IoT distributed programming made easy. IEEE Access **10**, 132763–132776 (2022). https://doi.org/10.1109/ACCESS.2022.3230287
11. Givoni, B.: Comfort, climate analysis and building design guidelines. Energy Build. **18**(1), 11–23 (1992)
12. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of a distributed system. ACM Trans. Comput. Syst., 63–75 (1985)
13. Babaoglu, O., Raynal, M.: Specification and verification of dynamic properties in distributed computations. J. Parallel Distrib. Comput. **28**(2), 173–185 (1995). https://doi.org/10.1006/jpdc.1995.1098
14. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
15. Aldini, A.: Design and verification of trusted collective adaptive systems. ACM Trans. Model. Comput. Simul. **28**(2) (2018). https://doi.org/10.1145/3155337
16. Audrito, G., Damiani, F., Stolz, V., Viroli, M.: On distributed runtime verification by aggregate computing. In: Ancona, D., Pace, G. (eds.) Proceedings of the Second Workshop on Verification of Objects at RunTime EXecution. EPTCS, vol. 302, pp. 47–61 (2019). https://doi.org/10.4204/EPTCS.302.4
17. Bortolussi, L., De Nicola, R., Galpin, V., Gilmore, S., Hillston, J., Latella, D., Loreti, M., Massink, M.: CARMA: collective adaptive resource-sharing Markovian agents. In: Proc. QAPL 2015, pp. 16–31 (2015). https://doi.org/10.4204/eptcs.194.2
18. Pasqua, M., Miculan, M.: Local reasoning and attribute-based memory updates for enforcing global invariants in collective adaptive systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Rigorous Engineering of Collective Adaptive Systems, pp. 351–367. Springer, Cham (2025)
19. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Form. Methods Syst. Des. **48**(1–2), 46–93 (2016). https://doi.org/10.1007/S10703-016-0253-8
20. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, pp. 494–503. IEEE Computer Society (2015). https://doi.org/10.1109/IPDPS.2015.95
21. Aceto, L., Achilleos, A., Anastasiadi, E., Francalanza, A., Gorla, D., Wagemaker, J.: Centralized vs decentralized monitors for hyperproperties. In: Majumdar, R., Silva, A. (eds.) 35th International Conference on Concurrency Theory, CONCUR 2024. LIPIcs, vol. 311. Schloss Dagstuhl (2024). https://doi.org/10.4230/LIPICS.CONCUR.2024.4
22. Clarkson, M., Schneider, F.: Hyperproperties. In: 21st IEEE Computer Security Foundations Symposium, pp. 51–65 (2008). https://doi.org/10.1109/CSF.2008.7

23. Bacci, G., Miculan, M.: Structural operational semantics for continuous state probabilistic processes. In: Proc. CMCS. Lecture Notes in Computer Science, vol. 7399, pp. 71–89. Springer (2012). https://doi.org/10.1007/978-3-642-32784-1_5
24. Bacci, G., Miculan, M.: Structural operational semantics for continuous state stochastic transition systems. J. Comput. Syst. Sci. **81**(5), 834–858 (2015). https://doi.org/10.1016/J.JCSS.2014.12.003

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.