Coordination of ECA Rules by Verification and Control

Julio Cano, Gwenaël Delaval, Eric Rutten

To cite this version:

Julio Cano, Gwenaël Delaval, Eric Rutten. Coordination of ECA Rules by Verification and Control. COORDINATION 2014 - 16th International Conference on Coordination Models and Languages, Jun 2014, Berlin, Germany. pp.33-48, 10.1007/978-3-662-43376-8_3 . hal-01290066

HAL Id: hal-01290066

https://hal.inria.fr/hal-01290066

Submitted on 17 Mar 2016

Coordination of ECA rules by verification and control

Julio Cano[1], Gwenaël Delaval[2], and Eric Rutten[1]

[1] INRIA, Grenoble, France,
julio-angel.cano-romero@inria.fr, eric.rutten@inria.fr
[2] LIG / UJF, Grenoble, France, gwenael.delaval@inria.fr

Abstract. Event-Condition-Action (ECA) rules are a widely used language for the high level specification of controllers in adaptive systems, such as Cyber-Physical Systems and smart environments, where devices equipped with sensors and actuators are controlled according to a set of rules. The evaluation and execution of every ECA rule is considered to be independent from the others, but interactions of rule actions can cause the system behaviors to be unpredictable or unsafe. Typical problems are in redundancy of rules, inconsistencies, circularity, or application-dependent safety issues. Hence, there is a need for coordination of ECA rule-based systems in order to ensure safety objectives. We propose a tool-supported method for verifying and controlling the correct interactions of rules, relying on formal models related to reactive systems, and Discrete Controller Synthesis (DCS) to generate correct rule controllers.

1 Coordination problems in ECA rules

Event-Condition-Action (ECA) rules are defined [13] as a set of rules where each of them *'autonomously reacts to actively or passively detected simple or complex events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happens and the condition is true'.* The form of the rule is: **ON** *Event* **IF** *Condition* **DO** *Action.* Some characteristics are that:

- a rule is activated only by events;
- its execution is autonomous and independent of other rules in the system;
- it implements a reaction to the incoming event;
- it contains a guarding condition to execute such actions.

Research on ECA rules is often related to active database management systems (ADBMS) [3,14], where events represent modifications produced in the database, and ECA rules are used to control the integrity. But they have also been used in different control environments [8] or adaptation frameworks [10], and there are many different implementations of ECA rule-based systems. ECA rules are a language derived from practice, and not constructed from a formal definition in the beginning. This is comparable to other cases like StateCharts and its multiple variants and implementations (e.g. in UML, or in the StateMate tools of iLogix [1], or in Stateflow/Simulink), or like works on the verification of implementation languages like Java or C. Therefore, there is no unique reference or formal semantics, and they can not be submitted to formal analysis as such.

Coordination problems The nature of ECA rule-based systems shows different problems in their execution, the most extended [18] being as follows.

Redundancy means that there are two (or more) rules in the system whose functionality is replicated. This can happen in large rule systems where rules are written by different persons. An example in a smart home automated system is to have two similar rules: one detects the presence of a person in a room and, if temperature is lower than 15 degrees, then turns on room heaters. The other rule does the same, but also closes the room door. This can be described in ECA syntax as follows (a concrete grammar is described later in Section 3):

```
ON presence IF (temperature_get < 15) DO heater_on
ON presence IF (temperature_get < 15) DO heater_on, door_close
```

This represents an overload in the rules system in the best of cases, and an undesired repetitive activation of orders on environment devices.

Inconsistency occurs when contradictory actions are sent to devices. This can occur if multiple rules are activated at the same time, and their execution order may render different final states in the system. An example is: upon the presence of a person in the room lights are activated, and TV will also be activated. A third rule will turn off the lights then the TV is turned on.

```
ON presence IF true DO lights_on
ON presence IF true DO TV_on
ON TV_light IF TV_on DO lights_off
```

Depending on the order of execution of rules, the final state of the system will be different. If rules 2 and 3 are activated before rule 1 is executed then the final state of lights will be different than executing rule 1 before rule 3. So the result of the execution of these rules is not predictable.

Circularity occurs when rules get activated continuously without reaching a stable system state that makes them finish their execution. For example, the first two rules will try to change the second light to a state different from the first light ; the third and fourth rules will try to maintain both in the same state.

```
ON light1_change IF light1_on DO light2_off
ON light1_change IF light1_off DO light2_on
ON light2_change IF light2_on DO light1_on
ON light2_change IF light2_off DO light1_off
```

Application-specific issues can be considered additionally in an environment. An example is ordering to open a windows and to turn on the room heaters. It can be considered as a contradiction by the user. In order to know which actions are contradictory, specific information must be provided about the environment. In this paper we will consider that multiple actions sent to the same device are contradictory. Only one action can be requested to every device at every instant.

Coordinating ECA rules is therefore necessary in order to enforce safety properties. One of the problems of ECA rules is that they are considered to be executed independently or autonomously. This means that possible interactions between rules and their effects are not controlled. In contrast, synchronous reactive languages, used to design and program control systems, provide some characteristics, such as determinism and verifiability [9]. This is useful for the safe execution of control systems. The objective of this work is to provide validation of the ECA rule system before and during the execution of the system, by relating them to synchronous languages. Here, safety is meant for the control system and people in the environment controlled by this control system. The system should not go into undesired states, and controlled devices are considered part this state.

Our approach proposed in this paper consists of a model transformation from an ECA rules description to a synchronous programming language, which will be used to validate the set of rules. ECA rule systems are validated, detecting the described issues. Our proposal constitutes a formal semantics of ECA rules, covering variants of rule engines, and defined concretely by translation into a formally defined language, for which formal tools are available. Rules execution is also controlled and coordinated to avoid the described problems at run-time. We will concentrate on small or home environment as target systems, although our results are generic enough to be applied in any ECA rule-based system.

The Heptagon/BZR programming language [2] is used here because of its capability to express behavioral invariants in the system in the form of contracts, which allows verifying the application by model checking as well as controlling or coordinating the execution according to the described invariants.

In the following, Section 3 formalizes the ECA rules used in this paper. Section 4 shows their translation to a synchronous program, on which Section 5 shows how we perform verification and control. Section 6 concludes.

2 State of the art

2.1 ECA rule based control systems and their validation

ECA rule systems are widely used to control the environment as well as to control reconfiguration of software systems. Here are described the closest proposals to our approach and ECA rule-based systems verification and validation. In [10], an adaptation framework is proposed. It detects the state of the system in the form of events. When these events are detected the associated rules can be applied. These rules will perform the required actions according to the detected state, to adapt the behavior of the system to the changes of the environment. In [16,17], a method is proposed to design applications with reconfiguration capabilities. At design time, invariants can be described for every state and transitions between states. These invariants are used in the design of Petri Nets representing the desired behavior of the application. Designed Petri Nets can be used to check the previously defined invariants and to create prototypes of the system. The system is supposed to be safe by design, if the design is correctly translated to the implementation. No control is performed at run-time about the specified invariants. A mixture of rule based system and utility functions is proposed in [5]. Rules are mainly used to change the priorities for the utility functions when a state change is detected. The number of possible available configurations can grow exponentially, so calculation of the utility functions at run-time is costly.

The following work cover basic aspects in verification and validation of ECA rule systems. In [15], a way to validate a set of rules in a knowledge based systems is proposed. It defines different types of rules to create a *rule net* consisting of chained rules, which explicitly invoke other rules. In the rule net, it checks if different paths contain *inconsistencies* according to the constraints defined in the system or other rules. In [11], an infrastructure is described to detect and solve static (compilation time) and dynamic (execution time) conflicts for a framework of WS-ECA. This framework is based in the use of ECA rules for Web Services. The existence of distributed devices with their own rules may lead to conflicting rules. No implementation is described for this infrastructure.

In [18], a more complete proposal is described to verify an ECA rule based system. It starts formalizing the system to be able to define the problems of *redundancy*, *inconsistency* and *circularity*. Three levels are described regarding the verification and validation of these problems. Level 1 refers only to rule set level, where no information about run-time execution is considered. Level 2 takes into account direct results of the execution of actions on the environment. This means actions that will directly provoke new events activating rules. Level 3 takes into account all the possible responses of the environment, which cannot be previously known because they are completely random or unpredictable. Certain problems can only be verified at some levels because of the required information to perform such verifications. In [4], a method is provided to verify ECA rule systems with formal methods, transforming the ECA rules set into a set of different kinds of automata for every part of the process, and using the automata verification tool Uppaal. This verification is limited to performing model checking of timed automata and their correspondence to the provided ECA rule set.

Every ECA rule-based system implementation imposes different execution semantics. These semantics can vary from parallel synchronized execution of rules to execution in depth first and discard of previously activated rules. So the result of the execution of a rule set differs depending on the execution policy of the implementation. All the proposals described above are centered in one kind of ECA rule system execution policy, or they do not take into account that results depend on the execution policy used by every different implementations. In this paper, we propose a solution that takes into account the desired execution policy of the target implementation to verify an ECA rules set.

2.2 Synchronous reactive programming and Heptagon/BZR

Reactive systems are interactive systems that constantly communicate with their environment taking into account the timing needs of this environment [9]. These systems will work reacting to received events or by sampling incoming signals. Synchronous programming languages allow programming of reactive systems using automata, where reactions will correspond to the automata transitions. Computations and transitions performed by composed automata are considered to occur in parallel at the same time instant. This intrinsic synchronism makes it easier to preserve the determinism, and allows these programming languages to be based on sound formal semantics. Thus, these languages are provided with tools for the verification (e.g., by automated test or model-checking) or control (e.g., by controller synthesis) of programs.

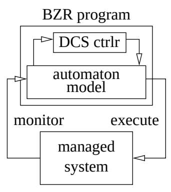
Heptagon/BZR [2] is a synchronous dataflow programming language with support for equations and automata. This language also provides a contract mechanism allowing the use of discrete controller synthesis (DCS) within the compilation, using the Sigali synthesis tool [12]. The discrete controller synthesis method is based in partitioning input variables into controllable and uncontrollable ones. For a given objective, such as staying in a subset of states, its DCS algorithm will automatically compute, by symbolic exploration of the state space, the constraint on controllable variables, so that the behavior satisfies the objective, whatever be the values of the inputs from the environment. Figure 1(a) represents the control loop. The automata-based program is in charge of controlling the environment. The behavior of the automata is constrained by the synthesized controller, which is in charge of maintaining the system in the desired subset of states. The main elements of a Heptagon/BZR program are:

– Nodes: blocks of equations or automata with input and output signals
– Equations: determining the value of node outputs. A set of equations in parallel are separated by semicolons.
– Automata: mode automata using states, input and output signals.
– Contracts: describing invariants to be enforced by control at execution.

The compiler generates executable code in C or Java. Figure 1(b) shows an example of Heptagon/BZR code. It contains a `delay` node, with an automaton which makes use of a controlled variable to delay the emission of a received signal. A `main` node makes use of this automaton. It includes a contract to enforce that both signals are not emitted at the same time. Heptagon/BZR makes use of Sigali at compilation time to synthesize the needed controller that will provide the correct value for every controlled variable (`c1` and `c2`), and hence forcing the delay of one of the signals. The control variable (`c`) in Figure 1(b) determines when the signal has to be delayed. When a `new_sig` is received, depending on the value of the controlled variable, the `new_sig` value is emitted (staying in the `Idle` state) or delayed (performing a transition to the `Waiting` state) until the controlled variable indicates that it can be released. The value of the `out` variable is described in function of `new_sig` and the controlled variable. Automata transitions will be effective in the next step of its execution. To avoid delays in the desired automaton output, the value of the `out` variable is described in function of `new_sig` and the controlled variable. This allows emitting the desired value in the same execution step.

Heptagon/BZR has been used by some work in smart home / environment context, to design safe control systems. In [19], Heptagon/BZR is proposed for the autonomic management of small environments. The behavior of devices is represented using automata and control objectives are described as *contracts*. For instance, it can avoid the request to turn on a device if it can generate an energy consumption higher than the specified. In [7], a similar approach is proposed

(a)



(b)

```
node delay
  (new_sig: bool; c:bool)
returns (out: bool)
let automaton
  state Idle
   do out=new_sig & c
   until new_sig & not c
                 then Waiting
    | new_sig & c then Idle
  state Waiting
   do out=c
   until c then Idle
end tel
```

```
node main
  (signal1, signal2: bool)
returns (d1, d2:bool)
contract
enforce not (d1 & d2)
with (c1,c2:bool)
let
 d1 = delay(signal1, c1);
 d2 = delay(signal2, c2);
tel
```

Fig. 1. Discrete control: (a) control model; (b) controlled automaton example.

to provide safe environment for disabled people. However, none of these works featured ECA rules as a high-level description language. In another domain, the coordination of multiple autonomic loops in adaptive computing systems has been approached with a discrete control approach [6].

3 Modeling ECA rules

Here we describe a formalization of an ECA rule-based system to be able to translate it into a Heptagon/BZR program. The ECA rule-based system is assumed to be connected to the physical world through devices that may work as sensors or actuators. The control system loop is generated providing devices information about the environment to the rules and then sending the result of rules again to devices. A rule based system $S = (R, E, D)$ is composed of a set of rules R, a set of events E and a set of devices D. We consider that rules, events, devices and signals are identified by unique names, taken in a name set N. Thus, events are names, i.e., $E \subset N$. Devices $d \in D$ are a virtual representation of physical devices in the system as sensors and actuators. A device $d = (n, I, O)$, named n, is composed of a set of input signals $I \subset N$ and a set of output signals $O \subset N$. In the following, we will denote by $\text{Expr}(O)$ the set of Boolean expressions defined on the set of output signals O. The function $\text{EventExpr} \in E \to \text{Expr}(O)$ maps events to boolean expressions based on output signals received from devices. The event $e \in E$ is activated whenever the expression $\text{EventExpr}(e)$ is true. Rules $r \in R$ are defined by a tuple $r = (n, e, c, A)$, where $n \in N$ is the name of the rule, $e \in E$ the activating event, $c \in \text{Expr}(O)$ the condition, and $A \subset I$ a set of actions to perform. The condition is a boolean expression based on the output of devices. If the event occurs and the condition is true, the corresponding actions will be performed.

Figure 2 shows the concrete grammar used by the implemented tool to translate a ECA rule-based system into a Heptagon/BZR program. The descriptions contains lists of events, rules and devices. Events can be internal (generated by rules as an action, indicated by the `INTERNAL` term) or be generated if the described expression becomes true (when the term `IF` is used in its description).

```
<ECA-system> ::= <events_list> <rules_list> <devices_list>
<event_lists> ::= <event> | <event> <event_lists>
<event> ::= EVENT <event_name> IF <expression> | EVENT <event_name> IS INTERNAL
<rules_list> ::= <rule> | <rule> <rule_list>
<rule> ::= ON <event_name> IF <condition> DO <action_list>
<action_list> ::= <action_name> | <action_name>, <action_list>
<device_list> ::= <device> | <device> <device_list>
<device> ::= DEVICE <device_name> [ SIMULTANEOUS ( DISCARD | DELAY )]
                        [INPUTS (<input_list>)] [OUTPUTS (<outputs_list>)]
```

Fig. 2. ECA rule-based system description grammar

Rules contain the event name that activates them (preceded by term `ON`), a boolean expression to determine if certain conditions apply (preceded by term `IF`), and the list of actions that have to be performed if event and condition are true (preceded by term `DO`). The device contains the device name, a specification of the policy to be used in the device when multiple simultaneous actions are sent to this device, a list of inputs and a list of outputs of the device. The term `SIMULTANEOUS` allows specifying the policy used when multiple signals are sent to the same device. `DISCARD` allows discarding all the signals but one. `DELAY` allows delaying all the signals but one. Delayed signals will be emitted later, in following executions of the controller, in order of priority. The priority in all cases is given by the order in which input and output signals are declared. Inputs and outputs are optional in the description of the device. At least one should be indicated. If the device policy is not specified, the default value is `DISCARD`.

Figure 3 shows an example including rules from Section 1: `light1_change` will be activated if light1 is turned on or off. The needed events and devices are also described: `presence` and `temperature` sensors are used, with only outputs.

```
EVENT presence:BOOL IF presence_get
EVENT TV_lights:BOOL IF TV_on
EVENT light1_change IF light1_on or light1_off
EVENT light2_change IF light2_on or light2_off

ON presence IF (temperature_get < 15) DO heater_on
ON presence IF (temperature_get < 15) DO heater_on, door_close
ON presence IF true DO light1_on
ON presence IF true DO TV_on
ON TV_lights IF TV_on DO light1_off
ON light1_change IF light1_on DO light2_off
ON light1_change IF light1_off DO light2_on
ON light2_change IF light2_on DO light1_on
ON light2_change IF light2_off DO light1_off

DEVICE presence OUTPUTS (get:BOOL)
DEVICE temperature OUTPUTS (get:BOOL)
DEVICE light1 SIMULTANEOUS DISCARD INPUTS (on:BOOL, off:BOOL) OUTPUTS (on:BOOL, off:BOOL)
DEVICE light2 SIMULTANEOUS DISCARD INPUTS (on:BOOL, off:BOOL) OUTPUTS (on:BOOL, off:BOOL)
DEVICE TV SIMULTANEOUS DELAY INPUTS (on:BOOL, off:BOOL) OUTPUTS (on:BOOL, off:BOOL)
```

Fig. 3. ECA rule-based system description example

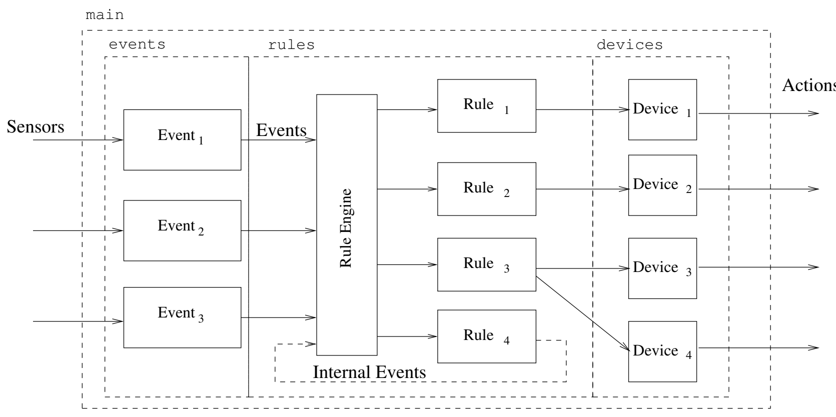


Fig. 4. Heptagon/BZR code detailed model

4 Transformation to synchronous language

We propose a transformation to Heptagon/BZR code from the described ECA rule-based system. As shown in Figure 4, the generated Heptagon/BZR program will be defined in the body of a `main` node. This node is structured into three sub-nodes named `events`, `rules` and `devices`. The `main` node will receive all the sensor signals from the devices. The `events` node will use them to determine if events occur according to their definition. Events and devices signals are then passed to the `rules` node, where rules are activated according to their firing event and condition. Actions corresponding to activated rules are then used in the `devices` node, to be processed before being sent to devices, according to the corresponding device policy. The Rule Engine contains the execution policy to be simulated, determining the behavior of the rule-based system. Its output is the activated internal events, according to the specified policy.

4.1 Code transformations

Figure 5 shows the skeleton for the code generated from a textual representation of an ECA rule-based system according to the grammar of Figure 2, with terms that are defined in the following. The order of the program is that nodes are defined before being used as sub-nodes in later nodes. The `main` node (e) invokes the sub-nodes `events` (defined in (a)), `rules` (c) and `devices` (d) nodes. The `rules` node (c) invokes the sub-node `rule_engine` (b), which models the execution policy, as described in 4.2. Event detectors, rules and devices are translated as lists of equations inside of the indicated nodes.

The developed tool implements several transformation schemes to translate events, rules and devices descriptions into Heptagon/BZR code, which are described for every element. We consider an ECA system $S = (R, E, D)$. The function *name* is used to define Heptagon/BZR variable names from the set

(a)

```
node events(<devices_ outputs>)
  returns (<event_ names>)
let <event_ detections>
tel
```

(b)

```
node rule_engine(<event_ names>)
  returns (<final_ event_ names>)
  <execution_ policy_ contracts>
let <execution_ policy>
tel
```

(c)

```
node rules(<event_ names>,
           <devices_ outputs>)
  returns (<request_ devices_ inputs>)
var <temp_ event_ names>:bool;
  <rule_ names>:bool;
let (<temp_ event_ names>)
      = rule_engine (<event_ names>) ;
  <rules_ activations> ;
  <signals_ activation>
tel
```

(d)

```
node devices(<request_ devices_ inputs>)
  returns (<final_ devices_ inputs>)
  contract
    enforce <device_ policy_ contracts>
    with <device_ policy_ controllables>
let
  <devices_ policies>
tel
```

(e)

```
node main(<devices_ outputs>)
returns (<final_ devices_ inputs>)
var <event_ names> ;
    <request_ devices_ inputs> ;
let
  (<event_ names>)
      = events(<devices_ outputs>) ;
  (<request_ devices_ inputs>)
      = rules(<event_ names>,
              <devices_ outputs>);
  (<devices_ inputs>)
      = devices(<request_ devices_ inputs>);
tel
```

Fig. 5. Program structure in Heptagon/BZR

N of rules, events and devices names. We consider for the sake of clarity and simplicity that Boolean expressions in the ECA language corresponds to the Heptagon/BZR ones, and thus they can be used as they are.

Figure 6 gives transformations to Heptagon/BZR for several terms. The *<device_ outputs>* list is generated over all the devices and their corresponding outputs. Similarly, we can generate *<device_ inputs>*. The *<final_ event_ names>* list is build similarly to *<event_ names>*, but adding the "`final_`" prefix, to differentiate inputs and outputs of the `rule_engine` node. Other lists are created similarly, *<request_ devices_ inputs>* and *<temp_ event_ names>*, but adding "`req_`" or "`temp_`" prefixes respectively to the names in their counterpart lists.

The *<rules_ activation>* defines the Boolean corresponding to activation of a rule if the corresponding event and its condition are true at the same time. Then, *<signals_ activation>* allows activating every device input if it has been

$$<\!devices\_\ outputs\!> = \{\ \mathrm{name}(n)\_\mathrm{name}(o) \mid (n, I, O) \in D, o \in O\ \}$$

$$<\!event\_\ names\!> = \{\ \mathrm{name}(e) \mid e \in E\ \}$$

$$<\!event\_\ detection\!> = \{\ \mathrm{name}(e)\ \texttt{=}\ \mathrm{EventExpr}(e) \mid e \in E\ \}$$

$$<\!rule\_\ names\!> = \{\ \mathrm{name}(n) \mid (n, e, c, A) \in R\ \}$$

$$<\!rules\_\ activation\!> = \{\ \mathrm{name}(n)\ \texttt{=}\ \texttt{final\_}\mathrm{name}(e)\ \texttt{\&}\ c \mid (n, e, c, A) \in R\ \}$$

$$<\!signals\_\ activation\!> = \{\ \mathrm{name}(a)\ \texttt{=}\ \bigvee_{(n,e,c,A)\in R \mid a \in A} \mathrm{name}(n) \mid \exists (n, I, O) \in D, a \in I\ \}$$

Fig. 6. Transformations for various sets

$<\!device\_policy\_contracts\!> = \texttt{true}$

$<\!device\_policy\_controllables\!> = \emptyset$

$$<\!device\_policy\!> = \Big\{\ \texttt{final\_name}(n)\texttt{\_name}(o)\ \texttt{=}\ \texttt{req\_name}(n)\texttt{\_name}(o)\ \texttt{\&}\ \texttt{not} \bigvee_{o' \in O, o \prec o'} \texttt{final\_name}(n)\texttt{\_name}(o')\ \mid (n, I, O) \in D, o \in O \Big\}$$

Fig. 7. Transformation for devices (discarding policy)

$<\!device\_policy\_contracts\!> =$

$$\bigwedge_{(n,I,O)\in D, o_1, o_2 \in O, o_1 \neq o_2} \texttt{not (final\_name}(n)\texttt{\_name}(o_1)\ \texttt{\&}\ \texttt{final\_name}(n)\texttt{\_name}(o_2)\ \texttt{)}$$

$<\!device\_policy\_controllables\!> = \{\ \mathrm{name}(n)\_\ \mathrm{name}(o)\texttt{\_c} \mid (n, I, O) \in D, o \in O\ \}$

$<\!device\_policy\!> = \{\texttt{final\_}m\texttt{= delay(req\_}m\texttt{,}m\texttt{\_c)}\ |(n, I, O) \in D, o \in O, m = \mathrm{name}(n)\_\ \mathrm{name}(o)\}$

Fig. 8. Transformation for devices (delaying policy)

requested by any of the rules. Given that multiple rules could activate the same signal, a disjunction is used to fuse them.

Transformation for devices differs depending on the specified execution policy to apply on signals sent to the device. Actions from rules correspond to devices inputs. The two specified strategies are *discarding* or *delaying* contradictory signals sent to a device. Equations are used in the first case, shown in Figure 7, to discard contradictory signals: the total order $\prec$ is used to give priorities. Once a signal with higher priority has already been activated, the rest are discarded. The $<$*device_policy*$>$ equations are composed for every device in the system. In the second case, shown in Figures 8, a contract is used to delay signals. Only one of them is sent to the device. The `delay` automaton from Figure 1 will store the input signal until it can be released and sent to the device. The generated controller will be in charge of selecting the right values for the controlled variables to send only one signal at a time and delay the others.

```
node rule_engine(<event_names>) returns (<final_event_names>)
  contract enforce $\bigwedge_{e_1,e_2 \in E, e_1 \neq e_2}$ not (final_name($e_1$) & final_name($e_2$))
    with { name($e$)_c | $e \in E$ }
let { final_name($e$) = delay(name($e$), name($e$)_c) | $e \in E$ }
tel
```

Fig. 9. Delayed execution model

```
node rule_engine(<event_names>) returns (<final_event_names>)
let { final_name(e) = name(e) | e ∈ E }
tel
```

Fig. 10. Parallel execution model

4.2 Execution models

As said before, the transformation is designed to support different execution policies in ECA rule-based systems. We currently support transformations for *parallel* and *delayed* execution. The code generation for the `rule_engine` node will differ for every case. For the delaying of events, as shown in Figure 9, the `delay` automaton is used as for the device node code generation. Only one event will be sent to the `rules` node for every execution of the controller. For the parallel execution model, as shown in Figure 10, all the events are allowed to be activated in the same execution step without restrictions. Other execution policies can be added in this node to simulate any ECA execution model.

5 ECA rule set verification and control

The previous transformation makes it possible to handle the problems described in Section 1 and to validate, verify or control the execution of the ECA rule set. Different kinds of verifications can be performed on the ECA rule based system, depending on the available information, as described in [18]. Verifications can be static (performed at compilation time) or dynamic (performed at run-time). They can also be classified as generic ECA rule verifications or domain specific issues that can verified. Diagnosis information about the detection of static errors can be extracted in the form of rule identifiers or line position, as well as identifiers of involved signals, by instrumenting the generated code.

5.1 Verifications at compilation time

The first verification to be performed is the detection of *syntax errors*. The use of undeclared events or unavailable device actions are examples of such errors in the declaration of rules. Syntax errors are easily detected by any compiler or interpreter when recognizing the ECA rules source code.

Redundancy of rules is detected when the condition and actions of one rule represent a subset of conditions and actions of the other rule. This means that having two rules $r_1 = (n_1, e_1, c_1, A_1)$ and $r_2 = (n_2, e_2, c_2, A_2)$ where $e_1 = e_2$, $c_1 \Rightarrow c_2$ and $A_1 \subseteq A_2$.

Redundant rules are not directly detected by Heptagon/BZR. Duplicated rules will be compiled and executed at run-time without problems. Rule actions will be activated using the `or` operator, so the results will not result in redundancy. Here is a simple example of redundancy:

```
ON presence IF true DO Tv_on
ON presence IF true DO Tv_on
```

This example generates the following Heptagon/BZR code:

```
rule6 = (presence) & (true);
rule7 = (presence) & (true);
req_tv_on = rule6 or rule7;
```

The Sigali tool is used to solve this problem. The redundancy can be better checked using this tool. The capability of working with equations [12] in Sigali is used to detect the situation where the condition of one rule is included in the condition of another rule, thus making them redundant. For Sigali this means that the solutions for the equation c_1 = true is a subset of the solutions of c_2 = true. Sigali code performing this check is generated for every couple of rules that fulfills the following conditions:

– Rules are activated by the same event
– The set of actions of one rule is a subset of actions of the other one.
– The set of variables used in the conditions of both rules are not disjoint.

This filtering also helps reducing the quantity of operations in Sigali.

Inconsistency is also detected at compilation time. It can be defined as the result of contradictory actions, provided as result of activation of different rules. It can be formalized as having two rules $r_1 = (n_1, e_1, c_1, A_1)$ and $r_2 = (n_2, e_2, c_2, A_2)$ where $e_1 = e_2$ and $c_1 = c_2$, but A_1 and A_2 are contradictory. As previously defined in Section 1, we consider as contradictory actions sending more than one signal to the same device at the same step of the rule system. Due to having the same event and condition to be activated, they will always be activated together, generating contradictory actions, even if not executed simultaneously. This verification is performed by compiling the corresponding Heptagon/BZR contracts on the device node, but not discarding or delaying signals. The Sigali tool will detect inconsistencies failing to generate the controller, indicating that the program it not executable regarding the contracts.

Circularity generated by internal events can be detected by Heptagon/BZR, for the case of the parallel execution model, as a causality error at compilation time. The following code generates a circularity problem:

```
ON internal1 IF true DO internal2
ON internal2 IF true DO internal3
ON internal3 IF true DO internal1
```

A dependency cycle occurs in the definition of rules in a way that these rules are always activating themselves. The circularity is detected independently of the number of involved events. The generated code is as follows:

```
rule0 = (internal3) & (true);
rule1 = (internal2) & (true);
rule2 = (internal1) & (true);
internal2 = rule2;
internal1 = rule0;
internal3 = rule1;
```

The dependency cycle will be detected by Heptagon/BZR as a causality error. Detection is conservative, so even if conditions in rules may avoid this dependency for some values, it is detected as a possible violation.

5.2 Control at run-time

It can not be foreseen at compilation time if two different rules with different events and contradictory actions will be activated at the same time instant at run-time. Coordination or control techniques have to applied in that case. Heptagon/BZR is designed to provide this kind of run-time control.

Inconsistency at run-time is controlled with the already described code generated for the devices. Contradictory signals can be discarded or delayed depending on the chosen policy. This provides more control on the execution of rules than avoiding the rules. In case that one rule has more than one action, only the inconsistent actions will be discarded or delayed, allowing the rest of actions to be performed. In this case, inconsistency is not only detected, as in the compilation time, but solved using the order priority to discard or delay signals.

Circularity is detected using automata and model checking capabilities of Heptagon/BZR, for two different circular behaviors. Automata are used to represent the behavior of devices. The actual state of the device and possible transitions are represented. Automata are designed to send the required signals to a device only if it represents a change of state in the signal. In this case the action of the corresponding rule is avoided, interrupting an endless chained execution of rules. Figure 11(a) represents an automaton for a devices with two states. The automaton receives requests to change the state. The outputs are the actual state of the device (`st_On` or `st_Off` become true or false respectively), and the signal to change the state of the device (`On` or `Off`). These signals are only emitted if the devices is not already in this state.

The other possibility is that rules continuously modify the state of a device. This represents an *oscillation* in the state of the device. An observer automaton is used to detect undesired oscillations. Oscillations can be considered undesired if they have not been generated by external events, this means not directly generated by the execution of rules. For instance, user actions are considered external events, while a light turned on by a rule is not an external event. In

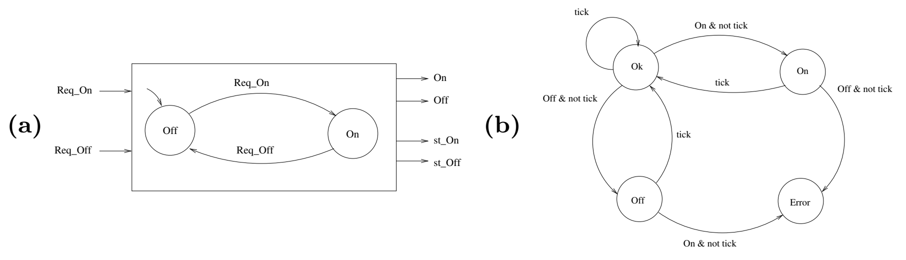


Fig. 11. **(a)** Two states device behavior **(b)** Observer for oscillation detection.

Figure 11(b) an automaton is represented. This is an observer automaton able to detect if the state of a two states devices is oscillating. A `tick` signal is used to represent the activation of an external event. In case that the oscillation is produced by the presence of external events, then it is considered as a desired oscillation. In case of the absence of the `tick` signal, the oscillation is considered as undesired. To avoid this situation, besides the automaton, a contract is included in the Heptagon/BZR code. This contract indicates that the `Error` state must not be reached. The use DCS in Heptagon/BZR can control the avoidance of this oscillation at run-time using the delaying automata.

```
node device1(on, off: bool; c:bool) returns (st_on, st_off: bool; power:int)
let
  automaton
    state Off do st_off = true; st_on = false; power = 0;
      until on & c then On
    state On do st_on = true; st_off = false; power = CONSUMPTION;
      until off & c then Off
  end
tel

node devices(req_d1_on, req_d1_off: bool; req_d2_on, req_d2_off: bool)
    returns (st1_on, st1_off: bool; ... ; power:int)
  contract
    enforce (power ≤ LIMIT)
    with (dev1_c, dev2_c:bool)
  var power1, power2:bool
let
  power = power1 + power2;
  (st1_on, st1_off, power1) = device1(req_d1_on, req_d1_off, dev1_c);
  (st2_on, st2_off, power2) = device1(req_d2_on, req_d2_off, dev2_c);
  ...
tel
```

Fig. 12. Application-specific scenario behavior control

Application-specific issues can be considered additionally to the above generic ECA rule-based system issues, for specific scenarios with specific requirements. These requirements can also be expressed in Heptagon/BZR, to provide more control on the ECA rule set execution. These requirements are also difficult to express in ECA rules, because of the lack of language support to describe them, but their violation would cause inconsistencies during the execution.

Additional information is required about the environment to be able to perform the specific control actions on the environment, following the approach in [19]. An example of such application-specific scenario requirements would be to forbid an energy power consumption higher than a given threshold `LIMIT`. A model should be provided in the form of automata representing the devices behaviors. Figure 12 shows an automaton called `device1`, representing the behavior of one device type, with states `On` and `Off`, associated with consumption levels, here valued for the example at 0 and `CONSUMPTION`. A node called `devices`, shown here only partially, describes a composite system with two such devices, each represented by an instantiation of the former node. The overall power consumption is defined as the sum of local power consumptions. A contract is then declaratively specified, as defined in Section 2.2, such that the global power is lower than the given limit. DCS is applied during the compilation of this program, to automatically solve the control problem. The generated controller will avoid entering the `On` state of a device if it makes the total power consumption to overcome the threshold. An ECA rule would be able to detect the situation when it is already occurring, whereas DCS performs an analysis predicting possible problems, and the controller generated by Heptagon/BZR will directly avoid reaching the undesired state.

6 Conclusions

We propose a novel method for coordination in ECA rule-based systems, by verification and control based on behavioral models, in order to avoid problems of redundancy, inconsistency, and circularity, as well as application-specific issues. This method is based on the use of model checking and a control technique (DCS) which provides safe control during the execution of the system. Verifications are performed at compilation time with simple transformations and model checking, ensuring that the desired system defined invariants apply. So, for the execution of the ECA rule set, the generated controller ensures that the desired properties will always apply. Our method also takes into account different possible execution models for the ECA rule-based system. These execution models can be modeled, ensuring that the final implementation of the system is correctly verified. Our work offers users with a combination of a high-level ECA rules language with the compiler and formal tool support of Heptagon/BZR, which can seen in both ways: formal support of ECA rules, and user-friendly language above Heptagon/BZR.

We are presently working on the integration of the generated controller, using the executable code in C or Java, in an experimental embedded platform for small or home environments where users could introduce ECA rules in the system to control home sensors and actuators using the automatically generated safe controller. We are also working on the possibility to automatically provide device models representing their behavior. This allows specifying, at the same level as ECA rules rather than in Heptagon/BZR, safety properties for application-specific scenarios as the one described in last section. Other perspective involves modular compilation and DCS, which can improve scalability of the approach, as well as distribution of the executable code, to design distributed controllers.

References

1. J.-R. Beauvais, E. Rutten, T. Gautier, R. Houdebine, P. Le Guernic, and Y.-M. Tang. Modeling statecharts and activitycharts as signal equations. *ACM Transactions on Software Engineering and Methodology*, 10(4):397–451, October 2001.
2. G. Delaval, É. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, December 2013.

3. K. Dittrich, S. Gatziu, and A. Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In *2nd Workshop on Rules in Databases*, pages 1–15, Athens, Greece, 1995.
4. A. Ericsson. *Enabling Tool Support for Formal Analysis of ECA Rules.* Phd thesis, University of Skövde, 2009.
5. F. Fleurey and A. Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS '09 Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, Berlin, 2009.
6. Soguy Mak-Karé Gueye, Noël de Palma, and Eric Rutten. Coordination control of component-based autonomic administration loops. In *Proceedings of the 15th International Conference on Coordination Models and Languages, COORDINATION,* 3-6 June, Florence, Italy, 2013.
7. S. Guillet, B. Bouchard, and A. Bouzouane. Correct by construction security approach to design fault tolerant smart homes for disabled people. *Procedia Computer Science*, 21(0):257–264, January 2013.
8. L. Gürgen, A. Cherbal, R. Sharrock, and S. Honiden. Autonomic management of heterogeneous sensing devices with ECA rules. In *2011 IEEE International Conference on Communications Workshops (ICC)*, pages 1 – 5, 2011.
9. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Springer, 2010.
10. J. Keeney and V. Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 3–14, 2003.
11. W. Lee, S. Lee, and K. Lee. Conflict Detection and Resolution method in WS-ECA framework. *The 9th International Conference on Advanced Communication Technology*, pages 786–791, February 2007.
12. H. Marchand, P. Bournai, M. Borgne, and P. Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, pages 1–26, 2000.
13. A. Paschke. ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. *Computer Research Repository*, abs/cs/061, 2006.
14. C. Turker and M. Gertz. Semantic Integrity Support in SQL-99 and Commercial Object- Relational Database Management Systems. *The International Journal on Very Large Data Bases*, 10(4):241–269, 2001.
15. J.P. Yoon. Techniques for data and rule validation in knowledge based systems. *Proceedings of the Fourth Annual Conference on Computer Assurance, 'Systems Integrity, Software Safety and Process Security*, pages 62–70, 1989.
16. J. Zhang and B. Cheng. Specifying adaptation semantics. *ACM SIGSOFT Software Engineering Notes*, pages 1–7, 2005.
17. J. Zhang and B. Cheng. Model-based development of dynamically adaptive software. *Proc. of the 28th international conference on Software Engineering*, 2006.
18. J. Zhang, J. Moyne, and D. Tilbury. Verification of ECA rule based management and control systems. *IEEE Int. Conf. Automation Science and Engineering*, 2008.
19. M. Zhao, G. Privat, E. Rutten, and H. Alla. Discrete Control for the Internet of Things and Smart Environments. In *Int. Workshop on Feedback Computing*, 2013.