vIRONy: a tool for analysis and verification of ECA rules in Intelligent Environments

Claudia Vannucchi*, Michelangelo Diamanti*, Gianmarco Mazzante*, Diletta Romana Cacciagrano*, Flavio Corradini*, Rosario Culmone*, Nikos Gorogiannis†, Leonardo Mostarda* and Franco Raimondi†
*Department of Computer Science, University of Camerino, Italy
Email: claudia.vannucchi@unicam.it, michelangel.diamanti@studenti.unicam.it, gianmarco.mazzante@studenti.unicam.it, diletta.cacciagrano@unicam.it, flavio.corradini@unicam.it, rosario.culmone@unicam.it, leonardo.mostarda@unicam.it
†Department of Computer Science, Middlesex University, London, UK
Email: n.gkorogiannis@mdx.ac.uk, f.raimondi@mdx.ac.uk

***Abstract*—Intelligent Environments (IE) are a very active area of research and a number of applications are currently being deployed in domains ranging from smart home to e-health and autonomous vehicles. In a number of cases, IE operate together with (or to support) humans, and it is therefore fundamental that IE are thoroughly verified. In this paper we present how a set of techniques and tools developed for the verification of software code can be employed in the verification of IE described by means of event-condition-action rules. In particular, we reduce the problem of verifying key properties of these rules to satisfiability and termination problems that can be addressed using state-of-the-art SMT solvers and program analysers. We introduce a tool called vIRONy that implements these techniques and we validate our approach against a number of case studies from the literature.**

I. Introduction

Intelligent Environments (IE) are growing fast as a multi-disciplinary field, allowing many areas of research to have a real beneficial influence in our society. IE encompass an heterogeneous range of scenarios and applications that include smart homes, e-healthcare, e-learning, smart factories, autonomous vehicles, etc. [1]. IE systems are specific examples of *reactive systems* [2], which react to any stimulus (or event) that occurs in the environment, maintaining a continuous interaction with it. A common approach to program such systems is via *rules* [3]. In general, these rules take the form of Event-Condition-Action (ECA) rules [4]: an *action* is executed if a certain *event* happens and a specific *condition* is met. Due to the vast range of applications, IE have a great impact in human day life. Therefore, it is fundamental to meet important requirements for these systems such as correctness, reliability, safety, security, desired reliable behaviour [5]. However, guaranteeing these properties for rule-based IE is a non-trivial task [6]. In fact, programming rule-based systems is a difficult and error-prone process (not only in IE) because the interactions of rule actions can cause the system behaviour to be unpredictable or unsafe. To face these challenges, the verification of consistency and safety properties of IE systems has became a necessity [7].

In this paper we present an approach for the verification of ECA rules in IE based on techniques for software verification and we introduce the tool vIRONy[1] that implements these techniques. vIRONy is based on the combination of formal methods and simulation techniques, with the aim of supporting programmers and end-users during the modelling and verification phases of IE systems based on ECA-rules. The features of vIRONy include:

- a *syntactic analyser* for checking the correctness of the source program and for enabling users to identify and avoid syntactic errors;
- a *formal verification component* based on different techniques (SMT solvers and program analysers) in order to check safety and correctness of the program expressed as a set of ECA rules;
- a *simulation environment* to generate and investigate specific behaviours of the system;
- a *semantic analyser* used to perform qualitative and quantitative analysis of the system in terms of number of rules invoked, energy efficiency, etc.

The rest of the paper is organised as follows: in Section II we provide a detailed review of the literature; in Section III we present the language IRON for ECA rules and its semantics; in Section IV we formally define the properties we verify and we present the verification algorithms; in Section V we present the main features of the tool vIRONy. We report experimental results obtained from several case studies in Section VI and we conclude in Section VIII.

II. Related Work

Due to the complexity of IE systems, it is necessary to apply appropriate techniques and methods that allow establishing correctness and safety properties [8]. Software testing [9], by far the most used technique to improve software quality, consists in executing only a sampling of all possible runs of the system to be checked. This means that testing is not an adequate technique for exhaustively checking the correctness of complex systems as IE, since the number of possible runs is intractable, in general, and the environment in which the system run is not predictable. Thus, the application of

[1]The current version of vIRONy is open source and it is available at https://gitlab.com/MichelangeloDiamanti/ecaProject.

formal methods is fundamental in analysiing and establishing the correctness of such systems, especially in applications in which safety is a critical issue, where a small error in their design could put human lives at risk [8]. To face these challenges, various approaches have been proposed in order to verify and analyse ECA-rule based systems. In this section we describe the ones that are closer to our work.

Authors of [10] propose an approach to analyse the dynamic behaviour of a set of ECA rules by first translating them into an extended Petri Net [11], then studying two fundamental correctness properties: termination and confluence. Authors in [12] investigate the possibility of using a pure Binary Decision Diagram ([13]) representation of integer values, and they focus on a particular class of programs, i.e., ECA rule-based programs with restricted numerical operations. In [14] a tool-supported method for verifying and controlling the correct interactions of ECA rules is presented. This method is based on formal models related to reactive systems to generate correct rule controllers. A formalisation of an ECA rule-based system is provided in order to perform the translation into a Heptagon/BZR [15] program. The work offers users with a combination of a high-level ECA rule language with the compiler and formal tool support for Heptagon/BZR. Then, coordination and control techniques are applied at run-time by using Heptagon/BZR in order to complete the verification. An approach based on formal methods is applied for the verification of ECA rule systems in [16]. In particular, a set of ECA rules is transformed into different kinds of automata and then the automata verification tool Uppaal [17] is applied. The approach is limited to performing model checking of timed automata and their correspondence to the provided ECA rule set.

The differences between each of these approaches and our solution are discussed in Section VII, once our work has been presented in detail.

III. Preliminaries and Notation

In this section we introduce IRON, a language for ECA rules, and a formal model for the execution of ECA rules in IE.

A. IRON

An Intelligent Environment is a physical or logical space that contains a potentially very large number of devices, such as sensors and actuators, that work together to provide users access to information and services. ECA-based languages have been proposed by a number of sources in order to program IE, because programmers and users in particular are quite at ease with them [3]. In order to allow programmers and end-users to develop ECA-rule based systems in a easy and accessible way, a clear and efficient domain-specific language is needed. Here we employ Integrated Rule ON data (IRON) as the underlining formalism for modelling IE. IRON is presented in [18]: it is a restricted first-order logic language that supports the categorisation of devices into sets [19], allows the definition of properties over sets and supports multicast and broadcast

```
1  Program ≡ ( Device | Rule | VarDecl )+;
2  Device ≡ PhysicalDevice | LogicalDevice | Set;
3  PhysicalDevice ≡ physical (sensor|actuator)
4  Type Id [= Exp]
5  node(Id Sep Id) [in id (Sep Id )*] [where BoolExp];
6  LogicalDevice ≡ logical (sensor|actuator)
7  Type Id = Exp[in Id (Sep Id )*] [where BoolExp];
8  Set ≡ set (sensor | actuator) Type Id;
9  Rule ≡ rule Id on Id (Sep Id)*when BoolExp then Action;
10 Action ≡ [Id = Exp ]+;
11 Exp ≡ BoolExp | IntExp;
```

Figure 1. The IRON extended BNF

abstractions. IRON programs are composed of two separate classes of specifications: static and dynamic. We report the main constructs of the IRON syntax in Figure 1 (for further details see [20]).

B. A formal model for ECA rules

In [21] we presented a model for ECA rule-based systems that exploits the features that are typical of IE. The model is based on IRON and it allows for a precise definition of formal requirements and for their efficient verification. For the sake of simplicity but without loss of generality, the formal model does not include the definition of sets and the distinction between logical and physical devices included in IRON (these could be introduced at the cost of additional notation but do not affect the overall partitioning strategy described below). The model exploits the features that are typical of IE, taking into account the fact that a generic action defined by the user can only change actuator configurations.

We consider an ECA-rule based system consisting of: (i) a set D of variables representing the input/output devices of the system, denoted with i and o respectively (to refer to a generic element of D we use the letter d); (ii) a set Inv of static constraints inv of the system identifying the admissible values for each device (each invariant is a restricted first-order logic predicate as defined by the IRON grammar); (iii) a set R of ECA rules of the form $Event[Condition]/Action$.

A *state* of the system is an assignment of values to the devices in D and the *universe* is the set of states. In detail:

Definition 1. *A* state *of the system is a function* $\varphi : D \rightarrow Val$ *where Val is a finite set of integer or boolean values.*

Definition 2. *The* universe Φ *of an ECA-rule based system is the set of all possible states of the system. In other words, it is the set of all possible functions* φ *defined in Def. 1.*

By adding constraints to the system, i.e. conditions that must be satisfied, we can define the *admissible state space*:

Definition 3. *Let* Φ *be the universe. The* admissible state space Φ_a *is the subset of* Φ *whose elements are all the states* φ *that satisfy the constraints of the system.*

Given a set D and a state space Φ, we consider a finite set R of labels for ECA rules $R = \{r_1, r_2, ..., r_k\}$ for $k \in \mathbb{N}_0$. A generic rule r in R is represented as $e_r[c_r]/a_r$, where e_r, c_r, a_r are labels for the event, the condition and the action of r respectively.

We refer to [21] for details about the evolution of the system. We observe that devices can change their values according to *external changes* (sensors) or *internal changes* (actuators change their values in response to ECA rules being triggered). As a consequence, the evolution of the system can be partitioned into two sets: the set of *artificial transitions* resulting from the activation of ECA rules, and the set of *natural transitions* that result from changes in the environment. According to this partitioning, we can distinguish between stable and unstable states: the system is in a *stable state* if only natural transitions can be applied, while *unstable states* are those states to which only artificial transitions can be applied. In addition, in order to avoid the exploration of states and transitions that are not relevant to the analysis of the properties we are interested in, we do not model the spontaneous evolution of the environment. Instead, we only give a representation of the natural evolution in terms of "the minimum natural transition that links a stable state to an unstable state". The representation of the evolution of the system is based on two important hypotheses: (1) the initial admissible configuration of the system is given by an external entity; (2) artificial transitions take much shorter time than natural transitions. Finally, since we consider a finite set of devices that can assume a finite set of values, it is possible to represent the result of the evolution as a Finite State Automaton (FSA) [22]. Its graphical representation is a directed graph whose edges correspond to the transitions of the system and the vertices are the states.

IV. Properties and verification algorithms

The application of formal verification techniques to ECA rule-based programs is essential to support the error-prone activity of defining ECA rules. Our aim is to avoid "unsafe" and "incorrect" situations deriving from erroneous definitions of ECA rules that may result in inefficient or potentially dangerous effects on the real world. Based on literature review of related and previous works (see for instance [18], [14]), we identify those properties that can be considered representative for "safety" and "correctness" of ECA rule-based systems. These properties are: *termination, consistency* and *determinism*. We formally define each of them and we present the verification algorithms below.

A. Properties

ECA rule-based systems for IE can respond to external and internal events with the application of one or more rules. While external events are generated by environmental changes (sensors), internal events are generated by the application of actions (e.g., changing the value of actuators). An erroneous specification of the system can lead to *infinite loops* where rules can be applied an infinite number of times.

1) Termination: We define the *termination* property as follows:

Definition 4. *An ECA rule-based system satisfies the* termination *property when all stable states (that satisfy the conditions of some rules) always lead (with the application of a finite number of rules) to a new stable state.*

Intuitively, as the number of rules is finite, we deduce that termination property is satisfied if there are no cycles.

2) Consistency: We recall the definition of *consistency* (see [20] for further details):

Definition 5. *An ECA rule-based system satisfies the* consistency *property if its rules are neither unusable nor incorrect nor redundant.*

The notions of *unused*, *incorrect* and *redundant* rule are defined as follows.

Definition 6. *An ECA rule $r \in R$ is called* unused *if the condition c is false for every state $\varphi \in \Phi_a$.*

Definition 7. Incorrect *rules are those rules that can lead to a state that is outside of Φ_a.*

Definition 8. *Given $r_i, r_j \in R$ such that $r_i : e_i[c_i]/a_i$, $r_j : e_j[c_j]/a_j$ we say that r_i is redundant with respect to r_j if the following conditions are met:*

1) *$e_i \subseteq e_j$;*
2) *$c_i \wedge Inv \Rightarrow c_j$ is satisfiable (where, by slight abuse of notation, we denote with Inv the conjunction of all invariants); and,*
3) *for every state φ satisfying $c_i \wedge Inv$, applying a_i to φ is equivalent to applying a_j to φ, which we write as $\varphi[a_i] = \varphi[a_j]$.*

3) Determinism: Due to the unpredictability of external changes (natural events) and the interactions between rules, it is difficult to guarantee determinism of the system. We can express this property for ECA-rule based systems as follows: a pair of rules r_i, r_j is non-deterministic if there is at least one admissible state in which both rules are triggered and the effects of their actions on the system are not the same. More formally, the definition can be expressed as follows.

Definition 9. *Given $r_i, r_j \in R$ such that $r_i : e_i[c_i]/a_i$, $r_j : e_j[c_j]/a_j$, we say that the system is non-deterministic if the following conditions are met:*

1) *$e_i \cap e_j \neq \emptyset$;*
2) *$c_i \wedge Inv \wedge c_j$ is satisfiable;*
3) *$\exists \varphi$ that satisfies $c_i \wedge Inv \wedge c_j$ and $\varphi[a_i] \neq \varphi[a_j]$.*

Condition 1. means e_i and e_j have at least a common label. Thus this condition, if met, guarantees that at least the occurrence of an event triggers both r_i and r_j. If both 1. and 2. are verified, than r_i and r_j are applicable to the same state. If the result of applying action a_i is different from the result of applying a_j for at least one state φ in $c_i \wedge Inv \wedge c_j$, then r_i and r_j make the system non-deterministic.

B. Verification algorithms

In this section we present the methods, techniques and verification algorithms that are used for verifying the properties previously defined.

1) Termination verification: In order to prove termination we make use of T2[2] (see [23], [24], [25]) a tool designed to prove temporal properties of programs, such as safety and termination. The tool implements the TERMINATOR-based approach to termination proving (see [26]) with some modifications as described in [23] and [25]. The idea of the technique is to reduce the checking of termination arguments to an incrementally evolving safety problem. T2 represents programs as graphs of program locations connected by transition rules with conditions (expressed by the command "`assume`") and assignments to a set of integer variables V. The canonical initial location is called `START`.

Given a generic rule-based program written in IRON syntax that has been defined in Fig. 1, we show here how it can be translated into T2 format. The translation algorithm can be found in Fig. 2. Consider a generic program in IRON

```
1  let D = I ∪ O the set of declared labels
2  let R := {r_1,....,r_k} the set of rules such that each r ∈ R is of
       the form r : e_r[c_r]a_r where e_r = {d_w1, ··· ,d_wf} ⊂ D and
     a_r = {o_α1 ← v_α1, ··· o_αp ← v_αp}, with o_α1, ... ,o_αp ∈ O
3  let Inv_set := {inv_1, ....,inv_v}
4  let o = val the initial value for o ∈ O
5  define Inv = ⋀_{j=1}^{v} inv_j
6  for each d ∈ D:
7    define d_changed
8  for each r ∈ R
9    for each o_αj ∈ a_r
10     define o_prime
11   for each d ∈ e_r
12     let d_check = (d_changed!= 0)
13 define eT2_r = ⋁_{d∈e_r} d_check
14 write
15 START: 0;
16 FROM: 0;
17 assume(Inv); //invariants
18 for each i ∈ I // sensor initial values
19   write i := nondet();
20 for each o ∈ O // actuator initial values
21   write o := val;
22 for each o_changed //actuators changes
23   write o_changed := 0;
24 for each i_changed //sensors changes
25   write i_changed := nondet();
26 TO : 1;
27 for each r ∈ R
28   write //rule r
29   FROM: 1;
30   assume((Inv) && (eT2_r) && (c_r));
31   for each o_αj ∈ a_r
32     write
33     o_prime := v_αj;
34     o_changed := (o-o_prime);
35     o := o_prime;
36     for each o ∈ O such that o ≠ o_αj ∈ a_r ∀j
37       write o_changed := 0;
38     for each i ∈ I
39       write i_changed := 0;
40   TO : 1;
41 end
```

Figure 2. Algorithm for translating input files from IRON to T2 format.

consisting of a set of devices D, a set of rules R and a set of invariants *Inv_set* (see lines $1-3$ in Fig. 2). We consider the set of initial states characterised by an assigned configuration

[2]available at https://github.com/mmjb/T2.

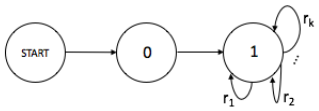

Figure 3. Execution model of IRON programs in T2.

of actuator values (line 4). We define a set of variables in lines $5-13$ that are used for the translation. Lines $14-40$ describe the algorithm for generating the program in T2 format. This program corresponds to the automaton represented in Fig. 3 that is characterised by two states that we generically name 0 and 1. The execution starts from state 1, characterised by: the assigned configuration of values, a generic configuration of sensors (the function $nondet()$ assigns values randomly). In addition, state 0 is admissible, as the invariants are assumed as valid (line 17). A natural transition (line 26) corresponds to a transition from state 0 to state 1. When a natural transition moves the state from 0 to 1, the rules in R, i.e., the artificial transitions of the system, as shown in Fig. 3, can be applied when their corresponding event is met (line 30).

Proposition 1. *The algorithm in Fig. 2 is correct.*

Proof. First, we show that if a generic ECA rule can be activated in an IRON program (P_{IRON}), then it can be activated in the corresponding T2 program (P_{T2}) too, and vice-versa. In fact, given a generic P_{IRON} and a generic ECA rule $r : e_r[c_r]a_r$ such that $r \in P_{IRON}$, the rule is activated if and only if: (i) the invariants are valid; (ii) the event e_r is triggered (i.e., if a change concerning the value of at least one of the labels in e_r occurs); (iii) the condition c_r is valid. As shown in Fig. 2, these three conditions are all reported verbatim in the `assume` at line 30. Now we show that there is an equivalence in the execution semantics of P_{IRON} and P_{T2} programs. If a natural change occurs, then ECA rules whose events capture this natural evolution are considered for an eventual activation. The natural transitions correspond in P_{T2} to the transition `FROM 0;[..] TO 1;` (line $16-26$), and the natural changes correspond to the assignments to sensor variables through the `nondet()` function (line 19). When a natural evolution occurs, and the program P_{T2} is in state 1 then, if conditions (i)-(iii) are met for a certain ECA rule, this rule is executed. Furthermore, the activation of the ECA rules in P_{IRON} is non-deterministic, and this non-determinism is maintained in P_{T2}, since all rules are applied to the state 1 (`FROM 1` at line 29). When an activation of an ECA rule in P_{IRON} is performed, the system is frozen, in the sense that natural transitions are not taken into account: this condition in P_{T2} is fixed at lines $38-39$. The T2 program does not allow any other kind of transition between states, and therefore the executions of the IRON and the T2 program are isomorphic. □

2) Consistency verification: The algorithms for verifying the consistency property are described in detail in [20]. The key insight of the verification approach is observing that it is

possible to perform consistency verification of ECA rule-based systems by using Satisfiability Modulo Theories (SMT) [27], [28] and predicate transformer techniques ([29], [30]). We briefly report in this paper the verification algorithms (see [20] for further details). An *SMT solver* is any software that implements a procedure for satisfiability *modulo* some given theory, for example the theory of linear arithmetic. Typically, SMT solvers support several fragments of First Order Logic (FOL). The solution of an SMT problem is an interpretation for the variables, functions and predicate symbols that make the formula true [28]. We use Z3 [31], a high-performance SMT Solver implemented in C++ and developed by Microsoft Research.

3) Determinism verification: The verification procedure of the determinism property is very similar to that one of redundancy, and it is performed under the same hypothesis. The algorithm is described in Fig. 4. All the pairs of distinct rules are considered. According to Definition 9, three conditions must be verified: if two rules are triggered together, i.e., the event parts have a least a common label, both conditions are met, and the actions have different effects on the system, then the system is non-deterministic. The procedure described below must be performed for all pairs of rules (line 4).

1 **let** $R := \{r_1, ..., r_k\}$
2 **let** $I := \{inv_1, ..., inv_v\}$
3 **define** $Inv = \bigwedge_{j=1}^{v} inv_j$
4 **for each** pair $r_i, r_j \in R^2$ such that $r_i \neq r_j$ **and** such that r_i, r_j are *usable*:
5 **if** $(e_i \cap e_j)$ is non-empty **and** $((c_i \wedge Inv \wedge c_j)$ is satisfiable)
6 **and** $\neg(c_i \wedge Inv \Rightarrow \Psi(a_i, a_j))$ is satisfiable):
7 **declare** the system *nondeterministic*
8 **end**

Figure 4. Algorithm for verifying determinism.

Proposition 2. *The algorithm in Fig. 4 is correct.*

Proof. The proof is similar to that one of the correctness of the algorithm for the verification of redundancy (see [20] for further details). Indeed, requirements (1) and (2) of Def. 9 are checked verbatim in the algorithm. Thus, it remains to show that the algorithm is correct w.r.t. requirement (3). It can be rewritten as follows: $\exists \varphi \not\models c_i \wedge Inv \wedge c_j, \varphi[a_i] = \varphi[a_j]$. Notice that requirement (3) of Def. 9 is equivalent to the negation of requirement (3) of Def. 8. □

V. The tool vIRONy

In this chapter we present vIRONy, the prototype tool that has been implemented to evaluate the proposed approach.

A. Graphical User Interface

Figure 5 depicts the graphical interface of the tool: users can select input files written in IRON syntax and select the desired functionality. In order to provide programmers and end-users with an easy to use tool for modelling and analysing ECA-rule based programs written in IRON syntax, the tool resembles the Eclipse IDE. The user interface and the components are

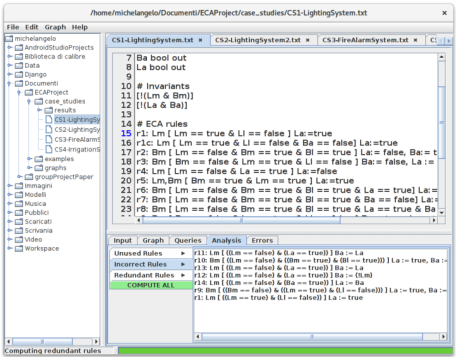

Figure 5. Graphical User Interface of vIRONy

implemented in Java Swing. When the user opens a file for the first time, a *TabbedPane* containing JavaSwing objects organised in sheets is opened. The main components of the *TabbedPane* are:

- *Development* area: it is a *TextArea* the user can use to write or modify the program.
- *Functionality* panel: the user can access the desired functions in order to analyse or generate a simulation of the program.

In detail, by accessing the *Functionality* panel the user can insert the initial configuration of values (*Input* tab), visualise the resulting graph of the simulation generated by using the *GraphStream* library (*Graph* tab), make queries on the resulting simulation (*Queries* tab), perform a formal analysis of the program (*Analysis* tab), check if the output console gives errors (*Errors* tab).

B. Syntactic analysis

The parser of vIRONy is implemented using Java Compiler Compiler[3]. The vIRONy parser only accepts input files written in IRON. As we showed in Section III, the IRON language consists of two main components, the static part which includes labels and invariants, and the dynamic part that is made up of ECA rules. Labels are used to uniquely identify the devices of the system. Each label declaration must contain the name of the device, the value type (*int* or *bool*) and the device category (*in* or *out*). The declaration of the invariants consists in a boolean expression enclosed by square brackets. The syntax of ECA rules is $rule : event[condition]action$, where *rule* is a string that uniquely identifies a rule, the *event* is a non-empty set of labels (guards) separated by commas, the *condition* is a boolean expression, the *action* is a set of assignments to actuators. The user can write the input file using the *TextArea* or visualise an existing one, and before performing any kind of operation (e.g. simulation, formal analysis, etc.) the parser is automatically called in order to check the syntactic correctness of the program. If there are errors, the chosen operation is cancelled and it is possible to visualise the errors found through the *Errors* tab that is automatically opened by the tool.

[3]See https://java.net/projects/javacc for further information.

C. Formal verification

The verification procedure is based on the formal techniques described above.

1) Termination verification: In order to use T2 for analysing termination, vIRONy provides the users with a procedure for translating the input file written in IRON syntax into the T2 format. The file is then stored in the computer, and available as input file for T2. T2 is run from the command line, using the `termination:` command line argument (to prove (non)termination). The user can select the initial configuration of values for the actuators, and then by submitting them, verification of termination is performed.

2) Consistency and determinism verification: As introduced in Section IV, the verification of consistency and determinism properties implemented in vIRONy makes use of the SMT solver Z3. From an implementation perspective, in order to use Z3, a recursive algorithm has been implemented to translate the expressions generated by the parser into expressions semantically equivalent to the initial ones that can be verified by using the SMT solver.

D. Simulation

Once an input file has been defined, it can be useful for the programmer to predict how the system would react to external stimuli, in order to observe the phases of the evolution of the system and for a deeper understanding of the reasons that cause undesired behaviours. However, it is very difficult to be predict the execution of the system, since rules interact in a complex way and the external environment changes in an unpredictable manner. For these reasons, we provide users and programmers with a function to simulate a possible behaviour of the system given a particular configuration of actuators.

The simulation generated by vIRONy is based on the formal model introduced in Section III and explored in detail in [21]. Before explaining the simulation procedure, we make a preliminary observation: for the sake of simulation and differently from the verification step using Z3 and T2, devices can assume only a finite set of values (we choose for integers the range $[-128, 127]$, but this arbitrary choice can be easily changed). From the user perspective, the *Simulation* menu allows to start a computation. After having pressed the button *Start simulation* the *Input* tab at the bottom is configured to allow the user to set the initial configuration of values. Then Z3 checks if it is admissible for the system: if the configuration is admissible, the simulator starts the procedure, otherwise an alert is generated for the user. If the user only gives the configuration of the actuators, the system automatically searches for the configuration of sensors such that the complete state is admissible (by default the system searches for the "minimum" sensor configuration according to the lexicographic order of the elements of the table containing only sensor labels). Once a configuration is found, the simulator starts the generation procedure of the transitions. Once the computation finishes, the tool automatically visualises the resulting graph in the corresponding tab. It is possible to export the graph in the GraphML format[4] or as an image by selecting the desired option from the *Graph* menu.

E. Analysis of the simulation

After the simulation is generated, end-users can analyse the resulting graph. We provide different strategies for a deeper understanding of the rules from a quantitative perspective that is strictly linked to energy saving problems. Different algorithms have been implemented and they have been grouped together under the *Queries* tab. We highlight the fact that the analysis is based only on the graph resulting from the simulation, so the results depend on the initial configuration chosen by the user. We mention here a subset of the algorithms available:

- *Most used rules.* This query is used to find out the rules that are used the maximum number of times during the simulation.
- *Initial rules.* This query asks the simulation for those rules that are triggered by a natural event.
- *Actuator updates.* This measure counts how many times the values of the actuators are modified by ECA rules.
- *Find cycles.* The query extracts the cycles from the graph and reports the graphical representation of each cycle.
- *Find paths.* This query explores the graph and finds out all possible paths reaching a certain state of the system starting from another one.

VI. Evaluation

In this section we consider four case studies taken from related and previous work and we present performance results. We report only results and we refer to the files available on-line for further details. All the experiments have been performed on an Intel Core i7-4700MQ CPU @ 3.4GHz with 8GB of RAM running Debian Linux.

The first case study (CS1) is a lighting control system in a simple scenario and it has been adapted from the example presented in [20]. The second case study (CS2) has been adapted from [10], where a light control subsystem in a smart home for senior housing is considered (see [20] for further details). The third case study (CS3) has been developed starting from the example presented in [32] and presented in [20]: a fire alarm system composed of temperature sensors, smoke detectors and sprinkler actuators is described by means of ECA rules. The fourth case study (CS4) consists of a Wireless Sensor and Actuator Network (WSAN) composed of five devices for an irrigation management system and controlled use of fertilizers.

In Table I we report some information about the case studies and the corresponding results of the formal analysis performed using the vIRONy tool. In detail, we report the cardinalities of the universe $|\Phi|$ and of the admissible state space $|\Phi_a|$. We also give the number of analysed ECA rules N_{tot}, the number N_{dev} of devices, the number of unusable, incorrect and redundant rules (denoted with N_{un}, N_{inc}, N_{red}) detected by the tool.

[4] http://graphml.graphdrawing.org

Table I
FORMAL ANALYSIS: SYNTHESIS OF THE RESULTS.

	CS1	CS2	CS3	CS4
$\|\Phi\|$	2^8	2^{30}	2^{16}	2^{19}
$\|\Phi_a\|$	$9 \cdot 2^4$	$1331 \cdot 2^{14}$	$2^8 \cdot 59$	$225 \cdot 2^8$
N_{tot}	20	17	14	17
N_{dev}	8	9	9	5
Unusable Rules	$r5$	$r14$	$r7, r8, r11, r12$	$r5, r8, r9$
N_{un}	1	1	4	3
T_{un} (ms)	135	141	105	141
Incorrect Rules	$r11, r10$ $r13, r12$ $r14, r9, r1$	$r2$	no	$r3, r4, r6, r7,$ $r7c, r1, r2$
N_{inc}	7	1	0	7
T_{inc} (ms)	235	252	173	230
Redundant Rules	$r4, r6, r7,$ $r8, r19$	$r11$	$r9, r10$	none
N_{red}	5	1	2	0
T_{red} (ms)	715	696	419	495
Determinism	non-det	non-det	non-det	non-det
T_{det} (ms)	183	190	125	228
Termination	Term./ nonterm. proof failed	Term. proof succeeded	Term./ nonterm. proof failed	Term. proof succeeded*
T_{ter} (s)	88	9	7	457

Table II
SIMULATION: SYNTHESIS OF THE RESULTS.

	CS1	CS2	CS3	CS4
T_{sim}	2066	5673	359002	5030
Initial Values	Bl: F, Bm: F, Bs: F, Ll: F, Lm: F, Ls: F, Ba: F, La: F	lgtsTmr: 0, intLgts: -128, Lgts: F, ChkExtLgt: F, ChkMtn: F, ChkSlp: F, Mtn: F, Slp: F, ExtLgt: 0	temperature: -9, smoke: F, presenceLiving: F, sprinkler: F, heating: F, tv: F, light: F, tempAlarm: F, smokeAlarm: F	c: 0, f: F, w: F, r: F, t: 0

Table III
SIMULATION ANALYSIS: SYNTHESIS OF THE RESULTS.

	CS1	CS2	CS4
Initial Rules	r10, r7, r17, r2 : 1 time	r8 : 3 times	r1c, r6, r7, r7c, r1 : 3 times
Time (ms)	21,5	16,8	84,4
Actuators Updates	La, Ba : ∞ times	ChkExtLgt, Lgts, ChkMtn : 0 time, ChkSlp : 24 time, lgtsTmr : 1 time, intLgts : 25 times	c, w : 15 times, f : 3 times
Time (ms)	67,9	2,9	0,7
Find cycles	3	0	0
Time (ms)	69,2	27,9	0,1
Find Paths Time (ms)	328,1	n/a	184,0

Furthermore, we measure the performance of the verification procedure in terms of time. The indicators T_{un}, T_{inc} and T_{red}, T_{det} (expressed in microseconds) refer, respectively, to the duration time of the verification for unusability, incorrectness, redundancy, determinism and T_{ter} is the duration time of the verification of the termination property.

The results in Table I show that our approach allows for the verification of the consistency property of non-trivial examples that include both boolean and integer variables in approximately 1 second (see [20] for further details). All the examples presented here are non-deterministic. The verification of this property requires a more-or-less constant time that is independent of the size of the state space and is only partially affected by the number of rules. The analysis of verification of termination requires special care: as it can be seen from the examples, the tool T2 is able to prove termination in 50% of the cases. This is expected, as proving termination is an undecidable problem. However, T2 is a sound tool and therefore, if an answer is provided, then we know that the result can be trusted. Verification of termination is a slower process if compared to the other properties. In particular, the process can take up to 10 minutes for larger examples (CS4). Moreover, it should be remarked that T2 is non-deterministic: this means that the tool may select different strategies even if it is invoked on the same example. As a result, in some cases (such as for CS4) it may happen that the tool sometimes finds a proof, and sometimes it fails with an out-of-memory or timeout error.

The evaluation is also used to assess the performance of the simulation environment of vIRONy. Table II reports for each case study the performances of the simulator implemented in vIRONy. The performances are measured in terms of time T_{sim} (expressed in milliseconds), and the initial configuration of actuators is detailed for each case study (the value "F" represents "false"). In Table III some of the results obtained by the semantic analysis performed on the generated simulations (ref. to Table II) are reported. End-users can benefit from these results as they allow to analyse the system in terms of energy consumption and efficiency of the system, for a deeper understanding of the rules and the system from a quantitative perspective that is strictly linked to energy saving problems.

VII. DISCUSSION

Compared to our approach, the approaches in [10] and [12] do not provide methods for verifying application-specific properties like redundancy, consistency and usability of rules. The work presented in [16] is not tailored to a specific rule language and requires a specific model checking tool, while the verification methods proposed in our work are based on a DSL for IE, and allow for defining and implementing verification algorithms directly using efficient techniques such as SMT solvers and theorem provers that enable the verification of application-specific properties. With respect to our approach, in [14] properties such as correctness and usability are not considered for verification. Redundant rules are not directly detected by Heptagon/BZR [15]. Duplicated rules are compiled and executed at run-time and rule actions are activated using the `or` operator. Instead, our approach enables end-users to identify redundant rules and decide how to modify the program; therefore it allows a deeper analysis and understanding in the design phase, giving to the programmer a greater control on the system he or she intends to develop.

Furthermore, they do not provide an implementation of their approach or experimental evaluation.

VIII. Conclusions

This paper proposes methods and a tool for modelling and verifying properties of rule-based systems in IE. In particular, the basis for this approach is a domain-specific language (IRON) that can be employed both by developers and by end-users to program and configure an ECA rule-based system for IE. The expressivity of IRON enables the application of high performance methods for verifying certain requirements that are specific for ECA-rule based systems for IE. We have implemented these methods in the tool vIRONy, which we have released as open source, and we have validate our approach and our tool against four cases studies from the literature.

References

[1] J. C. Augusto and A. Coronato, "Introduction to the inaugural issue of the journal of reliable intelligent environments," *Journal of Reliable Intelligent Environments*, vol. 1, no. 1, pp. 1–10, 2015. [Online]. Available: http://dx.doi.org/10.1007/s40860-015-0005-3

[2] K. Schneider, *Verification of Reactive Systems: Formal Methods and Algorithms*. SpringerVerlag, 2004.

[3] V. Callaghan, G. Clarke, M. Colley, H. Hagras, J. S. Y. Chin, and F. Doctor, "Inhabited Intelligent Environments," *BT Technology Journal*, vol. 22, no. 3, pp. 233–247, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:BTTJ.0000047137.42670.4d

[4] M. Berndtsson and J. Mellin, *ECA Rules*. Boston, MA: Springer US, 2009, pp. 959–960. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_504

[5] F. Corno and M. Sanaullah, "Design time methodology for the formal verification of intelligent domotic environments," in *Ambient Intelligence-Software and Applications*. Springer, 2011, pp. 9–16.

[6] F. Kausar, E. Al Eisa, and I. Bakhsh, "Intelligent home monitoring using rssi in wireless sensor networks," *International Journal of Computer Networks & Communications*, vol. 4, no. 6, p. 33, 2012.

[7] D. Preuveneers and W. Joosen, "Change impact analysis for context-aware applications in intelligent environments," in *Intelligent Environments*, 2015.

[8] J. C. Augusto and M. J. Hornos, "Using simulation and verification to inform the development of intelligent environments." in *Intelligent Environments (Workshops)*, 2012, pp. 413–424.

[9] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[10] X. Jin, Y. Lembachar, and G. Ciardo, "Symbolic verification of ECA rules," in *Joint Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13) and the International Workshop on Modeling and Business Environments (ModBE'13), Milano, Italy, June 24 - 25, 2013*, 2013, pp. 41–59. [Online]. Available: http://ceur-ws.org/Vol-989/paper17.pdf

[11] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[12] D. Beyer and A. Stahlbauer, "Bdd-based software verification," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 5, pp. 507–518, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10009-014-0334-1

[13] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on computers*, vol. 100, no. 6, pp. 509–516, 1978.

[14] J. Cano, G. Delaval, and E. Rutten, *Coordination Models and Languages: 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Coordination of ECA Rules by Verification and Control, pp. 33–48. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-43376-8_3

[15] G. Delaval, É. Rutten, and H. Marchand, "Integrating discrete controller synthesis into a reactive programming language compiler," *Discrete Event Dynamic Systems*, vol. 23, no. 4, pp. 385–418, 2013.

[16] A. Ericsson, "Enabling tool support for formal analysis of eca rules," Ph.D. dissertation, University of Skövde, 2009.

[17] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal?a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*. Springer, 1996, pp. 232–243.

[18] F. Corradini, R. Culmone, L. Mostarda, L. Tesei, and F. Raimondi, "A constrained eca language supporting formal verification of wsns," in *Advanced Information Networking and Applications Workshops (WAINA), 2015 IEEE 29th International Conference on*, March 2015, pp. 187–192.

[19] L. Mostarda, S. Marinovic, and N. Dulay, "Distributed Orchestration of Pervasive Services," in *24th IEEE IAINA 2010, Perth, Australia, 20-13 April 2010*, 2010, pp. 166–173.

[20] 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," *Journal of Reliable Intelligent Environments*, pp. 1–14, 2017. [Online]. Available: http://dx.doi.org/10.1007/s40860-017-0036-z

[21] C. Vannucchi, D. R. Cacciagrano, F. Corradini, R. Culmone, L. Mostarda, F. Raimondi, and L. Tesei, "A Formal Model for Event-Condition-Action Rules in Intelligent Environments," in *Proceedings of the 11th International Conference on Intelligent Environments*, 2016, pp. 56–65.

[22] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM journal of research and development*, vol. 3, no. 2, pp. 114–125, 1959.

[23] M. Brockschmidt, B. Cook, and C. Fuhs, "Better termination proving through cooperation," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 413–429.

[24] B. Cook and E. Koskinen, "Reasoning about nondeterminism in programs," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 219–230, 2013.

[25] B. Cook, A. See, and F. Zuleger, "Ramsey vs. lexicographic termination proving," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 47–61.

[26] B. Cook, A. Podelski, and A. Rybalchenko, "Terminator: beyond safety," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 415–418.

[27] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krsti, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke, "C.: The smt-lib standard: Version 2.0," Tech. Rep., 2010.

[28] L. De Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Brazilian Symposium on Formal Methods*. Springer, 2009, pp. 23–36.

[29] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[30] D. Gries, *The Science of Programming*, ser. Monographs in Computer Science. Springer New York, 1989.

[31] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[32] C. Vannucchi, D. R. Cacciagrano, R. Culmone, and L. Mostarda, "Towards a Uniform Ontology-Driven Approach for Modeling, Checking and Executing WSANs," *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, vol. 00, no. undefined, pp. 319–324, 2016.