Symbolic Verification of Event-Condition-Action Rules in Intelligent Environments

Claudia Vannucchi · **Michelangelo Diamanti** · **Gianmarco Mazzante** · **Diletta Cacciagrano** · **Rosario Culmone** · **Nikos Gorogiannis** · **Leonardo Mostarda** · **Franco Raimondi**

Abstract In this paper we show how state-of-the art SMT-based techniques for software verification can be employed in the verification of Event-Condition-Action rules in Intelligent Environments. Moreover, we exploit the specific features of Intelligent Environments to optimise the verification process. We compare our approach with previous work in a detailed evaluation section, showing how it improves both performance and expressivity of the language for Event-Condition-Action rules.

Keywords Event-Condition-Action rules · Symbolic Verification

1 Introduction

The term *Intelligent Environments* (IE) is used to encompass an heterogeneous range of scenarios and applications that include smart homes, smart factories, autonomous vehicles, etc. A common feature of IE is their ability to implement complex applications on top of an existing network of sensors and actuators. Several programming models and

C. Vannucchi · M. Diamanti · G. Mazzante · D. Cacciagrano · R. Culmone · L. Mostarda
Department of Computer Science
University of Camerino, Italy
E-mail: claudia.vannucchi@unicam.it
E-mail: michelangel.diamanti@studenti.unicam.it
E-mail: gianmarco.mazzante@studenti.unicam.it
E-mail: diletta.cacciagrano@unicam.it
E-mail: rosario.culmone@unicam.it
E-mail: leonardo.mostarda@unicam.it

N. Gorogiannis · F. Raimondi
Department of Computer Science
Middlesex University, London, UK
E-mail: n.gkorogiannis@mdx.ac.uk
E-mail: f.raimondi@mdx.ac.uk

middleware infrastructures exist and have been investigated in the past (see, for instance, [22] and references therein).

Domain specific languages have been developed to support the task of *programming* IE. The user interfaces provided to end users to control, for instance, the temperature and the humidity in an environment according to time slots and days of the week, can be considered as examples of such domain-specific languages that are often called *context-aware* [15].

In our work we consider Event-Condition-Action (ECA) rules as a model of a generic domain-specific language that can be employed both by developers and by end users to program and configure an IE. Our aim is to prove properties of applications for IE programmed using ECA rules. Focusing on the specific domain of IE allows us to develop efficient techniques.

In this work we employ IRON (Integrated Rule ON data), a domain-specific language that can be used both by developers and by end-users to program and configure an ECA rule-based system for IE. The aim of this work is to provide techniques and tools to guarantee application-specific properties of IE, something that is useful in a typical scenario of IE where non-expert users may autonomously adjust the behaviour of the system by updating the rules. Focusing on a domain-specific language allows us to develop efficient techniques and our main contribution is showing that verification methods based on SMT solvers enable the effective verification of ECA-rule based systems specified in the IRON language. In detail, the contributions of the work are the following:

- A formal model for ECA rules in Intelligent Environments and a formal definition of the properties that ECA rules should satisfy. These properties are based on a detailed review of existing literature.

– Verification algorithms for these properties, that treat sets of states *symbolically*. We employ techniques that are used in software verification and adapt them to the domain of ECA rules in IE. In particular, we employ the notion of weakest precondition and we make use of SMT solvers to prove properties.
– We provide an experimental evaluation using examples from the literature and we compare our results with those obtained in the past by other authors on the same examples. We also provide a prototype implementation.

The rest of the paper is organised as follows: in Section 2 we provide a detailed review of the literature; in Section 3 we present preliminary notation and the formalism employed in the paper; in Section 4 we introduce the properties that any set of ECA rules should satisfy and we present verification algorithms in Section 5. We describe case studies and report experimental results of our prototype implementation in Section 6.

2 Related Literature

IE are a very active area of research and a number of applications are currently being deployed in domains ranging from smart homes to e-health to autonomous vehicles. In a number of cases IE operate together with humans or to support them. For instance several systems for IE are more and more frequently designed for critical domains like houses for frail and elderly inhabitants, hospitals, emergency scenarios, etc. Therefore, it is fundamental to meet important requirements for these systems such as correctness, safety, security, desired reliable behaviour, something that may also be imposed by certification authorities before these systems are used in specific settings like hospitals etc.

IE systems are specific examples of *reactive systems*, i.e., systems that react to any *stimulus* (or *event*) that occurs in the environment maintaining a continuous interaction with it, since a common feature is their ability to implement complex applications on top of an existing network of sensors and actuators. Typically, the approach that is used to program such systems is via *rules*, that often take the form of ECA rules [2]: an *action* is executed if a certain *event* happens and a specific *condition* is met. ECA rules are an effective way for representing rules for IE, a good match with users mental model and can be useful and usable for end-user programming in IE.

However, programming rule-based systems is a difficult and error-prone process. In particular, looking at human daily life, these applications are typically developed and deployed by experts and engineers and are then left in the hands of end-users that are allowed to modify the behaviour of some of the components by adding or removing *rules* according to their requirements, as shown in [20].

Therefore, due to the complexity of IE systems, it is necessary to apply appropriate techniques and methods that allow to meet the specific requirements. Various approaches have been proposed in order to apply formal methods to ECA rule verification. In this section we describe the proposals that are closer to our approach.

Authors in [16] propose an approach to analyse the dynamic behaviour of a set of ECA rules by first translating them into an extended Petri Net (PN), then studying two fundamental correctness properties: termination and confluence. They have designed a language for ECA rules with the following features:

– It distinguishes *environmental* variables, that are used to represent environment states that can only be measured by sensors, and *local* variables, that can be both read and written by the system.
– It provide operations to set (absolute change) the value of local variables to an expression, or increase or decrease (relative change) it by an expression; these expressions may depend on environmental variables.
– Events can be external or internal. An external event can be activated when the value of an environmental variable crosses a threshold, and at that time it may take a snapshot of some environmental variables and read them into local variables to record their current values. An internal event can be activated only by an action of an ECA rule. Since these two types of events cannot be mixed within a single ECA rule, rules can be external or internal.
– The execution semantics of actions can be specified as any partial order described by a graph that is obtained through appropriate operators.
– Priorities for internal rules with respect to external ones can be specified.

According to the model in [16], a state is called *stable* if only external events can occur in it, *unstable* if actions of external or internal rules are being performed. For what concerns the evolution model, the system is initially stable and, after some external events trigger one or more external rules, it transitions to unstable states where internal events may be activated, triggering further internal rules. When all actions complete, the system is again in a stable state, waiting for environmental changes that will eventually trigger external events. In the presented model, the system is frozen during rule execution and does not respond to external events. Thus, rule execution is instantaneous, while in reality it obviously requires some time. However, from a verification perspective, environmental changes and external event occurrences are nondeterministic and asynchronous, thus their semantic allows the verification process to explore all possible combinations without missing errors due to the order in which events occur and environmental variables change. The set of ECA rules is translated into a PN, and then the

approach computes the initial states symbolically. Thanks to the nondeterministic semantics of Petri nets, the analysis is performed once starting from a single, but very large, set of initial states. Verification of termination and confluence for this obtained model is performed through their tool Smart.

Authors in [3] investigate the possibility of using a pure Binary decision diagram (BDD) [5] representation of integer values, and they focus on a particular class of programs, i.e. ECA rule-based programs with restricted operations. The subclass of ECA programs considered includes programs consisting of a single loop in which many conditional branches occur. In each of these branches, the condition is a boolean combination of equalities and negated equalities between variables and values, and the action is a sequence of assignments to variables, and all variables are of type integer. This means that all required operations are in fact efficiently supported by BDDs, and a symbolic representation using BDDs seems indeed appropriate for this particular class of programs. In order to define a configurable verifier, the authors define an iteration algorithm and a configurable program analysis which defines the abstract domain, the transition relation, as well as the merge and stop operators. A program is represented by a control-flow automaton (CFA) and different verification approaches are applied in order to solve a reachability problem: Explicit Value, Bounded Loops with bounded model checking, Predicate Abstraction, Predicate Impact [4], BDD. There is a number of (state) variables (with an initial value) that model the state of the system. Considering the BDD-based representation, since BDDs do not scale well for the multiplication operation [5], they partition the set of programs into two partitions: the first partition Eca-EQ contains all programs in which no multiplication of integer variables occurs, and the second partition Eca-MUL contains all programs that do not have that restriction in terms of operations on integer variables. In other words, Eca-EQ contains all programs whose variables are only used in equality expressions ($\neq$ and $==$) and not with other arithmetic operators. The results emphasise that BDD analysis work very well for Eca-EQ, while in the context of ECA-MUL the combination "BDD+Impact predicate" is the most efficient.

In [6] 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, and Discrete Controller Synthesis (DCS) 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 program. According to the proposed model, a generic device (sensor or actuator) is composed of a set of input signals and a set of output signals. The condition of an ECA rule is a Boolean expression of sensor or actuator output signals. Specific keywords are defined in the grammar in order to define the adopted execution policy.

The priority is given by the order in which input and output signals are declared. As a first step, verification is performed at compilation time by applying simple transformation and model checking in order to guarantee the satisfaction of certain properties. For instance, redundancy is verified by using the Sigali tool [18], consistency is verified by using Sigali and Heptagon [10], while circularity is verified by using Heptagon. The presented method also takes into account different possible execution models of ECA rule-based system. The work offers users with a combination of 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.

In [23] a framework is proposed for the validation of data and rules in knowledge-based systems. In particular, authors define different type of rules to create a "rule net" consisting of chained rules, and they present a way to validate a set of rules and check inconsistencies between different paths according to predefined constraints.

Conflict detection and resolution methods for distributed ECA rule processing are proposed in [17] in order to prevent the various kinds of abnormal situations. An infrastructure is described to detect and solve static and dynamic conflicts for a framework based on ECA rules for Web Services. No implementation is described for this infrastructure.

An approach for the verification of an ECA rule-based management and control system is proposed in [24]. A formal representation of the system is described in order to define problems like inconsistency, redundancy and circularity. Three levels are described regarding the verification and validation of these problems: the level related to the set of rules, the level based on the direct results of the execution of actions on the environment, and the level of all the possible responses of the environment.

An approach based on formal methods is applied for the verification of ECA rule systems in [13]. In particular, a set of ECA rules is transformed into different kinds of automata and then the automata verification tool Uppaal is applied. The approach is limited to performing model checking of timed automata and their correspondence to the provided ECA rule set.

Our work allows the verification of some of the properties presented in [24], [13] and [6] by using a different approach based on SMT solver. With respect to the state of the art of methods applied to the verification of sensor/actuators applications, SMT solvers have never be applied. Even if traditional model checkers can be employed to verify some of the properties we address, they are not specifically dedicated to IE and therefore cannot exploit the structure of ECA rules that define IE systems.

3 Preliminaries: ECA Rules and IRON

In this section we introduce preliminary notation and formalism that will be employed in the rest of the paper. We start by introducing a model for ECA rules, we then describe a language for ECA rules and we also include a short overview of the weakest precondition predicate transformer and SMT solvers.

3.1 State Space

Applications for IE are usually built starting from a set of sensors and actuators that interact by means of message passing over a network. In [21] we presented a formal model representing a system composed of devices of two categories: sensors and actuators.

We denote with D the set of variables that identify the devices of the system. In other words, D is the union of two disjoint sets, I and O, whose elements are, respectively, the sensors and the actuators of the system. We use the notation $D = I \cup O$ where $I = \{i_1, \ldots, i_m\}$ and $O = \{o_1, \ldots, o_n\}$ for some $m, n > 0$.

Definition 1 A *state* of the system is defined as the function $\varphi : D \to V$ where V is a finite set of integer or boolean values. In other words, φ is a standard first-order valuation of the variables in D.

For instance, if we consider the set

$$D_{house} = \{presence, light, tv\}$$

where *presence* is a boolean sensor and *light* and *tv* are boolean actuators, a state is a function of the type $\varphi : D_{house} \to \{0, 1\}$ that associates one value to each device in D_{house}.

We can represent the function φ as

$$\varphi = \{i_1 \mapsto u_1, \ldots, i_m \mapsto u_m, o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\}, \quad (1)$$

where $u_j, v_k \in V$ for $j = 1, \ldots, m$ and $k = 1, \ldots, n$. In other words, a state φ is a set of associations between variables in D and their specific values in V. We use the notation $\varphi(d)$ for representing the value v_d associated to the generic device d by means of φ. Given the set D_{house} defined above, in the following state

$$\varphi = \{presence \mapsto 1, \ldots, light \mapsto 0, tv \mapsto 1\}$$

two devices are on and one actuator is off. We will also use the notation $\varphi = < I_\varphi, O_\varphi >$ instead of $\varphi = I_\varphi \cup O_\varphi$, where

$$I_\varphi = \{i_1 \mapsto u_1, \ldots, i_m \mapsto u_m\}$$
$$O_\varphi = \{o_1 \mapsto v_1, \ldots, o_n \mapsto v_n\}.$$

Using the notion of state given in Def. 1, we can define, as usual, the standard notion of satisfaction of a first-order formula by a state. We will denote that φ satisfies the formula F, as usual, by $\varphi \models F$.

Given the definition of a state, we now introduce the definition of universe.

Definition 2 The *universe* Φ of a 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.

According to the running example D_{house}, the corresponding universe $\Phi house$ has cardinality given by 2^3. By adding invariants to the system, i.e. conditions that must be satisfied independently from the dynamic of the system, we can define the admissible state space as follows. Let us denote the set of the invariants of the system with

$$Inv = \{inv_1, inv_2, \ldots, inv_v\} \quad (2)$$

where an element $inv \in Inv$ is a restricted first-order logic predicate as defined in the IRON grammar (defined below in Section 3.3).

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.

For instance, if all sensors and actuators in D are boolean, the set V consists of two values and the set Φ has cardinality 2^{m+n}. By applying the static constraints we obtain the *admissible state space* $\Phi_a \subseteq \Phi$ having cardinality less or equal than 2^{m+n}. The set of static invariants must be satisfied independently from the set of ECA rules of the system. We will denote a generic state in Φ_a with φ. For example, if we define the following invariants for the system defined starting from D_{house}

$$Inv_{house} = \{((\text{not } light) \text{ or } presence), ((\text{not } tv) \text{ or } presence)\}$$

we can observe a reduction in the cardinality of the state space.

3.2 ECA rules

Given the set D and the state space Φ, we consider a finite set R of labels for ECA rules

$$R = \{r_1, r_2, \ldots, r_k\}, \quad k \in \mathbb{N}_0. \quad (3)$$

Since ECA rules are inspired by UML [12], we use the notation $Event[Condition]/Action$, to represent a generic ECA rule labelled with r. Therefore, a generic rule r in R is represented as

$$e[c]/a \quad (4)$$

where e, c, a are, respectively, labels for the event, the condition and the action of r. Let us now define each component of the ECA rule r. The generic event e is represented as a subset of variables in D, i.e.

$$e = \{d_{\alpha_1}, \cdots, d_{\alpha_f}\} \subseteq D.$$

The event is the trigger for the ECA rule, i.e. when a change concerning the value of at least one of the variables in e occurs, the condition is evaluated. The condition c is a restricted first-order logic predicate (as defined in the IRON grammar below) having variables in D, i.e.

$$c = P(d_{\beta_1}, \cdots, d_{\beta_l}), \quad d_{\beta_1}, \cdots, d_{\beta_l} \in D.$$

If the condition is true, the action is applied to the state of the system. A generic action a is defined as a set of assignments for a subset of actuators in O i.e.

$$a = \{o_\gamma \leftarrow E_\gamma \mid \text{where } o_\gamma \in O \text{ and } E_\gamma \text{ is a term over } D\}. \quad (5)$$

Note that each o_γ must appear at most once in a (i.e., we do not allow two or more assignments to the same variable). Additionally, we will write $o_\gamma \uparrow a$ if there is no E_γ such that $(o_\gamma \leftarrow E_\gamma) \in a$.

In other words, the action a assigns values to a subset of variables in O. In detail

- if o_γ is a boolean actuator, E_γ is a boolean expression over boolean variables in D,
- if o_γ is an integer actuator, E_γ is an integer expression over integer variables in D.

Let us denote with A the set of actions associated to rules in R. In order to represent the effect of the action a of the rule r on the admissible state φ defined in (1) we introduce the following operator

$$[\,] : \Phi_a \times A \to \Phi$$

such that

$$\varphi[a] = \varphi'$$

where

$$\varphi' = \{i_1 \mapsto u_1, \ldots, i_m \mapsto u_m, o_1 \mapsto v'_1, \ldots, o_n \mapsto v'_n\}$$

and for $\gamma = 1, \ldots, n$

$$v'_\gamma = \begin{cases} E_\gamma|_\varphi & \text{if } (o_\gamma \leftarrow E_\gamma) \in a \\ v_\gamma & \text{otherwise.} \end{cases}$$

where the $|$ operator is the first-order interpretation function:

$$E_\gamma|_\varphi = E_\gamma(\varphi(d_1), \ldots, \varphi(d_h)), \quad d_1, \ldots d_h \in D.$$

The operator $[\cdot]$ is *deterministic*: an action a transforms a state φ to exactly one state $\varphi[a]$.

In order to make more clear the contents of this subsection, we introduce some examples of ECA rules. If we consider the example D_{house} presented above, we can define the rule $presence[presence = 1] \setminus light \leftarrow 1, tv \leftarrow 1$ and $presence[presence = 0] \setminus light \leftarrow 0, tv \leftarrow 0$. The first rule turns on the light and the tv if someone enter the house, while the second rule turns them off if someone exists.

Notice that we have exploited here 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.

Moreover, for what concerns the execution semantics, the assignments that are part of the actions are executed in parallel. In detail, this means that all the assignments make use, in their right-hand side, of the values in state φ.

As no priority between rules is specified, in the verification step we explore all possible orderings. This means that whenever more than one rule is applicable, one rule is chosen non-deterministically and executed. In details, our execution semantic is described in [21] and is very similar to that one specified in [16].

3.3 IRON

We employ IRON (Integrated Rule on Data) as the underlining formalism for modelling IE. IRON is presented in [7]: it is a restricted first-order logic-based language that supports the categorisation of devices into sets [19], allows the definition of properties over sets and supports multicast and broadcast abstractions.

IRON programs are composed of two separate classes of specification: static and dynamic. We report the IRON syntax in Figure 1; in this grammar, $[x]$ means an optional occurrence of x, and boldface denotes keywords of the language.

The *static part* is composed of variables declarations (these variables can be sets, physical and logical devices) plus global constraints defined over them using restricted first order 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 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 in order 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 $9 - 11$ of the grammar).

The static part also includes the declaration of *constraints* (specified by the keyword *where*), i.e. laws that various variables, devices and sets must always satisfy. Constraints can be used in order to specify rules that bind variables together. The use of a constraint has two applications (1) it defines

```
1  Program ≡ ( Device | Rule | VarDecl )+;
2
3  Device  ≡ PhysicalDevice | LogicalDevice | Set;
4
5  PhysicalDevice  ≡ physical (sensor|actuator)
6      Type Id [= Exp] node(Id Sep Id ) [in id (Sep Id )*]
7      [where BoolExp];
8
9  LogicalDevice ≡ logical (sensor|actuator)
10     Type Id = Exp[in Id (Sep Id )*]
11     [where BoolExp];
12
13 Set ≡ set (sensor | actuator) Type Id;
14
15 Rule ≡ rule Id
16     on Id (Sep Id)* when BoolExp then Action;
17
18 Action ≡ [Id = Exp ]+;
19
20 Exp ≡ BoolExp | IntExp;
21
22 BoolExp ≡ CompIntExp | (BoolExp BoolBinaryOp BoolExp) |
23     (BoolUnaryOp BoolExp) | PrimaryBoolExp;
24 CompIntExp ≡ IntExp CompOp IntExp;
25 PrimaryBoolExp ≡ (LRB BoolExp RRB) | BoolConst | Id;
26
27 IntExp ≡ UnaryIntExp | (IntExp IntBinaryOp IntExp);
28 UnaryIntExp ≡ (IntUnaryOp UnaryIntExp)| PrimaryIntExp;
29 PrimaryIntExp ≡ (LRB IntExp RRB) | IntConst | Id;
30
31 Type ≡ integer | boolean;
32
33 IntConst ≡ [-] Digit (Digit)*;
34
35 Digit = [0-9];
36
37 BoolConst ≡ true | false;
38
39 Letter ≡ [A-Za-z];
40
41 Id ≡ Letter(Letter|Digit)*;
42
43 IntBinaryOp ≡ + | - | * | /;
44 BoolBinaryOp ≡ and | or;
45 BoolUnaryOp ≡ not;
46 CompOp ≡ == | != | < | > | <= | >=;
47 IntUnaryOp ≡ + | -;
48
49 Sep ≡ ,;
50 LRB  ≡ (;
51 RRB ≡ );
52
53 VarDecl ≡ BoolVarDecl | IntVarDecl ;
54 BoolVarDecl ≡ boolean Id [= BoolExp] [where BoolExp];
55 IntVarDecl ≡ integer Id [= IntExp] [where BoolExp];
```

Fig. 1 The IRON extended BNF

valid states of the system regardless of the rules that are de-
fined, and (2) it is used at run-time to verify whether any
physical device is providing erroneous data. *Sets* (line 3 of
the grammar) are considered to be logical devices and are
used to group together either sensors or actuators of the same
type (line 13 of the grammar). A programmer can assign val-
ues to a set that contains actuators. This assignment can be
used in order to instruct all the actuators to perform a spe-
cific action. Effectively, a set assignment is an abstraction
of a multicast communication primitive that can be used to
communicate an action to be performed to actuators. A pro-
grammer can read the value of a set of sensors in order to
define events and specify conditions. To this end various set
operators are introduced.

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
an identifier (line 15 of the grammar) and is composed of
three different parts that are *on*, *when* and *then* (line 16 of the
grammar). A list of variables follow the *on* keyword. When-
ever one of them changes its value, the boolean expression
that follows the keyword *when* is evaluated. When this ex-
pression is evaluated to true the rule can be applied and the
actions listed after the keyword *action* can be executed. A
boolean expression (line $22 - 25$ of the grammar) can in-
clude relational and logical operators, integers, devices and
variables. An action is a list of assignments to variables,
physical actuators and logical devices. Special operators are
used to support the definition of a boolean condition over a
set: *all*, *any*, *no*, *one* and *lone* (line 49 of the grammar). *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 order to specify
that at least one of the element of the set must satisfy the
condition. *No* (*one*) is used when we need to express that no
(respectively, exactly 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.

For the sake of simplicity but without loss of generality,
the model presented in [21] and briefly resumed in Section
3.1 and 3.2 does not include the definition of sets and the
distinction between logical and physical devices as detailed
in IRON. These could be introduced at the cost of additional
notation but do not affect the overall verification strategy we
propose in this paper.

3.4 Weakest preconditions and SMT solvers

In this section we briefly provide the theoretical foundations
for the proposed approach, i.e., predicate transformers and
Satisfiability Modulo Theories (SMT).

Dijkstra introduced the notion of *predicate transformer*
in [11] and developed a formalism for proving program prop-
erties by focussing on a particular predicate transformer, the
well-known *weakest precondition* transformer.

Given a statement S of some programming language, the
weakest (liberal) precondition operator transforms a predi-
cate Q to a predicate $\mathrm{wp}(S, Q)$, denoting the *largest* set of
states such that if S is executed at any of these states and it
terminates, then it necessarily does so at a state satisfying Q.

Equations satisfied by $\mathrm{wp}(S, Q)$ can be found in [11]. In
particular, the weakest precondition predicate transformer
semantics of an assignment $V := E$ is

$$\mathrm{wp}((V := E), Q) = Q[E/V].$$

However, in the context of ECA rules, statements are *multiple* assignments a, meaning that each action a may specify multiple simple assignments that occur simultaneously. The above rules are not adequate for working out the weakest preconditions of a multiple assignment. Let

$$\theta = E_1/V_1 \cdots E_n/V_n$$

stand for the *simultaneous substitution* of variable V_i with the expression E_i for $1 \leq i \leq n$. Gries [14] provides the following rule:

$$\mathrm{wp}((V_1 := E_1, \ldots, V_n := E_n), Q) = Q[\theta], \qquad (6)$$

i.e., the weakest precondition of a simultaneous substitution is a formula $Q[\theta]$ in which variables V_i are replaced by E_i, for all $i \in \{1, \ldots, n\}$.

Closely related to the problem of program verification, the topic of *Satisfiability Modulo Theories* (SMT) [1] is a growing area of automated deduction with many important applications, especially in static analysis and system verification. An SMT problem is the problem of checking the satisfiability of logical formulae over one or more theories. Given a certain logical formula, SMT solvers find satisfying assignments (or report that there are none).

An *SMT solver* is any software that implements a procedure for satisfiability *modulo* some given theory, for example the theory of linear arithmetic. This means that logical formulae of this logic are boolean combinations of linear arithmentic expressions. 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 makes the formula true [9]. For the purposes of our work, as described below, we are interested in *unquantified linear integer arithmetic*, i.e., boolean combinations of equalities, comparisons and inequalities between linear expressions over integer and boolean variables.

4 Properties of ECA rules in Intelligent Enviroments

The application of formal verification techniques to ECA-based programs is essential to support the error-prone activity of defining ECA rules. We want to avoid "bad" situations deriving from erroneous definitions of ECA rules that may result in inefficient or potentially dangerous effects on the real world.

In this work we focus on the verification of the *consistency* of a system of ECA rules. This property is defined as follows.

Definition 4 A system of ECA rules satisfies the *consistency* property if its rules are neither unused nor redundant nor incorrect.

In the next subsections we will properly define unusability, redundancy and incorrectness. We now give a brief overview of these properties with respect to the literature. All these properties are application specific, and they are relevant to guarantee fundamental requirements for IE systems and avoid possible errors.

According to [7], unusable rules are rules that can never be applied. The authors categorise them into inapplicable rules and rules with contradictory premises. A rule that is inapplicable has a condition that is only true for states that are outside the domain. A rule with a contradictory premise has some logical contradiction in its condition and thus it can never be true. They also define the concept of redundancy, as similarly done by authors of [6]. They state that state that redundancy of rules is detected when the condition and the action of one rule represent a subset of the condition and the action of the other rule. In oder words, this means that there are two or more rules in the system whose functionality is replicated. This could represent an overload in the rules system in the best cases, and an undesired repetitive activation of orders on environment devices. The concept of correctness concerns whether an application's behaviour meets its requirement specification, as stated in [13].

All these properties are important requirements to guarantee reliability of IE applications. In particular, the user could have defined a rule in an inappropriate way, and this error could lead to an undesired behaviour of the system. So the user can benefit from the identification of inconsistent rules.

4.1 Unused Rules

Informally, we can say that a generic rule $r \in R$ is called *unused* if it can never be applied to any state of the system. This property characterises those rules whose conditions are only true for the states that are outside the admissible universe Φ_a. For instance, we can consider a temperature physical sensor t of the type integer whose value belongs to the interval $[-80, +60]$ (e.g. expressed in degree celsius), that is its value cannot exceed sixty degree and be under minus eighty degree celsius. The condition ($t > 65$ or $t < -90$) is never met, since it is only valid outside the admissible interval for t. Another example is given by a rule whose condition is ($t < 5$ and $t > 20$): such a condition is never true since it is a logical contradiction.

We can formalize the definition of unused rules as follows:

Definition 5 An ECA rule $r \in R$ of the form (4) is called *unused* if the condition c is false for every state $\varphi \in \Phi_a$.

Since the set Φ_a is the set of all states in Φ satisfying the invariants (2) of the system, we can alternatively say that r

is unused if the logical predicate

$$P = c \wedge inv_1 \wedge inv_2 \wedge \cdots \wedge inv_v \tag{7}$$

is such that $P(\varphi)$ is false for all states φ.

4.2 Redundant Rules

In the IE context, redundancy means that there are two or more rules in the system whose outcome is replicated. As an example, we consider i_1, i_2 boolean sensors and o_1, o_2 boolean actuators. No constraints are defined. The following rules are given:

$r1 : i_1[(i_1 = true) \text{ and } (i_2 = false)] \setminus o_1 \leftarrow false$

$r2 : i_1, i_2[(i_1 = true) \text{ and } (i_2 = false)] \setminus o_1 \leftarrow false$

$r3 : i_1[i_1 = true] \setminus o_1 \leftarrow false$

$r4 : i_1[(i_1 = true) \text{ and } (i_2 = false)] \setminus o_2 \leftarrow false.$

Looking at them, we can observe that the rule $r1$ is identical to rule $r2$ apart from the variable in the event part. In fact $r1$ is applicable only if the value of i_1 changes, while rule $r2$ is applicable also when i_2 change its value. We can refer to this situation by using the term *redundancy*, i.e., r_1 is redundant w.r.t. r_2, in the sense that functionality of $r1$ is included in $r2$. Another case of redundancy happens when a rule subsumes another rule, that is whenever one rule is applicable, then the other is also applicable. This is the case of $r1$ and $r3$, since the condition $((i_1 = true) \text{ and } (i_2 = false))$ implies $(i_1 = true)$, and the event and action parts of these rules are identical. However, rules $r1$ and $r4$ are not redundant, since the action parts are different.

The redundancy property is defined as follows.

Definition 6 Given $r_i, r_j \in R$ such that

$r_i : \quad e_i[c_i]/a_i$

$r_j : \quad e_j[c_j]/a_j$

we say that r_i is redundant w.r.t. r_j if the following conditions are met:

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

Condition 1. means that variables in e_i belong to e_j too. Thus this condition, if met, guarantees that every time the occurrence of an event triggers r_i then it triggers r_j too. If both 1. and 2. are verified, than when r_i is applicable, so is r_j. If the result of applying action a_i is equal to the result of applying a_j, i.e., if given a certain state in $c_i \wedge Inv$, the state that results from the application of a_i is always equal to the state obtained by performing action a_j, then r_i is redundant

with respect to r_j, in the sense that r_i is "included" in r_j. A possible system behaviour is represented in figure 2. According to [21], empty circles represent admissible unstable states, while double circles are admissible stable states. Natural evolutions of the system, i.e., changes in sensor values, are represented by dotted arrows, while artificial transitions (ECA rules) are represented by solid arrows. States with the same configuration of sensor values are grouped into boxes.

According to the previous definition, the figure could represent a situation due to redundancy, in the sense that when r_2 is activated, also r_1 is applied, and there is a transition of the system that results from the application of r_2 but r_1 is not applicable. It could be that r_1 is redundant with respect to r_2, but formal verification is needed.
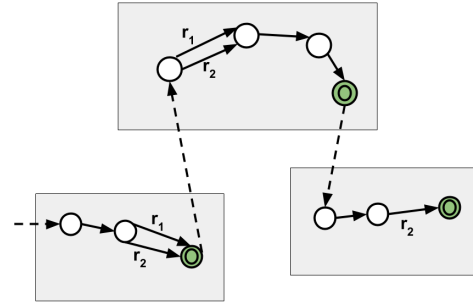

Fig. 2 Example of redundant rules

4.3 Incorrect Rules

In this work we use the term *incorrect* to refer to a rule r that can lead to a state whose values are not admissible. For instance, consider an actuator l for adjusting indoor light intensity whose integer value belongs to the interval $[0, 5]$. A rule r whose condition is always true for the admissible states and whose action assigns to l the value 10 is considered incorrect, since the action always leads the system outside of the valid domain. Another rule having the condition given by $(1 \leq l \leq 5)$ and the action $l \leftarrow l + 2$ is correct if applied for example to a state φ having the value 3 for l but it leads to a non-admissible state if the initial value is $l = 4$. We now define this property formally.

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

The term "incorrect" can be interpreted as "potentially incorrect" in the sense that the definition of incorrectness includes not only rules that always lead the system to a non-admissible state, but also rules that if applied to a certain

admissible state, lead to a state that deny the invariants of the system.
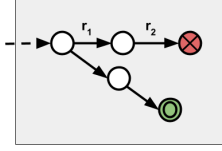

Fig. 3 Example of incorrect rules

In figure 3 for instance the rule r_2 leads to a non-admissible state (represented by a crossed circle).

5 Verification Algorithms

In this section we describe the algorithms that allow the verification of the properties defined in Section 4. The general approach is based on the following observations.

According to the model described above, our key insight is observing that it is possible to perform verification of ECA rules using the weakest precondition/strongest postcondition predicate transformers.

Before going into details, we want to emphasize the fact that the verification algorithms we will describe exploit the specific structure of ECA rules in IE. In fact, the actions only deal with output variables and this reduces the size of the state space in the verification as sensor variables can be discarded.

5.1 Unused Rules

The detection of unused rules is based on Definition 5. Given a generic rule $r : e[c]/a$, we have to check whether the condition c is satisfiable in Φ_a. This is equivalent to asking a (sound and complete) SMT solver whether there exists φ such that formula P defined in (7) is satisfiable. If P is satisfiable, the rule r can be used, otherwise it is unused. In Figure 4 the algorithm is described in detail.

```
1  let  R := {r_1,....,r_k}
2  let  I := {inv_1,....,inv_v}
                 v
3  define  Inv = ⋀ inv_j
                 j=1
4  for each  i = 1,...,k:
5    if  (c_i ∧ Inv) is unsatisfiable:
6      declare  r_i unused
7  end
```

Fig. 4 Unused rule verification

Proposition 1 *The algorithm in Fig. 4 is correct.*

Proof

$$
\begin{array}{lll}
& \forall \varphi \in \Phi_a.\, \varphi \not\models c & \text{(Def. 5)} \\
\Leftrightarrow & \forall \varphi.\, \varphi \models Inv \Rightarrow \varphi \not\models c & \text{(Def. } \Phi_a\text{)} \\
\Leftrightarrow & \forall \varphi.\, \varphi \not\models Inv \wedge c & \text{(Prop. Log.)} \\
\Leftrightarrow & \neg \exists \varphi.\, \varphi \models Inv \wedge c & \text{(Duality)} \\
\Leftrightarrow & Inv \wedge c \text{ is unsatisfiable.}
\end{array}
$$

5.2 Redundant Rules

The detection of redundant rules is performed under the following hypothesis on the ECA rule structure: with respect to the general definition in Section 3.2, if o_γ is an integer actuator, we will consider only expressions E_γ that are linear functions to integer variables, i.e. of the kind

$$ f : \mathbb{Z}^h \to \mathbb{Z} $$

where $h \in \mathbb{N}$ depends on the specific assignment and f is such that

$$ f(d_1,...,d_h) = k_1 d_1 + \ldots + k_h d_h + k_0, \;\; k_0, k_1, \ldots k_h \in \mathbb{N}. $$

In order to detect redundant rules, we have to check whether conditions 1-3 of Definition 6 are verified. The algorithm is described in Figure 6 and explained below. The algorithm makes use of the definition given in Figure 5.

$$
\psi_{a_i,a_j}(o) = \begin{cases}
\top & \text{if } o \uparrow a_i \text{ and } o \uparrow a_j \\
o = E_j & \text{if } o \uparrow a_i \text{ and } (o \leftarrow E_j) \in a_j \\
E_i = o & \text{if } (o \leftarrow E_i) \in a_i \text{ and } o \uparrow a_j \\
E_i = E_j & \text{if } (o \leftarrow E_i) \in a_i \text{ and } (o \leftarrow E_j) \in a_j
\end{cases}
$$

$$ \Psi(a_i, a_j) = \bigwedge_{o \in O} \psi_{a_i,a_j}(o) $$

Fig. 5 Preliminary defnition.

```
1  let  R := {r_1,....,r_k}
2  let  I := {inv_1,....,inv_v}
                 v
3  define  Inv = ⋀ inv_j
                 j=1
4  for each  ordered pair (r_i,r_j) ∈ R^2 such that  r_i ≠ r_j  and
5    such that  r_i,r_j are usable:
6    if  (e_i ⊆ e_j)  and  (¬(c_i ∧ Inv ⇒ c_j) is unsatisfiable)  and
7      ¬(c_i ∧ Inv ⇒ Ψ(a_i, a_j)) is unsatisfiable):
8      then declare  r_i redundant with respect to r_j
9  end
```

Fig. 6 Redundancy verification.

First of all, we consider all ordered pairs of distinct rules, i.e. (r_i, r_j) such that $r_i \neq r_j$, among those ones that are usable, i.e., not unused according to Definition 5. Rule r_i is redundant with respect to r_j if conditions 1-3 of Definition 6 are all verified. This fact is expressed at line 4 in Figure 6. Condition 1 simply corresponds to $e_i \subseteq e_j$.

The property $c_i \wedge Inv \Rightarrow c_j$ is checked by asking the solver whether its negation is satisfiable. This allows us to verify whether there exists a state in which $c_i \wedge Inv$ is true but c_j is not and, if this is the case, we can conclude that r_j is *potentially* redundant with respect to r_i:

$$\neg(c_i \wedge Inv \Rightarrow c_j) \text{ is satisfiable.}$$

We also need to check condition 3 to verify effective redundancy: the action parts of the considered rules must be in some sense equivalent within the domain $c \wedge Inv$. This is done by checking the validity of the formula $c_i \wedge Inv \Rightarrow \Psi(a_i, a_j)$, where $\Psi(a_i, a_j)$ is defined in Fig. 6.

If all conditions (line 6 of the algorithm in Figure 6) are verified, then the rule r_i is declared redundant with respect to r_j. The procedure described above must be performed for all pairs of rules (line 4).

Proposition 2 *The algorithm in Fig. 6 is correct.*

Proof Requirements (1) and (2) of Def. 6 are checked verbatim in the algorithm. Thus, it remains to show that the algorithm is correct w.r.t. requirement (3). Recall requirement 3:

$$\forall \varphi \models c \wedge Inv, \varphi[a_i] = \varphi[a_j].$$

This can be equivalently rewritten as

$$\forall o \in O. o|_{\varphi[a_i]} = o|_{\varphi[a_j]} \tag{8}$$

since we require that for all variables in O, their final values are the same after executing a_i and a_j.

Equally, the formula checked in the algorithm can be rewritten appropriately:

$$\varphi \models \Psi(a_i, a_j)$$
$$\Leftrightarrow \quad \varphi \models \bigwedge_{o \in O} \psi_{a_i, a_j}(o)$$
$$\Leftrightarrow \quad \forall o \in O. \varphi \models \psi_{a_i, a_j}(o)$$

Thus we need to show that for any $o \in O$, $o|_{\varphi[a_i]} = o|_{\varphi[a_j]}$ is equivalent to $\varphi \models \psi_{a_i, a_j}(o)$. We do this by case analysis.
Case $o \uparrow a_i$ **and** $o \uparrow a_j$. First, observe that by Fig. 6, $\psi_{a_i, a_j}(o) = \top$, thus trivially $\varphi \models \psi_{a_i, a_j}(o)$. Since o is not assigned by neither a_i nor a_j, this means $o|_\varphi = o|_{\varphi[a_i]} = o|_{\varphi[a_j]}$ and thus we are done.
Case $o \uparrow a_i$ **and** $(o \leftarrow E_j) \in a_j$. By Fig. 6, $\psi_{a_i, a_j}(o)$ is $o = E_j$. Thus we need to prove that $\varphi \models o = E_j$ iff $o|_{\varphi[a_i]} = o|_{\varphi[a_j]}$. But since $o \uparrow a_i$, it follows that $o|_\varphi = o|_{\varphi[a_i]}$. By the fact that

$(o \leftarrow E_j) \in a_j$, we can conclude that $o|_{\varphi[a_j]} = E_j|_\varphi$. We are done, since $\varphi \models o = E_j \Leftrightarrow o|_{\varphi[a_j]} = E_j|_\varphi$.
Case $(o \leftarrow E_i) \in a_i$ **and** $o \uparrow a_j$. Symmetric to previous case.
Case $(o \leftarrow E_i) \in a_i$ **and** $(o \leftarrow E_j) \in a_j$. Now, $\psi_{a_i, a_j}(o)$ is $E_i = E_j$; $o|_{\varphi[a_i]} = E_i|_\varphi$; and $o|_{\varphi[a_j]} = E_j|_\varphi$. Thus we need to show that $\varphi \models E_i = E_j \Leftrightarrow E_i|_\varphi = E_j|_\varphi$ which is true by construction.

5.3 Incorrect Rules

In order to verify the correctness of the generic rule $r : e[c]/a$ according to definition 7, we compute the weakest precondition P_{Inv} for the set of invariants, i.e., the formula

$$P_{Inv} = \text{wp}(a, Inv).$$

In order to declare the rule r correct, the set of states where the rule may apply must be contained into the set of states P_{Inv}, for otherwise there would exist a state where the rule applies, but from which we can reach a state outside those satisfying the invariants. Therefore, we translate this problem into the following SMT instance:

$$(c \wedge Inv) \wedge \neg P_{Inv}$$

and we verify that this is *not* satisfiable. If the solver answers that the proposition is satisfiable, then we conclude that the rule r is incorrect, otherwise we declare that r is correct. The algorithm is described in figure 7.

```
1  let  R := {r₁, ...., rₖ}
2  let  I := {inv₁, ...., invᵥ}
                      v
3  define  Inv = ⋀ invⱼ
                     j=1
4  for each  i = 1,...,k:
5      if  (cᵢ ∧ Inv) ∧ ¬wp(a,Inv)  is satisfiable:
6          declare  rᵢ  incorrect
7  end
```

Fig. 7 Incorrectness verification

Lemma 1 *Let a be an assignment as defined in (5), and let φ be a state. Then*

$$\varphi \models \text{wp}(a, P) \qquad \Leftrightarrow \qquad \varphi[a] \models P.$$

Proof By definition, $\varphi \models \text{wp}(a, P)$ iff for every state φ' that results by a terminating execution of a starting at φ, it is the case that $\varphi' \models P$. However, as argued in Section 3.2, the transition relation of any assignment a (the operator $[\cdot]$) is deterministic. In addition, assignments that involve linear functions over integers are always terminating. Therefore, $\varphi \models \text{wp}(a, P)$ iff $\varphi[a] \models P$.
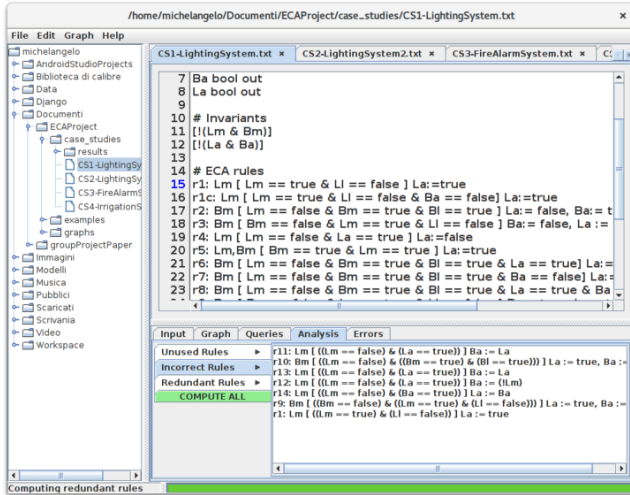
Proposition 3 *The algorithm in Fig. 7 is correct.*

Fig. 8 ECA tool

Proof

$$\exists \varphi \in \Phi_a. \varphi \models c \text{ and } \varphi[a] \notin \Phi_a \qquad \text{(Def. 7)}$$

$$\Leftrightarrow \quad \exists \varphi. \varphi \models Inv \wedge c \text{ and } \varphi[a] \not\models Inv \qquad \text{(Def. } \Phi_a\text{)}$$

$$e \Leftrightarrow \quad \exists \varphi. \varphi \models Inv \wedge c \text{ and } \varphi \not\models \mathtt{wp}(a, Inv) \qquad \text{(Lem. 1)}$$

$$\Leftrightarrow \quad \exists \varphi. \varphi \models Inv \wedge c \wedge \neg \mathtt{wp}(a, Inv) \qquad \text{(Prop. Log.)}$$

$$\Leftrightarrow \quad Inv \wedge c \wedge \neg \mathtt{wp}(a, Inv) \text{ is satisfiable}$$

6 Evaluation

6.1 The prototype tool

We have implemented a tool to evaluate our approach. We have released the tool open source[1]. Figure 8 depicts the graphical interface of the tool. Users only need to select input IRON files and then the tool reports the results of the analysis.

The tool makes use of Z3 [8], a high-performance SMT Solver implemented in C++ and developed at Microsoft Research. Z3 is released under the Microsoft Research Licence Agreement (MSR-LA) license[2]. Application Programming Interfaces (APIs) are available in C, C++, Java and others. Out tool is developed entirely in Java and we make heavy use of the Java API. To implement the algorithms described in Section 5 we consider the logic "QF_LIA" (quantifier-free linear arithmetic).

The verification proceeds in the following order: firstly, unused rules are detected, then among usable rules we look for incorrect rules, finally we check redundancy.

In the next section we describe four case studies taken from the literature and we present performance results. We

report only the key parts of each example and we refer to the files available on-line for the full source and additional details of each example.

6.2 Case study 1 (CS1)

The first case study models a smart home and it is an extension of the one described in [21]. In the simple scenario represented in figure 9 the house is composed of two rooms: the living room (L) and the bedroom (B). The entrance is in the living room and the bedroom is accessible from the living room. Both rooms contain a motion sensor (m), a light sensor (l), a light switch (s) to turn on and off the light manually, a light actuator (a) to automatically turn on and off the lamp.
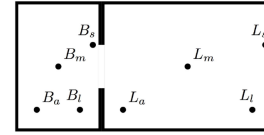

Fig. 9 Floormap.

We assume that only one person has access to the house at any given time. In figure 10 the invariants of the system are declared at lines $2 - 3$. The first constraint states that the person cannot stay in both rooms simultaneously. The second one states that light actuators cannot be both on at the same time. As a consequence, according to the first constraint, a state having both light actuators on is not admissible. At the end of the verification procedure, among the rules reported in figure 10, the rule $r5$ is declared unused, since the condition is never met, $r1$ is incorrect, since it could lead outside of the domain if applied to a state having $Ba := true$ (a possible correct version of $r1$ is $r1c$). Rules $r11$ and $r13$ are incorrect, since they lead the system to a non admissible state. Among usable and correct rules, the verification procedure declares rule $r7$ redundant with respect to $r2$, rules $r4$ and $r19$ are mutually redundant, rule $r6$ is redundant with respect to $r2, r7, r8$, the rule $r8$ is redundant with respect to $r2, r6, r7$.

6.3 Case study 2 (CS2)

The second case study has been adapted from [16], where a light control subsystem in a smart home for senior housing is considered. The number of devices is greater than that of CS1, and also the dynamic is more complex. Indeed, by using motion and pressure sensors (Mtn, Slp respectively), the system attempts to reduce energy consumption by turning off the lights in unoccupied rooms or if the occupant is

---

[1] it is available at https://gitlab.com/MichelangeloDiamanti/ecaProject

[2] http://research.microsoft.com/en-us/um/redmond/projects/z3/z3-commercial-license.pdf

```
 1 Bl bool in
 2 Bm bool in
 3 Bs bool in
 4 Ll bool in
 5 Lm bool in
 6 Ls bool in
 7 Ba bool out
 8 La bool out
 9
10 # Invariants
11 [!(Lm & Bm)]
12 [!(La & Ba)]
13
14 # ECA rules
15 r1: Lm [ Lm == true & Ll == false ] La:=true
16 r1c: Lm [ Lm == true & Ll == false & Ba == false] La:=
       true
17 r2: Bm [ Lm == false & Bm == true & Bl == true ] La:=
       false, Ba:= true
18 r4: Lm [ Lm == false & La == true ] La:=false
19 r5: Lm,Bm [ Bm == true & Lm == true ] La:=true
20 r6: Bm [ Lm == false & Bm == true & Bl == true & La ==
       true] La:= false, Ba:= true
21 r7: Bm [ Lm == false & Bm == true & Bl == true & Ba ==
       false] La:= false, Ba:= true
22 r8: Bm [ Lm == false & Bm == true & Bl == true & La ==
       true & Ba == false] La:= false, Ba:= true
23 r9: Bm [ Bm == false & Lm == true & Ll == false ] Ba:=
       true, La:= true
24 r11: Lm [ Lm == false & La == true ] Ba:= La
25 r14: Lm [ Lm == false & Ba == true ] La:= Ba
26 r19: Lm [ Lm == false & La == true ] La := !La
```

Fig. 10 (CS1) ECA rules for the lighting control system in a simple scenario.

```
 1 Mtn bool in
 2 ExtLgt int in
 3 Slp bool in
 4 lgtsTmr int out
 5 intLgts int out
 6 Lgts bool out
 7 ChkExtLgt bool out
 8 ChkMtn bool out
 9 ChkSlp bool out
10
11 # Invariants
12 [ ExtLgt >= 0 & ExtLgt <= 10 ]
13 [ lgtsTmr >= 0 & lgtsTmr <= 120 ]
14
15 # ECA rules
16 r2: Mtn, ExtLgt, Slp [ lgtsTmr >= 1 & Mtn == false ]
       lgtsTmr:=lgtsTmr+1
17 r2c: Mtn, ExtLgt, Slp [ lgtsTmr >= 1 & Mtn == false &
       lgtsTmr < 120 ] lgtsTmr:=lgtsTmr+1
18 r5: ChkExtLgt [ ChkExtLgt == true & Lgts == false &
       ExtLgt <= 5] Lgts:=true
19 r11: ChkExtLgt [ ChkExtLgt == true & Lgts == false &
       ExtLgt <= 4] Lgts:=true
```

Fig. 11 (CS2) ECA rules for the light control system of a smart home.

```
 1 temperature int in
 2 smoke bool in
 3 presenceLiving bool in
 4 sprinkler bool out
 5 heating bool out
 6 tv bool out
 7 light bool out
 8 tempAlarm bool out
 9 smokeAlarm bool out
10
11 # Invariants
12 [ temperature > -80 & temperature < 60]
13
14 # ECA rules
15 r1: temperature [ temperature < 16 ] heating:=true
16 r2: temperature [ temperature > 18 ] heating:=false
17 r7: temperature [ temperature < 30 & temperature > 30 ]
       tempAlarm:=true
18 r8: temperature [ temperature >= 1000 ] tempAlarm:=true
19 r9: temperature [ temperature == 14 ] heating:=true
20 r10: temperature [ temperature > 20 ] heating:=false
21 r11: temperature [ temperature < 15 & temperature > 50 ]
       tempAlarm:=true
```

Fig. 12 (CS3) ECA rules for a fire alarm system.

asleep, and it also provides automatic adjustment for indoor light intensity based on an outdoor light sensor (*ExtLgt*).

We made some changes in the admissible values for *lgtsTmr* and in the rules involving this variable (see line 3 in figure 11). Indeed, our tool automatically fixes upper and lower bounds (we choose the values of $+127, -128$ respectively) for those integer variables that have no limited values. We also defined some additional rules with respect to the original version of the case study in order to have a greater number of rules to be analysed.

We report in figure 11 a subset of the rules analysed. The verification procedure declares *r14* as unused, rule *r2* as potentially incorrect (since there is no upper bound for *lgtsTmr*, but for instance rule *r2c* is a possible correct version of *r2*). Finally, rule *r11* is redundant with respect to *r5*.

6.4 Case study 3 (CS3)

The case study we present in this subsection has been developed starting from the example presented in [7]. The dynamic of a fire alarm system composed of temperature sensors, smoke detectors and sprinkler actuators is described through ECA rules. When a temperature sensor reads a value that exceeds a specified threshold and a smoke sensor detects smoke all the sprinklers are activated. Among the rules defined in figure 12, rules *r7*, *r8*, *r11* are declared unused, there are no incorrect rules and for what concerns redundancy, *r9* is redundant with respect to *r1* and *r10* is redundant with respect to *r2*.

6.5 Case study 4 (CS4)

The fourth case study consists of a Wireless Sensor and Actuator Network (WSAN) composed of five devices for irrigation management system and controlled use of fertilizers. In detail the network is composed of a a rain sensor *r* to sense precipitation, a water valve actuator *w*, a fertilizer valve actuator *f*, a timer sensor *t* and a timer actuator *c* for the sprinkler. The property analysis gave the following results: *r5*, *r8*, *r9* are declared unused, *r2*, *r4*, *r7* are declared incorrect (for instance *r2c* is a possible correct version of *r2*), and there are no redundant rules.

```
1  r bool in    #rain
2  t int  in    #temperature
3  w bool out   #water
4  f bool out   #fertilizer
5  c int  out   #counter
6
7  # Invariants
8  [ (!f | w) ]
9  [ !(r & w) ]
10 [ c >= 0 & c < 120 ]
11 [ t >= 0 & t < 120 ]
12
13 # ECA rules
14 r2: t,w [ t-c > 2 & w == true ] w:=false,c:=t
15 r2c: t,w [ t-c > 2 & w == true & f == false ] w:=false,c
      :=t
16 r4: t,w [ t-c > 2 & w == true ]w:=false
17 r7: t,w [ t-c>8 ] w:=false, c:=t+1
18 r8: r [ r==true & w==true ]w:=false
19 r9: r [ r==true & w==true ] c:=t+1
```

Fig. 13 (CS4) ECA rules for an automatic irrigation system.

In figure 13 we define the invariants of the system and a set of ECA rules.

6.6 Results

In this section we present the results of the analysis of the case studies introduced in the previous subsections.

In table 1 we report some information about each problem and its results. In particular, we report the universe cardinality $|\Phi|$ and the dimension of the admissible state space $|\Phi_a|$. We also give the number of the analysed ECA rules N_{tot}, and the number of unsed, incorrect and redundant rules (denoted with N_{un}, N_{inc}, N_{red}) declared by the tool. Furthermore, we measure the performance of the verification procedure in terms of time (expressed in milliseconds). The indicators T_{un}, T_{inc} and T_{red} refer, respectively, to the duration time of the verification for unused, incorrect and redundant rules, and T_{tot} is the duration time of the entire procedure.

All the experiments have been performed on an Intel Core i7-4700MQ CPU @ 3.4GHz with 8GB of RAM running Debian Linux.

7 Discussion and Conclusion

We evaluated our approach by using some case studies taken from existing works in the area.

The results in Table 1 show that our approach allows for the verification of non-trivial examples that include both boolean and integer variables. The table reports the size of the examples, showing that a state space of 2^{30} and a number of rules comprised between 12 and 20 can be verified in approximately 1 second with our approach. The results also show that the running time does not seem to be affected by the size of the state space (compare CS1 with CS2), but rather by the *number of rules*. In particular, the verification

of redundancy is the most computationally expensive step, as the verification happens for each *pair* of rules, and thus it requires a number of iterations that is quadratic in the number of rules to be checked.

At a higher level, in this paper we have shown that techniques for software verification can be applied to the verification of ECA rules. We have identified key properties of ECA rules from a detailed analysis of related literature and we have implemented our algorithms in a tool that is publicly available.

In particular, we consider the 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. Focussing on IRON allows us to develop efficient techniques and our main contribution is showing that verification methods based on SMT solvers efficiently verify ECA-rule based systems specified in IRON language. With respect to the state of the art of methods applied to the verification of sensor/actuators applications, SMT solvers have never be applied.

Furthermore, we provide a user-friendly tool that has been implemented to test and validate our approach, and that can be used by developers of rule-based applications for IE for modelling and formally analysing their applications. In particular, by using formal methods, errors can be prevented from being introduced in the system in an early stage of development and thus we contribute to guarantee reliability of IE systems.

We are currently working at including non-linear expressions and we are also investigating additional properties of ECA rules, such as non-determinism, non-confluence and termination.

References

1. Barrett, C., Stump, A., Tinelli, C., Boehme, S., Cok, D., Deharbe, D., Dutertre, B., Fontaine, P., Ganesh, V., Griggio, A., Grundy, J., Jackson, P., Oliveras, A., KrstiÄĞ, S., Moskal, M., Moura, L.D., Sebastiani, R., Cok, T.D., Hoenicke, J.: C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010)

2. Berndtsson, M., Mellin, J.: ECA Rules, pp. 959–960. Springer US, Boston, MA (2009). DOI 10.1007/978-0-387-39940-9_504. URL http://dx.doi.org/10.1007/978-0-387-39940-9_504

3. Beyer, D., Stahlbauer, A.: BDD-based software verification. International Journal on Software Tools for Technology Transfer **16**(5), 507–518 (2014). DOI 10.1007/s10009-014-0334-1. URL http://dx.doi.org/10.1007/s10009-014-0334-1

4. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. impact. In: 2012 Formal Methods in Computer-Aided Design (FMCAD), pp. 106–113 (2012)

5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986). DOI 10.1109/TC.1986.1676819. URL http://dx.doi.org/10.1109/TC.1986.1676819

Table 1 Synthesis of the results.

Case study	State Space		ECA rules				Verification Time (ms)			
	$|\Phi|$	$|\Phi_a|$	N_{tot}	N_{un}	N_{inc}	N_{red}	T_{tot}	T_{un}	T_{inc}	T_{red}
CS1	2^8	$9 \cdot 2^4$	20	1	7	5	1297	197	331	769
CS2	2^{30}	$1331 \cdot 2^{14}$	17	1	1	1	1040	155	240	645
CS3	2^{16}	$139 \cdot 2^8$	14	3	0	2	827	113	177	537
CS4	2^{19}	$225 \cdot 2^8$	17	3	6	0	1085	256	279	550

6. Cano, J., Delaval, G., Rutten, E.: 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, chap. Coordination of ECA Rules by Verification and Control, pp. 33–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). DOI 10.1007/978-3-662-43376-8_3. URL http://dx.doi.org/10.1007/978-3-662-43376-8_3

7. Corradini, F., Culmone, R., Mostarda, L., Tesei, L., Raimondi, F.: A Constrained ECA Language Supporting Formal Verification of WSNs. In: 29th IEEE International Conference on Advanced Information Networking and Applications Workshops, AINA 2015 Workshops, Gwangju, South Korea, March 24-27, 2015, pp. 187–192 (2015). DOI 10.1109/WAINA.2015.109. URL http://dx.doi.org/10.1109/WAINA.2015.109

8. De Moura, L., Bjørner, N.: 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, TACAS'08/ETAPS'08, pp. 337–340. Springer-Verlag, Berlin, Heidelberg (2008). URL http://dl.acm.org/citation.cfm?id=1792734.1792766

9. De Moura, L., Bjørner, N.: Satisfiability modulo theories: An appetizer. In: Brazilian Symposium on Formal Methods, pp. 23–36. Springer (2009)

10. Delaval, G., Rutten, E., Marchand, H.: Integrating discrete controller synthesis into a reactive programming language compiler. Discrete Event Dynamic Systems **23**(4), 385–418 (2013). DOI 10.1007/s10626-013-0163-5. URL http://dx.doi.org/10.1007/s10626-013-0163-5

11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). DOI 10.1145/360933.360975. URL http://doi.acm.org/10.1145/360933.360975

12. Dumas, M., Hofstede, A.H.M.t.: UML Activity Diagrams as a Workflow Specification Language. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 76–90. Springer-Verlag, London, UK (2001). URL http://dl.acm.org/citation.cfm?id=647245.719456

13. Ericsson, A.: Enabling tool support for formal analysis of eca rules. Ph.D. thesis, University of Skövde (2009)

14. Gries, D.: The Science of Programming. Monographs in Computer Science. Springer New York (1989)

15. Gu, T., Wang, X.H., Pung, H.K., Zhang, D.Q.: An ontology-based context model in intelligent environments. In: Proceedings of communication networks and distributed systems modeling and simulation conference, vol. 2004, pp. 270–275. San Diego, CA, USA. (2004)

16. Jin, X., Lembachar, Y., Ciardo, G.: 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, pp. 41–59 (2013). URL http://ceur-ws.org/Vol-989/paper17.pdf

17. Lee, W.s., Lee, S.y., Lee, K.c.: Conflict detection and resolution method in WS-ECA framework. In: Advanced Communication Technology, The 9th International Conference on, vol. 1, pp. 786–791. IEEE (2007)

18. Marchand, H., Bournai, P., Borgne, M.L., Guernic, P.L.: Synthesis of discrete-event controllers based on the signal environment. Discrete Event Dynamic Systems **10**(4), 325–346 (2000). DOI 10.1023/A:1008311720696. URL http://dx.doi.org/10.1023/A:1008311720696

19. Mostarda, L., Marinovic, S., Dulay, N.: Distributed Orchestration of Pervasive Services. In: 24th IEEE IAINA 2010, Perth, Australia, 20-13 April 2010, pp. 166–173 (2010)

20. Sun, Y., Wang, X., Luo, H., Li, X.: Conflict detection scheme based on formal rule model for smart building systems. IEEE Transactions on Human-Machine Systems **45**(2), 215–227 (2015). DOI 10.1109/THMS.2014.2364613

21. Vannucchi, C., Cacciagrano, D.R., Corradini, F., Culmone, R., Mostarda, L., Raimondi, F., Tesei, L.: A Formal Model for Event-Condition-Action Rules in Intelligent Environments. In: Proceedings of the 11th International Conference on Intelligent Environments, pp. 56–65 (2016). DOI 10.3233/978-1-61499-690-3-56

22. Whitmore, A., Agarwal, A., Da Xu, L.: The internet of things—a survey of topics and trends. Information Systems Frontiers **17**(2), 261–274 (2015). DOI 10.1007/s10796-014-9489-2. URL http://dx.doi.org/10.1007/s10796-014-9489-2

23. Yoon, J.P.: Techniques for data and rule validation in knowledge based systems. In: Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', Proceedings of the Fourth Annual Conference on, pp. 62–70 (1989). DOI 10.1109/CMPASS.1989.76042

24. Zhang, J., Moyne, J., Tilbury, D.: Verification of ECA rule based management and control systems. In: 2008 IEEE International Conference on Automation Science and Engineering, pp. 1–7 (2008). DOI 10.1109/COASE.2008.4626431