Symbolic Verification of ECA Rules

Xiaoqing Jin, Yousra Lembachar, and Gianfranco Ciardo

Department of Computer Science and Engineering,
University of California, Riverside
{jinx,ylemb001,ciardo}@cs.ucr.edu

Abstract. Event-condition-action (ECA) rules specify a decision making process and are widely used in reactive systems and active database systems. Applying formal verification techniques to guarantee properties of the designed ECA rules is essential to help the error-prone procedure of collecting and translating expert knowledge. The nondeterministic and concurrent semantics of ECA rule execution enhance expressiveness but hinder analysis and verification. We then propose an approach to analyze the dynamic behavior of a set of ECA rules, by first translating them into an extended Petri net, then studying two fundamental correctness properties: *termination* and *confluence*. Our experimental results show that the symbolic algorithms we present greatly improve scalability.

Keywords: ECA rules, termination, confluence, verification

1 Introduction

Event-condition-action (ECA) [12] rules are expressive enough to describe complex events and reactions. Thus, this event-driven formalism is widely used to specify complex systems [1, 3], e.g., for industrial-scale management, and to improve efficiency when coupled with technologies such as embedded systems and sensor networks. Analogously, active DBMSs enhance security and semantic integrity of traditional DBMSs using ECA rules; these are now found in most enterprise DBMSs and academic prototypes thanks to the SQL3 standard [9]. ECA rules are used to specify a system's response to events, and are written in the format "on the occurrence of a set of events, if certain conditions hold, perform these actions". However, for systems with many components and complex behavior, it may be difficult to correctly specify these rules.

Termination, guaranteeing that the system does not remain "busy" internally forever without responding to external events, and *confluence*, ensuring that any possible interleaving of a set of triggered rules yields the same final result, are fundamental correctness properties. While termination has been studied extensively and many algorithms have been proposed to verify it, confluence is particularly challenging due to a potentially large number of rule interleavings [2].

Researchers began studying these properties for active databases in the early 90's [2, 4, 11, 14], by transforming ECA rules into some form of graph and applying various static analysis techniques on it to verify properties. These approaches

based on a static methodology worked well to detect redundancy, inconsistency, incompleteness, and circularity. However, since static approaches may not explore the whole state space, they could easily miss some errors. Also, they could find scenarios that did not actually result in errors due to the found error states may not be reachable. Moreover, they had poor support to provide concrete counterexamples and analyze ECA rules with priorities. [2] looked for cycles in the rule-triggering graph to disprove termination, but the cycle-triggering conditions may be unsatisfiable. [4] improved this work with an activation graph describing when rules are activated; while its analysis detects termination where previous work failed, it may still report false positives when rules have priorities. [5] proposed an algebraic approach emphasizing the condition portion of rules, but did not consider priorities. Other researchers [11, 14] chose to translate ECA rules into a Petri Net (PN), whose nondeterministic interleaving execution semantics naturally model unforeseen interactions between rule executions. However, as the analysis of the set of ECA rules was through structural PN techniques based on the incidence matrix of the net, false positives were again possible.

To overcome these limitations, dynamic analysis approaches using model checking tools such as SMV [15] and SPIN [6] have been proposed to verify termination. While closer to our work, these approaches require manually transforming ECA rules into an input script, assume a priori bounds for all variables, provide no support for priorities, and require the initial system state to be known; our approach does not have these limitations. [8] analyzes both termination and confluence by transforming ECA rules into Datalog rules through a "transformation diagram"; this supports rule priority and execution semantics, but requires the graph to be commutative and restricts event composition. However, most of these works show limited results, and none of them properly addresses confluence; we present detailed experimental results for both termination and confluence. UML Statecharts [17] provide visual diagrams to describe the dynamic behavior of reactive systems and can be used to verify these properties. However, event dispatching and execution semantics are not as flexible as for PNs [13].

Our approach transforms a set of ECA rules into a PN, then dynamically verifies termination and confluence and, if errors are found, provides concrete counterexamples to help debugging. It uses our tool SMART, which supports PNs with priorities to easily model ECA rules with priorities. Moreover, a single PN can naturally describe both the ECA rules as well as their nondeterministic concurrent environment and, while our MDD-based symbolic model-checking algorithms [18] require a finite state space, they do not require to know a priori the variable bounds (i.e., the maximum number of tokens each place may contain). Finally, our framework is not restricted to termination and confluence, but can be easily extended to verify a broader set of properties.

The rest of the paper is organized as follows: Sect. 2 recalls Petri nets; Sect. 3 introduces our syntax for ECA rules; Sect. 4 describes the transformation of ECA rules into a Petri net; Sect. 5 presents algorithms for termination and confluence; Sect. 6 shows experimental results; Sect. 7 concludes.

2 Petri Nets

As an intermediate step in our approach, we translate a set of ECA rules into a *self-modifying Petri net* [16] (PN) with *priorities* and *inhibitor arcs*, described by a tuple $(\mathcal{P}, \mathcal{T}, \pi, \mathbf{D}^-, \mathbf{D}^+, \mathbf{D}^\circ, \mathbf{x}_{init})$, where:

- $\mathcal{P}$ is a finite set of *places*, drawn as circles, and $\mathcal{T}$ is a finite set of *transitions*, drawn as rectangles, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$.
- $\pi : \mathcal{T} \to \mathbb{N}$ assigns a *priority* to each transition.
- $\mathbf{D}^- : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$, $\mathbf{D}^+ : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$, and $\mathbf{D}^\circ : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N} \cup \{\infty\}$ are the *marking-dependent* cardinalities of the *input*, *output*, and *inhibitor* arcs.
- $\mathbf{x}_{init} \in \mathbb{N}^{\mathcal{P}}$ is the *initial marking*, the number of *tokens* initially in each place.

Transition t has *concession* in marking $\mathbf{m} \in \mathbb{N}^{\mathcal{P}}$ if, for each $p \in \mathcal{P}$, the input arc cardinality is satisfied, i.e., $\mathbf{m}_p \geq \mathbf{D}^-(p, t, \mathbf{m})$, and the inhibitor arc cardinality is not, i.e., $\mathbf{m}_p < \mathbf{D}^\circ(p, t, \mathbf{m})$. If t has concession in $\mathbf{m}$ and no other transition t' with priority $\pi(t') > \pi(t)$ has concession, then t is *enabled* in $\mathbf{m}$ and can *fire* and lead to marking $\mathbf{m}'$, where $\mathbf{m}'_p = \mathbf{m}_p - \mathbf{D}^-(p, t, \mathbf{m}) + \mathbf{D}^+(p, t, \mathbf{m})$, for all places p (arc cardinalities are evaluated in the current marking $\mathbf{m}$ to determine the enabling of t and the new marking $\mathbf{m}'$). In our figures, $tk(p)$ indicates the number of tokens in p for the current marking, a thick input arc from p to t signifies a cardinality $tk(p)$, i.e., a *reset* arc, and we omit arc cardinalities 1, input or output arcs with cardinality 0, and inhibitor arcs with cardinality ∞.

The PN defines a discrete-state model $(\mathcal{S}_{pot}, \mathcal{S}_{init}, \mathcal{A}, \{\mathcal{N}_t : t \in \mathcal{A}\})$. The *potential state space* is $\mathcal{S}_{pot} = \mathbb{N}^{\mathcal{P}}$ (in practice we assume that the reachable set of markings is finite or, equivalently, that there is a finite bound on the number of tokens in each place, but we do not require to know this bound a priori). $\mathcal{S}_{init} \subseteq \mathcal{S}_{pot}$ is the set of *initial states*, $\{\mathbf{x}_{init}\}$ in our case (assuming an arbitrary finite initial set of markings is not a problem). The set of (asynchronous) model *events* is $\mathcal{A} = \mathcal{T}$. The *next-state function* for transition t is $\mathcal{N}_t$, such that $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{m}'\}$, where $\mathbf{m}'$ is as defined above if transition t is enabled in marking $\mathbf{m}$, and $\mathcal{N}_t(\mathbf{m}) = \emptyset$ otherwise. Thus, the next-state function for a particular PN transition is deterministic, although the overall behavior remains nondeterministic due to the choice of which transition should fire when multiple transitions are enabled.

3 ECA syntax and semantics

ECA rules have the format "**on** *events* **if** *condition* **do** *actions*". If the *events* are activated and the boolean *condition* is satisfied, the rule is triggered and its *actions* will be performed. In active DBMSs, events are normally produced by explicit database operations such as *insert* and *delete* [1] while, in reactive systems, they are produced by sensors monitoring environment variables [3], e.g., temperature. Many current ECA languages can model the environment and distinguish between **environmental** and **local** variables [2, 4, 6, 11, 14, 15]. Thus, we designed a language to address these issues, able to handle more general cases and allow different semantics for **environmental** and **local** variables (Fig. 1).

$$
\begin{aligned}
\mathit{env\_vars} &:= \textbf{environmental}\ \mathit{env\_var} && \text{READ-ONLY BOUNDED NATURAL}\\
\mathit{loc\_vars} &:= \textbf{local}\ \mathit{loc\_var} && \text{READ-AND-WRITE BOUNDED NATURAL}\\
\mathit{factor} &:= \mathit{loc\_var} \mid \mathit{env\_var} \mid (\ \mathit{exp}\ ) \mid \mathit{number}\\
\mathit{term} &:= \mathit{factor} \mid \mathit{term} * \mathit{term} \mid \mathit{term}\ /\ \mathit{term} && \text{“/” IS INTEGER DIVISION}\\
\mathit{exp} &:= \mathit{exp} - \mathit{exp} \mid \mathit{exp} + \mathit{exp} \mid \mathit{term} && \text{“}\mathit{number}\text{” IS A CONSTANT} \in \mathbb{N}\\
\mathit{rel\_op} &:= \geq \mid \leq \mid =\\
\mathit{assignment} &:= \mathit{env\_var}\ \textbf{into}\ \mathit{loc\_var}\ [, \mathit{assignment}]\\
\mathit{ext\_ev\_decl} &:= \textbf{external}\ \mathit{ext\_ev}\ [\ \textbf{activated when}\ \mathit{env\_var}\ \mathit{rel\_op}\ \mathit{number}\ ]\\
&\quad [\ \textbf{read}\ (\mathit{assignment})\ ]\\
\mathit{int\_ev\_decl} &:= \textbf{internal}\ \mathit{int\_ev}\\
\mathit{ext\_evs} &:= \mathit{ext\_ev} \mid (\mathit{ext\_evs}\ \textbf{or}\ \mathit{ext\_evs}) \mid (\mathit{ext\_evs}\ \textbf{and}\ \mathit{ext\_evs})\\
\mathit{int\_evs} &:= \mathit{int\_ev} \mid (\mathit{int\_evs}\ \textbf{or}\ \mathit{int\_evs}) \mid (\mathit{int\_evs}\ \textbf{and}\ \mathit{int\_evs})\\
\mathit{condition} &:= (\mathit{condition}\ \textbf{or}\ \mathit{condition}) \mid (\mathit{condition}\ \textbf{and}\ \mathit{condition}) \mid\\
&\quad \textbf{not}\ \mathit{condition} \mid \mathit{exp}\ \mathit{rel\_op}\ \mathit{exp}\\
\mathit{action} &:= \textbf{increase}\ (\mathit{loc\_var},\ \mathit{exp}) \mid \textbf{decrease}\ (\mathit{loc\_var},\ \mathit{exp}) \mid\\
&\quad \textbf{set}\ (\mathit{loc\_var},\ \mathit{exp}) \mid \textbf{activate}\ (\mathit{int\_ev})\\
\mathit{actions} &:= \mathit{action} \mid (\mathit{actions}\ \textbf{seq}\ \mathit{actions}) \mid (\mathit{actions}\ \textbf{par}\ \mathit{actions})\\
\mathit{ext\_rule} &:= \textbf{on}\ \mathit{ext\_evs}\ [\textbf{if}\ \mathit{condition}]\ \textbf{do}\ \mathit{actions}\\
\mathit{int\_rule} &:= \textbf{on}\ \mathit{int\_evs}\ [\textbf{if}\ \mathit{condition}]\ \textbf{do}\ \mathit{actions}\ [\textbf{with priority}\ \mathit{number}]\\
\mathit{system} &:= [\mathit{env\_vars}]^{+}[\mathit{loc\_vars}]^{*}[\mathit{ext\_ev\_decl}]^{+}[\mathit{int\_ev\_decl}]^{*}\\
&\quad [\mathit{ext\_rule}]^{+}[\mathit{int\_rule}]^{*}
\end{aligned}
$$

Fig. 1. The syntax of ECA rules.

Environmental variables are used to represent environment states that can only be measured by sensors but not *directly* modified by the system. For instance, if we want to increase the temperature in a room, the system may choose to turn on a heater, eventually achieving the desired effect, but it cannot directly change the value of the temperature variable. Thus, environmental variables capture the nondeterminism introduced by the environment, beyond the control of the system. Instead, local variables can be both read and written by the system. These may be associated with an actuator, a record value, or an intermediate value describing part of the system state; we provide operations to set (absolute change) their value to an expression, or increase or decrease (relative change) it by an expression; these expressions may depend on environmental variables.

Events can be combinations of atomic events activated by environmental or internal changes. We classify them using the keywords **external** and **internal**. An external event can be **activated when** the value of an environmental variable crosses a threshold; at that time, it may take a snapshot of some environmental variables and **read** them into local variables to record their current values. Only the action of an ECA rule can instead **activate** internal events. Internal events are useful to express internal changes or required actions within the system.

These two types of events cannot be mixed within a single ECA rule. Thus, rules are *external* or *internal*, respectively. Then, we say that a state is *stable* if only external events can occur in it, *unstable* if actions of external or internal rules are being performed (including the activation of internal events, which may then trigger internal rules). 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.

The condition portion of an ECA rule is a boolean expression on the value of environmental and local variables; it can be omitted if it is the constant true.

The last portion of a rule specifies which actions must be performed, and in which order. Most actions are operations on local variables which do not directly affect environmental variables, but may cause some changes that will conceivably be reflected in their future values. Thus, all environmental variables are read-only from the perspective of an action. Actions can also **activate** internal events. Moreover, to handle complex action operations, the execution semantics can be specified as any partial order described by a series-parallel graph; this is obtained through an appropriate nesting of **seq** operators, to force a sequential execution, and **par** operators, to allow an arbitrary concurrency. The keyword **with priority** enforces a priority for internal rules. If no priority is specified, the default priority of an internal rule is 1, the same as that of external rules.

We now discuss the choices of execution semantics for our language, to support the modeling of reactive systems. The first choice is how to couple the checking of events and conditions for our ECA rules. There are (at least) two options: *immediate* and *deferred*. The event-condition checking is *immediate* if the corresponding condition is immediately evaluated when the events occur, it is *deferred* if the condition is evaluated at the end of a cycle with a predefined frequency. One critical requirement for the design of reactive systems is that the system should respond to external events from the environment [10] as soon as possible. Thus, we choose immediate event-condition checking: when events occur, the corresponding condition is immediately evaluated to determine whether to trigger the rule. We stress that deferred checking can still be modeled using immediate checking, for example by adding an extra variable for the system clock and changing priorities related to rule evaluation to synchronize rule evaluations. However, the drawback of deferred checking is that the design must tolerate false ECA rule triggering or non-triggering scenarios. Since there is a time gap between event activation and condition evaluation, the environmental conditions that trigger an event might change during this period of time, causing rules supposed to be triggered at the time of event activation to fail because the "current " condition evaluation are now inconsistent.

Another important choice is how to handle and model the concurrent and nondeterministic nature of reactive systems. We introduce the concept of *batch* for external events, similar to the concept of transaction in DBMSs. Formally, the boundary of a batch of external events is defined as the end of the execution

of all triggered rules. Then, the system starts to receive external events and immediately evaluates the corresponding conditions. The occurrence of an external event closes a batch if it triggers one or more ECA rules; otherwise, the event is added to the current batch. Once the batch closes and the rules to be triggered have been determined, the events in the current batch are cleaned-up, to prevent multiple (and erroneous) triggerings of rules. For example, consider ECA rules r_a: "**on** a **do** $\cdots$" and r_{ac}: "**on** (a **and** c) **do** $\cdots$", and assume that the system finishes processing the last batch of events and is ready to receive external events for the next batch. If external events occur in the sequence "$c, a, \ldots$", event c alone cannot trigger any rule so it begins, but does not complete, the current batch. Then, event a triggers both rule r_a and r_{ac}, and thus, closes the current batch. Both rules are triggered and will be executed concurrently. This example shows how, when the system is in a stable state, the occurrence of a single external event may trigger one or more ECA rules, since there is no "contention" within a batch on "using" an external event: rule r_a and r_{ac} share event a and both rules are triggered and executed. If instead the sequence of events is "$a, c, \ldots$", event a by itself constitutes a batch, as it triggers rule r_a. This event is then discarded by the clean-up so, after executing r_a and any internal rule (recursively) triggered by it, the system returns to a stable state and the subsequent events "$c, \ldots$" begin the next batch. Under this semantic, all external events in one *batch* are processed concurrently. Thus, unless there is a termination error, the system will process all triggered rules, including those triggered by the activation of internal events during the current batch, before considering new external events. This batch definition provides maximum nondeterminism on event order, which is useful to discover design errors in a set of ECA rules.

We also stress that, in our semantics, 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 our 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.

3.1 Running example

We now illustrate the expressiveness of our ECA rules on a running example: a light control subsystem in a smart home for senior housing. Fig. 2 lists the requirements in plain English (R_1 to R_5). Using motion and pressure sensors, the system attempts to reduce energy consumption by turning off the lights in unoccupied rooms or if the occupant is asleep. Passive sensors emit signals when an environmental variable value crosses a significant threshold. The motion sensor measure is expressed by the boolean environmental variable *Mtn*. The system also provides automatic adjustment for indoor light intensity based on an outdoor light sensor, whose measure is expressed by the environmental variable $ExtLgt \in \{0, ..., 10\}$. A pressure sensor detects whether the person is asleep and is expressed by the boolean environmental variable *Slp*.

environmental	*Mtn, ExtLgt, Slp*
local	*lMtn, lExtLgt, lSlp, lgtsTmr, intLgts*
external	*SecElp* **read** (*Mtn* **into** *lMtn*, *ExtLgt* **into** *lExtLgt*, *Slp* **into** *lSlp*) *MtnOn* **activated when** $Mtn = 1$ *MtnOff* **activated when** $Mtn = 0$ *ExtLgtLow* **activated when** $ExtLgt \leq 5$
internal	*LgtsOff, LgtsOn, ChkExtLgt, ChkMtn, ChkSlp*
(R_1) When the room is unoccupied for 6 minutes, turn off lights if they are on.	
r_1	**on** *MtnOff* **if** ($intLgts > 0$ **and** $lgtsTmr = 0$) **do set** (*lgtsTmr*, 1)
r_2	**on** *SecElp* **if** ($lgtsTmr \geq 1$ **and** $lMtn = 0$) **do increase** (*lgtsTmr*, 1)
r_3	**on** *SecElp* **if** ($lgtsTmr = 360$ **and** $lMtn = 0$) **do** (**set** (*lgtsTmr*, 0) **par activate** (*LgtsOff*))
r_4	**on** *LgtsOff* **do** (**set** (*intLgts*, 0) **par activate** (*ChkExtLgt*))
(R_2) When lights are off, if external light intensity is below 5, turn on lights.	
r_5	**on** *ChkExtLgt* **if** ($intLgts = 0$ **and** $lExtLgt \leq 5$) **do activate** (*LgtsOn*)
(R_3) When lights are on, if the room is empty or a person is asleep, turn off lights.	
r_6	**on** *LgtsOn* **do** (**set** (*intLgts*, 6) **seq activate** (*ChkMtn*))
r_7	**on** *ChkMtn* **if** ($lSlp = 1$ **or** ($lMtn = 0$ **and** $intLgts \geq 1$)) **do activate** (*LgtsOff*)
(R_4) If the external light intensity drops below 5, check if the person is asleep and set the lights intensity to 6. If the person is asleep, turn off the lights.	
r_8	**on** *ExtLgtLow* **do** (**set** (*intLgts*, 6) **par activate** (*ChkSlp*))
r_9	**on** *ChkSlp* **if** ($lSlp = 1$) **do set** (*intLgts*, 0)
(R_5) If the room is occupied, set the lights intensity to 4.	
r_{10}	**on** *MtnOn* **do** (**set** (*intLgts*, 4) **par set** (*lgtsTmr*, 0))

Fig. 2. ECA rules for the light control subsystem of a smart home.

MtnOn, *MtnOff*, and *ExtLgtLow* are external events activated by the environmental variables discussed above. *MtnOn* and *MtnOff* occur when *Mtn* changes from 0 to 1 or from 1 to 0, respectively. *ExtLgtLow* occurs when *ExtLgt* drops below 6. External event *SecElp* models the system clock, occurs every second, and takes a snapshot of the environmental variables into local variables *lMtn*, *lExtLgt*, and *lSlp*, respectively. Additional local variables *lgtsTmr* and *intLgts* are used. Variable *lgtsTmr* is a timer for R_1, to convert the continuous condition "the room is unoccupied for 6 minutes" into 360 discretized *SecElps* events. Rule r_1 initializes *lgtsTmr* to 1 whenever the motion sensor detects no motion and the lights are on. The timer then increases as second elapses, provided that no motion is detected (rule r_2). If the timer reaches 360, internal event *LgtsOff* is activated to turn off the lights and to reset *lgtsTmr* to 0 (rule r_3). Variable *intLgts* acts as an actuator control to adjust the internal light intensity.

Our ECA rules contain internal events to model internal system actions or checks not observable from the outside. *LgtsOff*, activated by rule r_3 or r_7, turns the lights off and activates another check on outdoor light intensity through internal event *ChkExtLgt* (rule r_4). *ChkExtLgt* activates *LgtsOn* if $lExtLgt \leq 5$

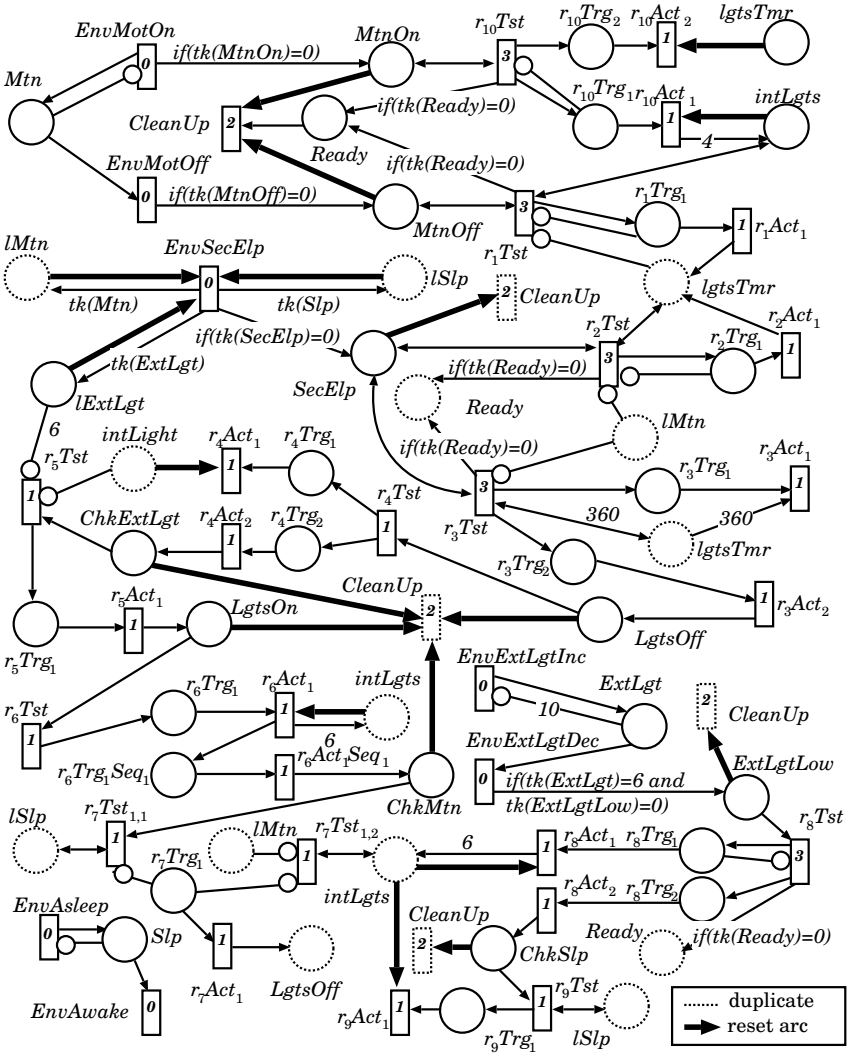

Fig. 3. The PN for ECA rules in Fig. 2.

(rule r_5). *ChkSlp* is activated by rule r_8 to check whether a person is asleep. If true, the event triggers an action that turns the lights off (rule r_9). Internal event *ChkMtn*, activated by rule r_6, activates *LgtsOff* if the room is unoccupied and all lights are on, or if the room is occupied but the occupant is asleep (rule r_7).

4 Transforming a set of ECA rules into a PN

We now explain the procedure to transform a set of ECA rules into a PN. First, we put each ECA rule into a *regular* form where both *events* and *condition* are

disjunctions of conjunctions of events and relational expressions, respectively. All rules in Fig. 2 are in this form. While this transformation may in principle cause the expressions for events and conditions to grow exponentially large, each ECA rule usually contains a small number of events and conditions, hence this is not a problem in practice. Based on the immediate event-condition checking assumption, a rule is triggered iff "*trigger* $\equiv$ *events* $\wedge$ *condition*" holds.

Next, we map variables and events into places, and use PN transitions to model event testing, condition evaluation, and action execution. Any change of variable values is achieved through input and output arcs with appropriate cardinalities. Additional control places and transitions allow the PN behavior to be organized into "phases", as shown next. ECA rules r_1 through r_{10} of Fig. 2 are transformed into the PN of Fig. 3 (dotted transitions and places are duplicated and arcs labeled "*if*(*cond*)" are present only if *cond* holds).

4.1 Occurring phase

This phase models the occurrence of external events, due to environment changes over which the system has no control. The PN firing semantics perfectly matches the nondeterministic asynchronous nature of these changes. For example, in Fig. 3, transitions *EnvMotOn* and *EnvMotOff* can add or remove the token in place *Mtn*, to nondeterministically model the presence or absence of people in the room (the inhibitor arc from place *Mtn* back to transition *EnvMotOn* ensures that at most one token resides in *Mtn*). Firing these environmental transitions might nondeterministically enable the corresponding external events. Here, firing *EnvMotOn* generates the external event *MtnOn* by placing a token in the place of the same name, while firing *EnvMotOff* generates event *MtnOff*, consistent with the change in *Mtn*. To ensure that environmental transitions only fire if the system is in a stable state (when no rule is being processed) we assign the lowest priority, 0, to these transitions. As the system does not directly affect environmental variables, rule execution does not modify them. However, we can take snapshots of these variables by copying the current number of tokens into their corresponding local variables using marking-dependent arcs. For example, transition *EnvSecElp* has an output arc to generate event *SecElp*, and arcs connected to local variables to perform the snapshots, e.g., all tokens in *lMtn* are removed by a reset arc (an input arc that removes all tokens from its place), while the output arc with cardinality $tk(Mtn)$ copies the value of *Mtn* into *lMtn*.

4.2 Triggering phase

This phase starts when *trigger* $\equiv$ *events* $\wedge$ *condition* holds for at least one external ECA rule. If, for rule r_k, *events* and *condition* consist of n_d and n_c disjuncts, respectively, we define $n_d \cdot n_c$ test transitions $r_k Tst_{i,j}$ with priority $P+2$, where i and j are the index of a conjunct in *events* and one in *condition*, while $P \geq 1$ is the highest priority used for internal rules (in our example, all internal rules have default priority $P = 1$). Then, to trigger rule r_k, only one of these transitions, e.g., $r_7 Tst_{1,1}$ or $r_7 Tst_{1,2}$, needs to be fired (we omit i and j if $n_d = n_c = 1$). Firing

a test transition means that the corresponding events and conditions are satisfied and results in placing a token in each of the *triggered* places $r_k\,Trg_1, \ldots, r_k\,Trg_N$, to indicate that Rule r_k is triggered, where N is the number of outermost parallel actions (recall that **par** and **seq** model parallel and sequential actions). Thus, $N = 1$ if r_k contains only one action, or an outermost sequential series of actions. Inhibitor arcs from $r_k\,Trg_1$ to test transitions $r_k Tst_{i,j}$ ensure that, even if multiple conjuncts are satisfied, only one test transition fires. The firing of test transitions does not "consume" external events, thus we use double-headed arrows between them. This allows one batch to trigger multiple rules, conceptually "at the same time". After all enabled test transitions for external rules have fired, place *Ready* contains one token, indicating that the current batch of external events can be cleared: transition *CleanUp*, with priority $P+1$, fires and removes all tokens from external and internal event places using reset arcs, since all the rules that can be triggered have been marked. This ends the triggering phase and closes the current batch of events.

4.3 Performing phase

This phase executes all actions of external rules marked in the previous phase. It may further result in triggering and executing internal rules. Transitions in this phase correspond to the *actions* of rules with priority in $[1, P]$, the same as that of the corresponding rule. An action activates an internal event by adding a token to its place. This token is consumed as soon as a test transition of any internal rule related to this event fires. This is different from the way external rules "use" external events. Internal events not consumed in this phase are cleared when transition *CleanUp* fires in the next batch. When all enabled transitions of the performing phase have fired, the system is in a stable state where environmental changes (transitions with priority 0) can again happen and the next batch starts.

4.4 ECA rules to PN translation

The algorithm in Fig. 4 takes external and internal ECA rules $\mathbf{R}_{ext}$, $\mathbf{R}_{int}$, with priorities in $[1, P]$, environmental and local variables $\mathbf{V}_{env}$, $\mathbf{V}_{loc}$, and external and internal events $\mathbf{E}_{ext}$, $\mathbf{E}_{int}$, and generates a PN. After normalizing the rules and setting P to the highest priority among the rule priorities in $\mathbf{R}_{int}$, it maps environmental variables $\mathbf{V}_{env}$, local variables $\mathbf{V}_{loc}$, external events $\mathbf{E}_{ext}$, and internal events $\mathbf{E}_{int}$, into the corresponding places (Lines 5, 6, and 8). Then, it creates phase control place *Ready*, transition *CleanUp*, and reset arcs for *CleanUp* (Lines 4-5). We use arcs with marking-dependent cardinalities to model expressions. For example, together with inhibitor arcs, these arcs ensure that each variable $v \in \mathbf{V}_{env}$ remains in its range $[v_{min}, v_{max}]$ (Lines 8-10). These arcs also model the **activated when** portion of external events (Line 17), rule conditions (Line 33), and assignments of environmental variables to local variables (Lines 19-20 and lines 14-15). The algorithm models external events and environmental changes (Lines 11-24); it connects environmental transitions such as t_{vInc} and t_{vDec} to their corresponding external event places, if any, with an arc

TransformECAintoPN $(\mathbf{R}_{ext}, \mathbf{R}_{int}, \mathbf{V}_{env}, \mathbf{V}_{loc}, \mathbf{E}_{ext}, \mathbf{E}_{int})$

1 normalize $\mathbf{R}_{ext}$ and $\mathbf{R}_{int}$ into regular form and set P to the highest rule priority
2 create a place $Ready$ • *to control "phases" of the net*
3 create transition $CleanUp$ with priority $P+1$ and $Ready$ –[1]→ $CleanUp$
4 **for each** event $e \in \mathbf{E}_{ext} \cup \mathbf{E}_{int}$ **do**
5 create place p_e and p_e –[$tk(p_e)$]→ $CleanUp$
6 create place p_v, for each variable $v \in \mathbf{V}_{loc}$
7 **for each** variable $v \in \mathbf{V}_{env}$ with range $[v_{min}, v_{max}]$ **do**
8 create place p_v and transitions t_{vInc} and t_{vDec} with priority 0
9 create t_{vDec} –[if($tk(p_v) > v_{min}$)1 else 0]→ p_v
10 create t_{vInc} –[1]→ p_v and p_v –[v_{max}]–○ t_{vInc}
11 **for each** event $e \in \mathbf{E}_{ext}$ **activated when** $v\ op\ val$, for $op \in \{\geq \mid =\}$ **do**
12 create t_{vInc} –[if($tk(p_v)\ op\ val$)1 else 0]→ p_e
13 **if** e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ **then**
14 create $p_{v'}$ –[if($tk(p_v)\ op\ val$)$tk(p_{v'})$ else 0]→ t_{vInc}
15 create t_{vInc} –[if($tk(p_v)\ op\ val$)$tk(p_v)$ else 0]→ $p_{v'}$
16 **for each** event $e \in \mathbf{E}_{ext}$ **activated when** $v\ op\ val$, for $op \in \{\leq \mid =\}$ **do**
17 create t_{vDec} –[if($tk(p_v)\ op\ val$)1 else 0]→ p_e
18 **if** e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ **then**
19 create $p_{v'}$ –[if($tk(p_v)\ op\ val$)$tk(p_{v'})$ else 0]→ t_{vDec}
20 create t_{vDec} –[if($tk(p_v)\ op\ val$)$tk(p_v)$ else 0]→ $p_{v'}$
21 **for each** event $e \in \mathbf{E}_{ext}$ without an **activated when** portion **do**
22 create t_e and t_e –[if($tk(p_e) = 0$)1 else 0]→ p_e
23 **if** e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ **then**
24 create $p_{v'}$ –[$tk(p_{v'})$]→ t_e and t_e –[$tk(p_v)$]→ $p_{v'}$
25 **for each** rule $r_k \in \mathbf{R}_{ext} \cup \mathbf{R}_{int}$ with n_d event disjuncts, n_c condition disjuncts, actions A, and priority $p \in [1, P]$ **do**
26 create trans. $r_k\,Tst_{i,j}, i \in [1, n_d], j \in [1, n_c]$,w/priority $P+2$ if $r_k \in \mathbf{R}_{ext}$, else p
27 **for each** event e in disjunct i **do**
28 create p_e –[1]→ $r_k\,Tst_{i,j}$
29 create $r_k\,Tst_{i,j}$ –[1]→ p_e, if $e \in \mathbf{E}_{ext}$
30 **for each** conjunct $v \leq val$ or $v = val$ in disjunct j **do**
31 create p_v –[$val + 1$]–○ $r_k\,Tst_{i,j}$
32 **for each** conjunct $v \geq val$ or $v = val$ in disjunct j **do**
33 create p_v –[val]→ $r_k\,Tst_{i,j}$ and $r_k\,Tst_{i,j}$ –[val]→ p_v
34 **if** actions A is "$(A_1$ *par* $A_2)$" **then** $n_a = 2$ **else** $n_a = 1$, $A_1 = A$
35 **for each** $l \in [1, n_a]$ **do**
36 create places $r_k\,Trg_l$ and transitions $r_k\,Act_l$ with priority p
37 create $r_k\,Trg_l$ –[1]→ $r_k\,Act_l$ and $r_k\,Tst_{i,j}$ –[1]→ $r_k\,Trg_l$
38 $SeqSubGraph(A_l, "r_k\,Acl_l", l, p)$
39 **for each** $r_k \in \mathbf{R}_{ext}, i \in [1, n_d], j \in [1, n_c]$ **do**
40 create $r_k\,Tst_{i,j}$ –[if($tk(Ready) = 0$)1 else 0]→ $Ready$ and $r_k\,Trg_1$ –[1]–○ $r_k\,Tst_{i,j}$

Fig. 4. Transforming ECA rules into a PN: a –[k]→ b means "an arc from a to b with cardinality k; a –[k]–○ b means "an inhibitor arc from a to b with cardinality k.

whose cardinality evaluates to 1 if the corresponding condition becomes true upon the firing of the transition and the event place does not contain a token already, 0 otherwise (e.g., the arcs from *EnvExtLigDec* to *ExtLgtLow*).

Next, rules are considered (Lines 25-40). A rule with n_d event disjuncts and n_c condition disjuncts generates $n_d \cdot n_c$ testing transitions. To model the parallel-sequential action graph of a rule, we use mutually recursive procedures (Fig. 5 and Fig. 6). Procedure *SeqSubGraph* first tests all atomic actions, such as "set",

```
ParSubGraph(Pars, Pre, p)                    • Pars: parallel actions, Pre: prefix
 1  for each l ∈ {1,2} do     • according to the syntax Pars_2 has two components
 2    create place PreAct_l Trg_l Seq_l and transition PreAct_l w/priority p
 3    create Pre –[1]→ PreAct_l and PreAct_l –[1]→ PreAct_l Trg_l Seq_l
 4    SeqSubGraph(Pars_l, "PreAct_l Trg_l Seq_l", l, p);
```

Fig. 5. Processing **par**.

```
SeqSubGraph(Seqs, Pre, i, p)                 • Seqs: sequential actions, Pre: prefix
 1  if Seqs sets variable v to val then
 2    create p_v –[tk(p_v)]→ Pre and Pre –[val]→ p_v
 3  else if Seqs increases variable v by val then
 4    create Pre –[val]→ p_v
 5  else if Seqs decreases variable v by val then
 6    create p_v –[val]→ Pre
 7  else if Seqs activates an internal event e then
 8    create Pre –[1]→ p_e
 9  else if the outermost operator of Seqs is par then
10    ParSubGraph(Seqs, "Pre", p)                  • Recursion on parallel part
11  else if the outermost operator of Seqs is seq then
12    SeqSubGraph(Seqs_1, "Pre", 1, p)          • Seqs_1 is the first part of Seq
13    create place PreTrg_i Seq_1 and transition PreAct_i Seq_1
14    create Pre –[1]→ PreTrg_i Seq_1 and PreTrg_i Seq_1 –[1]→ PreAct_i Seq_1
15    SeqSubGraph(Seqs_2, "PreTrg_i Seq_1", 2, p)   • Seqs_2 is the second part of Seq
```

Fig. 6. Processing **seq**.

"increase", "decrease", and "activate". Then, it recursively calls *ParSubGraph* at Line 10 if it encounters parallel actions. Otherwise, it calls itself to unwind another layer of sequential actions at Line 12 and Line 15 for the two portions of the sequence. Procedure *ParSubGraph* creates control places and transitions for the two branches of a parallel action and calls *SeqSubGraph* at Line 4.

5 Verifying properties

The first step towards verifying correctness properties is to define $\mathcal{S}_{init}$, the set of initial states, corresponding to all the possible initial combinations of system variables (e.g., *ExtLgt* can initially have any value in $[0, 10]$). One could consider all these possible values by enumerating all legal stable states corresponding to possible initial combinations of the environmental variables, then start the analysis from each of these states, one at a time. However, in addition to requiring the user to explicitly provide the set of initial states, this approach may require enormous runtime, also because many computations are repeated in different runs. Our approach instead computes the initial states symbolically, thanks to the nondeterministic semantics of Petri nets, so that the analysis is performed once starting from a single, but very large, set $\mathcal{S}_{init}$.

To this end, we add an initialization phase that puts a nondeterministically chosen legal number of tokens in each place corresponding to an environmental variable. This phase is described by a subnet consisting of a transition *InitEnd*

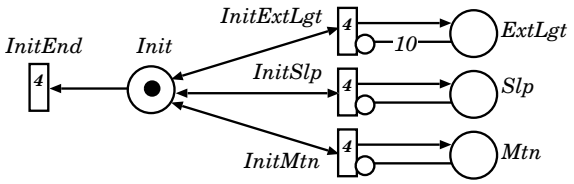

Fig. 7. The initialization phase for the smart home example.

with priority $P+3$, a place *Init* with one initial token, and an initializing transition with priority $P+3$ for every environmental variable, to initialize the number of tokens in the corresponding place. Fig. 7 shows this subnet for our running example. We initialize the Petri net by assigning the minimum number of tokens to every environmental variable place and leaving all other places empty, then we let the initializing transitions nondeterministically add a token at a time, possibly up to the maximum legal number of tokens in each corresponding place. When *InitEnd* fires, it disables the initializing transitions, freezes the nondeterministic choices, and starts the system's normal execution.

This builds the set of initial states, ensuring that the PN will explore all possible initial states, and avoids the overhead of manually starting the PN from one legal initial marking at a time. Even though the overall state space might be larger (it equals the union of all the state spaces that would be built starting from each individual marking), this is normally not the case, and, anyway, having to perform just one state space generation is obviously enormously better.

After the initialization step, we proceed with verifying termination and confluence using our tool SMART, which provides symbolic reachability analysis and CTL model checking with counterexample generation [7].

5.1 Termination

Reactive systems constantly respond to external events. However, if the system has a *livelock*, a finite number of external events can trigger an infinite number of rule executions (i.e, activate a cycle of internal events), causing the system to remain "busy" internally, a fatal design error. When generating the state space, all legal batches of events are considered, due to the PN execution semantics, again avoiding the need for an explicit enumeration, this time, of event batches.

> A set $\mathcal{G}$ of ECA rules satisfies *termination* if no infinite sequence of internal events can be triggered in any possible execution of $\mathcal{G}$. This can be expressed in CTL as $\neg\mathsf{EF}(\mathsf{EG}(\mathit{unstable}))$, stating that there is no cycle of unstable states reachable from an initial, thus stable, state.

Both traditional breadth-first-search (BFS) and saturation-based [19] algorithms are suitable to compute the EG operator. Fig. 8 uses saturation, which tends to perform much better in both time and memory consumption when analyzing large asynchronous systems. We encode transitions related to external

```
bool Term(mdd S_init, mdd2 N_int)
 1  mdd S_rch ← StateSpaceGen(S_init, N_ext ∪ N_int);
 2  mdd S_unst ← Intersection(S_rch, ExtractUnprimed(N_int));
 3  mdd S_p ← EF(EG(S_unst));
 4  if S_p ≠ ∅ then return false                          • provide error trace
 5  else return true;
```

```
mdd ExtractUnprimed(mdd2 p)                              • p unprimed
 1  if p = 1 then return 1;
 2  if CacheLookUp(EXTRACTUNPRIMED, p, r) return r;
 3  foreach i ∈ V_p.v do
 4    mdd r_i ← 0;
 5    if p[i] ≠ 0 then                  • p[i] is the node pointed edge i of node p
 6      foreach j ∈ V_p.v s.t. p[i][j] ≠ 0 do
 7        r_i ← Union(r_i, ExtractUnprimed(p[i][j]))
 8  mdd r ← UniqueTableInsert({r_i : i ∈ V_p.v});
 9  CacheInsert(EXTRACTUNPRIMED, p, r);
10  return r;
```

Fig. 8. Algorithms to verify the termination property.

events and environmental variable changes into $\mathcal{N}_{ext}$. Thus, the internal transitions are $\mathcal{N}_{int} = \mathcal{N} \backslash \mathcal{N}_{ext}$. After generating the state space $\mathcal{S}_{rch}$ using constrained saturation [18], we build the set of states $\mathcal{S}_{unst}$ by symbolically intersecting $\mathcal{S}_{rch}$ with the unprimed, or "from", states extracted from $\mathcal{N}_{int}$. Then, we use the CTL operators EG and EF to identify any nonterminating path (i.e., cycle).

5.2 Confluence

Confluence is another desirable property to ensure consistency in systems exhibiting highly concurrent behavior.

> A set $\mathcal{G}$ of ECA rules satisfying termination also satisfies confluence if, for any *legal* batch b of external events and starting from any particular stable state s, the system eventually reaches a *unique* stable state.

We stress that what constitutes a legal batch b of events depends on state s, since the condition portion of one or more rules might affect whether b (or a subset of b) can trigger a rule (thus close a batch). Given a legal batch b occurring in stable state s, the system satisfies *confluence* if it progresses from s by traversing some (nondeterministically chosen) sequence of unstable states, eventually reaching a stable state uniquely determined by b and s. Checking confluence is therefore expensive [2], as it requires verifying the combinations of all stable states reachable from $\mathcal{S}_{init}$ with all legal batches of external events when the system is in that stable state. A straightforward approach enumerates all legal batches of events for each stable state, runs the model, and checks that the set of reachable stable states has cardinality one. We instead only check that, from each reachable unstable state, exactly one stable state is reachable; this avoids enumerating all legal batches of events for each stable state. Since

```
bool ConfExplicit(mdd S_st, mdd S_unst, mdd2 N_int)
 1  foreach i ∈ S_unst
 2    mdd S_i ← StateSpaceGen(i, N_int);
 3    if Cardinality(Intersection(S_i, S_st)) > 1 then return false; • provide error trace
 4  return true;
```

```
bool ConfExplicitImproved(mdd S_st, mdd S_unst, mdd2 N_int, mdd2 N)
 1  mdd S_frontier ← Intersection(RelProd(S_st, N), S_unst);
 2  while S_frontier ≠ ∅ do                    • if S_frontier is empty, it explores all S_unst
 3    pick i ∈ S_frontier;
 4    mdd S_i ← StateSpaceGen(i, N_int);
 5    if Cardinality(Intersection(S_i, S_st)) > 1 then return false; • provide error trace
 6    else S_frontier ← S_frontier \ Intersection(S_i, S_unst);     • exclude all unstable
      states reached by i
 7  return true;
```

Fig. 9. Explicit algorithms to verify the confluence property.

```
bool ConfSymbolic(mdd S_st, mdd S_unst, mdd2 N_int)
 1  mdd2 TC ← ConstraintedTransitiveClosure(N_int, S_unst);
 2  mdd2 TC_u2s ← FilterPrimed(TC, S_st);
 3  return CheckConf(TC_u2s);
```

```
bool CheckConf(mdd2 p)
 1  if p = 1 then return true
 2  if CacheLookUp(CHECKCONF, p, r) return r;
 3  foreach i ∈ V_p.v, s.t. exist j, j' ∈ V_p.v, j ≠ j', p[i][j] ≠ 0, p[i[j'] ≠ 0 do
 4    foreach j, j' ∈ V_p.v, j ≠ j' s.t. p[i][j] ≠ 0, p[i[j'] ≠ 0 do
 5      if p[i][j] = p[i][j'] return false;          • Confluence does not hold
 6      mdd f_j ← ExtractUnprimed(p[i][j]);            • Result will be cached
 7      mdd f_j' ← ExtractUnprimed(p[i][j']);        • No duplicate computation
 8      if Intersection(f_i, f_j') ≠ 0 then return false;
 9  foreach i, j ∈ V_p.v s.t. p[i][j] ≠ 0 do
10    if CheckConf(p[i][j]) = false return false;
11  CacheInsert(CHECKCONF, p, true);
12  return true;
```

Fig. 10. Fully symbolic algorithm to verify the confluence property.

nondeterministic execution in performing phase is the main reason to violate confluence and the system is in unstable states in our definition, checking the evolution starting from unstable states will fulfill the purpose.

The brute force algorithm *ConfExplicit* in Fig. 9 enumerates unstable states and generates reachable states only from unstable states using constrained saturation [18]. Then, it counts the stable states in the obtained set. We observe that, starting from an unstable stable u, the system may traverse a large set of unstable states before reaching a stable state. If unstable state u is reachable, so are the unstable states reachable from it. Thus, the improved version *ConfExplicitImproved* first picks an unstable state $\mathbf{i}$ and, after generating the states reachable from $\mathbf{i}$ and verifying that they include only one stable state, it excludes all visited unstable states (Line 6). Furthermore, it starts only from states $\mathbf{i}$ in the *frontier*, i.e., unstable states reachable in one step from stable

Termination (time: sec, memory: MB)					
Model	$\lvert\mathcal{S}_{rch}\rvert$	T_t	T_c	M_p	M_f
PN_t	$2.66{\cdot}10^6$	0.009	9.665	358.68	88.36
PN_c	$2.61{\cdot}10^6$	0.005	9.497	344.42	87.88
PN_1	$8.99{\cdot}10^6$	0.010	11.559	391.52	89.78
PN_2	$1.78{\cdot}10^7$	0.010	24.477	673.33	158.66
PN_3	$1.78{\cdot}10^7$	0.010	85.171	1686.46	559.52
PN_4	$5.02{\cdot}10^7$	0.010	14.541	491.80	105.90

Confluence (time: min, memory: GB, –: out of memory)							
Model	$\lvert\mathcal{S}_{rch}\rvert$	Best Explicit			Symbolic		
		T_{be}	M_p	M_f	T_s	M_p	M_f
PN_c	$2.38{\cdot}10^6$	4.51	4.24	4.10	5.11	2.02	0.22
PN_1	$8.12{\cdot}10^6$	40.25	14.53	14.33	6.40	2.31	0.27
PN_2	$1.61{\cdot}10^7$	34.40	0.85	0.08	10.11	2.59	0.25
PN_3	$2.33{\cdot}10^7$	>120.00	–	–	60.09	2.59	0.25
PN_4	$4.55{\cdot}10^7$	>120.00	–	–	23.33	4.66	0.52

Table 1. Results to verify the ECA rules for a smart home.

states (all other unstable reachable states are by definition reachable from this frontier). However, we stress that these, as most symbolic algorithms, are heuristics, thus they are not guaranteed to work better than the simpler approaches.

Next, we introduce a fully symbolic algorithm to check confluence in Fig. 10. It first generates the transition transitive closure (TC) set from $\mathcal{N}_{int}$ using constrained saturation [19], where the "from"' states of the closure are in $\mathcal{S}_{unst}$ (Line 1). The resulting set encodes the reachability relation from any reachable unstable state without going through any stable state. Then, it filters this relation to obtain the relation from reachable unstable states to stable states by constraining the "to" states to set $\mathcal{S}_{st}$. Thus, checking confluence reduces to verifying whether there exist two different pairs $(\mathbf{i}, \mathbf{j})$ and $(\mathbf{i}, \mathbf{j}')$ in the relation: Procedure *CheckConf* implements this check symbolically. While computing TC is a an expensive operation [19], this approach avoids separate searches from distinct unstable states, thus is particularly appropriate when $\mathcal{S}_{unst}$ is huge.

6 Experimental results

Table 1 reports results for a set of models run on an Intel Xeon 2.53GHz workstation with 36GB RAM under Linux. For each model, it shows the state space size ($\lvert\mathcal{S}_{rch}\rvert$), the peak memory ($M_p$), and the final memory (M_f). For termination, it shows the time used to verify the property (T_t) and to find the *shortest*

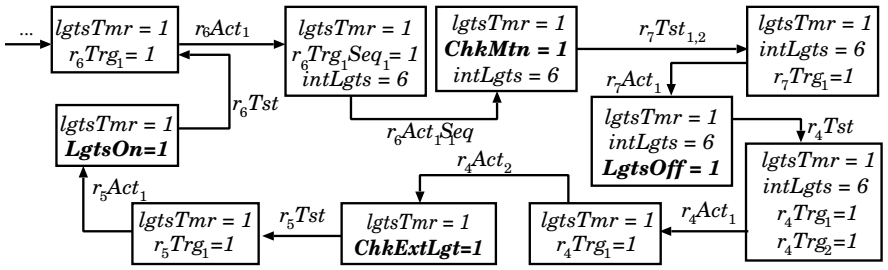

Fig. 11. A termination counterexample (related to rules r_4 to r_7).

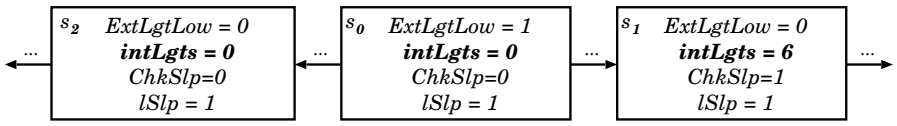

Fig. 12. A confluence counterexample (related to rules r_8 and r_9).

counterexample (T_c). For confluence, it reports the best runtime between our two explicit algorithms (T_{be}) and for our symbolic algorithm (T_s). Memory consumption accounts for both decision diagrams and operation caches.

Net PN_t is the model corresponding to our running example, and fails the termination check. Even though the state space is not very large, counterexample generation is computationally expensive [20] and consumes most of the runtime. The shortest counterexample generated by SMART has a long tail consisting of 1885 states and leads to the 10-state cycle of Fig. 11 (only the non-empty places are listed for each state, and edges are labeled with the corresponding PN transition). Analyzing the trace, we can clearly see (in bold) that, when lights are about to be turned off due to the timeout, $lMtn = 0$, and the external light is low, $ExtLgt \leq 5$, the infinite sequence of internal events $(LgtsOff, ChkExtLgt, LgtsOn, ChkMtn)^\omega$ prevents the system from terminating. Thus, rules r_4, r_5, r_6, and r_7 need to be investigated to fix the error. Among the possible modifications, we choose to replace rule r_5 with r_5' : **on** *ChkExtLgt* **if** ((*intLgts*= 0 **and** $lExtLgt \leq 5$) **and** $lMtn = 1$) **do activate** (*LgtsOn*), resulting in the addition of an input arc from *lMtn* to $r_5\,Tst$. The new corrected model is called PN_c in Table 1, and SMART verifies it holds the termination property.

We then run SMART on PN_c to verify confluence, and found 72,644 bad states. Fig. 12 shows one of these unstable states, s_0, reaching two stables states, s_1 and s_2. External event *ExtLgtLow* closes the batch in s_0 and triggers rule r_8, which sets *intLgt* to 6 and activates internal event *ChkSlp*, which in turn sets *intLgt* to 0 (we omit intermediate unstable states from s_0 to s_1 and to s_2). We correct rules r_8 and r_9, replacing them with r_8': **on** *ExtLgtLow* **if** *lSlp*=0 **do set** (*intLgt*, 6) and r_9': **on** *ExtLgtLow* **if** *lSlp*=1 **do set** (*intLgt*, 0); resulting in model PN_{fc}. Checking this new model against for confluence, we find that the number of bad states decreases from 72,644 to 24,420. After investigation, we determine that the remaining problem is related to rules r_2 and r_3. After

changing rule r_1 to **on** *SecElp* **if** ((*lgtsTmr* ≥ 1 **and** *lgtsTmr* ≤ 359) **and** *lMtn* $= 0$) **do increase** (*lgtsTmr*, 1), the model passes the check. This demonstrates the effectiveness of counterexamples to help a designer debug a set of ECA rules.

We then turn our attention to larger models, which extend our original model by introducing four additional rules and increasing variable ranges. In PN_1 and PN_2, the external light variable *ExtLgt* ranges in $[0, 20]$ instead of $[0, 10]$; for PN_4, it ranges in $[0, 50]$. PN_2 also extends the range of the light timer variable *lgtTmr* to $[0, 720]$; PN_3 to $[0, 3600]$. We observe that, when verifying termination or confluence, the time and memory consumption tends to increase as the model grows; also, our symbolic algorithm scales much better than the best explicit approach when verifying confluence. For the relatively small state space of PN_t, enumeration is effective, since computing TC is quite computationally expensive. However, as the state space grows, enumerating the unstable states consumes excessive resources. We also observe that the supposedly improved explicit confluence algorithm sometimes makes things worse. The reason may lie in the fact that a random selection of a state from the frontier has different statistical properties than for the original explicit approach, and also in the fact that operation caches save many intermediate results. However, both explicit algorithms run out of memory on PN_3 and PN_4. Comparing the results for PN_3 and PN_4, we also observe that larger state spaces might require less resources. With symbolic encodings, this might happen because the corresponding MDD is more regular than the one for a smaller state space.

7 Conclusion

Verifying termination and confluence of ECA rule bases for reactive systems is challenging due to their highly concurrent and nondeterministic nature. We proposed an approach to verify these properties using a self-modifying PN with inhibitor arcs and priorities. Our approach is general enough to give precise answers to questions about other properties, certainly those that can be expressed in CTL. As an application, we showed how a light control system can be captured by our approach, and we verified termination and confluence for this model using SMART. In the future, we would like to improve our approach in the following ways. The confluence algorithm must perform constrained state space generation starting from each unstable state, which is not efficient if $\mathcal{S}_{unst}$ is large. In that case, a simulation-based falsification approach might be more suitable, using intelligent heuristic sampling and searching strategies. However, this approach is sound only if the entire set $\mathcal{S}_{unst}$ is explored. Another direction to extend our work is the inclusion of abstraction techniques to reduce the size of the state space.

Acknowledgement
Work supported in part by UC-MEXUS under grant *Verification of active rule bases using timed Petri nets* and by the National Science Foundation under grant CCF-1018057.

References

1. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
2. A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. *Proc. ACM SIGMOD*, pp. 59–68. ACM Press, 1992.
3. J. C. Augusto and C. D. Nugent. A new architecture for smart homes based on ADB and temporal reasoning. *Toward a Human-Friendly Assistive Environment*, vol. 14, pp. 106–113, 2004.
4. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. *Rules in Database Systems*, LNCS 985, pp. 163–181. 1995.
5. E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM TODS*, 25(3):269–332, Sept. 2000.
6. E.-H. Choi, T. Tsuchiya, and T. Kikuno. Model checking active database rules under various rule processing strategies. *IPSJ Digital Courier*, 2:826–839, 2006.
7. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pp. 78–97. 2003.
8. S. Comai and L. Tanca. Termination and confluence by rule prioritization. *IEEE TKDE*, 15:257–270, 2003.
9. K. G. Kulkarni, N. M. Mattos, and R. Cochrane. Active database features in SQL3. *Active Rules in Database Systems*, pp. 197–219. Springer, New York, 1999.
10. E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, UC Berkeley, May 2007.
11. X. Li, J. M. Marín, and S. V. Chapa. A structural model of ECA rules in active database. *Proc. Second Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pp. 486–493. Springer, 2002.
12. D. McCarthy and U. Dayal. The architecture of an active database management system. *ACM SIGMOD Record*, 18(2):215–224, 1989.
13. T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.
14. D. Nazareth. Investigating the applicability of Petri nets for rule-based system verification. *IEEE TKDE*, 5(3):402–415, 1993.
15. I. Ray and I. Ray. Detecting termination of active database rules using symbolic model checking. *Proc. East European Conference on Advances in Databases and Information Systems*, pp. 266–279. Springer, 2001.
16. R. Valk. Generalizations of Petri nets. *Mathematical foundations of computer science*, LNCS 118, pp. 140–155. 1981.
17. D. Varró. A formal semantics of UML statecharts by model transition systems. *Proc. Int. Conf. on Graph Transformation*, pp. 378–392. Springer, 2002.
18. Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. *Proc. ATVA*, LNCS 5799, pp. 368–381. 2009.
19. Y. Zhao and G. Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *ISSE*, 7(2):141–150, 2011.
20. Y. Zhao, J. Xiaoqing, and G. Ciardo. A symbolic algorithm for shortest EG witness generation. *Proc. TASE*, pp. 68–75. IEEE Comp. Soc. Press, 2011.