Internet of Things 9 (2020) 100020

Contents lists available at ScienceDirect

Internet of Things

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

IRON: Reliable domain specific language for programming IoT devices

D.R. Cacciagrano, R. Culmone*

Computer Science Department, University of Camerino, Camerino, Italy

ARTICLE INFO

Article history:
Received 31 August 2018
Accepted 9 September 2018
Available online 25 September 2018

Keywords:
Domain-specific languages
IoT
Formal specification
ECA rules

ABSTRACT

A domain-specific language (DSL) is a programming language that is specialized to a particular application domain. IRON is a DSL for the IoT domain which allows not only to program in an easy way using the Event-Condition-Action (ECA) rules but also to prevent incorrect actions. In this paper, we formally describe the semantics of IRON.

The anomalies that IRON prevents are: (i) the presence of cycles that determine the non-termination, (ii) the ambiguous actions that do not allow the definition of a final configuration, (iii) the breaking of invariances. In addition to the formal description of IRON, an interpreter was created in a host language (LUA) that captures and manages the three anomalies. This provides a general scheme for the implementation of languages based on ECA rules.

1. Introduction

In the fields of embedded systems and Internet of Things (IoT), smart environments [1] are systems which react to changing conditions and users behaviours. Such reactive systems can be specified with Event-Condition-Action (ECA) rules [2,3], which provides for a high-level and flexible description. An ECA rule performs an action A if the condition C is satisfied when the event E occurs. It enables systems to respond to events that occur in an arbitrary order as well as combinations of events. Programming ECA rules is an error-prone process [4,5]. For instance, the interaction of rules that make use of the same component and are defined by different users can easily create safety problems [6,7].

The execution of ECA rule-based systems can generate different problems [8–10] such as redundancy, inconsistency and circularity, as well as application-specific safety issues. Very often ECA-based languages only implement procedural semantics, lacking a precise formal semantics. This is a severe drawback for many real-world applications where validation and traceability of the effects that are generated by events and the triggered by actions are crucial. Thus, it is essential to provide a complete formal semantics that can determine the reliability of the produced results.

In this paper we face this challenge. Taking into account a subset of IRON (Integrated Rule On Data) [11–13], a Domain-specific language (DSL) for programming smart environments by means of ECA rules, we program an interpreter of the given sub-language able to signal several program anomalies (in detail, contradictory, cyclical and ambiguous program configurations) at runtime.

We first proceed defining a formal semantics for the considered sublanguage. Such a semantics formally describes the execution model of IRON programs as well as detects inadmissible, cyclical and ambiguous program configurations. Then,

* Corresponding author.
E-mail addresses: diletta.cacciagrano@unicam.it (D.R. Cacciagrano), rosario.culmone@unicam.it (R. Culmone).

https://doi.org/10.1016/j.iot.2018.09.006
2542-6605/

```
physical sensor logical motion node(1,1)
logical actuator integer c
rule inc1 on motion when true then c=c+1
```

Listing 1. No safe.

```
physical sensor logical motion node(1,1)
logical actuator integer counter where counter < MAXINT
rule inc2 on motion when true then c=c+1
```

Listing 2. With invariance.

```
physical sensor logical motion node(1,1)
logical actuator integer counter where counter < MAXINT
rule inc3 on motion when c < MAXINT then c=c+1
```

Listing 3. Safe.

the interpreter is built on the top of the given semantics, inheriting automatically the capability to detect such anomalies as program states with no semantics.

To give an idea, consider the following three programs in IRON.

Each of the above programs increments a counter whenever a motion sensor changes its value. However, the interpreter signals semantic errors in Listings 1 and 2, but not in Listing 3. In detail, in Listing 1 the interpreter stops on the first configuration where the counter value exceeds the greatest expressible integer (inadmissible configuration), and in Listing 2 when the counter value breaks the invariance `counter < MAXINT` (breaking invariance configuration). On the contrary, in Listing 3 the rule *inc3* is applied until the counter value is the greatest expressible value, without any errors.

It is worthy of noting that the interpreter is programmed in LUA [14] and IRON programs, as a consequence, are translated in LUA too to be executed by the interpreter. The choice of translating the whole IRON programming framework (programs and interpreter) in LUA is not casual. In fact, the IRON programming paradigm is thought to go towards a distributed execution on board of IOT devices. The LUA code can be helpful to fulfil this idea, being a lightweight and portable code even on smaller devices, i.e., suitable for embedded devices which typically populate an IoT scenario.

1.1. Plan of the paper

In Section 2 we briefly recall the IRON [11–13] language. Section 3 is the core of the paper. Here, we first define the formal semantics [15] of a subset of IRON, in such a way specific program anomalies and application-specific safety issues can be intercepted. Then we propose an interpreter in LUA language built on the top of the defined semantics, obtaining an interpreter able to detect anomalous configurations as states with no semantics. Finally, Section 4 provides conclusions and future work.

2. IRON

In this section we briefly recall IRON [11–13]. This is a language based on the first-order predicate language. IRON supports the categorisation of devices into sets, allows the definition of properties over sets and supports multicast and broadcast abstractions. To ease the presentation, we use an extended BNF form to describe the IRON syntax. The operator $(x)^*$ means zero or more repetitions of x, the expression $(x)+$ stands for one or more repetitions of x while $[x]$ represents an optional occurrence of x. The grammar of IRON is shown in Table 1.

2.1. IRON static part

An IRON program has a static and a dynamic part. The *static part* is composed of variables declarations (these variables can be sets, physical and logical devices) plus global constraints defined over them by using first-order logic formulae. A *physical device* defines a piece of hardware that is physically installed in the environment, it has a type (i.e., integer or boolean) and can be either a sensor or an actuator. A physical device has a name and is characterised by the syntax *node*(*id*; *id*) where the first *id* is an identifier that uniquely identifies the physical node while the second *id* uniquely identifies a sensor/actuator that is installed on the node. The keyword *in* can be added to specify a list of sets the physical device belongs to. IRON also supports the definition of *logical devices*. A logical device can be set according to the values observed over different sensors and actuators, and thus it produces information that would be impossible to get by considering a single physical device. A logical sensor/actuator does not specify any *node*(*id, id*) keyword but must specify an initial value (line 6 – 8 of the grammar). The static part also includes the declaration of *constraints* (specified by the keyword *where*), i.e.,

Table 1
Syntax of IRON.

```
1  Program ::= (Device|Rule|VarDecl)+ ;
2  Device ::= PhysicalDevice|LogicalDevice|Set
3  PhysicalDevice ::= physical(sensor|actuator) Type Id [= Exp]
4    node(Id Sep Id) [in id (Sep Id )*] [whereBoolExp] ;
5  LogicalDevice ::= logical(sensor|actuator) Type Id = Exp
6    [in Id(Sep Id )*] [where BoolExp]
7  Set ::= set(sensor|actuator)Type Id
8  Rule ::= rule Id on Id(Sep Id)* when
   BoolExp then Action [else Action]
9  Action ::= [Id = Exp ]+
10 Exp ::= BoolExp|IntExp
11 BoolExp ::= CompIntExp| (BoolExp BoolBinaryOp BoolExp)|
12   (BoolUnaryOp BoolExp)|PrimaryBoolExp
13 CompIntExp ::= IntExp CompOp IntExp
14 PrimaryBoolExp ::= (LRB BoolExp RRB) | BoolConst|Id
15 IntExp ::= UnaryIntExp | (IntExp IntBinaryOp IntExp)
16 UnaryIntExp ::= (IntUnaryOp UnaryIntExp) | PrimaryIntExp
17 PrimaryIntExp ::= (LRB IntExp RRB) | IntConst | Id
18 Type ::= integer | boolean
19 IntConst ::= [-] Digit (Digit)*
20 Digit ::= 0|1|2|3|4|5|6|7|8|9
21 BoolConst ::= true|false
22 Letter ::= <character>
23 Id ::= Letter(Letter|Digit)*
24 IntBinaryOp ::=  +|-|*|/
25 BoolBinaryOp ::= and|or
26 BoolUnaryOp :== not
27 CompOp ::= ==|!=|<|>|<=|>=
28 IntUnaryOp ::= +|-
29 SetOp ::= all|any|one|no|lone
30 Sep ::= ,
31 LRB ::= (
32 RRB ::= )
33 VarDecl ::= BoolVarDecl|IntVarDecl ;
34 BoolVarDecl ::= boolean Id [= BoolExp] [where BoolExp]
35 IntVarDecl ::= integer Id [= IntExp] [where BoolExp]
```

laws that various variables, devices and sets must always satisfy. Constraints can be used to specify rules that bind variables together. The use of a constraint has two applications:

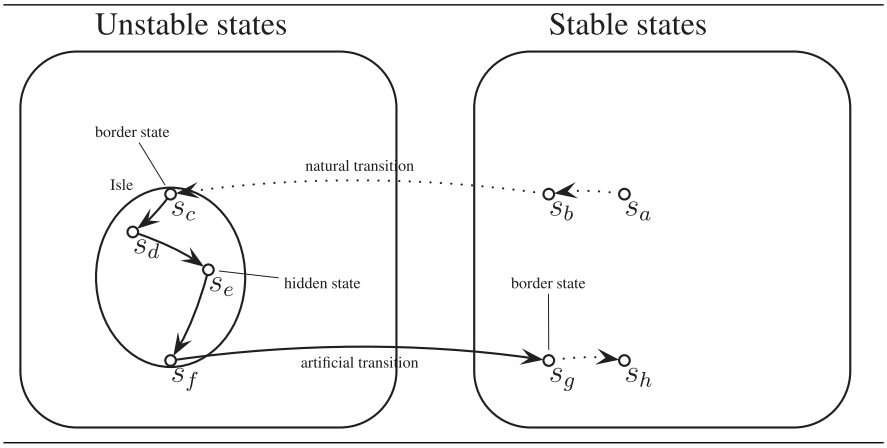
- it defines valid states of the system regardless of the rules that are defined
- it is used at run-time to verify whether any physical device is providing erroneous data.

Sets (line 2 of the grammar) are considered to be logical devices and are used to group together either sensors or actuators of the same type (line 9 of the grammar). A programmer can assign values to a set that contains actuators. This assignment can be used to instruct all the actuators to perform a specified action. Effectively, a set assignment is an abstraction of a multicast communication primitive that can be used to communicate to actuators an action to be performed. A programmer can read the value of a set of sensors to define events and specify conditions. To this end various set operators are introduced in the next section.

2.2. IRON dynamic part

The *dynamic part* of IRON is composed of ECA rules that are defined by the programmer. The monitoring and control actions are specified by using ECA rules. A rule has a name and is composed of three different parts that are *on, when* and *then* (line 10 of the grammar). A list of variables follows the *on* keyword. Whenever one of them changes its value, the boolean expression that follows the keyword *when* is evaluated. When this expression is evaluated to true the rule can be applied, and the actions listed after the keyword *action* can be executed. A boolean expression can include relational and logical operators, integers, devices, variables and functions. An action is a list of assignments to variables, physical actuators and logical devices. Special operators are used to supporting the definition of a boolean condition over a set: *all, any, no, one* and *lone*. *All* is a universal operator that allows the definition of conditions that must be satisfied by all devices belonging to the set. *Any* is an existential operator that can be used in to specify that at least one of the element of the set must satisfy

Table 2
The IRON program behaviour.



the condition. *No* (*one*) is used when we need to express that no (respectively, precisely one) element of the set must verify the specified condition. *lone* is used when we need to express that at most one element of the set must verify the specified condition.

2.3. IRON informal semantics

In this section we informally describe the IRON execution model. Without loss of generality, we will give the semantics on a syntax without syntactic sugar and we assume systems which do not include the definition of sets and the distinction between logical and physical devices as detailed in Table 1. These could be introduced at the cost of additional notation but do not affect the overall partitioning strategy described below.

Since we consider a finite set of devices that can assume a finite set of values, it is possible to represent the behaviour of a generic IRON program as a Finite State Automaton (FSA) [16], where the edges correspond to (two kinds of) transitions of the system and the vertices to its states (in our case, a state is a function φ: $D \rightarrow V$ where D is the set of labels that identify the sensors and the actuators of the system, and V is a finite set of integer or boolean values). Fig. 2 is an example of IRON FSA, where:

- Vertexes are all states (i.e., configurations of sensors and actuators) which satisfy all invariants;
- There are two different kinds of transitions: *artificial transitions*, resulting from the activation of ECA rules, and *natural transitions*, resulting from changes in the environment. According to this partition, we can distinguish between *stable* and *unstable* states. The system is in a *stable state* if only natural transitions can be applied whereas the system is in *unstable states* when only artificial transitions can be applied.

The representation of the evolution of the system is based on two essential hypotheses: (1) the initial admissible configuration of the system is given by an external entity; (2) artificial transitions take a much shorter time than natural transitions.

An IRON program behaviour always matches the execution schema shown in Fig. 2.

3. A concrete interpreter for IRON

In this section we build a simple IRON interpreter in LUA [14] embedding the verification of *loop, break invariances and ambiguity* conditions.

We start defining the IRON semantics by a set of formal rules allowing us to generate IRON program behaviours in the form of FSAs as in Fig. 2, where no state is *inadmissible, involved in a loop* and *ambiguous*. In detail, we say that a state

- is *admissible*, when it breaks no invariance;
- is *not involved in a loop*, when it does not belong to any cycle of transitions, generated by applying ECA rules;
- is not *ambiguous*, when it is a stable state and it is the unique stable state generated by the application of a given set of ECA rules.

The semantic rules are defined to assign no semantics to states violating at least one of the above conditions.

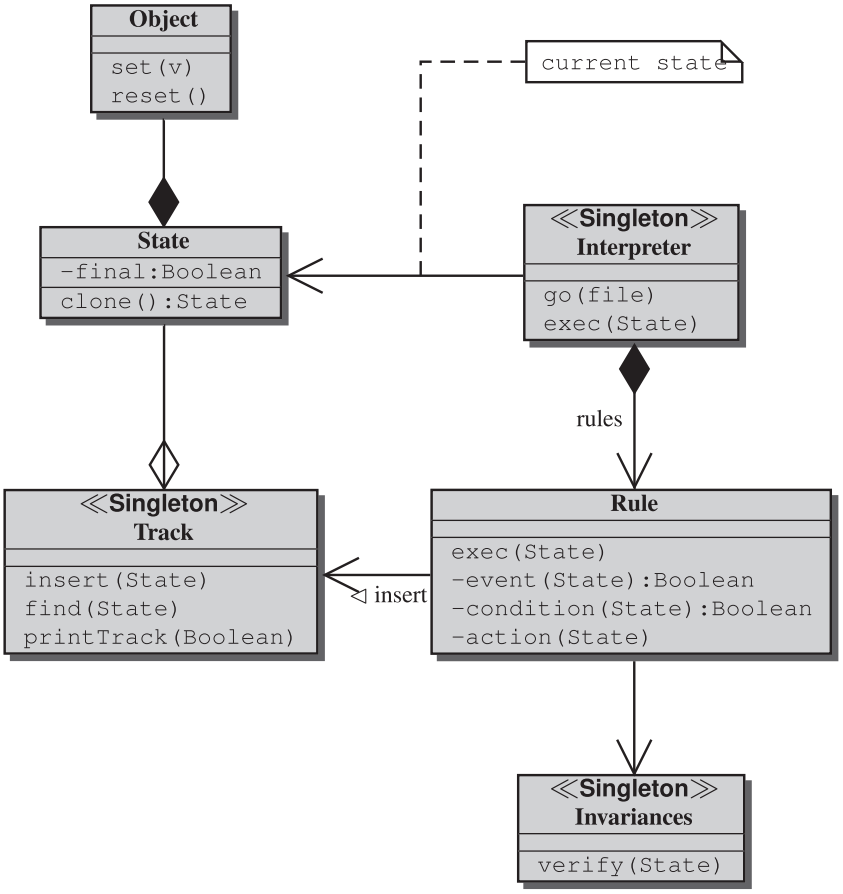


Fig. 1. Class diagram of Interpreter.

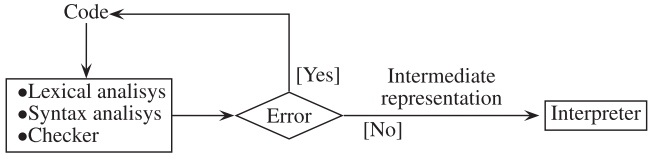


Fig. 2. Diagram for a possible language implementation.

3.1. Formal operational semantics step by step

The IRON operational semantics is defined starting from the IRON basic constructs, i.e., arithmetic expression, invariants, events, conditions and actions. The operational semantics of basic constructs allow us to define the semantics of composed constructs (e.g., rules, set of rules), up until natural and artificial transitions.

Declarations. Let $\mathcal{D}$ be the set of all system devices that are composed of two disjoint sets $\mathcal{I}$ and $\mathcal{O}$ whose elements are, respectively, the sensors ($\mathcal{I}$ means *Input*) and the actuators ($\mathcal{O}$ means *Output*). These sets are defined in the first part of a program as declarations that are generated by the following syntax:

Δ	::=	$\delta \mid \delta\ ;\ \Delta$	declarations
δ	::=	i in (int | bool)	sensor
	|	o out int = z	int actuator
	|	o out bool = (true | false)	bool actuator

where i and o are variable identifiers and z is a literal representing an integer number. Note that only output devices require an explicit initialization because sensors will take an initial value when the system starts. A list of declarations Δ denotes a pair $\langle\theta, \varphi_0\rangle$ where θ is a function recording the type of the declared variables and φ_0 is a state recording their initial value. A state φ is a function that maps each variable in $\mathcal{D}$ into its value. The set of all states is denoted by Φ. The set of all possible variable values is $\text{Val} = \mathbb{Z} \cup \{\texttt{tt}, \texttt{ff}\} \cup \{\omega\}$, where ω is the undefined value, i.e., the initial value of

all sensor variables. The actuator variables are assigned, according to the declaration, with the integer number denoted by the corresponding literal z or with the corresponding boolean value. In the following we suppose that $\mathcal{I} = \{i_1, \ldots, i_n\}$ and $\mathcal{O} = \{o_1, \ldots, o_m\}$ such that $\mathcal{I} \cap \mathcal{O} = \emptyset$ and $\mathcal{D} = \mathcal{I} \cup \mathcal{O}$. Thus, the evaluation of a declaration produces an initial state φ_0 that can be represented as follows:

$$\varphi_0 = \{i_1 \mapsto \omega, \ldots, i_n \mapsto \omega, o_1 \mapsto v_0^1, \ldots, o_m \mapsto v_0^m\}$$

The evaluation of a declaration also produces a total function $\theta : \mathcal{D} \to \{\texttt{int}, \texttt{bool}\}$ recording the declared type of all variables.

Arithmetic expressions. They are defined, with the usual operators, by the following grammar:

exp	::=	d	integer variable
	|	z	integer literal
	|	exp # exp	arithmetic operators

$$\mathcal{E}[\![\text{exp}]\!]_\varphi = \begin{cases} \bot & \text{if} \quad \text{exp} = d \wedge \theta(d) \neq \texttt{int} \\ \varphi(d) & \text{if} \quad \text{exp} = d \wedge \theta(d) = \texttt{int} \\ z & \text{if} \quad \text{exp} = \texttt{z} \\ \mathcal{E}[\![\text{exp}_1]\!]_\varphi \ \# \ \mathcal{E}[\![\text{exp}_2]\!]_\varphi & \text{if} \quad exp = \text{exp}_1 \ \# \ \text{exp}_2 \end{cases}$$

Conditions. A condition c is a boolean expression whose truth value depends on the values of the variables in $\mathcal{D}$. These values are defined by the state φ. The definition of c is based on the following grammar:

c	::=	d	boolean variable
	|	(true | false)	boolean constant
	|	c $\wedge$ c | c $\vee$ c | !c	logical operators
	|	exp @ exp	relational operators

where $@ \in \{=, <, \leq, >, \geq, <>\}$. The semantics of a condition is given by a semantic evaluation function $\mathcal{C}[\![c]\!]_\varphi$ which gives the boolean value $b \in \{\texttt{tt}, \texttt{ff}\}$ of an expression c in a state φ and is defined as follows[1]:

$$\mathcal{C}[\![c]\!]_\varphi = \begin{cases} \bot & \text{if} \quad c = d \wedge \theta(d) \neq \texttt{bool} \\ \varphi(d) & \text{if} \quad c = d \wedge \theta(d) = \texttt{bool} \\ \texttt{tt (ff resp.)} & \text{if} \quad c = \texttt{true (false resp.)} \\ \mathcal{C}[\![c_1]\!]_\varphi \wedge (\vee \text{ resp.}) \ \mathcal{C}[\![c_2]\!]_\varphi & \text{if} \quad c = c_1 \wedge \ (\vee \text{ resp.}) \ c_2 \\ !\mathcal{C}[\![c_1]\!]_\varphi & \text{if} \quad c = !c_1 \\ \mathcal{E}[\![\text{exp}_1]\!]_\varphi \ @ \ \mathcal{E}[\![\text{exp}_2]\!]_\varphi & \text{if} \quad c = e_1 \ @ \ e_2 \end{cases}$$

Moreover, we define a satisfaction operator $\models$ such that $\varphi \models c$ if and only if $\mathcal{C}[\![c]\!]_\varphi = \texttt{tt}$.

Invariants. Invariants μ are particular conditions that are used to express constraints on the possible values of variables. Invariants assign the values of sensors or actuators variables (of type int) to finite integer intervals. Invariants also express environmental constraints such as "if the light is on then the light sensor must be greater than 5". They are generated by the following grammar:

μ	::=	[c]	single invariant
	|	[c] μ	list of invariants

We denote with Inv the set of invariants generated by the above grammar. The semantics of an invariant $\mu = [c_1] \cdots [c_n]$ is defined by a semantic evaluation function $\mathcal{M}$ as follows:

$$\mathcal{M}[\![\mu]\!] = \{\varphi \in \Phi \mid \forall i \in \{1, \ldots, n\} \ \varphi \models c_i\}$$

Thus, an invariant denotes the set of all *admissible* states, i.e., those states that satisfy the conditions $c_1, \ldots, c_n$. The satisfaction relation $\models$ is naturally extended to invariants in the following way: $\varphi \models \mu$ if and only if $\varphi \in \mathcal{M}[\![\mu]\!]$.

Events. An *event* e is defined as a subset of $\mathcal{D}$. The notation $e = \{d_1, \ldots, d_k\}$ represents the fact that all variables $d_i \in e$ changed their values with respect to a previous reading. The values of sensor variables can be changed only by the *environment*. Their changing is monitored during a specific time window at the end of which an event $e \subseteq \mathcal{I}$ is generated. The values of actuator variables can only be changed by the execution of a set R of all activated ECA rules. Thus, the execution of R generates an event $e \subseteq \mathcal{O}$ containing only the variables that changed their values. We denote with E the set of events.

Actions. Action has the following form: $a = \{o_{i_1} \leftarrow \eta_{i_1}, \ldots, o_{i_h} \leftarrow \eta_{i_h}\}$. where $\eta_{i_1}, \ldots, \eta_{i_h}$ are assignments to a subset of actuators. We denote with Act the set of actions: notice that the action in Act can only change the value of actuator variables.

Rules. The rules in IRON programs are defined by the following ECA syntax:

r	::=	e [c]/a	where e $\in$ E and a$\in$ Act
R	::=	r | r ; R	list of rules

[1] As for arithmetic expressions, with abuse of notation we denote by $\wedge$, $\vee$, ! and @ both the syntactic and the semantics operators.

```
initState=State.new {tree=Object.new(true,0)}
Invariances={function(s) return s.tree.value < 4; end}
Rules={Rule.new(function(s) return s.tree.event end,--event
                function(s) return s.tree.value > -5 and s.tree.value < 5 end,--condition
                function(s) s:reset('tree'); s:set('tree',s.tree.value-1); end),--action
       Rule.new(function(s) return s.tree.event end,
                function(s) return s.tree.value > -5 and s.tree.value < 5 end,
                function(s) s:reset('tree'); s:set('tree',s.tree.value+1); end)}
```

Listing 4. Simple program.

i.e., r denotes an ECA rule and R represents a set of rules. The condition c must be non-empty (eventually equal to "true"). An IRON program is defined as follows:

Prog	::=	$\Delta^+\ \mu^*\ r^+$	IRON program

Semantic domains. We define the following semantic domains:

Int	$\subseteq$	$\mathbb{Z}$	Int is a finite set of integers
Bool	=	{tt, ff}	true and false values
Val	=	Int $\cup$ Bool	
Φ	=	$\{\ \varphi \mid \varphi\colon D \to Val\ \}$	

Semantics of admissible states. We denote by $\mathcal{M}[\![\mu]\!]$ the set of all *admissible* states, i.e., states φ that satisfy the invariant μ. Formally φ is an admissible state iff $Feasible_\mu(\varphi)$, where

$$Feasible_\mu(\varphi) = \begin{cases} \texttt{tt} & \text{if} \quad \varphi \in \mathcal{M}[\![\mu]\!] \\ \texttt{ff} & \text{if} \quad \varphi \notin \mathcal{M}[\![\mu]\!] \end{cases}$$

We can finally define the semantics of IRON programs.

Semantics of an action. Let ξ, ξ' be subsets of labels and Σ a set of φ. We introduce the symbol $\longrightarrow_a$ and we define the semantics of an action as follows:

$$\frac{\begin{array}{c} a = \{o_{i_1} \leftarrow \eta_{i_1}, .., o_{i_h} \leftarrow \eta_{i_h}\},\ \forall j = 1...h, \mathcal{E}[\![\eta_{i_j}]\!]_\varphi = \mathrm{v}_{i_j}, \\ \varphi' = \varphi[\mathrm{v}_{i_j}/o_{i_j}],\ \xi' = \{o_{i_j} \mid j = 1...h\},\ \varphi' \notin \Sigma \end{array}}{< a, \varphi, \xi, \Sigma > \longrightarrow_a < \varphi', \xi', \Sigma \cup \{\varphi'\} >} \tag{1}$$

An action a (in the state φ) that has events labelled in ξ changes the state from φ to φ' and generates new events ξ' only if φ' is not already in Σ. This condition avoids to give semantics to a state φ' that has been already generated (i.e., belonging to a loop).

Semantics of a rule. Let $R = \{r_1, \ldots, r_n\}$ be a set of rules, we introduce the symbol $\overset{i}{\longrightarrow}_r$ and we define the semantics of a rule r_i as follows:

$$\frac{r_i = e[c]/a,\ \xi \cap e \neq \emptyset,\ \varphi \models c,\ < a, \varphi, \xi, \Sigma > \longrightarrow_a < \varphi', \xi', \Sigma' >}{< r_i, \varphi, \xi, \Sigma > \overset{i}{\longrightarrow}_r < \varphi', \xi', \Sigma' >} \tag{2}$$

The evaluation of the rule r_i changes the state φ in φ' and activates the events ξ' if there exists a list of events ξ and the evaluation of the condition c in the state φ is true.

Semantics of a set of rules R. The execution of rules in R for IRON programs is non-deterministic. At each step, it is possible to apply any rule in R among those ones that can be activated. When there are no rules that can be activated, the final state is said to be *stable*.

More precisely, all rules in R must be applied by using the semantic rule (3) (in any order) until a stable state φ is reached (this is denoted by $Stable_R(\varphi)$).

Note that a set of rules R produces a single stable state φ'. In cases of non-confluence (i.e., ambiguity) the semantic rule (3) is inapplicable. This is the way we do not assign semantics to ambiguous states.

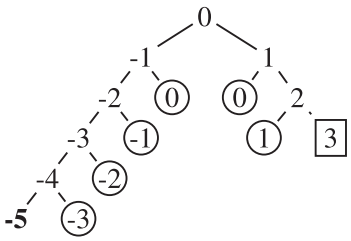
$$\frac{\begin{array}{c} R = \{r_1, \ldots, r_n\}, \\ n > 0\ \ \exists i \in \{1, \ldots, n\}, < r_i, \varphi, \xi, \Sigma > \overset{i}{\longrightarrow}_r < \varphi', \xi', \Sigma' > \end{array}}{< R, \varphi, \xi, \Sigma > \overset{i}{\longrightarrow}_R < R, \varphi', \Sigma' >} \tag{3}$$

Formally φ is a stable state w.r.t. a set of rules $R = \{r_1, \ldots, r_n\} (n > 0)$ iff $Stable_R(\varphi)$, where

$$Stable_R(\varphi) = \begin{cases} \texttt{ff} & \text{if} \quad \exists \xi, \Sigma \text{ such that } < R, \varphi, \xi, \Sigma > \overset{i}{\longrightarrow}_R \\ \texttt{tt} & \text{otherwise} \end{cases}$$

Table 3
Output and trace.

```
Ambiguity
Loop
Break invariance
[1] tree=-1[true] final=false
[2] tree=-2[true] final=false
[3] tree=-3[true] final=false
[4] tree=-4[true] final=false
[5] tree=-5[true] final=true
[6] tree=0[true] final=false
[7] tree=1[true] final=false
[8] tree=2[true] final=false
[9] tree=3[true] final=true
```



Semantics of artificial transitions. Artificial transitions are defined by using the following rule:

$$\frac{Feasible_{\mu}(\varphi), Feasible_{\mu}(\varphi') \quad R=\{r_1,\ldots,r_n\}, \quad <R,\varphi,\xi,\Sigma> \xrightarrow{i}_R <R,\varphi',\xi',\Sigma'>}{\varphi \xrightarrow{i}_A \varphi'} \tag{4}$$

It is possible to prove that $Stable_R(\varphi))$ if and only if $\varphi \not\xrightarrow{i}_A$. In other words, a stable state is a state which cannot evolve via artificial transitions.

Semantics of natural transitions. Natural transitions are introduced to describe the semantic execution of IRON programs in response to the natural evolution of the system, i.e., sensors change their values because of environmental changes.

$$\frac{Feasible_{\mu}(\varphi),\ Feasible_{\mu}(\varphi'),\ Stable_R(\varphi),\ Stable_R(\varphi') \quad R=\{r_1,\ldots,r_n\},\ \varphi'=\varphi[^{v_1}/_{i_1},\ldots,^{v_n}/_{i_n}],\ n>0}{<R,\varphi,\xi> \longrightarrow_{jump} <R,\varphi',\{i_1,\ldots,i_n\}>} \tag{5}$$

$$\frac{Feasible_{\mu}(\varphi),\ Feasible_{\mu}(\varphi'),\ Stable_R(\varphi),\ \neg Stable_R(\varphi') \quad R=\{r_1,\ldots,r_n\},\ \varphi'=\varphi[^{v_1}/_{i_1},\ldots,^{v_n}/_{i_n}],\ n>0}{<R,\varphi,\xi> \longrightarrow_{jump} <R,\varphi,\{i_1,\ldots,i_n\},\{\varphi\}>} \tag{6}$$

$$\frac{<R,\varphi,\xi> \longrightarrow_{jump} <R,\varphi',\xi'> \text{ or } <R,\varphi,\xi> \longrightarrow_{jump} <R,\varphi',\xi',\Sigma>}{\varphi \longrightarrow_N \varphi'} \tag{7}$$

Rule (7) says that a natural transitions always starts from a stable state and can produce a state which can be stable (rule (5)) or unstable (rule (6)). The first case enables again rule (7), while in the latter only rule (6) can be applied, skipping to the artificial transition evaluation (rule (4)). Notice that both rules (5) and (6) capture the changes of sensors.

To sum it up, the behaviour of an IRON program equipped with ECA rules R starts from an admissible configuration of sensor and actuator values (it can be considered the initial state). If the initial state is unstable w.r.t. R, rule (4) is repeatedly applied producing admissible and not ambiguous states – where only actuator values change along the computational branch – until a stable state is generated or until a loop is detected. If the initial state or the generated state is stable w.r.t. R, rule (7) is repeatedly applied producing admissible states – where only sensor values change along the computational branch.

Notice that, differently from the previous case (i.e., in the case of artificial transitions), in this case (i.e., in the case of natural transitions) we admit the possibility that the environment can change sensor values in a cyclic way. If along this computation branch an unstable state is generated, we switch again applying rule (4), and so on. Notice that rules (4) and rule (7) are mutually exclusive.

3.2. Interpreter

Now we are ready to describe the interpreter which embeds the semantics defined in the previous section. The state diagram of a program is dynamically generated by the algorithm in a Breadth-First-Search like manner. It means that states are recursively and concurrently generated trying to apply all possible rules in a non-deterministic way (i.e., implementing rule (4)) until no rule can be applied (i.e., implementing rule (5)). In such a case the semantic is given by the last generated state (or states). The termination is guaranteed since any state element ranges over a finite set of values.

In the class diagram in Fig. 1 the *Rule::exec(State)* is the recursive function that is calling the first time on an initial state. The *Rule::exec(State)* applies all rules in *Rules*. When a rule is called with *Rule::exec(State)* and is performed, then a copy of

```
Object = {}; Object.__index = Object -- Define a node for status element
function Object.new(e,v) -- Costructor for status each element with value, flag for event
  return setmetatable({event=e or false, value=v or false},Object)
end
State = {}; State.__index=State -- Define of status as set of elements
function State.new(l) -- Create a new State form a table of objects
  l.final = false
return setmetatable(l,State) end
function State:set(e,v) -- Operator for change the value of a element
  self[e].value=v; self[e].event=true end
function State:reset(e) -- Reset the event flag
  self[e].event=false end
 function State:clone() -- Return a clone from a State
  local s={final=self.final}
  for i,o in pairs(self) do
    if i ~= 'final' then
    s[i]=Object.new()
    for k,v in pairs(o) do s[i][k]=v end
    end
  end
  return setmetatable(s,State)
end
State.__eq=function(o1,o2) -- Define the equal operator for compare two State
  if getmetatable(o1)~=getmetatable(o2) then return false end
  if #o1 ~= #o2 then return false end
  for k1,v1 in pairs(o1) do
    if k1 ~= 'final' then
      if o2[k1].event~=v1.event or
         o2[k1].value~=v1.value then return false end
         end
  end
  return true
end
function State:print(n) -- Print the State with optional the position in track
  s = "[".. tostring(n) .."]␣" or ""
    for k,v in pairs(self) do
      if k == 'final' then s=s.."final="..tostring(v).."␣" else
      s=s..k.."="..tostring(v.value).."["..tostring(v.event).."]␣"
      end
    end
  print(s)
end
Track={} -- Define a singleton table for all states
function find(s) -- Look for a state in the Track
  for _,v in ipairs(Track) do if v==s then return true end end
  return false
end
function printTrack(f) -- Print with [nil] or [false] all states, [true] only final states
  for i,v in ipairs(Track) do
    if not f then if v.final then v:print(i) end
    else v:print(i) end
  end
end
Rule={}; Rule.__index=Rule -- Define a singleton table for all rules
function Rule.new(e,c,a) -- Costructor for ECA rules
  return setmetatable({events=e, condition=c, action=a},Rule)
end
function execInvariances(s) -- Predicate for evaluation all invariaces for a state
  for _,v in ipairs(Invariances) do if not v(s) then return true end end
  return false
end
function Rule:exec(s) -- Execution of a rule in a state
  if self.events(s) and self.condition(s) then
    local s1=s:clone(); self.action(s1)
    if execInvariances(s1) then error.invariance = true; return false end
    if find(s1) then error.loop = true; return false end
    return true, s1
  end
  end
function exec(s) -- Execution of all rules in a state, return the number of final states
  local all=true
  for _,v in ipairs(Rules) do
    local result, newState = v:exec(s)
    if result then
      table.insert(Track,newState); all=false; s.final=false; exec(newState) end
  end
  if all then s.final = true end
end
function go(f) -- Executor
  error = {loop=false,invariance=false}
  local program = loadfile(f)
  program(); exec(initState);
  local count = 0
  for _,v in ipairs(Track) do if v.final then count=count+1 end end
  if count > 1 then print("Ambiguity") end
  if error.loop then print("Loop"); end
  if error.invariance then print("Break␣invariance"); end
end
go('program.lua') -- Run
printTrack(true)
```

Listing 5. Simple interpreter in LUA.

current state is created. On this copy the actions of *Rule::exec(State)* are applied and on the new state resulting is applied the *Invariances::verify(State)*. If no break of invariances results then the new state created is added in *Track* by *Track::insert(State)*.

The function *Track::insert(State)* is called before the evaluation of predicate *Track::find(State)* for checking if the new state is just in *Track*. In the positive case there is a *loop* and the evaluation finishes with an error signal. In the action performed by a rule activated there is the switch off of events through the *Object::reset()*. The *Object::reset()* prevents other rules are activated incorrectly. However, the management of the non-determinism in the application of the rules that are activated for the same events is guaranteed by the recursive execution of *Rule::exec(State)* and in the creation of copies of the current state. If no rule is activated by *Rule::exec(State)* then the state is marked *final=true* and the evaluation finishes with success. The function *go(f)* processes the file with the definition of objects and the initial state, invariances and rules. The function *Track::printTrace(Boolean)* prints all states if *param* is *true* or only final state if *param* is *false*. In Listing 4 a simple example of program for the evaluator in Listing 5. For neatness the syntax of the program is itself in LUA language.

In the following, Listing 5 shows an implementation in LUA of the interpreter.

An example of execution can be found in Table 3. In the file *tree.lua* there are three tables: the table tree.lua which contains the declaration of sensors and actuators, the singleton table Invariances which contains the list of *invariances*, the singleton table Rules which contains the list of rules.

In the *tree.lua* example the initState contains the declaration of actuator *tree* at value *0*. In Invariances a single invariance constrains the value of *tree* at less to *4*. In Rules two rules together trigger when the changed value of *tree* is between -5 and *5*. The first rule increases *tree*, the latter decreases *tree*.

The execution of Listing 5 on the code in Listing 4 produces the output of Table 3. In the left side, the output and in the right side, the tree of states with 9 *inner states*, 6 states in loops, 1 state breaking the invariance and 1 ambiguous state. The function printTrack(true) writes the final states, in our case two final states detecting the presence of *ambiguity*.

4. Conclusions and future work

We presented IRON, a language for the easy management and programming of IoT objects. Together with the syntax and the formal semantics, a simple interpreter was presented. The interpreter, consistently with the semantics, captures the anomalies (loop, break of invariances, ambiguity) that may come from a wrong writing of the ECA rule in IRON. Then, although the interpreter correctly implements the language semantics for fault handling, the latter are only detected at runtime. This implies that the running program can signal anomalies or no action. To allow the user to know if a program has anomalies, it is possible to create a checker that verifies the presence of the three forms of anomalies (loop, break invariances, ambiguity). How to make this checker, is described in [11–13,17,18]. Therefore, the most appropriate implementation of IRON consists in the implementation of an IRON compiler for to verify the properties of absence of loops, break invariances and ambiguity. If there are no anomalies, then the compiler generate an intermediate code on which the interpreter is applied. In this way, if the error-free compilation phase is exceeded, it is possible, not only to have a lighter interpreter (simplification of lexical and syntactic analysis) but to implement distributed action policies on the [19] actuators or to implement energy saving mechanisms [20]. We also plan to equip IRON language with a graphical programming environment and improving its usability by taking advantage of end-user development techniques as in [21].

References

[1] C. Evans, L. Brodie, J.C. Augusto, Requirements engineering for intelligent environments, in: Proceedings of the 2014 International Conference on Intelligent Environments (IE), 2014, pp. 154–161.
[2] G. Russello, L. Mostarda, N. Dulay, ESCAPE: a component-based policy framework for sense and react applications, in: Proceedings of the Eleventh International Symposium on Component-Based Software Engineering, CBSE 2008, Karlsruhe, Germany, October 14–17, 2008, pp. 212–229.
[3] J. Cano, É. Rutten, G. Delaval, Y. Benazzouz, L. Gürgen, ECA rules for iot environment: a case study in safe design, in: Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, September 8–12, 2014, 2014, pp. 116–121.
[4] W.-S. Lee, S.-Y. Lee, K.-C. Lee, Conflict detection and resolution method in ws-eca framework, in: Proceedings of the Ninth International Conference on Advanced Communication Technology, 1, IEEE, 2007, pp. 786–791, doi:10.1109/ICACT.2007.358468.
[5] M. Berndtsson, J. Mellin, ECA Rules, Springer US, Boston, MA, pp. 959–960. doi:10.1007/978-0-387-39940-9_504.
[6] J.P. Yoon, Techniques for data and rule validation in knowledge based systems, in: Proceedings of the Fourth Annual Conference on Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', 1989, pp. 62–70.
[7] J. Zhang, J. Moyne, D. Tilbury, Verification of ECA rule based management and control systems, in: Proceedings of the 2008 IEEE International Conference on Automation Science and Engineering, 2008, pp. 1–7.
[8] A. Aiken, J. Widom, J.M. Hellerstein, Behavior of database production rules: termination, confluence, and observable determinism, 21, ACM, 1992.
[9] S. Comai, L. Tanca, Termination and confluence by rule prioritization, IEEE Trans. Knowl. Data Eng. 15 (2) (2003) 257–270.
[10] J. Cano, G. Delaval, E. Rutten, Coordination models and languages, in: Proceedings of the Sixteenth IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the Ninth International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3–5, 2014, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 33–48.
[11] F. Corradini, R. Culmone, L. Mostarda, L. Tesei, F. Raimondi, A constrained ECA language supporting formal verification of WSNS, in: Proceedings of the 2015 IEEE Twenty-ninth International Conference on Advanced Information Networking and Applications Workshops (WAINA), 2015, pp. 187–192.
[12] 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: Proceedings of the 2017 International Conference on Intelligent Environments, IE 2017, Seoul, Korea (South), August 21–25, 2017, 2017a, pp. 92–99.
[13] C. Vannucchi, M. Diamanti, G. Mazzante, D. Cacciagrano, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, Symbolic verification of event–condition–action rules in intelligent environments, J. Reliab. Intell. Environ. 3 (2) (2017b) 117–130.
[14] R. Ierusalimschy, L.H.D. Figueiredo, W. Celes, Lua 5.1 Reference Manual, Lua.Org, 2006.
[15] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60–61 (2004) 17–139.

[16] M.O. Rabin, D. Scott, Finite automata and their decision problems, IBM J. Res. Dev. 3 (2) (1959) 114–125.
[17] D.R. Cacciagrano, F. Corradini, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, C. Vannucchi, Analysis and verification of ECA rules in intelligent environments, JAISE 10 (3) (2018) 261–273, doi:10.3233/AIS-180487.
[18] C. Vannucchi, D.R. Cacciagrano, R. Culmone, L. Mostarda, Towards a uniform ontology-driven approach for modeling, checking and executing wsans, in: Proceedings of the AINA Workshops IEEE Computer Society, 2016, pp. 319–324.
[19] N. Dulay, M. Micheletti, L. Mostarda, A. Piermarteri, Pico-mp: de-centralised macro-programming for wireless sensor and actuator networks, in: Proceedings of the Thirty-second IEEE International Conference on Advanced Information Networking and Applications, 2019, pp. 1–100.
[20] M. Micheletti, L. Mostarda, A. Piermarteri, Rotating energy efficient clustering for heterogeneous devices (reechd), in: Proceedings of the Thirty-second IEEE International Conference on Advanced Information Networking and Applications, 2018, pp. 1–100.
[21] F. Buti, D. Cacciagrano, M.C.D. Donato, F. Corradini, E. Merelli, L. Tesei, Bioshape: end-user development for simulating biological systems, in: Proceedings of the Third International Symposium on End-User Development, IS-EUD 2011, Torre Canne (BR), Italy, June 7–10, 2011, 2011, pp. 379–382, doi:10.1007/978-3-642-21530-8_45.