Distributed Programming of Smart Systems with Event-Condition-Action Rules (Short Paper)*

Marino Miculan[1,2], Michele Pasqua[3]

[1]University of Udine, Italy
[2]Ca' Foscari University of Venice, Italy
[3]University of Verona, Italy

Abstract

In recent years, event-driven programming languages, e.g. those based on *Event Condition Action* (ECA) rules, have emerged as a promising paradigm for implementing smart systems, such as IoT devices. Still, actual implementations are bound to a centralized infrastructure, limiting scalability and security.

In this work, we present *attribute-based memory updates* (AbU), a new interaction mechanism aiming to extend the ECA programming paradigm to distributed systems. It relies on *attribute-based communication*, that is similar to broadcast, but receivers are selected "on the fly" by means of predicates over their attributes. With AbU, smart devices can be easily programmed via ECA rules and, at the same time, they can be deployed to a distributed network. Hence, a centralized infrastructure is not needed anymore: the computation is moved on the *edge*, improving reliability, scalability, privacy and security.

Keywords

ECA rules, Attribute-based communication systems, Formal methods, Autonomic computing

1. Introduction

Event Condition Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. In the ECA programming paradigm, the behaviour of a system is defined by a set of rules of the form "**on** *Event* **if** *Condition* **do** *Action*" which means: when *Event* occurs, if *Condition* is verified then execute *Action*. This paradigm is particularly suited for programming *smart systems*, such as in IoT scenarios [2, 3, 4]: ECA systems react to inputs (as events) from the environment performing *internal* actions (updating the node local memory) and *external* actions, which influence the environment itself. Indeed, all main platforms in the field of Home/Automotive IoT (e.g., IFTTT, Samsung SmartThings, Microsoft PowerAutomate, Zapier, etc.) adopt this programming style.

Despite this simplicity, actual ECA platforms adopt a centralized infrastructure: IoT devices

Proceedings of the 23rd Italian Conference on Theoretical Computer Science, Rome, Italy, September 7-9, 2022

* This is an short version of the paper that we presented at ICTAC 2021 [1]. Work supported by the Italian MIUR project PRIN 2017FTXR7S *IT MATTERS (Methods and Tools for Trustworthy Smart Systems).*

marino.miculan@uniud.it (M. Miculan); michele.pasqua@univr.it (M. Pasqua)
https://users.dimi.uniud.it/~marino.miculan/ (M. Miculan); https://michelepasqua.github.io/ (M. Pasqua)
0000-0003-0755-3444 (M. Miculan); 0000-0002-9475-4836 (M. Pasqua)

are managed by a central coordinator node (often deployed on the cloud). This leads to several issues: scalability (as the IoT devices will increase exponentially), availability (what happens when the central node is not reachable?) and privacy (users' data are continuously sent to remote, unknown, servers, not under user's control). Thus, in these situations we may prefer to move computation closer to the edge of the network, akin *fog computing*: the ECA rules should be stored and executed directly on the nodes, in a truly distributed setting. This approach reduces data transfers between the edge and the center of the network—in fact, there can be no center at all, thus increasing scalability and resilience—but, on the other hand, it requires a distributed coordination and communication of these components.

To mitigate these issues, we introduce AbU (for "Attribute-based Updates"), a new calculus aiming at merging the simplicity of ECA programming with distributed coordination mechanisms in the spirit of *attribute-based communication* (AbC) [5]. Attribute-based communication is a time-coupled, space-uncoupled interaction model recently introduced for coordinating large numbers of components. The key aspect of attribute-based communication is that the actual receivers are selected "on the fly" by means of *predicates*.

Integrating attribute-based communication in the ECA paradigm is not obvious, since the first adopts a message-passing mechanism while the latter uses memory-based events. To solve the problem, in AbU attribute-based communication is reduced to events of the same kind ECA programs already deal with, that is, memory updates. For instance, the following AbU rule

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

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

This smooth integration of paradigms makes it easier to extend to the distributed setting known results and techniques. As an example, we will provide a simple syntactic check to guarantee *stabilization*, i.e., that a chain of rule executions triggered by an external event will eventually terminate.

2. Attribute-based Memory Updates

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

2.1. Syntax and Semantics

In AbU, a smart device is modeled by a *node* $R\langle\Sigma, \Theta\rangle$. R is a list of *ECA rules* of the form $\mathsf{evt} \gg \mathsf{task}$, where evt is the *event* and task is the *task*. The latter is a pair $\mathsf{cnd} : \mathsf{act}$, where cnd is the *condition* and act is the *action*. The full syntax of AbU ECA rules is depicted in Fig. 1. The configuration of the node is given by a *state* Σ, mapping resources to values, and an *execution pool* Θ, which is a set of *memory updates*. An update is a finite list of pairs (x, v), meaning

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

Figure 1: AbU syntax for ECA rules.

that the resource x will take the value v after the execution of the update. When one of the resources in a rule event is modified, the rule is *fired* and its task *evaluated*. Evaluation does not change the resource states immediately; instead, it yields update operations that are added to the execution pools, and applied later on.

A task *condition* is a boolean expression, optionally prefixed with @: 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, the action is evaluated on every node where the condition holds. An action is a finite list of assignments of value expressions to *local* x or *remote* $\overline{x}$. The evaluation of an action yields an update, which is added to the node pool of the local or remote nodes, in the case of local or remote tasks, respectively. As an example, the (remote) task $@\top : \overline{x} \leftarrow \overline{x} + x$ means "add the value of the current node x to the x of every other node". Finally, an AbU system S is just the parallel composition of AbU nodes, one for each smart device to model.

The behavior of an AbU system is specified by means of (small-step) semantics, modeled as a *labeled transition system* (LTS) $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_2$. The semantics is *distributed*, in the sense that each node does not have a global knowledge about the system. In the AbU semantics, that can be found in the original paper [1], a rule (Exec) executes an update picked from the pool; while a rule (Input) models an external modification of some resources. The execution of an update, or an external change in the resources, may trigger some rules of the nodes. Hence, after updating a node state, the semantics of a node launches a *discovery phase*, with the goal of finding new updates to add to the local pool (or some pools of remote nodes), given by the activation of some rules. The discovery phase is composed by two parts. Firstly, a node $R\langle\Sigma, \Theta\rangle$ performs a *local* discovery to add to the local pool Θ all updates originated by some rules in R that contain local tasks. Then, the node computes a list of remote tasks that may update *external* nodes and sends it to all nodes in the system. This is modeled with the LTS labels $\triangleright T$ and $\blacktriangleright T$, produced by the rules (Exec) and (Input), respectively. On the other side, when a node receives a list T of tasks, executing the rule (Disc) with a LTS label T, it evaluates them and adds to its pool the actions generated by the tasks whose condition is satisfied. Finally, the rule (Step) completes (on all nodes in the system) a discovery phase launched by a given node.

2.2. A Working Example

Consider the scenario sketched in the introduction, where an "access" node aims at sending its local access time to all "logger" nodes in the system. In other words, this node is activated when $accessT$ changes, i.e., when a user performs access. Suppose that the node, together with the

time-stamp, aims at sending the IP address of the user and the name of the accessed resource. On the other side, the logger nodes record the access time, the IP address and the resource name. Furthermore, these nodes contain a black-list of IP addresses. A logger node noticing an access from a black-listed IP is in charge of notifying an *intrusion detection system* (IDS). The system is formalized in AbU as follows, supposing to have two access and two logger nodes.

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

At the beginning, in the AbU system $\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4$ all pools are empty, so there is nothing to compute. At some point, an access is made on the resource `camera`, so the rule (Input) is applied on S_1, namely $R_a\langle\Sigma_1, \varnothing\rangle \xrightarrow{\blacktriangleright T} R_a\langle\Sigma_1', \varnothing\rangle$, where $\Sigma_1' = [accessT \mapsto 15:07:00 \;\; res \mapsto \texttt{camera} \;\; IP \mapsto 167.123.23.2]$ and $T = @(role = \texttt{logger}) : log \leftarrow \mathsf{append}\ log\ |167.123.23.2; 15:07:00; \texttt{camera}|$. Now, a discovery phase is performed on all other nodes. In particular, we have: $R_a\langle\Sigma_2, \varnothing\rangle \xrightarrow{T} R_a\langle\Sigma_2, \varnothing\rangle$, $R_l\langle\Sigma_3, \varnothing\rangle \xrightarrow{T} R_l\langle\Sigma_3, \Theta\rangle$, and $R_l\langle\Sigma_4, \varnothing\rangle \xrightarrow{T} R_l\langle\Sigma_4, \Theta\rangle$. Here, the pool Θ is $\{(log, |167.123.23.2; 15:07:00; \texttt{camera}|)\}$.

Now, the third and the fourth nodes can apply an execution step, since their pools are not empty. Suppose the third node is chosen, namely we have $R_l\langle\Sigma_3, \Theta\rangle \xrightarrow{\triangleright\varepsilon} R_l\langle\Sigma_3', \varnothing\rangle$, by applying the rule (Exec), and $\Sigma_3' = [role \mapsto \texttt{logger}\ log \mapsto |167.123.23.2; 15:07:00; \texttt{camera}|\ Blist \mapsto \varnothing \;\; IDS \mapsto \varepsilon]$. Note that, in this case, no rule is triggered by the executed update. Since there is nothing to discover, all the other nodes do not have to update their pool. Finally, the fourth node can execute, namely we have that $R_l\langle\Sigma_4, \Theta\rangle \xrightarrow{\triangleright\varepsilon} R_l\langle\Sigma_4', \Theta'\rangle$, by applying the rule (Exec). Here, $\Sigma_4' = [role \mapsto \texttt{logger} \;\; log \mapsto |167.123.23.2; 15:07:00; \texttt{camera}|\ Blist \mapsto 167.123.23.2; \;\; IDS \mapsto \varepsilon]$ and $\Theta' = \{(IDS, |167.123.23.2; 15:07:00; \texttt{camera}|)\}$. In this case, the execution of the update triggers a rule of the node but the rule is local so, also in this case, the discovery phase does not have effect. Since all pools are empty, the execution stops, until a new input (i.e., an external change to resources) is performed.

2.3. Waves and Stabilization

An AbU system S is *stable*, when no more execution steps can be performed, namely when all execution pools of all nodes in S are empty. In the case of a stable system, only the rule (Input) can be applied, i.e., an external environment change is needed to (re)start the computation.

We can define a (big-step) semantics $\mathsf{S} \rightsquigarrow \mathsf{S}'$ between stable systems, dubbed *wave semantics*, in terms of the small-step semantics of AbU (see again the original paper [1] for details). The idea is that a (stable) system reacts to an external stimulus by executing a series of tasks (a "wave"), until it becomes stable again, waiting for the next stimulus. Note that, in the wave semantics inputs do not interleave with internal steps: this leaves the system the time to reach stability

before the next input. If we allow arbitrary input steps during the computation, a system may never reach stability since the execution pools could be never emptied. This assumption has a practical interpretation: in the IoT context, usually internal computation steps take much less time of changes in the environment, i.e., input from sensors [6].

The wave semantics (and, hence, an AbU system) may exhibit *internal divergence*: once an input step starts the computation, the subsequent execution steps may not reach a stable system, even if intermediate inputs are not performed. In order to prevent this situation, we define a simple syntactic condition on the ECA rules, that guarantees (internal) termination. Each system satisfying the condition eventually becomes stable, after an initial input and without further interleaving inputs. The idea is to build an *ECA dependency graph* expressing dependencies between the resources handled by an AbU system: a resources x in the graph is linked to another resource y in the graph when a change of x may potentially affect the value of y.

To build such graph we need: the *output resources* of an ECA rule, namely the resources involved in the actions performed by the rule (they are given by the resources assigned in the rule task); and the *input resources* of an ECA rule, namely the resources that the rule depends on (they are the resources in the rule event). Then, the ECA dependency graph of a system S is a directed graph (N, E) where nodes N are the input and output resources of all ECA rules in S, while the edges E link two resources x and y if x is an input resource of a rule in S and y is an output resource of the same rule. Indeed, this means that y is potentially assigned by a rule in S when x is changed. The sufficient syntactic condition for the termination of the wave semantics (i.e., stabilization) consists in the acyclicity of the ECA dependency graph.

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

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

3. Conclusions and Future Work

In this paper we have presented AbU, a new calculus merging the simplicity of ECA programming with *attribute-based memory updates*. This provides us an effective method for programming IoT smart devices in a fully-distributed setting, without sacrificing the simplicity typical of ECA rules. Moreover, AbU can be used as a solid foundational model for edge computation.

Future work First, we are interested in porting to AbU the verification techniques developed for other ECA languages [7, 8, 9]. Another interesting issue is *distributed runtime verification and monitoring*, in order to detect violations at runtime of given correctness properties, e.g., expressed in temporal logics like the μ-calculus [10]. Similarly, we can define syntactic criteria and corresponding verification mechanisms to guarantee *confluence*. Indeed, in practical IoT scenarios, it is important to ensure that the rules execution order does not impact the overall behavior (which is a sort of rule determinism). Along this line, in IoT systems it is often important to guarantee that inputs are processed within precise time bounds; to this end,

we can think of adding quantitative aspects to the AbU semantics, following [11]. Finally, a new generalization of attribute-based communication has been presented in [12]; it could be interesting to investigate how to apply that interaction model to the ECA programming.

References

[1] M. Miculan, M. Pasqua, A calculus for attribute-based memory updates, in: A. Cerone, P. C. Ölveczky (Eds.), Theoretical Aspects of Computing – ICTAC 2021, Springer International Publishing, Cham, 2021, pp. 366–385. doi:10.1007/978-3-030-85315-0_21.

[2] J. Cano, E. Rutten, G. Delaval, Y. Benazzouz, L. Gurgen, ECA rules for IoT environment: A case study in safe design, in: 8th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops, IEEE, USA, 2014, pp. 116–121. doi:10.1109/SASOW.2014.32.

[3] M. Balliu, M. Merro, M. Pasqua, Securing cross-app interactions in IoT platforms, in: 32nd IEEE Computer Security Foundations Symposium, Hoboken, NJ, USA, IEEE, 2019, pp. 319–334. doi:10.1109/CSF.2019.00029.

[4] M. Balliu, M. Merro, M. Pasqua, M. Shcherbakov, Friendly fire: Cross-app interactions in IoT platforms, ACM Trans. Priv. Secur. 24 (2021) 16:1–16:40. doi:10.1145/3444963.

[5] Y. Abd Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, R. Vigo, A calculus for attribute-based communication, in: 30th Symposium on Applied Computing, ACM, 2015, pp. 1840–1845. doi:10.1145/2695664.2695668.

[6] D. R. Cacciagrano, R. Culmone, IRON: Reliable domain specific language for programming IoT devices, Internet Things 9 (2020) 100020. doi:10.1016/j.iot.2018.09.006.

[7] X. Jin, Y. Lembachar, G. Ciardo, Symbolic verification of ECA rules, in: D. Moldt (Ed.), Joint Proceedings of PNSE '13 and ModBE '13, Milano, Italy, volume 989, CEUR-WS.org, 2013, pp. 41–59. URL: http://ceur-ws.org/Vol-989/paper17.pdf.

[8] C. Vannucchi, M. Diamanti, G. Mazzante, D. R. Cacciagrano, F. Corradini, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, vIRONy: A tool for analysis and verification of ECA rules in intelligent environments, in: Int. Conf. on Intell. Environ., S. Korea, IEEE, 2017, pp. 92–99. doi:10.1109/IE.2017.32.

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

[10] M. Miculan, On the formalization of the modal μ-calculus in the Calculus of Inductive Constructions, Inf. Comput. 164 (2001) 199–231. doi:10.1006/inco.2000.2902.

[11] M. Miculan, M. Peressotti, Structural operational semantics for non-deterministic processes with quantitative aspects, Theor. Comput. Sci. 655 (2016) 135–154. doi:10.1016/j.tcs.2016.01.012.

[12] M. Miculan, M. Paier, A calculus of subjective communication, in: U. Dal Lago, D. Gorla (Eds.), 23rd Italian Conference on Theoretical Computer Science (ICTCS), Proceedings, CEUR Workshop Proceedings, CEUR-WS.org, 2022, pp. 1–13.