IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 November 2022, accepted 9 December 2022, date of publication 16 December 2022, date of current version 27 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3230287

METHODS

The AbU Language: IoT Distributed Programming Made Easy

MICHELE PASQUA[1], MASSIMO COMUZZO[2], AND MARINO MICULAN[2]
[1]Department of Computer Science, University of Verona, 37134 Verona, Italy
[2]Department of Mathematics, Computer Science and Physics, University of Udine, 33100 Udine, Italy

Corresponding author: Michele Pasqua (michele.pasqua@univr.it)

This work was supported by the Italian MIUR Project PRIN 2017FTXR7S IT MATTERS (Methods and Tools for Trustworthy Smart Systems).

ABSTRACT Event-driven programming based on *Event-Condition-Action (ECA)* rules allows users to define complex automation routines in a simple, declarative way; for this reason, in recent years ECA rules have been adopted by the majority of companies in the *Internet of Things (IoT)* industry as a promising paradigm for implementing ubiquitous and pervasive systems. Unfortunately, programming simplicity comes to a price: most implementations of ECA rules are bound to a strongly centralized communication infrastructure, that poses serious limitations on the application scenarios for the IoT, due to scalability, security and availability issues. To mitigate these issues, recent works introduced abstractions for communication and coordination of ensembles of IoT devices in a *decentralized* setting, effectively moving the computation towards the *edge* of the network without sacrificing the programming simplicity prerogative of ECA rules. In particular, *Attribute-based memory Updates* is a communication model transparently enhancing ECA rules-based systems with an interaction mechanism where communication is similar to broadcast but actual receivers are selected on the spot, by means of predicates (i.e., properties) over devices attributes. In this paper, we introduce AbU-dsl, a *Domain Specific Language for the IoT* that compiles on top of an implementation of Attribute-based memory Updates. In this way, AbU-dsl provides a practical development interface, based on ECA rules, to effectively program IoT devices in a fully decentralized setting, by exploiting a full-fledged attribute-based interaction model. Thus, programmers can specify interactions between devices in a declarative way, abstracting from details such as devices identity, number, or even their existence, without the need for a central controlling service.

INDEX TERMS IoT programming, ECA rules, attribute-based memory updates, distributed systems, edge computing, domain specific language.

I. INTRODUCTION

The *Internet of Things (IoT)* is nowadays an integral part of our daily life, and it is composed by many computational objects communicating with each other. These objects, usually called *smart devices*, can interact with the physical environment (by means of sensors and actuators), with users and with other computational systems. Indeed, the peculiarity of smart devices is their *computational power*: they are not just data collectors, but they can elaborate the collected data and send it to external entities.

The associate editor coordinating the review of this manuscript and approving it for publication was Jad Nasreddine.

Event-driven programming has emerged as the prominent paradigm for the development of IoT smart devices [1], [2]; indeed, this paradigm can be found in various commercial IoT frameworks like IFTTT, Samsung SmartThings, Microsoft Power Automate, Zapier, Google Home, etc. In this approach, the behavior of a smart device is defined by some rules (called also ''applets'', ''zaps'', ''routines'', ''flows'', and so on) adopting the *Event Condition Action (ECA)* programming style:

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

This rule pattern basically means: when *Event* occurs, if *Condition* is verified, then execute *Action*. Thus, an ECA rules-based smart device can react to an event (e.g., a variable

change or a signal from a sensor) by executing one or more actions, that can update the internal state of the device or act on the environment via an actuator. Of course, the effect caused by the rule action can trigger other rules, and so on.

However, in most current models and implementations of this paradigm, the ECA rules are stored on, and executed by, a central computing entity, possibly hosted on some server on the *cloud* (as in IFTTT, Samsung SmartThings, Microsoft Power Automate, Zapier and Google Home) and accessible via the Internet. Thus, the components of the IoT system cannot directly communicate: the coordination between devices is demanded to the central node/service.

Although simple, such a centralized architecture inherently affects crucial aspects like scalability, availability, privacy and security. Indeed, the increase of IoT smart devices (think of *smart cities*, *smart farming*, etc..) is going to produce a massive amount of data [3], and transferring, storing and processing this data in the cloud will overload servers and network channels. As a consequence, cloud servers will not be able to guarantee acceptable transfer rates and response times; moreover, sending all this data back and forth on the network is a big waste of energy. The dependence on Internet connections and on a central node/service hinders availability, which is also a critical requirement for many IoT and other pervasive and autonomic applications; e.g., a smart door lock may be stuck because the server is not reachable. Smart devices often deal with sensible and personal data (like health sensors, surveillance cameras, etc.) that the user would prefer to do not share with some untrusted server on the cloud; in fact, the need for a service in the cloud raises concerns about *data sovereignty*. Finally, the dependence on external services increases the attack surface of an IoT system; e.g., an attacker can open a house front door, taking advantage of some vulnerability on the server communicating with the door, and unknown to the user.

In this context, a natural evolution of the IoT would bring the underlying infrastructure closer to the so-called *edge computing* paradigm [3], [4], which aims to move the computation away from cloud data centers towards the "edge" of the network, i.e., the smart devices which are the sources of data. This approach allows to mitigate the previously mentioned issues, as it reduces data transfers between the edge and the center of the network —in fact, there can be no center at all.

At the same time, placing the application logic on many devices in a truly distributed and decentralized setting introduces new issues and challenges. In particular, it requires suitable mechanisms and abstractions for communication and coordination of (possibly large) ensembles of distributed components. In this respect, *Attribute-based memory Updates (AbU)* [5] is a time-coupled, space-uncoupled interaction model recently introduced for coordinating large numbers of components which are not supposed to have a global knowledge of the system, in the spirit of Attribute-based Communication (AbC) [6], [7]. The latter is a loosely coupled message-oriented interaction model specifically designed for coordinating large numbers of components. The key aspect of Attribute-based Communication is that the actual receivers are selected "on the spot" by means of predicates over node attributes; dually, a node can "filter" incoming messages by means of predicates. For instance, in AbC we have primitives to express "send (the value of) expression e to all nodes satisfying predicate Π" and, dually, "receive the message x when it satisfies the predicate Π". Many interaction models, such as channels, agents, pub/sub, broadcast and multicast, can be readily implemented by using Attribute-based Communication [7], [8].

In [5], the authors introduced the *AbU calculus*, a formal model aiming to integrate ECA programming with Attribute-based Communication. In this calculus, interactions are reduced to events of the same kind ECA programs already deal with, i.e., *memory updates*. More precisely, an AbU system is composed by a set of agents, called *nodes*, each endowed with a local memory (representing the local variables, sensors and actuators) and a set of ECA rules. When a rule of the form $x_1, \ldots, x_n \gg \Pi : \mathsf{act}$ is triggered on a given node due to a change in some local variable x_i, the action act can update the memory of that node (like in normal ECA programming), but it can also update the memory of other nodes, selected upon their memories by the predicate Π. For instance, an AbU rule like the following:

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

means "when (my local) variable *accessTime* changes, append its value to the variable *log* of all nodes whose variable *role* has value `logger`". Clearly, the update of *log* may trigger other rules on these (remote) nodes, and so on.

In this paper, we introduce AbU-dsl, a *Domain Specific Language (DSL)* based on the AbU calculus of [5]. This results in a new programming language merging the simplicity of ECA rules with distributed (decentralized) coordination and communication mechanisms, in the spirit of Attribute-based Communication. Hence, AbU-dsl can serve as a reference language for programming IoT smart devices, in an easy yet powerful way.

For instance, the AbU rule introduced above can be encoded in AbU-dsl as follows:

```
rule LogAccess
  on accessTime
  for all (ext.role == "logger")
     do ext.log = ext.log :: this.accessTime
```

that is very concise and easy to understand.

Paper Structure: In Section II we introduce AbU-dsl, a new ECA-inspired DSL extended with Attribute-based memory Updates. Its syntax and operational semantics are presented in Subsections II-A and II-B, respectively. In Section III we use AbU-dsl in some practical IoT scenarios. In Section IV we discuss how the DSL is compiled to a target language and deployed on a target architecture. Finally, in Section V we draw some conclusions, discuss related work and outline some future directions.

II. THE AbU LANGUAGE

AbU-dsl is a Domain Specific Language for the Internet of Things, particularly suitable for autonomic systems. It follows the Event Condition Action programming paradigm and, at the same time, it can be deployed in a truly distributed and decentralized network. Devices interaction is implemented by means of *Attribute-based memory Updates* [5], which is the memory-based counterpart of Attribute-based Communication [6]. In particular, AbU-dsl builds upon the *AbU calculus* [5], the archetypal calculus with Attribute-based memory Updates.

In the rest of the section, in order to present more concisely the DSL syntax and semantics, we adopt the following notation. The expression $(x)^*$ means zero or more repetitions of x, while $(x)^+$ means one or more repetitions of x. The expression $[x]$ represents an optional occurrence of x, while $x \mid y$ stands for either x or y. The symbol $\triangleq$ stands for "is defined as".

A. SYNTAX

An AbU-dsl *program* consists in a non-empty list of IoT devices, preceded by a (possibly empty) list of type declarations and followed by a (possibly empty) list of ECA rules. The syntax of AbU-dsl programs is given in Figure 1. We suppose to have a denumerable set Identifiers of identifiers (i.e., alphanumeric non-quoted strings), ranged over by Id, DevId and RuleId.

Each *device* Dev is equipped with some ECA rules that act on the device. These ECA rules are specified by means of *rule references* Refs (i.e., a non-empty list $(\text{RuleId})^+$ of rule identifiers) written after the optional keyword **has**. The latter is optional since a node may have no specific rule acting on it (e.g., an actuator that can only be changed by external devices). Furthermore, a device has a unique identifier DevId, a description Descr (i.e., a quoted string describing the device functionalities), a *resources declaration* ResDecl (i.e., a non-empty list $(\text{PhysRes} \mid \text{LogRes} \mid \text{CompRes})^+$ of internal resources) and an *invariant* Bexp (that is, a boolean expression), written after the optional keyword **where**, that resources have to fulfill (if present).

As an example, the following AbU-dsl code:

```
watering : "A simple device managing a water pump" {
  # Resources declaration.
  physical input decimal moisture
  ...
  where
  # Device invariant.
  moistMinLevel <= moistMaxLevel
} has openValve closeValve
```

defines a device called `watering` that manages a water valve, in order to maintain the soil moisture level within an given interval (`moistMinLevel` and `moistMaxLevel`). The latter can be modified at run-time by the user, so the invariant `moistMinLevel <= moistMaxLevel` assures that only valid intervals can be specified. Furthermore, the device is equipped with two ECA rules `openValve` and `closeValve`, that open and close the water valve, respectively.

A *resource* can be physical, logical or compound. A *physical resource* PhysRes can be used either as input (**input**), modeling a sensor; or as output (**output**), modeling an actuator. An input resource is supposed to be read-only, an output resource is supposed to be write-only and a *logical resource* LogRes has no constraints. Logical resources can be used as local variables. Logical and physical output resources have to be declared with an initialization value Val, while physical input resources do not. All physical and logical resources have a name Id and a *primitive* type PType, that can be either: **boolean**, for a boolean resource, e.g., `true` or `false`; **integer**, for an integer resource, e.g., `42` or `-42`; **decimal**, for a decimal resource, e.g., `3.14` or `-3.14`; or **string**, for a string resource, e.g., `"sTr1nG"`. Elements $\text{Val} \in \text{Values}$ belong to primitive types.

A *compound resource* CompRes is a structured object, whose schema (i.e., the description of the object fields) is defined in a type declaration. A *type declaration* TypeDecl reserves an identifier CType, written after the keyword **define**, to the new compound type, and its schema is enclosed between curly brackets, after the keyword **as**. In particular, a *schema* FieldDecl is a non-empty list of field declarations, that can be: **physical input** PType, for physical input fields; **physical output** PType, for physical output fields; and **logical** PType, for logical fields.[1] When compound resources are declared (in a device), their fields that require initialization must be provided with a value (at declaration time). In particular, compound resources are initialized by a constructor Const, that is a list of comma-separated field initializations, of the form $\text{Id} = \text{Val}(, \text{Id} = \text{Val})^*$. A constructor may be empty, when no field requires initialization (i.e., **physical input** fields). Compound resource fields are accessed *à la* Python, namely by using square brackets.

As an example, the following AbU-dsl code:

```
define PresenceSensor {
  mode : physical output string
  movement : physical input boolean
}
```

defines a compound resource having two fields, `mode` and `movement`. The first requires an initialization value, while the second does not, hence a resource `prSen` of type `PresenceSensor` can be initialized with the constructor `PresenceSensor prSens = (mode = "idle")`, inside a node resource declaration. The field `movement` can be accessed by writing `prSens[movement]`.

Each *ECA rule* Rule has a unique identifier RuleId, that can be referred by multiple devices. Rules are guarded by an *event* Evt, which is a non-empty list of simple resources or compound resource fields. Note that, Evt represents a list of resources/fields that the rule continuously check for changes. When one of the resources/fields is updated with a new value, then one (or more) task, among those in the tasks list specified

[1] From these declarations, we cannot have nested compound resources.

```
Programs ∋ Prg ::= (TypeDecl)* (Dev [has Refs])+ (Rule)*
TypeDeclarations ∋ TypeDecl ::= define CType as { (FieldDecl)+ }
FieldDeclarations ∋ FieldDecl ::= Id : (physical input PType | physical output PType | logical PType)
PrimitiveTypes ∋ PType ::= boolean | integer | decimal | string
Devices ∋ Dev ::= DevId : Descr { ResDecl [where Bexp] }
ResourceDeclarations ∋ ResDecl ::= (PhysRes | LogRes | CompRes)+
PhysicalResources ∋ PhysRes ::= physical output PType Id = Val | physical input PType Id
LogicalResources ∋ LogRes ::= logical PType Id = Val
CompoundResources ∋ CompRes ::= CType Id = ( [Const] )
Constructors ∋ Const ::= Id = Val | Const, Const
Rules ∋ Rule ::= rule RuleId on Evt (Task)+
Events ∋ Evt ::= Id | Id[Id] | Evt Evt
Tasks ∋ Task ::= for [all] Cnd do Act
Actions ∋ Act ::= Assign | Act, Act
```

FIGURE 1. Syntax for single devices and ensembles of devices.

after the rule event, may be executed, depending on a given condition contained in the task.

As an example, the following AbU-dsl code:

```
rule nightAlarm
  on light prSens[movement]
```

defines a rule named `nightAlarm` that is activated when either the resource `light` or the field `movement` of the (compound) resource `prSens` is modified.

In a *task* Task, an action is performed when the condition Cnd is true. The optional modifier **`all`** means that the task may act on remote (external) devices. In this case, the condition and the action in the task may reference resources on external devices, by prefixing them with the **`ext`**. keyword. When the modifier **`all`** is omitted, the condition and the action are considered on the local (current) device only. With task conditions, the **`all`** modifier and local/remote resource access we can express Attribute-based memory Updates [5].

An *action* Act is a non-empty comma-separated list of assignments on different resources/fields. An *assignment* Assign can be on the local device **`this`**.Id = Exp or on a remote device **`ext`**.Id = Exp (this applies also for compound resources, with local **`this`**.Id[Id] = Exp and remote **`ext`**.Id[Id] = Exp field assignments). Notice that, on each node the assignments of an action are executed simultaneously as a single step, not sequentially.

As an example, the following AbU-dsl code:

```
rule nightAlarm
  on light prSens[movement]
  for all (ext.node == "alarmRing")
     do ext.ring = true, this.status = "alarm"
```

defines the behavior of the rule `nightAlarm` previously introduced. In particular, all devices tagged `alarmRing` are forced to ring the alarm (via a remote update **`ext`**.`ring = true`), while the current device changes status (via the local update **`this`**.`status = "alarm"`).

Expressions Exp can be numeric expressions Nexp, boolean expressions Bexp or string expressions Sexp. They comprises logical connectives (**`and`**, **`or`**, **`not`**), arithmetic operators (**`+`**, **`-`**, **`*`**, **`/`**), strings concatenation (**`::`**), and comparison operators (**`==`**, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**). We also have built-in modulo (**`%`**) and absolute value (**`absint`**, for integers, and **`absdec`**, for decimals) operators. The standard operators/connectives composition priority can be overridden by using round brackets.

An access to the resource Id on the local device can be made by writing **`this`**.Id (or just Id). Similarly, an access on the field of a local compound resource can be made by writing **`this`**.Id[Id] (or just Id[Id]). Conditions Cnd are boolean expressions that can access a resource Id on remote devices, by writing **`ext`**.Id, or by writing **`ext`**.Id[Id] in the case of compound resource fields access.

AbU-dsl is provided with *foreign functions*, allowing code written in the compilation target language (see Section IV for details) to be used inside AbU-dsl expressions. The syntax for foreign functions is **`foreign`**(FuncName, ParamLst), where FuncName is a quoted string containing the name of the function to call and ParamLst is a, possibly empty, comma-separated list of AbU-dsl identifiers or values. Note that, foreign functions are assumed to be *pure* w.r.t. AbU-dsl, so that they cannot introduce side-effects in the caller AbU-dsl program. The return value of foreign functions should be compatible with an AbU-dsl primitive type.

As an example, the following AbU-dsl code:

```
rule computeVolume
  on radius, height
  for (true)
     do volume = foreign("math.Pow", radius, 2) *
                 height * foreign("math.Pi")
```

defines a rule that computes the volume of a cylinder when either its radius or its height change. The code make use of the

Golang `math` library to retrieve the value of π and to compute exponentiation. Note that, we could had called a user-defined Golang function that directly computes the volume of the cylinder, using a single **`foreign`** instance.

Finally, comments can be inserted inline, after the keyword `#`, while multi-line comments are enclosed within the delimiters `\@ ... @\`.

Detailed examples of AbU-dsl programs will be provided in Section III, while a complete presentation of the syntax can be found on the DSL GitHub repository.[2]

1) SYNTACTIC SUGAR

To ease code writing, we can define the following macros on top of the previously defined syntactic constructs. With:

`rule` RuleId **`on`** Evt **`default`** Act (Task)*

we denote a rule that has *default* code Act to execute when the event happens, independently from tasks condition. This is a shorthand for:

`rule` RuleId **`on`** Evt **`for`** `true` **`do`** Act (Task)*

Similarly, classic *if-then-else* conditional statements can be encoded with ECA rules that have the following form:

`rule` RuleId **`on`** Evt
  `for` Cnd **`do`** Act_1 **`for not`** Cnd **`do`** Act_2
`rule` RuleId **`on`** Evt
  `for all` Cnd **`do`** Act_1 **`for all not`** Cnd **`do`** Act_2

In this case, we can write the latter in a simpler way:

`rule` RuleId **`on`** Evt **`for`** Cnd **`do`** Act_1 **`owise`** Act_2
`rule` RuleId **`on`** Evt **`for all`** Cnd **`do`** Act_1 **`owise`** Act_2

Finally, we can easily introduce a *let* construct to simplify rules coding. Indeed, a rule like the following:

`rule` RuleId **`on`** Evt
  `let` $\mathsf{Id}_1 := \text{Exp}_1 ; \ldots ; \mathsf{Id}_n := \text{Exp}_n$ **`in`** $(\text{Task})^+$

where $\mathsf{Id}_1, \ldots \mathsf{Id}_n$ are fresh resource identifiers, can be pre-processed substituting for each $i \in [1..n]$ the occurrences of Id_i in $(\text{Task})^+$ with the corresponding expression Exp_i. Then, **`let`** $\mathsf{Id}_1 := \text{Exp}_1 ; \ldots ; \mathsf{Id}_n := \text{Exp}_n$ **`in`** is dropped. Note that, the pre-processing does not perform any evaluation of the expressions, it is just a mere syntactic substitution.

B. SEMANTICS

In order to present the semantics of AbU-dsl, we need some auxiliary functions and definitions. The set of *semantic values* comprises integers ($\mathbb{Z}$), decimals ($\mathbb{Q}$), booleans (tt, ff), character strings ($\mathbb{S}$) and an undefined value ($\perp$), namely the value domain is $\mathbb{V} \triangleq \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{S} \cup \{\mathsf{tt}, \mathsf{ff}\} \cup \{\perp\}$. A generic value is denoted by v.

In the following, we denote by `ruleOf`(RuleId) the ECA rule (code) that has RuleId as rule name in an intended AbU-dsl program. Given a device Dev, we denote with `inv`(Dev) its invariant, if present. Similarly, given an ECA rule Rule, we denote with `task`(Rule) the set of its tasks, and with `event`(Rule) the set of resources in its event.

[2]Available at https://github.com/abu-lang/abudsl.

1) OPERATIONAL (SMALL STEP) SEMANTICS

Following [5], the semantics of AbU-dsl builds upon execution states and pools. A *device state* $\sigma \in \text{Identifiers} \to \mathbb{V}$, is a map from resource (names) to values, while a *device pool* $\theta \subseteq \bigcup_{n \in \mathbb{N}} \mathbb{U}^n$ is a set of *updates*. An update upd is a finite list of pairs $(\mathsf{Id}, v) \in \mathbb{U}$, meaning that the resource Id will take the value v after the commitment of the update.

Let $\text{Act} = \text{[}\mathbf{\texttt{this}}\text{.]}\mathsf{Id}_1 = \text{Exp}_1, \ldots, \text{[}\mathbf{\texttt{this}}\text{.]}\mathsf{Id}_n = \text{Exp}_n$ be a (local) task action, its evaluation $[\![\text{Act}]\!]$ in the device state σ returns an update. Formally, $[\![\text{Act}]\!]\sigma \triangleq$

$$(\mathsf{Id}_1, [\![\text{Exp}_1]\!]\sigma) \ldots (\mathsf{Id}_n, [\![\text{Exp}_n]\!]\sigma)$$

The evaluation semantics for value expressions Exp is standard. As we will see in a moment, the semantic function $[\![\cdot]\!]$ is applied only to local task actions, that do not contain instances of external resources **`ext`**.Id.

Given a list $\text{Refs} = \mathsf{RuleId}_1 \ldots \mathsf{RuleId}_k$ of ECA rules (names) and a set X of resources that have been modified, we define the set of *active rules* as $\mathsf{Active}(\text{Refs}, X) \triangleq$

$$\left\{ \text{Rule} \;\middle|\; \begin{array}{l} \text{Rule} = \texttt{ruleOf}(\mathsf{RuleId}_i) \wedge i \in [1..k] \\ \wedge\, \texttt{event}(\text{Rule}) \cap X \neq \varnothing \end{array} \right\}$$

namely the rules in Refs that listen on resources in X and, hence, that may be fired.

The *local updates* are the updates originated from the tasks of the active rules in Refs that act only locally (i.e., the modifier **`all`** is not present in the tasks condition) and that satisfy the task condition, namely, $\mathsf{LocUpds}(\text{Refs}, X, \sigma) \triangleq$

$$\left\{ [\![\text{Act}]\!]\sigma \;\middle|\; \begin{array}{l} \text{Rule} \in \mathsf{Active}(\text{Refs}, X) \wedge \text{Task} \in \texttt{task}(\text{Rule}) \\ \wedge\, \text{Task} = \mathbf{\texttt{for}}\ \text{Bexp}\ \mathbf{\texttt{do}}\ \text{Act} \wedge \sigma \models \text{Bexp} \end{array} \right\}$$

The satisfiability relation is defined as usual: $\sigma \models \text{Bexp} \triangleq [\![\text{Bexp}]\!]\sigma = \mathsf{tt}$ (the evaluation semantics for boolean expressions Bexp is standard as well). Note that, the *default* updates, namely the updates originated by actions under the scope of **`default`** are local updates, by definition.

When we have a task containing the modifier **`all`**, a remote device is needed to evaluate the task condition. In the AbU-dsl semantics, when a device needs to evaluate a task involving remote devices, it partially evaluates the task (with its own state) and then it sends the partially evaluated task to all other devices. The latter, receive the task and complete the evaluation, potentially adding updates to their pool. In particular, the partial evaluation of tasks works as follows. With $\{\!|\text{Task}|\!\}\sigma$ we denote the task obtained from Task with each occurrence of resources [**`this`**.]Id in the task condition and the right-hand side of the assignments in the task action replaced with the value $\sigma(\mathsf{Id})$. After that, each instance of **`ext`**.Id in the task condition and action is replaced with **`this`**.Id. Finally, the

$$\text{(Exec)}\ \frac{\mathsf{upd} = (\mathsf{Id}_1, v_1)\ldots(\mathsf{Id}_k, v_k) \quad \sigma' = \sigma[v_1/\mathsf{Id}_1\ldots v_k/\mathsf{Id}_k] \quad \mathtt{inv}(\mathsf{Dev}) = \mathsf{Bexp} \quad \sigma' \models \mathsf{Bexp} \quad \theta'' = \theta \setminus \{\mathsf{upd}\} \quad X = \{\mathsf{Id}_i \mid i \in [1..k] \wedge \sigma(\mathsf{Id}_i) \neq \sigma'(\mathsf{Id}_i)\} \quad \theta' = \theta'' \cup \mathsf{LocUpds}(\mathsf{Refs}, X, \sigma') \quad T = \mathsf{RemTasks}(\mathsf{Refs}, X, \sigma')}{\mathsf{Dev}\ \mathbf{has}\ \mathsf{Refs} \vdash \langle \sigma, \theta\rangle \xrightarrow{\mathsf{upd} \triangleright T} \langle \sigma', \theta'\rangle}\ \mathsf{upd} \in \theta$$

$$\text{(Exec-Fail)}\ \frac{\mathsf{upd} = (\mathsf{Id}_1, v_1)\ldots(\mathsf{Id}_k, v_k) \quad \sigma' = \sigma[v_1/\mathsf{Id}_1\ldots v_k/\mathsf{Id}_k] \quad \mathtt{inv}(\mathsf{Dev}) = \mathsf{Bexp} \quad \sigma' \not\models \mathsf{Bexp} \quad \theta' = \theta \setminus \{\mathsf{upd}\}}{\mathsf{Dev}\ \mathbf{has}\ \mathsf{Refs} \vdash \langle \sigma, \theta\rangle \xrightarrow{\mathsf{upd} \triangleright \epsilon} \langle \sigma, \theta'\rangle}\ \mathsf{upd} \in \theta$$

$$\text{(Input)}\ \frac{\sigma' = \sigma[v_1/\mathsf{Id}_1\ldots v_k/\mathsf{Id}_k] \quad X = \{\mathsf{Id}_1, \ldots, \mathsf{Id}_k\} \quad \theta' = \theta \cup \mathsf{LocUpds}(\mathsf{Refs}, X, \sigma') \quad T = \mathsf{RemTasks}(\mathsf{Refs}, X, \sigma')}{\mathsf{Dev}\ \mathbf{has}\ \mathsf{Refs} \vdash \langle \sigma, \theta\rangle \xrightarrow{\mathsf{upd} \blacktriangleright T} \langle \sigma', \theta'\rangle}\ v_1, \ldots, v_k \in \mathbb{V}$$

$$\text{(Propagate)}\ \frac{\theta'' = \{[\![\mathsf{Act}]\!]\sigma \mid \exists i \in [1..n]\,.\,\mathsf{Task}_i = \mathbf{for}\ \mathsf{Bexp}\ \mathbf{do}\ \mathsf{Act} \wedge \sigma \models \mathsf{Bexp}\} \quad \theta' = \theta \cup \theta''}{\mathsf{Dev}\ \mathbf{has}\ \mathsf{Refs} \vdash \langle \sigma, \theta\rangle \xrightarrow{\mathsf{Task}_1 \ldots \mathsf{Task}_n} \langle \sigma, \theta'\rangle}$$

$$\text{(Comm)}\ \frac{\mathsf{Dev}_i\ \mathbf{has}\ \mathsf{Refs}_i \vdash \langle \sigma_i, \theta_i\rangle \xrightarrow{\alpha} \langle \sigma_i', \theta_i'\rangle \quad \forall j \in [1..n] \setminus \{i\}\,.\,\mathsf{Dev}_j\ \mathbf{has}\ \mathsf{Refs}_j \vdash \langle \sigma_j, \theta_j\rangle \xrightarrow{T} \langle \sigma_j, \theta_j'\rangle \quad \Sigma = \sigma_1\ldots\sigma_n \quad \Theta = \theta_1\ldots\theta_n \quad \Sigma' = \Sigma[\sigma_i'/\sigma_i] \quad \Theta' = \theta_1'\ldots\theta_n'}{\mathsf{Dev}_1\ \mathbf{has}\ \mathsf{Refs}_1 \ldots \mathsf{Dev}_n\ \mathbf{has}\ \mathsf{Refs}_n \vdash \langle \Sigma, \Theta\rangle \xrightarrow{\alpha} \langle \Sigma', \Theta'\rangle}\ i \in [1..n]\ \ \alpha \in \{\mathsf{upd} \triangleright T, \mathsf{upd} \blacktriangleright T\}$$

FIGURE 2. Operational semantics for single devices and ensembles of devices (actual ECA rules code is omitted to simplify the presentation).

modifier `all` is dropped. For instance, consider the task $\mathsf{Task} \triangleq$

```
for all(this.x < ext.x) do ext.y = x + ext.y
```

Then, considering a device state $\sigma \triangleq [\mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0]$, assigning 1 to `x` and 0 to `y`, we have that $\{\!|\mathsf{Task}|\!\}\sigma$ is:

```
for (1 < this.x) do this.y = 1 + this.y
```

Note that, once the task is partially evaluated and sent to other devices, it becomes "syntactically local" for the receiving devices and, hence, its action can be evaluated with the semantic function $[\![\cdot]\!]$.

Finally, the *remote tasks* are the pre-evaluated tasks of active rules in Refs whose condition contains `all` (i.e., tasks that require a remote device to be evaluated), namely, $\mathsf{RemTasks}(\mathsf{Refs}, X, \sigma) \triangleq$

$$\left\{ \{\!|\mathsf{Task}|\!\}\sigma \;\middle|\; \begin{array}{l} \mathsf{Rule} \in \mathsf{Active}(\mathsf{Refs}, X) \wedge \mathsf{Task} \in \mathtt{task}(\mathsf{Rule}) \\ \wedge\ \mathsf{Task} = \mathbf{for\ all}\ \mathsf{Cnd}\ \mathbf{do}\ \mathsf{Act} \end{array} \right\}$$

Let Prg be a program with devices Dev_1 [`has` Refs_1] ... Dev_n [`has` Refs_n] and ECA rules $\mathsf{Rule}_1 \ldots \mathsf{Rule}_k$. We define the *execution state* Σ of Prg as a list of device states, one for each device defined in the program. Formally, $\Sigma = \sigma_1 \ldots \sigma_n$, where for each $i \in [1..n]$ we have that σ_i is a device state for Dev_i. Similarly, the *execution pool* Θ of Prg is a list of device pools, one for each device defined in the program, i.e., $\Theta = \theta_1 \ldots \theta_n$, where for each $i \in [1..n]$ we have that θ_i is a device pool for Dev_i.

The small-step operational semantics of a program Prg is modeled as a *Labeled Transition System (LTS)*. In particular, $\mathsf{Prg} \vdash \langle \sigma, \theta\rangle \xrightarrow{\alpha} \langle \sigma', \theta'\rangle$ means that the program Prg evolves, producing the label α. Here, labels are given by:

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

where T is a finite (possibly empty) list of tasks and upd an update. The semantics is *distributed*, in the sense that each device does not have a global knowledge about the system. The semantics is depicted in Figure 2, where rules (Exec), (Exec-Fail), (Input) and (Propagate) model the evolution of single devices, while the rule (Comm) models the evolution of an ensemble of devices (i.e., of a program). To simplify the presentation of the transition rules, in Figure 2 we omit the list of ECA rules (code) that devices refer to.

The rule (Exec) executes an update picked from the pool; while a rule (Input) models an external modification of some resources. The execution of an update, or the modification of resources in general, may trigger some other ECA rules. Hence, after updating its state, a device launches a *discovery phase*, to find new updates to add to the local pool (or some pools of remote devices), given by the activation of some ECA rules. The discovery phase is composed by two parts, the local and the remote one. A device performs a local discovery by means of the function LocUpds, that adds to the local pool all updates originated by the activation of some local rules. Then, by means of the function RemTasks, the device computes a list of tasks that may update remote devices and sends it to all devices of the program. This is modeled with the labels $\mathsf{upd} \triangleright T$, produced by the rule (Exec), and $\mathsf{upd} \blacktriangleright T$, produced by the rule (Input). On the other side, when a device receives a list of tasks, executing the rule (Propagate) with a label T, it evaluates them and adds to its pool the actions

generated by the tasks whose condition is satisfied. Note that, not necessarily all devices have to modify their pool (indeed, a task condition may not hold in a remote device). The rule (Comm) synchronizes the whole discovery phase, originated by a change in the state of a device of the program. When a device executes an action originating only local updates, the rule (Comm) is applied with $\Theta' = \Theta$, producing the label $\mathsf{upd} \triangleright \epsilon$ or the label $\mathsf{upd} \blacktriangleright \epsilon$ (i.e., with an empty tasks list ϵ). The latter, is matched by a label $T = \epsilon$, that all devices can generate by applying the rule (Propagate). Attentive readers may notice that in the rule (Exec) we start the discovery on the resources that have been modified only (the set X performs such check), while in the rule (Input) we start the discovery on all resources (no checks on X). This is due to the assumption that inputs from the environment reflect an actual change in some external components.[3]

Finally, the semantics also checks the fulfillment of invariants, at run-time. Indeed, when an update would break an invariant, the rule (Exec) is not applicable; instead, the rule (Exec-Fail) is performed, that ignores the update (i.e., it is removed from the pool without commitment). This fact is observable with labels of the form $\mathsf{upd} \triangleright \epsilon$.

2) WAVE SEMANTICS

On top of the operational semantics described above, we can define a big-step semantics, dubbed *wave semantics*, that hides internal computation steps. This semantics represents only state modifications resulting from inputs from the surrounding environment. An AbU-dsl *configuration* $\langle \Sigma, \Theta \rangle$ is a pair consisting of an execution state Σ and an execution pool Θ, for a given AbU-dsl program. A configuration is said *stable* when no more execution steps can be performed, namely when all device pools in Θ are empty. The wave semantics transforms stable configurations in stable configurations. Let $\rightarrow^*$ be the transitive closure of $\rightarrow$, without occurrences of labels of the form $\mathsf{upd} \blacktriangleright T$, namely $\rightarrow^*$ denotes a finite sequence of internal execution steps (with the corresponding discovery phases), without interleaving input steps. The wave semantics for a program Prg is:

$$\text{(Wave)}\ \frac{\mathsf{Prg} \vdash \langle \Sigma, \varnothing \rangle \xrightarrow{\mathsf{upd}} \langle \Sigma'', \Theta \rangle \quad \mathsf{Prg} \vdash \langle \Sigma'', \Theta \rangle \rightarrow^* \langle \Sigma', \varnothing \rangle}{\mathsf{Prg} \vdash \Sigma \overset{\mathsf{upd}}{\rightsquigarrow} \Sigma'}$$

The idea is that a stable system, in the state Σ, reacts to an external stimulus upd by executing a series of tasks that propagate across the devices like a "wave", until it becomes stable again, in the state Σ', waiting for the next stimulus. Note that, in the wave semantics inputs do not interleave with internal steps: the system needs to reach stability before processing the next input. If we allow arbitrary input steps during the computation, a system may never reach stability since the execution pools could be never emptied. This assumption has a practical explanation: in the IoT context, usually, external changes (in sensors) take much more time than internal computation steps [9].

[3]In the implementation of the DSL we will describe in Section IV we enforce this assumption by avoiding to start the discovery on idempotent inputs, if any.

C. OPTIMIZATION

The semantics described so far follows precisely the semantics of the AbU calculus [5], but it could be optimized for a more efficient implementation. Indeed, when a device of a program performs a local update (either an execution or an input), i.e., when the committed update does not trigger ECA rules on remote devices, the semantics in Figure 2 forces *all* devices to synchronize, by means of an empty discovery. In other words, when a device performs an update upd that does not trigger ECA rules on remote devices, i.e., when it emits a label $\mathsf{upd} \triangleright \epsilon$ (or $\mathsf{upd} \blacktriangleright \epsilon$), then all other devices must perform a (Propagate) step matching ϵ. To improve the performance of AbU-dsl, we can slightly modify the language semantics, allowing devices to *asynchronously* execute local updates. This translates in applying the transition rule (Comm) only when $T \neq \epsilon$, and to apply the following transition rule otherwise, i.e., when $\alpha \in \{\mathsf{upd} \triangleright \epsilon, \mathsf{upd} \blacktriangleright \epsilon\}$:

$$\text{(Local)}\ \frac{\begin{array}{c} i \in [1..n] \quad \mathsf{Dev}_i\ \mathtt{has}\ \mathsf{Refs}_i \vdash \langle \sigma_i, \theta_i \rangle \xrightarrow{\alpha} \langle \sigma, \theta \rangle \\ \Sigma = \sigma_1 \ldots \sigma_n \quad \Sigma' = \Sigma[\sigma/\sigma_i] \\ \Theta = \theta_1 \ldots \theta_n \quad \Theta' = \Theta[\theta/\theta_i] \end{array}}{\mathsf{Dev}_1\ \mathtt{has}\ \mathsf{Refs}_1 \ldots \mathsf{Dev}_n\ \mathtt{has}\ \mathsf{Refs}_n \vdash \langle \Sigma, \Theta \rangle \xrightarrow{\alpha} \langle \Sigma', \Theta' \rangle}$$

Note that, the added rule does not change the semantics of AbU-dsl in Figure 2, but it improves performance since less device communications are performed.

III. IoT PROGRAMMING EXAMPLES

In this section, we provide some IoT application scenarios for AbU-dsl. In particular, we model in AbU-dsl IoT devices that need to autonomously interact with each other, without the need of a central controlling node.

A. SWARM OF ROBOTS

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

The position of drones is given by a user-defined type `Coords`, that has two fields `latitude` and `longitude`. For each drone we have an AbU-dsl device with a resource `battery`, indicating the battery level of the drone; a (compound) resource `pos`, indicating where is located the drone; a resource `mode`, indicating in which operative state is the drone; a resource `help`, indicating the position of a drone

```
1  # AbU custom types declaration.
2
3  define Coords as {
4     latitude : physical output integer
5     longitude : physical output integer
6  }
7
8  # AbU devices definition.
9
10 droneA : "A robot taking measurements" {
11    physical input integer battery
12    Coords pos = (latitude = 2, longitude = 2)
13    Coords help = (latitude = -1, longitude = -1)
14    logical string mode = "measure"
15    logical string node = "droneA"
16    logical integer threshold = 7
17 } has batteryCheck setRescue
18
19 droneB : "A robot taking measurements" {
20    physical input integer battery
21    Coords pos = (latitude = 12, longitude = 5)
22    Coords help = (latitude = -1, longitude = -1)
23    logical string mode = "measure"
24    logical string node = "droneB"
25    logical integer threshold = 6
26 } has batteryCheck setRescue
27
28 droneC : "A robot taking measurements" {
29    physical input integer battery
30    Coords pos = (latitude = 5, longitude = 2)
31    Coords help = (latitude = -1, longitude = -1)
32    logical string mode = "measure"
33    logical string node = "droneC"
34    logical integer threshold = 7
35 } has batteryCheck setRescue
36
37 droneD : "A robot taking measurements" {
38    physical input integer battery
39    Coords pos = (latitude = 5, longitude = 5)
40    Coords help = (latitude = -1, longitude = -1)
41    logical string mode = "measure"
42    logical string node = "droneD"
43    logical integer threshold = 6
44 } has batteryCheck setRescue
45
46 \@
47    AbU (ECA) rules definition.
48    Rules can be referenced by multiple devices.
49 @\
50
51 rule batteryCheck on battery
52    for all (battery < 5 and ext.battery > 80)
53       do ext.help[latitude] = pos[latitude];
54          ext.help[longitude] = pos[longitude]
55
56 rule setRescue on helpLat helpLong
57    let diffLat := (pos[latitude] - help[latitude]);
58        diffLong := (pos[longitude] - help[longitude])
59    in
60    for (absint diffLat < threshold and
61         absint diffLong < threshold)
62       do mode = "rescue"
```

LISTING 1. A program modeling a swarm of robots.

that needs help; and a resource `threshold`, indicating when another drone is considered close to the current. Formally, the AbU-dsl program modeling the drone-swarm scenario is depicted in Listing 1. Now suppose that the battery levels of the drones are the following: `battery = 4` for `droneA`; `battery = 81` for `droneB`; `battery = 97` for `droneC`; and `battery = 65` for `droneD`. We assume that drones have an embedded battery sensor that updates the drone resource `battery`.

The ECA rule `batteryCheck` says that when the current drone battery level is low (i.e., when `battery < 5`), then the current drone has to send to all neighbors (using `all`) that have some energy to share (i.e., that have `ext.battery > 80`) its position, performing a remote update (composed by the action at lines 53 – 54).

In the example, `droneA` can fire the rule, since its battery level is low. Then, it pre-evaluates the task condition, yielding `4 < 5~and battery > 80`, which is sent to the other drones, together with the pre-evaluation of the task action, namely `help[latitude] = 2` and `help[longitude] = 2`. Among all possible receivers, only `droneB` and `droneC` are interested in the communication, since they are the only drones with battery level greater than `80`. So, they both add to their pool the update `(help[latitude], 2)(help[longitude], 2)`. This ends the discovery phase originated by `droneA`.

The rule `setRescue`, instead, is fired when a drone receives a help request (i.e., when its resource `help` changes) and basically checks if the current drone position is close to the requester drone position (by checking that the difference between latitude and longitude values is less than `threshold`). If it is the case, the current drone enters the rescue mode performing the local update `mode = "rescue"`.

In the example, when `droneB` and `droneC` execute the update `(help[latitude], 2)(help[longitude], 2)` the task of the rule `setRescue` may be executed. For `droneB` this does not happen, since `absint (12 - 2) < 6` at line 60 is not satisfied (the drone is too far from `droneA`). Instead, the conditions `absint (5 - 2) < 7` and `absint (2 - 2) < 7` at lines 60 – 61 are both satisfied for `droneC`, hence the latter can execute the rule task, adding to its pool the update `(mode, "rescue")`.

B. SMART HVAC SYSTEM

We provide an AbU-dsl implementation of a *Heating, Ventilation and Air Conditioning (HVAC)* system. 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. In particular, `node` takes the values `"system"`, `"tempSens"` and `"humSens"` on the HVAC control system, the temperature sensor and the humidity sensor, respectively.

The AbU-dsl code modeling the scenario is depicted in Listing 2. The rule `notifyTemp` 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 `node` equals to `"system"`. The rule `notifyHum` do the same for the resource `humidity` on the humidity sensor device.

All other rules in Listing 2 are related to the HVAC control system. The latter 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 $18°C$ (`this.temperature < 18`) the rule `cool` activates the heating (`this.heating = true`). Instead, when the temperature is greater than $27°C$ (`this.temperature > 27`), then the rule `warm` deactivates the heating (`this.heating = false`). The air

```
1  system : "An HVAC control system" {
2     physical output boolean heating = false
3     physical output boolean conditioning = false
4     logical integer temperature = 0
5     logical integer humidity = 0
6     physical input boolean airButton
7     logical string node = "hvac"
8     where not (conditioning and heating)
9  } has cool warm dry stopAir
10
11 tempSens : "A temperature sensor" {
12    physical input integer temperature
13    logical string node = "tempSens"
14 } has notifyTemp
15
16 humSens : "A humidity sensor" {
17    physical input integer humidity
18    logical string node = "humSens"
19 } has notifyHum
20
21 rule cool on temperature
22    for (this.temperature < 18) do this.heating = true
23
24 rule warm on temperature
25    for (this.temperature > 27) do this.heating = false
26
27 rule dry on humidity; temperature
28    for (2 + 0.5 * this.temperature < this.humidity and
29          38 - this.temperature < this.humidity)
30       do this.conditioning = true
31
32 rule stopAir on airButton
33    for (this.airButton) do this.conditioning = false
34
35 rule notifyTemp on temperature
36    for all (ext.node == "system")
37       do ext.temperature = this.temperature
38
39 rule notifyHum on humidity
40    for all (ext.node == "system")
41       do ext.humidity = this.humidity
```

LISTING 2. A program modeling an HVAC system.

conditioning is turned on (`this.conditioning = true`), by means of the ECA rule `dry`, when the humidity exceeds the upper bound of the Givoni's *comfort zone* [10], i.e., when the condition at lines 28 – 29 is satisfied.

The HVAC control system is also bestowed with a physical button for manually stopping the air conditioning. Indeed, the rule `stopAir` stops the air conditioning (`this.conditioning = false`) when the button is pressed (`this.airButton` is `true`). Finally, by means of the invariant `not (conditioning and heating)` we specify that no update can result in the activation of both heating and air conditioning simultaneously.

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-dsl with a single device comprising all resources introduced in Listing 2 and transforming remote rules into local ones. This highlights the flexibility of AbU-dsl, that is able to model both distributed and centralized ensembles of devices.

C. PROGRAMMING A RASPBERRY Pi

In this last example, we show how AbU-dsl allows to easily program a Raspberry Pi equipped with physical sensors and actuators. We actually tested the example on real devices, compiling the AbU-dsl program of Listing 3 with the AbU compiler, that we will present in Section IV. In particular, the code has been deployed on two Raspberry Pi 3b, one equipped with a brushed DC motor with L293 driver; and the other equipped with two LEDs and two GPIO buttons.

The `wheel` device (i.e., one of the two Raspberry Pi), that is equipped with the brushed DC motor, contains a (compound) resource `motor` initialized with the actual physical PIN numbers connected to the Raspberry (fields `forwardPin` and `backwardPin`) and the initial electric tension (expressed by an integer between 0 and 255) applied to the PINs (fields `forwardPace` and `backwardPace`). These tensions are used to set the rotation speed of the motor, as described in the L293 motor datasheet.

When `motor[forwardPace]` is greater than 0 then the motor spins clockwise, with the given pace, while when `motor[backwardPace]` is greater than 0 then the motor spins counterclockwise, with the given pace. The specification of the L293 driver does not describe what happens when both PINs are simultaneously powered, so, to prevent motor damages, we enforce at programming level (i.e., by means of AbU-dsl code) that only one PIN at a time can be powered, by using the invariant at lines 20 – 21. The device is controlled by two ECA rules `drive` and `brake`: the first continuously increases the speed of the motor on a given spinning direction; while the second stops the motor rotation, when the maximum speed is reached.

To change the motor rotation direction we need to press a combination of buttons on the device `controls` (i.e., the other Raspberry Pi), that is equipped with two GPIO buttons and two one-color LEDs. Similarly to the previous case, we have to initialize buttons and LEDs resources with the physical PIN numbers connected to the Raspberry (setting the `pin` field of the `LED` and `GPIOButton` resources). Then, the boolean field `status` of `LED` resources can be set to `true` and `false` indicating that the LED is on and off, respectively. Similarly, the boolean field `status` of `GPIOButton` resources can be set to `true` and `false` indicating that the button is pressed and released, respectively.

Buttons are used to control the rotation direction of the motor and to turn on and off the LEDs. In particular, the ECA rule `changeDir` sets the motor spin counterclockwise, by performing the actions `ext.motor[forwardPace] = 0` and `ext.motor[backwardPace] = 1`, when the first button is pressed and the second one is not pressed; while the rule sets the motor spin clockwise, by performing the actions `ext.motor[forwardPace] = 1` and `ext.motor[backwardPace] = 0`, when the second button is pressed and the first one is not pressed. Note that, the rule potentially affects all devices in the network that have a brushed DC motor, by selecting the devices such that `ext.node == "DCmotor"`. Finally, the ECA rule `toggleLed` toggles the status of the two LEDs (lines 46 – 47), when both buttons are pressed together (`buttonA[status] and buttonB[status]`).

```
define DCMotor as {
   forwardPin : physical output integer
   backwardPin : physical output integer
   forwardPace : logical integer
   backwardPace : logical integer
}
define GPIOButton as {
   pin : physical output integer
   status : logical boolean
}
define LED as {
   pin : physical output integer
   status : logical boolean
}

wheel : "A brushed DC motor with L293 driver" {
   DCMotor motor = (forwardPin = 13, backwardPin = 11,
                    forwardPace = 0, backwardPace = 0)
   logical string node = "DCmotor"
   where not (motor[forwardPace] > 0 and
              motor[backwardPace] > 0)
} has drive brake

controls : "A device managing LEDs and buttons" {
   # Two simple one color LEDs
   LED ledOne = (pin = 36, status = false)
   LED ledTwo = (pin = 37, status = false)
   # Two GPIO on\off buttons
   GPIOButton buttonA = (pin = 38, status = false)
   GPIOButton buttonB = (pin = 40, status = false)
   logical string node = "LEDsAndButtons"
} has changeDirection toggleLed

rule changeDir on buttonA[status] buttonB[status]
   for all (ext.node == "DCmotor" and buttonA[status]
            and not buttonB[status])
      do ext.motor[forwardPace] = 0;
         ext.motor[backwardPace] = 1
   for all (ext.node == "DCmotor" and buttonB[status]
            and not buttonA[status])
      do ext.motor[forwardPace] = 1;
         ext.motor[backwardPace] = 0

rule toggleLed on buttonA[status] buttonB[status]
   for (buttonA[status] and buttonB[status])
      do ledOne[status] = not ledOne[status];
         ledTwo[status] = not ledTwo[status]

rule drive on motor[forwardPace] motor[backwardPace]
   for (motor[forwardPace] > 0 and
        motor[forwardPace] < 255)
      do motor[forwardPace] = motor[forwardPace] + 8
   for (motor[backwardPace] > 0 and
        motor[backwardPace] < 255)
      do motor[backwardPace] = motor[backwardPace] + 8

rule brake on motor[forwardPace] motor[backwardPace]
   for (motor[forwardPace] >= 255)
      do motor[forwardPace] = 0
   for (motor[backwardPace] >= 255)
      do motor[backwardPace] = 0
```

LISTING 3. An program modeling two devices (Raspberry Pi) equipped with a DC motor, two LEDs and two GPIO buttons.

IV. A DISTRIBUTED IMPLEMENTATION

AbU-dsl is a (decentralized) distributed language, hence any implementation has to deal with the intrinsic issues of distributed systems. In particular, by the CAP theorem [11] we cannot have, at the same time, consistency, availability and partition-tolerance. Hence, some compromises have to be taken, depending on the application context. For instance, in a scenario with low network traffic we can aim for correctness, implementing a robust, but slow, communication protocol. Vice versa, when devices exchange data at a high rate (or when the network is not stable), communication should take very short time, hence we may prefer to renounce to consistency in favor of eventual consistency.

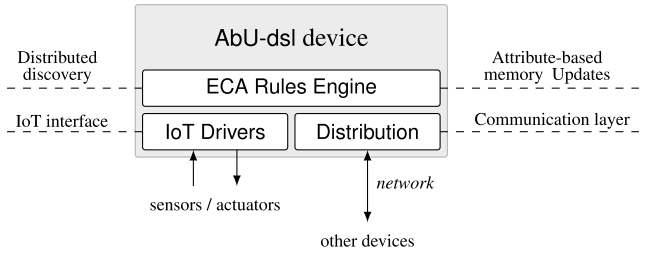

FIGURE 3. High-level view of an AbU-dsl device architecture.

For these reasons, a flexible and modular implementation is mandatory, where modules can be implemented in different ways, depending on the application context. Hence, we designed a modular architecture suitable to implement AbU-dsl devices, as depicted in Figure 3. An AbU-dsl device consists in a *device state* (mapping resources to values), a *device pool* (a set of updates to execute) and a *list of ECA rules* (modeling the device behavior). An *ECA Rules Engine* module is in charge of executing the updates in the pool and to discover new rules to trigger, potentially on remote devices (distributed discovery). This module also implements Attribute-based memory Updates and deals with IoT inputs (from sensors) and outputs (to actuators), which are accessed by means of a dedicated interface. A separate *IoT Drivers* module translates low-level IoT devices primitives to high-level signals for the rule engine and vice versa. The *Distribution* module is in charge of joining a cluster of AbU-dsl devices and exchanging messages with them. It embodies all distributed infrastructure-related aspects, that can be tuned to meet the desired context-related requirements. Moreover, it provides the communication APIs needed by the rule engine to implement the (distributed) discovery phase (and, in turn, Attributed-based memory Updates). For instance, the labels upd $\blacktriangleright$ T and upd $\triangleright$ T of the AbU-dsl semantics may generate a broadcast communication.

We opted for an *exogenous* language design, meaning that the DSL provides an abstraction layer for an existing full-fledged general-purpose programming language. In this way, we can reuse existing code (or fast-developing new code), demanding to AbU-dsl only distributed communication and coordination aspects. Local computations on devices are implemented using the underlying general-purpose language.

As a first prototype, we have developed GoAbU, an implementation of AbU-dsl in Golang, together with several open-source support tools for AbU-dsl: `abuc`, an AbU-dsl compiler; and `abusim`, a simulator for AbU-dsl devices. The extension of AbU-dsl source code files is `.abu`. The code of GoAbU and these tools is publicly accessible on GitHub.[4]

A. GOLANG-BASED PROTOTYPE

GoAbU[5] implements the architecture described in Figure 3. In particular, the ECA Rules Engine module is built on

[4]Available at https://github.com/abu-lang.
[5]Available at https://github.com/abu-lang/goabu.

top of the Hyperjump's **Grule** [12] library, a stable ECA rules engine providing an incremental evaluation strategy. Incremental evaluation enhances efficiency in rule evaluation by avoiding the repetition of previous computations and by exploiting the structural similarity between rules.

The IoT Drivers module is built on top of **GOBOT** [13], a Go library for the IoT ecosystem, with a great availability of device drivers. The library provides data abstractions for programming IoT devices in a simple way. Devices behavior is specified by implementing callback functions, that can be compiled to all supported platforms (e.g., arm64).

Finally, the Distribution module is based on HashiCorp's **Memberlist** [14], a popular Go library for cluster membership and failures detection that uses a SWIM-like gossip protocol [15]. In such protocol, to achieve a lightweight and scalable cluster membership solution, every membership update message (carrying information about joins, leaves or failures) is distributed by piggybacking the update on the messages used to implement the failure detection protocol. This allows for solutions relying on cluster membership, with *eventual consistency*, that are able to scale on larger systems.

GoAbU provides a correct implementation of the semantics depicted in Figure 2, where the whole discovery phase is executed in a single atomic step. This implies that the partially evaluated tasks must be delivered to other devices by means of an atomic (''all or none'') operation, as when *reliable broadcast* is performed. In the **GoAbU** implementation, we achieve reliable broadcast by using a customized transactional two-phase commit protocol [16].

B. COMPILATION AND DEPLOYMENT

In this section, we describe the compilation and deployment pipeline for **AbU-dsl** programs. A graphical representation of the whole compilation process is reported in Figure 4.

Before compilation, we need a little pre-processing. First, we have a *de-sugaring phase*, that removes the syntactic constructs added to simplify coding, as explained in Subsection II-A. Then, we have a *splitting phase*, that separates the devices of the program, together with the corresponding rules code. In particular, a program consisting in n devices will result in n ''partial'' **AbU-dsl** programs consisting in one device only. Finally, the code of the rules referenced by a device is copied in the corresponding single-device program.

The resulting ''partial'' **AbU-dsl** programs are compiled to the target language. The compiler `abuc`[6] supports different targets, that may yield *standalone* or *intermediate* compilations. In the first case, the **AbU-dsl** program is translated to the target machine code and the IoT run-time is added, obtaining a standalone executable. In the second case, the **AbU-dsl** program is translated to the target programming language code, so that it can be extended or linked into other projects.

At the time of writing, `abuc` supports the following targets: `go`, namely the output of the compilation is a ready-to-use

[6]Available at https://github.com/abu-lang/abuc.

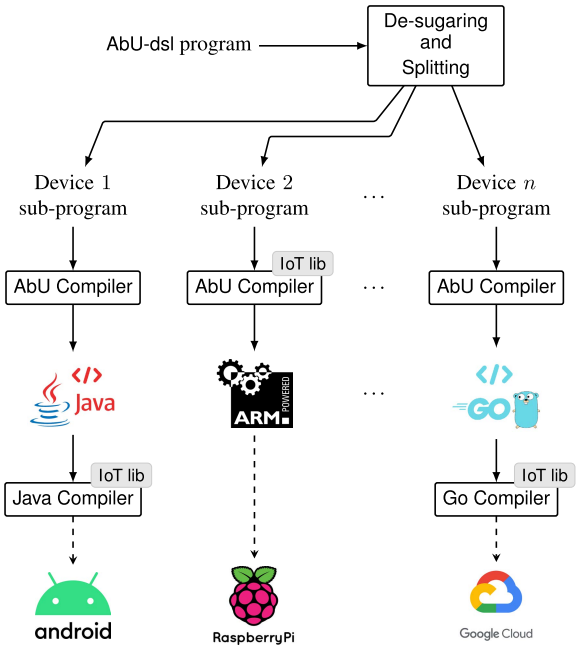

FIGURE 4. High-level view of AbU-dsl programs compilation.

GoAbU code, that can be imported to other Golang project and manually customized; `amd64`, namely the output of the compilation is an amd64 executable; and `arm64`, namely the output of the compilation is an arm64 executable.

As a matter of example, by using the following command on the **AbU-dsl** program in Listing 2:

```
$ abuc -o hvacGo -t go hvac.abu
```

we obtain three Golang source files `hvacGo-system.go`, `hvacGo-tempSens.go` and `hvacGo-humSens.go`. The first contains (the **GoAbU** code of) the `system` device and the rules `cool`, `warm`, `dry` and `stopAir`; the second contains the `tempsSens` device and the rule `notifyTemp`; while the third contains the `humSens` device and the rule `notifyHum`.

The compiler accepts also an optional parameter `-c` (or `-config`), that is used to specify a configuration files for linking IoT libraries, as explained next.

1) LINKING IoT LIBRARIES

In order to map (low-level) IoT resources with (high-level) **AbU-dsl** resources, we have to link devices driver libraries. In particular, for each custom type defined in **AbU-dsl** (e.g., the `DCMotor` type in Listing 3) we have to link a corresponding driver written in the target language. This information is provided to the compiler by using a `.json` configuration file. In Listing 4 we show an example of such configuration file, for the **AbU-dsl** program in Listing 3. The first line of the file specifies the necessary libraries (in the example, the **GOBOT** driver for the Raspberry Pi), while the second line imports the target language file containing the interfaces to which **AbU-dsl** types should be mapped to (in the example, we have a Golang implementation of the **GOBOT**-based drivers). The third line

```
{
   "Imports": ["gobot.io/x/gobot/platforms/raspi"],
   "AdditionalFiles": ["customResources.go"],
   "StateInitializer":
        "MakeCustomResources(raspi.NewAdaptor())",
   "Mappings": {
      "DCMotor": {
         "GoabuType": "L293Motor",
         "Fields": {
            "forwardPin": "fPin",
            "backwardPin": "bPin",
            "forwardPace": "fSpeed",
            "backwardPace": "bSpeed"
         },
         "Args": ["forwardPin", "backwardPin",
                  "forwardPace", "backwardPace"]
      },
      "GPIOButton": {
         "GoabuType": "Button",
         "Fields": {
            "pin": "pin",
            "status": "pressed"
         },
         "Args": ["pin", "status"]
      },
      "LED": {
         "GoabuType": "Led",
         "Fields": {
            "pin": "pin",
            "status": "active"
         },
         "Args": ["pin", "status"]
      }
   }
}
```

LISTING 4. An configuration file example for the AbU-dsl compiler.

is an initialization string that is required by the specific target GoAbU. The rest of the file comprises the mappings. For each custom AbU-dsl type we specify the corresponding target language interface (in the example, for instance, we associate the AbU-dsl type `DCMotor` with the Golang struct `L293Motor`) and the mappings for the AbU-dsl type fields (in the example, for instance, we associate the field `forwardPin` with the Golang resource `fPin` of `L293Motor`). Finally, `Args` specifies the order of the parameters required by the target language type constructor.

As a matter of example, by using the following command on the AbU-dsl program in Listing 3:

```
$ abuc -t arm64 -c config.json rasp-pi.abu
```

where `config.json` is the configuration file in Listing 4, we obtain two arm64 executables[7] `wheel` and `controls`. The first can be deployed on the Raspberry Pi 3b equipped with the brushed DC motor, while the second can be deployed the other Raspberry Pi equipped with the two LEDs and the two GPIO buttons.

C. SIMULATION ENVIRONMENT

In order to test AbU-dsl implementations, we have developed `abusim`,[8] a simulator that automatically deploys and executes AbU-dsl devices. The simulator is configured by means of a dedicated `.yaml` file, and can be used with different implementations of AbU-dsl. Indeed, it is sufficient to provide the simulator with Docker images containing the devices code (e.g., GoAbU compiled code).

[7]File names are taken from the devices in the source code if the option `-o` is not present.

[8]Available at https://github.com/abu-lang/abusim.

V. CONCLUSION

In this paper we have introduced AbU-dsl, a new Domain Specific Language for the IoT merging the simplicity of ECA programming with Attribute-based memory Updates [5]. The latter is a new time-coupled, space-uncoupled interaction mechanism where nodes communication is performed without a global knowledge of network participants, and it fits neatly within the ECA programming paradigm. We have shown how practical IoT systems of smart devices can be easily programmed in AbU-dsl, even in the case of complex scenarios. Then, we have described the compilation and deployment process of an AbU-dsl program, considering a particular target (implementation) language, i.e., Golang. A prototype implementation of AbU-dsl in Golang (i.e., GoAbU), together with a compiler and a simulation environment, is publicly available in the language GitHub repository:

`https://github.com/abu-lang`

A. RELATED WORK

To the best of our knowledge, the only work aiming at merging the ECA programming paradigm with attribute-based interaction is [5], which is indeed the theoretical model at the basis of the DSL presented in this paper. In turn, the model of [5] has its root in the AbC calculus, introduced and studied in [6], [7], and [8] as a core calculus for SCEL [17]. Various extensions of AbC has been proposed [18], [19], as well as correct implementations in Erlang and Golang [20], [21], [22].

Concerning ECA programming for IoT systems, a notably example is IRON [23], whose formal semantics is defined in [9] and [24]. Most works dealing with ECA rules try to assess properties such as termination, confluence, absence of redundant or contradicting rules. To this aim, [23], [25], and [26] implement verification mechanisms to check these properties on IRON programs. Other works propose approaches to verify systems based on ECA rules by using Petri Nets [27] and Binary Decision Diagrams [28]. In [1] and [29], the authors present a tool-supported method for verifying and controlling the correct interactions of ECA rules. A formalization of an ECA rule-based system is provided in order to transform the translation into a Heptagon/BZR program.

Finally, recent work [30], [31] on the Reactive Data model adopts a declarative attribute-based interaction similar to AbU-dsl's. In this model, ECA rules are given by declarative Response Relations, while attribute-based interaction is obtained by using dynamic end-points of the relations, defined by graph query languages. However, these approaches are not intended to be used for distributed programming.

B. FUTURE WORK

First, we plan to develop a *type system* for AbU-dsl, with the aim of performing devices and ECA rules well-formedness checks, together with expressions type checks. Furthermore, we plan to extend the AbU-dsl compiler, targeting more languages and architectures. Note that, AbU-dsl can be considered as a target language for the implementation of other distributed languages, e.g., agent based, hence we may think of developing compilers for other known DSL, e.g., JADEL [32], to AbU-dsl.

In order to guarantee termination of AbU-dsl programs, we plan to implement a *static verification mechanism* to ensure stabilization, as defined in [5] for the AbU calculus. Similarly, we may implement for AbU-dsl verification mechanisms for *safety* and *security* IoT requirements, defined, for instance, in terms of behavioral equivalences (e.g., *bisimulations*) between AbU-dsl devices, as done in [33]. Along this line, in IoT systems it is often important to guarantee that inputs are processed within precise time bounds; to this end, following [34], we can think of adding quantitative aspects to the semantics of AbU-dsl, in order to provide precise estimations of stabilization times.

The smooth integration of an attribute-based interaction mechanism within the ECA paradigm should simplify the porting to the distributed setting of many known results and techniques from the ECA literature. In particular, we are interested in porting to AbU-dsl the verification techniques developed for ECA languages, such as IRON [25], [26], [27].

Another interesting issue is distributed *runtime verification*, in order to detect violations at runtime of given correctness properties, e.g., expressed in temporal logics like CTL or the μ-calculus [35]. Finally, a new generalization of Attribute-based Communication has been presented in [36]; it could be interesting to investigate how to apply that interaction model to the ECA programming.

REFERENCES

[1] J. Cano, E. Rutten, G. Delaval, Y. Benazzouz, and L. Gurgen, "ECA rules for IoT environment: A case study in safe design," in *Proc. IEEE 8th Int. Conf. Self-Adapt. Self-Organizing Syst. Workshops*, Sep. 2014, pp. 116–121.

[2] M. Balliu, M. Merro, M. Pasqua, and M. Shcherbakov, "Friendly fire: Cross-app interactions in IoT platforms," *ACM Trans. Privacy Secur.*, vol. 24, no. 3, pp. 1–40, Aug. 2021.

[3] D. R. John and G. J. Rydning. (2018). *Data Age 2025 IDC's Whitepaper, Sponsored by Seagate Technology LLC*. [Online]. Available: https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html

[4] B. Gill and D. Smith. (2018). *The Edge Completes the Cloud: A Gartner Trend Insight Report*. Gartner. [Online]. Available: https://emtemp.gcom.cloud/ngw/globalassets/en/doc/documents/3889058-the-edge-completes-the-cloud-a-gartner-trend-insight-report.pdf

[5] M. Miculan and M. Pasqua, "A calculus for attribute-based memory updates," in *Theoretical Aspects of Computing—ICTAC* (Lecture Notes in Computer Science), A. Cerone and P. Ölveczky, Eds. Nur-Sultan, Kazakhstan: Springer, 2021.

[6] Y. A. Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, and R. Vigo, "A calculus for attribute-based communication," in *Proc. 30th Annu. ACM Symp. Appl. Comput.*, Apr. 2015, pp. 1840–1845, doi: 10.1145/2695664.2695668.

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

[8] Y. A. Alrahman, R. De Nicola, and M. Loreti, "Programming interactions in collective adaptive systems by relying on attribute-based communication," *Sci. Comput. Program.*, vol. 192, Jun. 2020, Art. no. 102428.

[9] D. R. Cacciagrano and R. Culmone, "IRON: Reliable domain specific language for programming IoT devices," *Internet Things*, vol. 9, Mar. 2020, Art. no. 100020, doi: 10.1016/j.iot.2018.09.006.

[10] B. Givoni, "Comfort, climate analysis and building design guidelines," *Energy Buildings*, vol. 18, no. 1, pp. 11–23, Jan. 1992.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, doi: 10.1145/564585.564601.

[12] Hyperjump.tech. *Grule*. Accessed: Oct. 21, 2022. [Online]. Available: https://github.com/hyperjumptech/grule-rule-engine/

[13] gobot.io. *GOBOT*. Accessed: Oct. 21, 2022. [Online]. Available: https://gobot.io/

[14] hashicorp.com. *Memberlist*. Accessed: Oct. 21, 2022. [Online]. Available: https://github.com/hashicorp/memberlist/

[15] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 303–312.

[16] J. N. Gray, *Notes on Data Base Operating Systems*. Berlin, Germany: Springer, 1978, pp. 393–481.

[17] R. De Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, and A. Vandin, "The SCEL language: Design, implementation, verification," in *Software Engineering for Collective Autonomic Systems* (Lecture Notes in Computer Science), vol. 8998, M. Wirsing, M. Holzl, N. Koch, and P. Mayer, Eds. Cham, Switzerland: Springer, 2015, pp. 3–71.

[18] Y. A. Alrahman, R. De Nicola, and M. Loreti, "A calculus for collective-adaptive systems and its behavioural theory," *Inf. Comput.*, vol. 268, Oct. 2019, Art. no. 104457.

[19] Y. A. Alrahman and G. Garbi, "A distributed API for coordinating AbC programs," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 4, pp. 477–496, Feb. 2020.

[20] R. De Nicola, T. Duong, and M. Loreti, "Provably correct implementation of the AbC calculus," *Sci. Comput. Program.*, vol. 202, Feb. 2021, Art. no. 102567. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642320301751

[21] Y. Abd Alrahman, R. De Nicola, and G. Garbi, "GoAt: Attribute-based interaction in Google go," in *Leveraging Applications of Formal Methods, Verification and Validation, Distributed Systems* (Lecture Notes in Computer Science), vol. 11246, T. Margaria and B. Steffen, Eds. Cham, Switzerland: Springer, 2018, pp. 288–303.

[22] giulio-garbi.github.io. *GoAt*. Accessed: Oct. 21, 2022. [Online]. Available: https://giulio-garbi.github.io/goat/

[23] F. Corradini, R. Culmone, L. Mostarda, L. Tesei, and F. Raimondi, "A constrained ECA language supporting formal verification of WSNs," in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Mar. 2015, pp. 187–192.

[24] D. R. Cacciagrano and R. Culmone, "Formal semantics of an IoT-specific language," in *Proc. 32nd Int. Conf. Adv. Inf. Netw. Appl. Workshops (WAINA)*, May 2018, pp. 579–584.

[25] C. Vannucchi, M. Diamanti, G. Mazzante, D. R. Cacciagrano, F. Corradini, R. Culmone, N. Gorogiannis, L. Mostarda, and F. Raimondi, "VIRONy: A tool for analysis and verification of ECA rules in intelligent environments," in *Proc. Int. Conf. Intell. Environ. (IE)*, Aug. 2017, pp. 92–99, doi: 10.1109/IE.2017.32.

[26] C. Vannucchi, M. Diamanti, G. Mazzante, D. Cacciagrano, R. Culmone, N. Gorogiannis, L. Mostarda, and F. Raimondi, "Symbolic verification of event–condition–action rules in intelligent environments," *J. Reliable Intell. Environ.*, vol. 3, no. 2, pp. 117–130, Aug. 2017, doi: 10.1007/s40860-017-0036-z.

[27] X. Jin, Y. Lembachar, and G. Ciardo, "Symbolic verification of ECA rules," in *Proc. Int. Workshop Petri Nets Softw. Eng. (PNSE), Int. Workshop Model. Bus. Environ. (ModBE)*, vol. 989, D. Moldt, Ed. Milano, Italy, 2013, pp. 41–59. [Online]. Available: http://ceur-ws.org/Vol-989/paper17.pdf

[28] D. Beyer and A. Stahlbauer, "BDD-based software verification," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 5, pp. 507–518, Oct. 2014, doi: 10.1007/s10009-014-0334-1.

[29] J. Cano, G. Delaval, and E. Rutten, "Coordination of ECA rules by verification and control," in *Coordination Models and Languages*, E. Kuhn and R. Pugliese, Eds. Berlin, Germany: Springer, 2014, pp. 33–48.
[30] J. C. Seco, S. Debois, T. Hildebrandt, and T. Slaats, "RESEDA: Declaring live event-driven computations as reactive semi-structured data," in *Proc. IEEE 22nd Int. Enterprise Distrib. Object Comput. Conf. (EDOC)*, Oct. 2018, pp. 75–84.
[31] L. Galrinho, J. C. Seco, S. Debois, T. Hildebrandt, H. Norman, and T. Slaats, "ReGraDa: Reactive graph data," in *Coordination Models and Languages*, F. Damiani and O. Dardha, Eds. Cham, Switzerland: Springer, 2021, pp. 188–205.
[32] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "Agent-oriented model-driven development for JADE with the JADEL programming language," *Comput. Lang., Syst. Struct.*, vol. 50, pp. 142–158, Dec. 2017.
[33] M. Pasqua and M. Miculan, "On the security and safety of AbU systems," in *Software Engineering and Formal Methods* (Lecture Notes in Computer Science), vol. 13085, R. Calinescu and C. S. Pasareanu, Eds. Cham, Switzerland: Springer, 2021, pp. 178–198.
[34] M. Miculan and M. Peressotti, "Structural operational semantics for non-deterministic processes with quantitative aspects," *Theor. Comput. Sci.*, vol. 655, pp. 135–154, Dec. 2016.
[35] M. Miculan, "On the formalization of the modal μ-calculus in the calculus of inductive constructions," *Inf. Comput.*, vol. 164, no. 1, pp. 199–231, Jan. 2001, doi: 10.1006/inco.2000.2902.
[36] M. Miculan and M. Paier, "A calculus of subjective communication," in *Proc. 23rd Italian Conf. Theor. Comput. Sci. (ICTCS)*, U. D. Lago and D. Gorla, Eds. 2022, pp. 1–13.

MICHELE PASQUA received the B.Sc. and M.Sc. degrees in computer science and engineering and the Ph.D. degree in computer science from the University of Verona, Italy, in 2013, 2015, and 2019, respectively.

From 2019 to 2020, he was a Postdoctoral Researcher at the University of Verona, working on security aspects of IoT systems. From 2020 to 2021, he was a Postdoctoral Researcher at the University of Udine, Italy, working on new decentralized programming paradigms for the IoT. He is currently an Assistant Professor at the Department of Computer Science, University of Verona. His research interests include security of computational systems, program verification (based on abstract interpretation), mathematical foundations of computer science, programming languages design, cybersecurity, and software testing.

MASSIMO COMUZZO received the B.Sc. and M.Sc. degrees in computer science from the University of Udine, Italy, in 2018 and 2022, respectively. His M.Sc. thesis is focused on the implementation of a new language for distributed programming based on ECA-rules and Attribute-based memory Updates.

His research interests include learning and working in the fields of distributed systems and the IoT.

MARINO MICULAN received the Ph.D. degree in computer science from the University of Pisa, Italy, in 1997.

He is currently an Associate Professor at the University of Udine, where he directs the Laboratory of Models and Application of Distributed System and the local group of the National Cybersecurity Laboratory. He is the author or coauthor of about 80 publications in international scientific journals and conference proceedings with peer review. His research interests include semantic models, formal methods and programming languages for concurrent, distributed, and autonomous systems, especially for the analysis and verification of security aspects. He serves regularly in the program committees of international workshops and conferences.

Open Access funding provided by 'Università degli Studi di Verona' within the CRUI CARE Agreement