# Local Reasoning and Attribute-Based Memory Updates for Enforcing Global Invariants in Collective Adaptive Systems

Michele Pasqua[1(✉)] and Marino Miculan[2]

[1] Department of Computer Science, University of Verona, Verona, Italy
michele.pasqua@univr.it
[2] Department of Mathematics, Computer Science and Physics, University of Udine, Udine, Italy
marino.miculan@uniud.it

**Abstract.** We address the problem of *enforcing global invariants*, i.e., system-level properties, in Collective Adaptive Systems, such as distributed and decentralized Internet of Things (IoT) solutions. In particular, we propose a novel approach adopting *Attribute-based memory Updates* (AbU), a calculus modeling declarative, *event-driven* systems with attribute-based communication.

Our methodology leverages a combination of precise node-level scheduling and local reasoning, with *local invariants* derived from the system-level property to guarantee. This distributed and decentralized approach promotes efficient enforcing while ensuring desired system-wide behavior, without the need for a central controlling authority.

**Keywords:** Autonomic systems · Distributed verification · ECA rules

## 1 Introduction

Pervasive systems, like the *Internet of Things* (IoT), smart homes, and autonomous agents, present unique challenges due to their inherent complexity, being characterized by distributed computing, dynamic network structures, context awareness, and real-time data processing. To address this complexity, the paradigm of *event-driven programming* has emerged, since it aligns well with the reactive nature of pervasive systems which constantly interact with their environment through events like sensor readings or actuation commands [10,12].

*Event Condition Action* (ECA) languages represent an intuitive, yet powerful, paradigm for programming reactive systems. The fundamental construct of ECA languages are rules of the form "**on** *Event* **if** *Condition* **do** *Action*" which means: when *Event* occurs, if *Condition* is verified then execute *Action*. ECA systems receive inputs (as events) from the external environment and react by performing *internal* actions, updating the node's local memory, or *external* actions, which influence the environment itself. Due to their reactive nature,


T. Margaria and B. Steffen (Eds.): ISoLA 2024, LNCS 15220, pp. 351–367, 2025.
https://doi.org/10.1007/978-3-031-75107-3_21

ECA languages are well-suited for programming smart systems, in particular in IoT scenarios [10,12]. Indeed, this paradigm can be found, often under the name of *trigger-action platforms*, in various commercial IoT frameworks like IFTTT, Samsung SmartThings, Microsoft Power Automate, Zapier, to name a few.

Despite the actual underlying infrastructure of most pervasive systems (e.g., IoT solutions) is strongly centralized, the ever-growing increase of such systems size, in terms of number of components, clearly advocates for a decentralized settings, due to evident scalability issues. This poses systems like IoT solutions closer to the computation paradigm of a *Collective Adaptive System* (CAS), that is decentralized in nature. In CASs, a large number of agents autonomously interact in pursuit of an individual or collective goal, where the behavior of the system arises as an emergent property of the collective intelligence. This paradigm shift has been recently performed by the introduction of *Attribute-based memory Updates* (AbU) [17,20], a communication mechanism designed for reactive and decentralized programming, derived from Attribute-based Communication (AbC) [1,2]. In this model, nodes (e.g., IoT devices) can directly communicate with each other and self-coordinate, in a truly decentralized setting, without the need of a central entity. AbU also retains the programming simplicity of ECA rules, by casting the AbC communication model to a declarative, event-driven setting. In particular, in AbU an event on a node can cause the update of the states of (possibly many) *remote* nodes, selected "on the fly" by means of ECA rule conditions. For instance, the following rule:

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

means "when the (local) variable *login* changes, on every node whose *role* is 'logger' append my current (local) *time* to the (remote) variable *log*". Therefore, AbU allows to propagate effects to collections of nodes at once, abstracting from their identities (or even their existence), in a pure CAS fashion.

As a downside, the absence of a central entity makes challenging to enforce correctness guarantees (e.g., functional or safety properties) of the whole decentralized IoT system or to retrieve emerging properties of CASs. In this respect, AbU nodes come with *local invariants* [21], that is, properties of single nodes that are automatically enforced by AbU semantics. Nevertheless, ensuring system-wide properties for AbU is far from trivial. In this paper, we propose an approach purely based on AbU nodes local reasoning to enforce AbU *global invariants*, that is, properties over the whole AbU system. By exploiting attribute-based memory updates, such a system-level invariant is enforced without resorting to a central authority. In particular, by projecting the global invariant into an ensemble of local invariants, the enforced-by-design local invariants of AbU nodes guarantee the fulfillment of the system-wide property, when a suitable distribution of knowledge is adopted. The latter, is modeled by means of AbU memory updates, the distinctive synchronization primitive of AbU which does not hinder decentralization and node anonymity.

*Synopsis.* In Sect. 2 we recall AbU, an ECA-inspired calculus with attribute-based memory updates well suited for modeling Collective Adaptive Systems.

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

**Fig. 1.** Grammar of ECA rules in the AbU calculus.

In Sect. 3 we define system-level invariants in AbU, and we provide a syntactic transformation of AbU code adding node-level invariants that, under specific assumptions, imply the system-level one. In Sect. 4 we show that if the local schedulers respect some *priority* policies, the enforcing of node-level invariants guarantees the enforcing of a system-level invariant. Finally, Sect. 5 recalls some related work and presents directions for future work.

## 2 Attribute-Based Memory Updates and CASs

### 2.1 The AbU Calculus

AbU [17,20] 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 [3]. 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. 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.

In addition, the calculus has been extended in [21] 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.

An AbU system $\mathsf{S}$ 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: a set $R$ of ECA rules; an invariant $\iota$, 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 $v \in \mathbb{V}$; an *execution pool* $\Theta \subseteq (\mathbb{X} \times \mathbb{V})^*$, that is a set $\Theta = \{\mathsf{upd}_1, \ldots, \mathsf{upd}_n\}$ of lists of pairs of the form $((\mathrm{x}_1, v_1) \ldots (\mathrm{x}_m, v_m))$. Each list, called an *update*, represents a simultaneous multiple update waiting to be applied to the state. In the following we will denote the set of updates as

$\mathbb{U} = (\mathbb{X} \times \mathbb{V})^* = \bigcup_{i \in \mathbb{N}} (\mathbb{X} \times \mathbb{V})^n$. When a node has no invariant, that is equivalent to have $\iota = \mathtt{tt}$, we simply write $R\langle \Sigma, \Theta \rangle$ instead of $R, \mathtt{tt}\langle \Sigma, \Theta \rangle$.

An AbU rule is defined by the grammar[1] in Fig. 1. Each ECA rule $\mathsf{evt} \gg \mathsf{task}$ has an *event* evt, which is a list of resources the rule is listening on: when one of the resources in evt is modified, the rule is fired, namely rule's tasks task are evaluated. Evaluation does not change the resource states immediately; instead, it yields update operations which are added to the execution pools, and applied later on. A task consists in a condition cnd and an action act. A *condition* is a boolean expression, optionally prefixed with the modifier @: when @ is not present, the task is *local*; otherwise the task is *remote*. In local tasks, the condition is checked in the local node and, if it holds, the action is evaluated. For remote tasks, on every node where the condition holds, the action is evaluated. An action is a list of assignments of value expressions to *local* x or *remote* $\overline{\mathsf{x}}$ resources. The evaluation of an action yields an update, which is added to the current node pool in the case of local tasks; and added to remote nodes pools in the case of remote tasks.

AbU semantics is modeled as a labeled transition system $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_2$ whose labels $\alpha$ are given by the grammar $\alpha ::= T \mid \mathsf{upd} \rhd T \mid \mathsf{upd} \blacktriangleright T$. Here, $T$ is a finite list of tasks and upd is an update. 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 new updates to add to the local pool (or some pools of remote nodes), given by the activation of some rules.

The discovery phase is composed by two parts, the local and the external one. A node $R, \iota\langle \Sigma, \Theta \rangle$ performs a local discovery by means of the functions LocalUpds, that add to the local pool $\Theta$ 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[2]. External task spreading is modeled with the labels $\mathsf{upd} \rhd T$, produced by the rule (Exec), and $\mathsf{upd} \blacktriangleright T$, produced by the rule (Input). 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. Updates in the pools are consumed asynchronously, but communication between nodes is synchronous.

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

[1] The syntactic category for invariants $\iota$ will be introduced in Sect. 3.

[2] We refer to [17] for the formal definition of LocalUpds and ExtTasks.

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 [18] for some implementation details.

The AbU semantics also checks the fulfillment of invariants. Indeed, the rule (Exec) is applied only when the state modified by the update still satisfies the invariant (i.e., $\Sigma' \models \iota$); otherwise, rule (Exec-F) is applied. In this case, the update that would lead to a "bad" state is discarded and removed from the pool.

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

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

$$(\textsc{Input})\ \frac{v_1, \dots, v_k \in \mathbb{V} \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad X = \{\mathrm{x}_1, \dots, \mathrm{x}_k\} \quad \Theta' = \Theta \cup \mathsf{LocalUpds}(R, X, \Sigma') \quad T = \mathsf{ExtTasks}(R, X, \Sigma')}{R, \iota\langle \Sigma, \Theta\rangle \xrightarrow{(\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \blacktriangleright T} R, \iota\langle \Sigma', \Theta'\rangle}$$

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

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

**Fig. 2.** LTS Semantics of the AbU calculus.

## 2.2 CAS Examples in AbU

*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. We can model this scenario in AbU as follows (without the energy transfer phase, due to space reasons).

Suppose to have four rovers. For each rover we have an AbU node with a resource battery, indicating the battery level of the rover; a resource position, indicating the rover position; a resource mode, indicating in which operative state is the rover; and a resource helpPos, indicating the position of a rover that needs help. Formally, the AbU system modeling the rover-swarm scenario is

$$\mathsf{S} = R\langle \Sigma_1, \varnothing\rangle \parallel R\langle \Sigma_2, \varnothing\rangle \parallel R\langle \Sigma_3, \varnothing\rangle \parallel R\langle \Sigma_4, \varnothing\rangle$$

where $R$ contains, among the others, the following two AbU rules:

$$\mathrm{battery} \gg @(\mathrm{battery} < 5 \wedge \overline{\mathrm{battery}} > 80) : \overline{\mathrm{helpPos}} \leftarrow \mathrm{position} \quad (1)$$

$$\mathrm{helpPos} \gg (|\mathrm{position} - \mathrm{helpPos}| < 7.0) : \mathrm{mode} \leftarrow \textit{`rescue'} \quad (2)$$

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

$$\Sigma_1 = [\,\mathrm{battery} \mapsto 4 \;\; \mathrm{position} \mapsto 2.0 \;\; \mathrm{mode} \mapsto \textit{`measure'} \;\; \mathrm{helpPos} \mapsto 0.0\,]$$
$$\Sigma_2 = [\,\mathrm{battery} \mapsto 81 \;\; \mathrm{position} \mapsto 15.0 \;\; \mathrm{mode} \mapsto \textit{`measure'} \;\; \mathrm{helpPos} \mapsto 0.0\,]$$
$$\Sigma_3 = [\,\mathrm{battery} \mapsto 97 \;\; \mathrm{position} \mapsto 6.0 \;\; \mathrm{mode} \mapsto \textit{`measure'} \;\; \mathrm{helpPos} \mapsto 0.0\,]$$
$$\Sigma_4 = [\,\mathrm{battery} \mapsto 65 \;\; \mathrm{position} \mapsto 8.0 \;\; \mathrm{mode} \mapsto \textit{`measure'} \;\; \mathrm{helpPos} \mapsto 0.0\,]$$

The rule (1) says that when the current rover battery level is low (i.e., $\mathrm{battery} < 5$), then the current 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 the 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, i.e., $\mathrm{helpPos} \leftarrow 2.0$. Among all receivers, only the second and the third rovers are interested in the communication, since they are the only with battery level greater than 80. So they both add to their pool the update $(\mathrm{helpPos}, 2.0)$. This ends the discovery phase originated by the first rover.

The rule (2), instead, is fired when a rover receives a help request (i.e., when its resource helpPos changes) and basically 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: $\mathrm{mode} \leftarrow$ *'rescue'*. In the example, when the second and the third rovers execute the update $(\mathrm{helpPos}, 2.0)$, the task of the 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 first one). Instead, $|6.0 - 2.0| < 7.0$ holds and the third rover can execute the rule task, adding to its pool the update $(\mathrm{mode}, \textit{`rescue'})$.

*Smart HVAC System.* In this example, we provide an AbU implementation of a Heating, Ventilation and Air Conditioning (HVAC) system, that makes use of device invariants, namely 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:

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

while its ECA rules $R_s$ are:

$$\text{temperature} \gg (\text{temperature} < 18) : \text{heating} \leftarrow \texttt{tt} \tag{3}$$
$$\text{temperature} \gg (\text{temperature} > 27) : \text{heating} \leftarrow \texttt{ff} \tag{4}$$
$$\text{airButton} \gg (\text{airButton} = \texttt{tt}) : \text{conditioning} \leftarrow \texttt{ff} \tag{5}$$
$$\text{humidity temperature} \gg (2 + 0.5 * \text{temperature} < \text{humidity} \wedge 38 - \text{temperature} < \text{humidity}) : \text{conditioning} \leftarrow \texttt{tt} \tag{6}$$

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 $< 18$) the rule (3) activates the heating with the update: heating $\leftarrow$ tt. Instead, when the temperature is greater than $27°C$ (i.e., temperature $> 27$), then the rule (4) deactivates the heating with the update: heating $\leftarrow$ ff. The air conditioning is turned on (with the update conditioning $\leftarrow$ tt), by means of the rule (6), when the humidity exceeds the upper bound of the Givoni's *comfort zone* [16].

Execution states and ECA rules for temperature and humidity sensors are:

$$\Sigma_t = [\,\text{temperature} \mapsto 19 \quad \text{node} \mapsto \textit{'tempSens'}\,]$$
$$\Sigma_h = [\,\text{humidity} \mapsto 40 \quad \text{node} \mapsto \textit{'humSens'}\,]$$

$$R_t \triangleq \text{temperature} \gg @(\text{node} = \textit{'system'}) : \overline{\text{temperature}} \leftarrow \text{temperature} \tag{7}$$
$$R_h \triangleq \text{humidity} \gg @(\text{node} = \textit{'system'}) : \overline{\text{humidity}} \leftarrow \text{humidity} \tag{8}$$

The rule (7) on the temperature sensor device is simply 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'*; while the rule (8) do the same for the resource humidity on the humidity sensor device.

The HVAC control system is also bestowed with a physical button for manually stopping the air conditioning. Indeed, the rule (5) stops the air conditioning (with the update conditioning $\leftarrow$ ff) when the button is pressed (i.e., airButton $=$ tt). Finally, by means of the invariant[3]

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

on the HVAC control system device we specify that no update can result in the activation of both heating and air conditioning simultaneously. The complete AbU system is $R_s, \iota_s\langle\Sigma_s, \varnothing\rangle \parallel R_t\langle\Sigma_t, \varnothing\rangle \parallel R_h\langle\Sigma_h, \varnothing\rangle$.

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] The formal definition of invariants will be introduced in Sect. 3.

## 3 Distributed Verification of System-Level Invariants

We model *local invariants* as systems of linear inequalities over AbU expressions $\varepsilon$, where expressions can only range over the resources of the node the invariants belong to. That is, invariants $\iota$ are defined by the following grammar.

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

Here expressions $\varepsilon$ cannot contain remote resource lookup $\overline{\mathrm{x}}$. We can express inequalities between arbitrary expressions with simple syntactic refactoring. For instance, we can express $\varepsilon_1 < \varepsilon_2$ as $\varepsilon_1 - \varepsilon_2 < 0$. Similarly, we can express equality by negating an inequality. For instance, we can express $\varepsilon_1 = \varepsilon_2$ as $\neg(\varepsilon_1 - \varepsilon_2 \neq 0)$.

In the following, we assume to have boolean, numeric (integers and decimals) and string values in $\mathbb{V}$, and to have basic arithmetic operations (addition, subtraction, multiplication and division) over numeric values, as well as basic operations (length, substring and concatenation) over string values. Comparison operators $<$ and $\leq$ are defined for numeric values only (with the usual semantics), while $\neq$ is defined for all values.

*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. 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. Nevertheless, 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.

*Smart HVAC System Revisited.* Suppose to modify the HVAC example of Subsect. 2.2, 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:

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

The ECA rules $R_t$ for the temperature sensor node are:

$$\mathrm{temperature} \gg (\texttt{tt}) : \overline{\mathrm{temperature}} \leftarrow \mathrm{temperature} \tag{9}$$
$$\mathrm{temperature} \gg (\mathrm{temperature} < 18) : \mathrm{heating} \leftarrow \texttt{tt} \tag{10}$$
$$\mathrm{temperature} \gg (\mathrm{temperature} > 27) : \mathrm{heating} \leftarrow \texttt{ff} \tag{11}$$

The ECA rules $R_h$ for the humidity sensor node are:

$$\text{airButton} \gg (\text{airButton} = \texttt{tt}) : \text{conditioning} \leftarrow \texttt{ff} \tag{12}$$

$$\text{humidity temperature} \gg (2 + 0.5 * \text{temperature} < \text{humidity} \wedge 38 - \text{temperature} < \text{humidity}) : \text{conditioning} \leftarrow \texttt{tt} \tag{13}$$

This formulation of the problem is equivalent to the one presented in Subsect. 2.2, except for the invariant. Indeed, 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 $\text{conditioning}_h$ of the humidity node and the resource $\text{heating}_t$ of the temperature node cannot be `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.

### 3.1 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, 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 AbU remote updates for each resource in the scope of global invariants.

Algorithm 1 describes how an AbU system $\mathsf{S}$ can be modified in order to fulfill a global invariant $I$ by means of an ensemble of local invariants, added to the nodes in $\mathsf{S}$. 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 $\iota_i$ are either of the form $\mathsf{ineq}$ or $\neg\iota$. The outer loops at lines **1..2** try to add each conjunct of $I$ to each node of the system $\mathsf{S}$. 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 $\iota_j$ to the local invariant $\hat{\iota}_i$ of the $i^{\text{th}}$ node of $\mathsf{S}$. 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 $\Sigma_i$ (line

**Algorithm** `DecentralizeInvariant`$(\mathsf{S}, I)$

```
   /* 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           */
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 ∈ type(x)
               end
7              for all x in vars(ι_j) ∩ vars(Σ_i) do
8                  R_i := R_i :: x ≫ @(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.

**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 belong to the modified node (condition at line **7**) can be 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 $i^{\text{th}}$ node in S (line **8**). The added rule here is a special rule having higher priority, since synchronization updates must be considered before normal updates. A rule $\mathsf{evt} \gg \mathsf{task}$ has the same meaning of a standard AbU rule $\mathsf{evt} \gg \mathsf{task}$ except for the fact that the AbU semantics will consider it with higher priority. This would need to add a priority-based update scheduling mechanism to AbU, as we will see in Sect. 4.

Coming back to the revisited smart HVAC system introduced at the beginning of the section, by applying Algorithm 1 we obtain the following 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:

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

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

$$\text{heating} \gg @(\texttt{tt}) : \overline{\text{heating}} \leftarrow \text{heating} \tag{14}$$

is added to $\Sigma_t$ and the rule

$$\text{conditioning} \gg @(\texttt{tt}) : \overline{\text{conditioning}} \leftarrow \text{conditioning} \tag{15}$$

is added to $\Sigma_h$. These synchronization 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 the 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., $I = \neg(\text{conditioning} \wedge \text{heating}) \wedge (\text{brightness} < 125)$), then the local invariants on those nodes and the global invariant would have been different.

### 3.2 Aerial Drone Coalition

To showcase the generality of AbU 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 position, modeling the current GPS position of the drone (updated by the physics engine of the drone); a sensor destination, modeling the target GPS position the drone is aiming at (updated remotely by the control room); and an actuator movement, modeled as a GPS coordinates shift (that will instruct the physics engine on how to move the drone).

Each drone is then equipped with the following AbU rule:

$$\begin{aligned}&\text{position destination} \gg (|\text{destination} - \text{position}| \geq \delta) :\\ &\quad \text{movement} \leftarrow (\text{destination} - \text{position})/|\text{destination} - \text{position}|\end{aligned} \tag{16}$$

The rule says that each time the drone position or the destination change, if their distance is greater than a given threshold $\delta$ (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 $\Delta_i$ the distance shift, computed as in (16), for the drone $i \in [1..n]$, i.e., $\Delta_i \triangleq (\text{destination}_i - \text{position}_i)/|\text{destination}_i - \text{position}_i|$.

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

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

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

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

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

$$I_{\text{cg}} \triangleq \bigwedge_{i\in[1..n]} |(\text{position}_i + \Delta_i) - (\sum_{j\in[1..n]}^{j\neq i} \text{position}_j)/n| < l$$

In the first case $I_{gcs}$, Algorithm 1 will simply add to each node $i \in [1..n]$ the local invariant $|(\text{position}_i + \Delta_i) - G| < l$. No AbU synchronization rules will be added, since no coordination between drones is needed to fulfill the (global) invariant. In the second case $I_{ca}$, Algorithm 1 will add to each node $i \in [1..n]$ the local invariant $\bigwedge_{j\in[1..n]\setminus\{i\}} |(\text{position}_i + \Delta_i) - \text{position}_j| > t$, together with the resources in the scope of the added invariant missing from node $i$, that is, the resources $\text{position}_j$ such that $j \in [1..n] \setminus \{i\}$. Finally, the AbU synchronization rule $\text{position}_i \gg @(\texttt{tt}) : \overline{\text{position}_i} \leftarrow \text{position}_i$ will be added. The third case $I_{cg}$ is analogous to the previous one.

## 4 Priority Scheduling and Correctness Guarantees

In order to enforce a global invariant by enforcing local invariants, updates execution should respect some *priority ordering*. Coming again to the revised HVAC example, when we turn on the air cooling system and, subsequently, we turn on the heater, we should guarantee that the update $(\text{conditioning}, \texttt{tt})$ is delivered and executed on all nodes before the update $(\text{heating}, \texttt{tt})$. That is, synchronization updates must be executed with higher priority than normal updates.

AbU semantics guarantees atomicity within the discovery phase, i.e., nodes performing updates cannot be interrupted by discovery from other nodes[4]. This ensures a causal order for updates delivery (any update execution are delivered in order to all nodes), but it does not inherently guarantee a specific priority for update execution. This is because each node relies on its local scheduler to determine the execution order of the updates in its pool. This design choice in AbU promotes decoupling the theoretical model from practical implementation concerns. However, it can lead to inconsistencies when updates need to be executed under a specific priority order to maintain global system property (e.g., a global invariant). To address this, in this section we propose a scheduling strategy, with a slight modification of AbU semantics, that enforces priority ordering of update execution, ensuring consistency and local-to-global invariant enforcement.

[4] This guarantee can be realized by means of distributed transactions, as done in the implementation available at https://github.com/abu-lang.

### 4.1 A Scheduling Strategy for Priority Ordering

A simple scheduler guaranteeing priority to synchronization updates can be implemented with a slight modification of AbU semantics. In this case, a node pool is modeled as a pair of sets of updates $(\hat{\Theta}, \Theta)$, instead of a single set of updates. A *high priority pool* $\hat{\Theta}$ is devoted to contain *high priority updates*, while another pool $\Theta$ to contain normal updates (i.e., updates with low priority). To syntactically identify high priority updates, we must add to the AbU syntax *high priority ECA rules* denoted $\mathsf{evt} \gg \mathsf{task}$. The meaning of such rules is the same of standard rules, except for the fact that the updates contained in $\mathsf{task}$ will have high priority. In other words, updates resulting from high priority rules will be added to the high priority pool, while updates resulting from normal rules will be added to the normal, low priority, pool. The AbU semantics scheduler, will first execute all updates in the high priority pool, and then the remaining low priority updates in the normal pool. This is formalized by the modified AbU semantic in Fig. 3 and the modified discovery function in Fig. 4. In the latter, $\{\!|\mathsf{task}|\!\}\Sigma$ denotes the task obtained from $\mathsf{task}$ with each occurrence of a resource $\mathrm{x}$ in the task condition and the right-hand side of assignments in the task action replaced with the value $\Sigma(\mathrm{x})$ (after that, each instance of $\overline{\mathrm{x}}$ in the task action is replaced with $\mathrm{x}$ and the modifier @ is dropped).

When external updates are propagated in the network, their priority level should preserved. This is done by modifying LTS labels, that now model both high priority and normal updates. Formally, the AbU semantics with priority is modeled as a labeled transition system $\mathsf{S}_1 \overset{\alpha}{\rightarrow}_{\bullet} \mathsf{S}_2$ whose labels $\alpha$ are now of the form: $(P, T)$; $\mathsf{upd} \triangleright (P, T)$; or $\mathsf{upd} \blacktriangleright (P, T)$. Here, $(P, T)$ is a pair of finite lists of tasks (and $\mathsf{upd}$ is an update). In these labels, $P$ represents the list of high priority updates, while $T$ represents the list of low priority updates.

### 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 AbU system $\mathsf{S}$, the AbU system $\mathsf{S}_\ell = \texttt{DecentralizeInvariant}(\mathsf{S}, I)$ is guaranteed to not violate $I$, for all its possible executions. This is done by enforcing all local invariants in $\mathsf{S}_\ell$. We assume that $\mathsf{S}_\ell$ preserves the semantics of the original system $\mathsf{S}$ up-to the added synchronization resources. Indeed, the ECA rules added by Algorithm 1 do not affect the resources of the original system; more formally, it is possible to prove that Algorithm 1 is semantics-preserving by defining a suitable hiding bismulation [19,21] masquerading high priority updates.

**Theorem 1. (Local Invariants Soundness).** *Let $\mathsf{S}_\ell$ be a system obtained from an AbU system $\mathsf{S}$ by decentralizing the invariant $I$ as per Algorithm 1. If $\mathsf{S}_\ell$ satisfies $I$, then for all $\mathsf{S}'$ reachable from $\mathsf{S}_\ell$, $\mathsf{S}'$ satisfies $I$.*

$$(\textsc{ExecP})\ \frac{\mathsf{upd} \in \hat{\Theta} \quad \mathsf{upd} = (\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad \Sigma' \models \iota \\ X = \{\mathrm{x}_i \mid i \in [1..k] \wedge \Sigma(\mathrm{x}_i) \neq \Sigma'(\mathrm{x}_i)\} \quad \mathsf{LocalUpds}(R, X, \Sigma') = (\hat{\Theta}'', \Theta'') \\ \hat{\Theta}' = (\hat{\Theta} \setminus \{\mathsf{upd}\}) \cup \hat{\Theta}'' \quad \Theta' = \Theta \cup \Theta'' \quad \mathsf{ExtTasks}(R, X, \Sigma') = (P, T)}{R, \iota\langle \Sigma, (\hat{\Theta}, \Theta)\rangle \xrightarrow{\mathsf{upd} \rhd (P,T)}_{\bullet} R, \iota\langle \Sigma', (\hat{\Theta}', \Theta')\rangle}$$

$$(\textsc{Exec})\ \frac{\hat{\Theta} = \varnothing \quad \mathsf{upd} \in \Theta \quad \mathsf{upd} = (\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad \Sigma' \models \iota \\ X = \{\mathrm{x}_i \mid i \in [1..k] \wedge \Sigma(\mathrm{x}_i) \neq \Sigma'(\mathrm{x}_i)\} \quad \mathsf{LocalUpds}(R, X, \Sigma') = (\hat{\Theta}'', \Theta'') \\ \hat{\Theta}' = \hat{\Theta} \cup \hat{\Theta}'' \quad \Theta' = (\Theta \setminus \{\mathsf{upd}\}) \cup \Theta'' \quad \mathsf{ExtTasks}(R, X, \Sigma') = (P, T)}{R, \iota\langle \Sigma, (\hat{\Theta}, \Theta)\rangle \xrightarrow{\mathsf{upd} \rhd (P,T)}_{\bullet} R, \iota\langle \Sigma', (\hat{\Theta}', \Theta')\rangle}$$

$$(\textsc{ExecP-F})\ \frac{\mathsf{upd} \in \hat{\Theta} \quad \mathsf{upd} = (\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad \Sigma' \not\models \iota \quad \hat{\Theta}' = \hat{\Theta} \setminus \{\mathsf{upd}\}}{R, \iota\langle \Sigma, (\hat{\Theta}, \Theta)\rangle \xrightarrow{\mathsf{upd} \rhd (\epsilon,\epsilon)}_{\bullet} R, \iota\langle \Sigma, (\hat{\Theta}', \Theta)\rangle}$$

$$(\textsc{Exec-F})\ \frac{\hat{\Theta} = \varnothing \quad \mathsf{upd} \in \Theta \quad \mathsf{upd} = (\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \\ \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad \Sigma' \not\models \iota \quad \Theta' = \Theta \setminus \{\mathsf{upd}\}}{R, \iota\langle \Sigma, (\hat{\Theta}, \Theta)\rangle \xrightarrow{\mathsf{upd} \rhd (\epsilon,\epsilon)}_{\bullet} R, \iota\langle \Sigma, (\hat{\Theta}, \Theta')\rangle}$$

$$(\textsc{Input})\ \frac{v_1, \dots, v_k \in \mathbb{V} \quad \Sigma' = \Sigma[v_1/\mathrm{x}_1 \dots v_k/\mathrm{x}_k] \quad X = \{\mathrm{x}_1, \dots, \mathrm{x}_k\} \quad \mathsf{ExtTasks}(R, X, \Sigma') = (P, T) \\ \mathsf{LocalUpds}(R, X, \Sigma') = (\hat{\Theta}'', \Theta'') \quad \hat{\Theta}' = \hat{\Theta} \cup \hat{\Theta}'' \quad \Theta' = \Theta \cup \Theta''}{R, \iota\langle \Sigma, (\hat{\Theta}, \Theta)\rangle \xrightarrow{(\mathrm{x}_1, v_1) \dots (\mathrm{x}_k, v_k) \blacktriangleright (P,T)}_{\bullet} R, \iota\langle \Sigma', (\hat{\Theta}', \Theta')\rangle}$$

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

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

**Fig. 3.** LTS Semantics of the AbU calculus with priority.

**Proof.** Let us consider a (possibly infinite) sequence of transitions $\mathsf{S}_\ell \xrightarrow{\alpha_1}_{\bullet} \mathsf{S}_1 \xrightarrow{\alpha_2}_{\bullet} \mathsf{S}_2 \dots$. If the label $\alpha_1$ does not contain any high priority task (i.e., $\alpha_1$ is of the form $\mathsf{upd} \rhd (\epsilon, T)$), it means that the update executed at the first step does not involve any variable of the invariant $I$, because the only high priority updates are those generated by the rules added by Algorithm 1. Hence $I$ is still valid for $\mathsf{S}_1$; in this case, we can repeat the argument starting from $\mathsf{S}_1$.

Let us consider the case when $\alpha_1$ contains some high priority task, i.e., $\alpha_1$ is of the form $\mathsf{upd} \rhd (P, T)$ with $P$ non-empty; this means that some variable of $I$ has been modified at the first step. Let $\mathrm{x}$ one of these variables, and let us consider any node $R, \iota \wedge \iota_\ell\langle , , (\hat{\Theta}, \Theta)\rangle$ in $S_1$ where $\iota_\ell$ is the part of invariant added by Algorithm 1. Clearly, its high priority pool $\hat{\Theta}$ contains $(\mathrm{x}, v)$, to update the local copy of $\mathrm{x}$ with the new value. This update can be executed by (ExecP) because it does not violate neither the local invariant $\iota$ (since $\mathrm{x}$ has been freshly added to the store $\Sigma$) nor $\iota_\ell$ (otherwise the update would have violated the invariant on the originating node at $\mathsf{S}_\ell$ already). Therefore, the update $(\mathrm{x}, v)$ is executed before any update in $\Theta$ (due to priority), yielding to state $\mathsf{S}_1$, where the invariant still holds. The same argument can be repeated until all updates in all high priority pools are executed; this leads to some state $\mathsf{S}_n$, where there

$$\mathsf{LocalUpds}(R, X, \Sigma) \triangleq (\hat{\Theta}, \Theta) \text{ such that } \hat{\Theta} = \{[\![\mathsf{act}]\!]\Sigma \mid \exists \mathsf{evt} \gg \varphi : \mathsf{act} \in \mathsf{Active}(R, X) \,.\, \Sigma \models \varphi\}$$
$$\text{and } \Theta = \{[\![\mathsf{act}]\!]\Sigma \mid \exists \mathsf{evt} \gg \varphi : \mathsf{act} \in \mathsf{Active}(R, X) \,.\, \Sigma \models \varphi\}$$

$$\mathsf{ExtTasks}(R, X, \Sigma) \triangleq (\{\!|\hat{\mathsf{task}}_1|\!\}\Sigma \ldots \{\!|\hat{\mathsf{task}}_n|\!\}\Sigma, \{\!|\mathsf{task}_1|\!\}\Sigma \ldots \{\!|\mathsf{task}_m|\!\}\Sigma) \text{ given that:}$$
$$\forall i \in [1..n]\, \exists \mathsf{evt} \gg \hat{\mathsf{task}}_i \in \mathsf{Active}(R, X) \,.\, \mathsf{task}_i = @\varphi : \mathsf{act}$$
$$\forall j \in [1..m]\, \exists \mathsf{evt} \gg \mathsf{task}_j \in \mathsf{Active}(R, X) \,.\, \mathsf{task}_j = @\varphi : \mathsf{act}$$

$$\mathsf{Active}(\mathsf{rule}_1 \ldots \mathsf{rule}_n, X) \triangleq \{\mathsf{rule}_i \mid \exists i \in [1..n] \,.\, (\mathsf{rule}_i = \mathsf{evt} \gg \mathsf{task} \vee \mathsf{rule}_i = \mathsf{evt} \gg \mathsf{task}) \wedge \mathsf{evt} \cap X \neq \varnothing\}$$

**Fig. 4.** Discovery functions for the AbU calculus semantics with priority.

are no further synchronization updates to apply, and invariant $I$ still holds. Now we can repeat the argument starting from $\mathsf{S}_n$, proving the thesis. □

## 5 Conclusion

In this paper, we presented a novel approach to enforce global invariants, i.e., system-level properties, of Collective Adaptive Systems such as decentralized IoT solutions. Global invariants stipulate correctness requirements (e.g., functional or safety properties) that must hold for all components of the distributed system. This is particularly challenging in CASs, due to the lack of a central authority. To overcome the problem, we leveraged AbU to combine local node reasoning with an efficient distribution mechanism, which dispenses from central entities and node identity. The former corresponds to AbU node invariants, while latter corresponds to AbU remote updates. In particular, we provided a syntactic translation of AbU systems in order to embed a global system invariant into an ensemble of local node invariants. By assuming a suitable synchronization mechanism based on a prioritized scheduling of updates execution, the enforcement mechanism of local invariants provided by the AbU semantics guarantees the fulfillment of the global, system-level invariant.

*Related Work.* The literature about distributed systems verification is consistent, starting from the classic work of Chandy and Lamport [13], Babaoglu and Raynal [7] to more recent approaches; see Francalanza et al. [15] for a survey. Most such approaches consider a centralized scenario, or assume a shared knowledge of the underlying system topology between nodes, thus do not fit the CAS scenario. Little has been done in the CAS community about (decentralized) verification. Aldrini [4] proposes a framework for designing and verifying trust within CAS, but global/local invariants of the system are not considered. Audrito et al. [6] argues that aggregate computing is particularly well-suited for distributed runtime verification in the context of CAS. Aggregate computing is a programming paradigm that views a collection of devices as a single computational unit. It abstracts away the specifics of individual devices and their locations, focusing on the overall computational process. However, this model is difficult to apply in scenarios where different devices are assigned with specific

tasks and are supposed to fulfill specific constraints (e.g., in many IoT applications). Finally, Bortolussi et al. [11] propose CARMA, a framework designed for modeling the dynamic and adaptive behavior of CASs, allowing for analysis of the system's performance and identification of potential implementation issues. Nevertheless, CARMA does not allow to define global/local invariants for the system.

*Future work* We plan to investigate how the proposed approach can be generalized to enforce program aspects that go beyond invariants, such as liveness properties [5] (e.g., fairness) or hyperproperties [14] (e.g., confidentiality leaks). Another possible line of work may include the weakening of the AbU semantics, by relaxing the atomicity constraint of updates distribution. This may result in different scheduling policies needed to guarantee the relation between global and local invariants discussed in Section 4. Finally, it would be interesting to consider also quantitative aspects in the semantics, along the lines of [8,9].

**Acknowledgments.** This research has been partially supported by the Department Strategic Project on Artificial Intelligence of the University of Udine (2020-25), and the project SERICS (PE00000014) under the NRRP MUR program funded by EU-NGEU.

## References

1. 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)
2. 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
3. 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)
4. Aldini, A.: Design and verification of trusted collective adaptive systems. ACM Trans. Model. Comput. Simul. **28**(2) (2018). https://doi.org/10.1145/3155337
5. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21**(4), 181–185 (1985). https://doi.org/10.1016/0020-0190(85)90056-0
6. 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
7. Babaoglu, O., Raynal, M.: Specification and verification of dynamic properties in distributed computations. Journal of Parallel and Distributed Computing **28**(2), 173–185 (1995). https://doi.org/10.1006/jpdc.1995.1098
8. Bacci, G., Miculan, M.: Structural operational semantics for continuous state probabilistic processes. In: Proceedings of Coalgebraic Methods in Computer Science (CMCS). Lecture Notes in Computer Science, vol. 7399, pp. 71–89. Springer (2012). https://doi.org/10.1007/978-3-642-32784-1_5
9. 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

10. Balliu, M., Merro, M., Pasqua, M., Shcherbakov, M.: Friendly fire: Cross-app interactions in IoT platforms. ACM Trans. Priv. Secur. **24**(3) (2021), https://doi.org/10.1145/3444963
11. 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. p. 16–31 (2015). https://doi.org/10.4204/eptcs.194.2
12. 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, USA (2014), https://doi.org/10.1109/SASOW.2014.32
13. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of a distributed system. ACM Transactions on Computer Systems pp. 63–75 (1985)
14. Clarkson, M., Schneider, F.: Hyperproperties. In: 21st IEEE Computer Security Foundations Symposium. pp. 51–65 (2008). https://doi.org/10.1109/CSF.2008.7
15. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems, pp. 176–210. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
16. Givoni, B.: Comfort, climate analysis and building design guidelines. Energy and Buildings **18**(1), 11–23 (1992)
17. 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 (2021). https://doi.org/10.1007/978-3-030-85315-0_21
18. 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
19. Pasqua, M., Miculan, M.: On the security and safety of AbU systems. In: Proc. 19th SEFM. Lecture Notes in Computer Science, vol. 13085, pp. 178–198. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_11
20. Pasqua, M., Miculan, M.: AbU: A calculus for distributed event-driven programming with attribute-based interaction. Theoretical Computer Science pp. 1–32 (2023). https://doi.org/10.1016/j.tcs.2023.113841
21. Pasqua, M., Miculan, M.: Behavioral equivalences for AbU: Verifying security and safety in distributed IoT systems. Theoretical Computer Science pp. 1–32 (2024). https://doi.org/https://doi.org/10.1016/j.tcs.2024.114537