Symbolic termination and confluence checking for ECA rules

Xiaoqing Jin[1], Yousra Lembachar[1], and Gianfranco Ciardo[2]

[1] Dept. of Computer Science and Engineering, University of California, Riverside
[2] Department of Computer Science, Iowa State University
{jinx,ylemb001}@cs.ucr.edu,ciardo@iastate.edu

Abstract. Event-condition-action (ECA) rules can specify decision processes 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. However, while the nondeterministic and concurrent semantics of ECA rule execution enhances expressiveness, it also makes analysis and verification more difficult. We 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) [18] rules are expressive enough to describe complex events and reactions. Thus, this event-driven formalism is widely used to specify complex systems [1,3] such as 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 [15]. 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 all interleavings of a set of triggered rules yields the same result, are fundamental correctness properties. While termination has been studied extensively and many algorithms have been proposed to verify it, confluence is more challenging due to a potentially large number of rule interleavings [2] and its complex semantics, which cannot be expressed in the temporal logics LTL or CTL [11].

Researchers began studying these properties for active databases in the 90's, by transforming ECA rules into a graph and applying static analysis techniques on it to verify properties [2,4,17,20]. These approaches based on static methods can detect redundancy, inconsistency, incompleteness, and circularity. However, as they may not explore the whole state space, they can easily miss some errors. Also, they can find scenarios that do not actually result in errors because the discovered error states may not be reachable. Moreover, they have poor support in providing concrete counterexamples and analyzing ECA rules with priorities. [2] looks for cycles in the rule-triggering graph to disprove termination, but the cycle-triggering conditions may be unsatisfiable. [4] improves 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] proposes an algebraic approach emphasizing the condition portion of rules, but does not consider priorities either. Other researchers [17,20] chose to translate ECA rules into a Petri Net (PN), whose nondeterministic interleaving execution semantics naturally models unforeseen interactions between rule executions. However, as they analyze the ECA rules through structural PN techniques based on the incidence matrix, false positives are again possible.

Model checking tools such as SMV [21] and SPIN [7] have also been proposed. While closer to our work, these dynamic approaches manually transform ECA rules into an input script, assume a priori bounds for all variables, do not support priorities, and require the initial system states to be known; our approach does not have these limitations. [12] 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. 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 [23] 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 [19].

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 *multiway decision diagram* (MDD) [14] based symbolic model-checking algorithms [24] require a finite state space, they do not require to know a priori the variable bounds (i.e., the maximum number of tokens that 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 briefly recalls Petri nets, symbolic algorithms, and CTL model checking; 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 Preliminaries

This section provides the formal definition of the specific class of Petri nets used in our approach, as well as a brief background on key symbolic data structures and algorithms for their analysis. We use the symbols $\mathbb{N}$ and $\mathbb{B}$ to denote the natural numbers and the set $\{0, 1\}$, respectively.

2.1 Self-modifying Petri nets with priorities and inhibitor arcs

A *self-modifying Petri net* [22] (PN) with *priorities* and *inhibitor arcs* is described by a tuple $(\mathcal{P}, \mathcal{T}, \pi, \mathbf{D}^-, \mathbf{D}^+, \mathbf{D}^\circ, \mathbf{s}^{init})$, 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.
- $\mathbf{s}^{init} \in \mathbb{N}^{\mathcal{P}}$ 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 represents an arc with cardinality $tk(p)$, i.e., a *reset* arc. We omit arcs with cardinality 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 $\mathcal{S}_{pot} = \mathbb{N}^{\mathcal{P}}$ (we assume that the reachable set of markings is finite or, equivalently, that the number of tokens in each place is bounded, but we do not require to know the bound a priori). $\mathcal{S}_{init} \subseteq \mathcal{S}_{pot}$ is the set of *initial states*, $\{\mathbf{s}^{init}\}$ 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 $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{m}'\}$ if transition t is enabled in marking $\mathbf{m}$, where $\mathbf{m}'$ is as defined above, 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.

2.2 Multiway decision diagrams and the saturation algorithm

Traditional symbolic algorithms for state-space generation or temporal-logic model checking use *binary decision diagrams* (BDDs) [6] to encode sets of (or relations on) states and compute sets of interest as the fixpoint of *breadth-first* iterations. Our approach uses instead *multiway decision diagrams* (MDDs) [14], which are more natural when encoding variables with unknown range (e.g., the number of tokens in a place), and, more fundamentally, computes fixpoints using *saturation* [9], which tends to be very efficient for largely asynchronous systems. We introduce these two concepts, see our recent survey [10] for more details.

Consider a sequence of *domain variables* $(v_L, \ldots, v_1)$ with an order "$\succ$" defined on them, such that $l > k$ implies $v_l \succ v_k$, and where v_k has finite domain $\mathcal{X}_k = \{0,1,...,n_k - 1\}$ for some $n_k > 0$. An *(ordered) multiway decision diagram* (MDD) over $(v_L, \ldots, v_1)$ is an acyclic directed edge-labeled graph where:

- The only *terminal* nodes can be **0** and **1**, and are associated with the *range* variable $\mathbf{0}.var = \mathbf{1}.var = v_0$, satisfying $v_k \succ v_0$ for any domain variable v_k.
- A *nonterminal* node p is associated with a domain variable $p.var$.
- For each $i \in \mathcal{X}_k$, a nonterminal node p associated with v_k has an outgoing edge labeled with i and pointing to a child $p[i]$, where $p.var \succ p[i].var$.

If $p.var = v_k$, node p encodes function $f_p : \mathcal{X}_L \times \cdots \times \mathcal{X}_1 \to \mathbb{B}$, recursively given by $f_p(i_L, ..., i_1) = p$, if $k = 0$, and $f_p(i_L, ..., i_1) = f_{p[i_k]}(i_L, ..., i_1)$, otherwise (we write f_p as a function of L variables even when $k < L$, to stress that *any* variable v_h not explicitly appearing on a path from p to a terminal node is a "don't care" for f_p, regardless of whether $h < k$ or $h > k$).

Canonical MDDs, which have no *duplicate nodes* (if $p.var = q.var = v_k$ and, for each $i \in \mathcal{X}_k$, $p[i] = q[i]$, then $p = q$) and are in *quasi-reduced form* (if $p.var = v_k$, then $p[i].var = v_{k-1}$ and $k = L$ if p is a root) or in *fully-reduced form* (if $p.var = v_k$, then there are indices $i, j \in \mathcal{X}_k$, s.t. $p[i] \neq p[j]$), are quite efficient and compact: if functions f and g are encoded with canonical MDDs, *satisfiability* (is $f \neq 0$?) and *equivalence* (is $f = g$?), are answered in $O(1)$ time, while the MDD encoding the result of elementwise operations such as *conjunction* ($f \wedge g$) is built in time and space proportional to the product of the number of nodes in f and g.

Of course, in symbolic system analysis, the domain $\mathcal{X}_{pot} = \mathcal{X}_L \times \cdots \times \mathcal{X}_1$ is the *potential state space* (states that the system *may* reach during its evolution), and an MDD root p encodes the set $\mathcal{X} = \{(i_L, ..., i_1) : f_p(i_L, ..., i_1) = \mathbf{1}\} \subseteq \mathcal{X}_{pot}$. Fig. 1, adapted from [10], gives an example of a potential state space and the set of states $\mathcal{X}$ encoded by a quasi-reduced or a fully-reduced MDD. A relation $\mathcal{T}$ over states can also be encoded by MDDs, using two *interleaved* copies of the domain variables, so that a path $(i_L, i'_L, ..., i_1, i'_1)$ in such MDD signifies $((i_L, ..., i_1), (i'_L, ..., i'_1)) \in \mathcal{T}$. If $\mathcal{T}$ represents the *transition relation*, this would mean that the system can move from $(i_L, ..., i_1)$ to $(i'_L, ..., i'_1)$ in one step.

Many sets of interest are fixpoints. For example, the reachable state space $\mathcal{X}_{rch}$ is the smallest fixpoint of $\mathcal{X} = \mathcal{X}_{init} \cup \mathcal{X} \cup \mathcal{T}(\mathcal{X})$, where $\mathcal{X}_{init}$ is the set of initial states and $\mathcal{T}(\mathcal{X})$ is the image of $\mathcal{X}$ through the transition relation $\mathcal{T}$.

Potential state space $\mathcal{X}_{pot}$:
$\{0,1,2,3\} \times \{0,1,2\} \times \{0,1\} \times \{0,1,2\}$

Set $\mathcal{X}$ encoded by either MDD:
$\{0210, 1000, 1010, 1100, 1110,$
$1210, 2000, 2010, 2100, 2110,$
$2210, 3010, 3110, 3200, 3201,$
$3202, 3210, 3211, 3212\}$

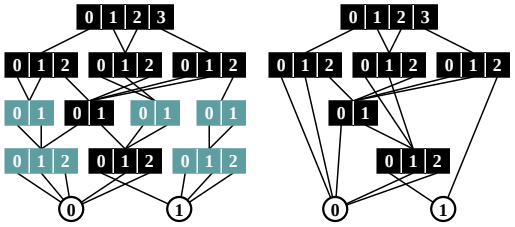

Fig. 1. A set of states encoded as a quasi-reduced (left) or fully-reduced (right) MDD.

Fixpoints also play a key role in CTL [11]. A CTL state formula is inductively defined as: $\phi := \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{A}\,\varphi \mid \mathsf{E}\,\varphi$, where $\top$ indicates the "true" proposition that holds in every state, a is an atomic proposition that may hold in each particular state, and φ is a path formula of the form $\varphi := \mathsf{X}\,\phi \mid \phi_1\,\mathsf{U}\,\phi_2$, where ϕ, ϕ_1, and ϕ_2 are state formulas. Intuitively, s satisfies $\mathsf{A}\,\varphi$ if φ holds on all paths starting at s, while it satisfies $\mathsf{E}\,\varphi$ if φ holds on at least one path starting at s; for path formulas, $\mathsf{X}\,\phi$ holds on a path σ if the next state on σ satisfies ϕ, while it satisfies $\phi_1\,\mathsf{U}\,\phi_2$ if there is a state s on σ that satisfies ϕ_2 and all the states before s on σ satisfy ϕ_1. In CTL, the set of states $\mathcal{X}_{\mathsf{E}a\mathsf{U}b} = \{i : \exists d > 0, \exists i^{(1)} \rightarrow i^{(2)} \rightarrow \cdots \rightarrow i^{(d)} \wedge i = i^{(1)} \wedge i^{(d)} \in \mathcal{B} \wedge \forall c, 1 \leq c < d, i^{(c)} \in \mathcal{A}\}$ is the *smallest* fixpoint of $\mathcal{X} = \mathcal{B} \cup (\mathcal{A} \cap \mathcal{T}^{-1}(\mathcal{X}))$, where $\mathcal{T}^{-1}$ is the *backward* transition relation and $\mathcal{A}$ and $\mathcal{B}$ are the states satisfying a or b, respectively, while $\mathcal{X}_{\mathsf{EG}a} = \{i : \forall d > 0, \exists i^{(1)} \rightarrow i^{(2)} \rightarrow \cdots \rightarrow i^{(d)} \wedge i = i^{(1)} \wedge \forall c, 1 \leq c \leq d, i^{(c)} \in \mathcal{A}\}$ is the *largest* fixpoint of $\mathcal{X} = \mathcal{A} \cap \mathcal{T}^{-1}(\mathcal{X})$.

Consider now, for example, the fixpoint equation for state-space generation. Its obvious symbolic implementation initializes $\mathcal{X}$ to $\mathcal{X}_{init}$, iteratively computes $\mathcal{T}(\mathcal{X})$, and adds the resulting set of states to $\mathcal{X}$, until no new states are found, i.e., $\mathcal{X}$ does not change. This *breadth-first* search, requiring d_{max} iterations (each one performing a relational product and a set union), where d_{max} is the largest distance of any reachable state from the initial states, has been experimentally found to suffer from a large *peak size*. In other words, while the final result $\mathcal{X}_{rch}$ might have a compact MDD representation, some intermediate $\mathcal{X}$, corresponding to the reachable states at distance up to d from $\mathcal{X}_{init}$, might require a huge MDD.

The *saturation* algorithm [9], instead, *disjunctively* decomposes $\mathcal{T}$ into a set $\{\mathcal{T}_k : L \geq k \geq 1\}$ of asynchronous events, where $\mathcal{T}_k$ only depends and affects state variable v_k and possibly some of $\{v_{k-1}, ..., v_1\}$. Then, instead of using "global" relational products on the entire MDD, saturation computes "local" fixpoints on the nodes of the initial MDD. More specifically, it computes the fixpoint with respect to $\mathcal{T}_1$ of the nodes associated with v_1 in the MDD encoding $\mathcal{X}_{init}$; then, it computes the fixpoint with respect to $\mathcal{T}_2$ of the nodes associated with v_2, with the proviso that any newly-created node associated with v_1 is immediately saturated by computing its fixpoint with respect to $\mathcal{T}_1$, and so on, until the root node associated with v_L is saturated, at which point it encodes the desired result. This approach tends to create many fewer MDD nodes, each of which is saturated, thus has at least a chance of being in the final MDD (which, encoding a fixpoint, contains only saturated nodes by definition). Empirically, saturation

$$
\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. 2. The syntax of ECA rules.

has been shown to build much smaller peak MDDs, sometimes by many orders of magnitude, in turn leading to enormously faster runtimes.

3 A language to describe ECA rules

We now present the syntax and the semantics of our language to define ECA rules and illustrate them with a running example.

3.1 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* are 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 [2,4,7,17,20,21]. 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. 2).

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 their value to an expression (absolute change), or increase or decrease it by an expression (relative change); 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 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 [16] 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 rules r_a and r_{ac}, closing 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.

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

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.

3.2 Running example

We illustrate the expressiveness of our ECA rules on a running example: a light control subsystem in a smart home for senior housing. Fig. 3 lists the requirements in natural language (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*.

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 (r_2). If the timer reaches 360, internal event *LgtsOff* is activated to turn off the lights and to reset *lgtsTmr* to 0 (r_3). Variable *intLgts* acts as an actuator control to adjust the internal light intensity.

We use internal events to model events not observable outside. *LgtsOff*, activated by r_3 or r_7, turns the lights off and activates an outdoor light intensity check through *ChkExtLgt* (r_4). *ChkExtLgt* activates *LgtsOn* if $lExtLgt \leq 5$ (r_5). *ChkSlp* is activated by r_8 to check whether a person is asleep, in which case it triggers an action to turn the lights off (r_9). *ChkMtn*, activated by r_6, activates *LgtsOff* if the room is empty and lights are on, or if the occupant is asleep (r_7).

4 Transforming a set of ECA rules into a PN

We now explain how to transform a set of ECA rules into a PN. First, we put each ECA rule into a "regular" form where both the *events* and the *condition* portion of the rule are disjunctions of conjunctions (of events or relational expressions, respectively). All rules of the smart home example in Fig. 3 are in this form. While this transformation may in principle grow exponentially large, each ECA rule usually contains a small number of events and conditions, hence it is not a problem in practice. Due to our 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 in a variable's value 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. 3 are transformed into the PN of Fig. 4 (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

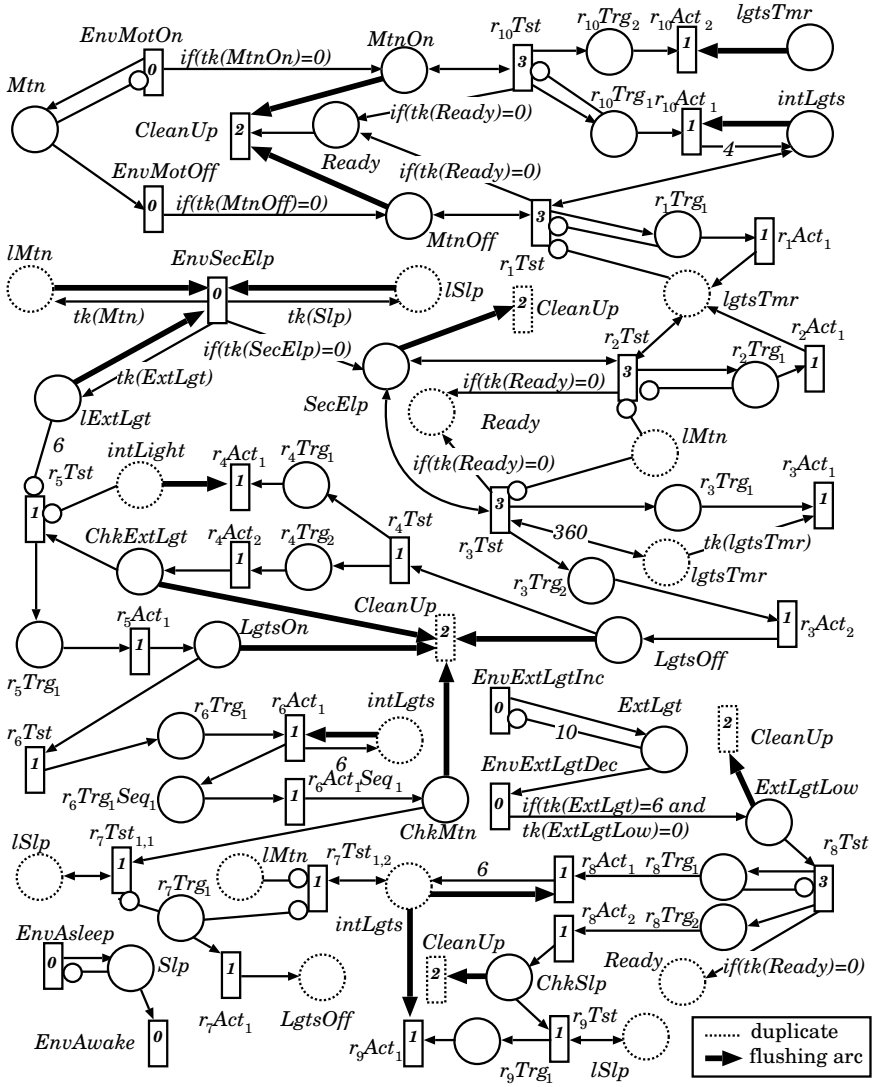

Fig. 4. The PN for ECA rules in Fig. 3.

the nondeterministic asynchronous nature of these changes. For example, in Fig. 4, 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 one conjunct in *events* and one in *condition*, respectively, 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 puts 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 and 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

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 $Ready$ $-[1]\rightarrow$ $CleanUp$
4 foreach event $e \in \mathbf{E}_{ext} \cup \mathbf{E}_{int}$ do
5 create place p_e and p_e $-[tk(p_e)]\rightarrow$ $CleanUp$
6 create place p_v, for each variable $v \in \mathbf{V}_{loc}$
7 foreach 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 $0]\rightarrow$ p_v
10 create t_{vInc} $-[1]\rightarrow$ p_v and p_v $-[v_{max}]-\circ$ t_{vInc}
11 foreach 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 $0]\rightarrow$ 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 $0]\rightarrow$ t_{vInc}
15 create t_{vInc} $-[$if$(tk(p_v)$ op $val)tk(p_v)$ else $0]\rightarrow$ $p_{v'}$
16 foreach 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 $0]\rightarrow$ 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 $0]\rightarrow$ t_{vDec}
20 create t_{vDec} $-[$if$(tk(p_v)$ op $val)tk(p_v)$ else $0]\rightarrow$ $p_{v'}$
21 foreach 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 $0]\rightarrow$ p_e
23 if e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ then
24 create $p_{v'}$ $-[tk(p_{v'})]\rightarrow$ t_e and t_e $-[tk(p_v)]\rightarrow$ $p_{v'}$
25 foreach 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 foreach event e in disjunct i do
28 create p_e $-[1]\rightarrow$ $r_k\,Tst_{i,j}$
29 create $r_k\,Tst_{i,j}$ $-[1]\rightarrow$ p_e, if $e \in \mathbf{E}_{ext}$
30 foreach conjunct $v \leq val$ or $v = val$ in disjunct j do
31 create p_v $-[val+1]-\circ$ $r_k\,Tst_{i,j}$
32 foreach conjunct $v \geq val$ or $v = val$ in disjunct j do
33 create p_v $-[val]\rightarrow$ $r_k\,Tst_{i,j}$ and $r_k\,Tst_{i,j}$ $-[val]\rightarrow$ p_v
34 if actions A is "$(A_1$ *par* $A_2)$" then $n_a = 2$;
35 else $n_a = 1$, $A_1 = A$;
36 foreach $l \in [1, n_a]$ do
37 create places $r_k\,Trg_l$ and transitions $r_k\,Act_l$ with priority p
38 create $r_k\,Trg_l$ $-[1]\rightarrow$ $r_k\,Act_l$ and $r_k\,Tst_{i,j}$ $-[1]\rightarrow$ $r_k\,Trg_l$
39 $SeqSubGraph(A_l$, "$r_k Acl_l$", $l, p)$
40 foreach $r_k \in \mathbf{R}_{ext}, i \in [1, n_d], j \in [1, n_c]$ do
41 create $r_k\,Tst_{i,j}$ $-[$if$(tk(Ready) = 0)1$ else $0]\rightarrow$ $Ready$ and $r_k\,Trg_1$ $-[1]-\circ$ $r_k\,Tst_{i,j}$

Fig. 5. Transforming ECA rules into a PN: a $-[k]\rightarrow$ b means "an arc from a to b with cardinality k; a $-[k]-\circ$ b means "an inhibitor arc from a to b with cardinality k.

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 Translating ECA rules into a PN

The algorithm in Fig. 5 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

```
ParSubGraph(Pars, Pre, p)                    • Pars: parallel actions, Pre: prefix
 1  foreach 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. 6. 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. 7. Processing **seq**.

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 also 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 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 25-41). 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, one for parallel actions in Fig. 6 and the other for sequential actions in Fig. 7. Procedure *SeqSubGraph* first tests all atomic actions, such as "set", "increase", "decrease", and "activate". Then, it recursively calls *ParSubGraph* at Line 10 if it encounters

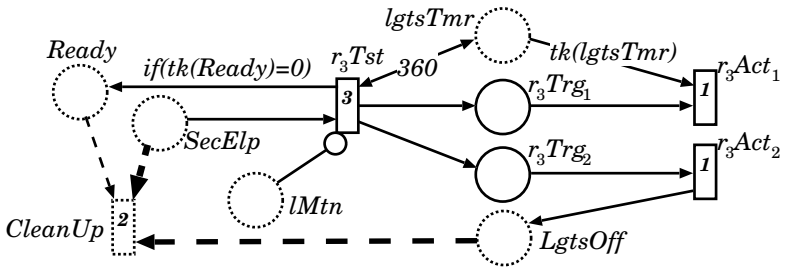

Fig. 8. The PN corresponding to rule r_3

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.

We use rule r_3: **on** *SecElp* **if** (*lgtsTmr* = 360 **and** *lMtn* = 0) **do** (**set**(*lgtsTmr*,0) **par activate** (*LgtsOff*)), with default priority $P = 1$, to better illustrate how to transform a rule into our model. Fig. 8 shows the resulting PN. Event places *SecElp* and *LgtsOff*, variable places *lMtn* and *lgtsTmr*, control place *Ready*, and control transition *CleanUp*, with priority $P + 1 = 2$, are created during the pre-processing step. The subnets for the external event occurring phase and the event cleanup are also created then. These are shown as dotted places and transitions in Fig. 8.

Test transition r_3Tst with priority $P + 2 = 3$ is created for the triggering phase. Place *SecElp* is connected to r_3Tst with an input arc to represent the occurrence of event *SecElp*, while place *lMtn* is connected to r_3Tst with an inhibitor arc and *lgtsTmr* is connected to r_3Tst with both an input and an output arc, with cardinality 360, to represent r_3's triggering condition: $lgtsTmr = 360$ **and** $lMtn = 0$. The PN semantics naturally implements the conjunction of events and conditions. r_3Tst is connected to control place *Ready* with a self-modifying arc having cardinality $if(tk(Ready) = 0)$ to indicate the end of a batch, which enables transition *CleanUp*, that in turn clears up all the tokens in the event places (with the corresponding reset arcs). r_3 has two parallel actions, thus there are two places r_3Trg_1 and r_3Trg_2 that contain a token when r_3 is triggered. Then, two action transitions (r_3Act_1 and r_3Act_2 with priority $P = 1$) are needed for the performing phase. After the event cleanup, when transition *CleanUp* with priority 2 fires, only action transitions (having priority 1) are enabled. r_3Trg_1 is connected to r_3Act_1 and *lgtsTmr* to r_3Act_1 with an arc having cardinality $tk(lgtsTmr)$ to model the action **set** $(lgtsTmr, 0)$. r_3Trg_2 is connected to r_3Act_2 and r_3Act_2 to *LgtsOff* to model action **activate** (*LgtsOff*).

The translation process we just described generates a PN from a set of ECA rules augmented with an interpretation of the external environment where the value of environmental variables can change asynchronously, nondeterministically, and independently of the values of variables used to define the ECA rules. Because the PN contains additional machinery needed to enforce the semantics

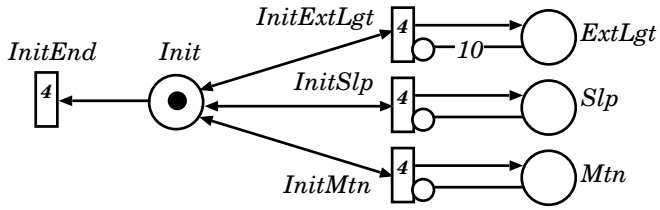

Fig. 9. The initialization phase for the smart home example.

of ECA rules (such as the *CleanUp* transition), the two models are only weakly bisimilar, but this is enough to preserve CTL properties [13]

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 PNs, 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* 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. 9 shows this subnet for our running example. We initialize the PN 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 real 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. Having to perform just one state space generation is obviously enormously better.

```
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
 6  if p = 1 then return 1;
 7  if CacheLookUp(ExtractUnprimedCode, p, r) return r;
 8  foreach i ∈ V_p.v do
 9    mdd r_i ← 0;
10    if p[i] ≠ 0 then          • p[i] is the node pointed edge i of node p
11      foreach j ∈ V_p.v s.t. p[i][j] ≠ 0 do
12        r_i ← Union(r_i, ExtractUnprimed(p[i][j]))
13  mdd r ← UniqueTableInsert({r_i : i ∈ V_p.v});
14  CacheInsert(ExtractUnprimedCode, p, r);
15  return r;
```

Fig. 10. Algorithms to verify the termination property.

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 [8].

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, we can again avoid the need for an explicit enumeration, this time, of event batches.

Definition 1. A set $\mathcal{G}$ of ECA rules satisfies termination if no infinite sequence of internal events can be triggered in any possible execution of $\mathcal{G}$. □

In CTL, this can then be expressed as $\neg\mathsf{EF}(\mathsf{EG}(\mathit{unstable}))$, which states that there is no cycle of unstable states reachable from an initial, thus stable, state. Since we observed at the end of Sect. 4.4 that the PN obtained by the translation process is weakly-bisimilar to the ECA rules (plus a representation of the environment), and since weak-bisimilarity preserves CTL properties, termination can be verified directly on the PN. Both traditional breadth-first-search (BFS) and saturation-based [25] algorithms are suitable to compute the EG operator. Algorithm *Term* in Fig. 10 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 events and environmental variable changes into $\mathcal{N}_{ext}$. Thus, the internal transitions are $\mathcal{N}_{int} = \mathcal{N} \setminus \mathcal{N}_{ext}$. After generating the state space $\mathcal{S}_{rch}$ using constrained saturation [24], we build the set of states $\mathcal{S}_{unst}$ by symbolically intersecting $\mathcal{S}_{rch}$ with the unprimed, or

```
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
 4      return false;                                  • provide error trace
 5  return true;
```

```
bool ConfExplicitImproved(mdd S_st, mdd S_unst, mdd2 N_int, mdd2 N)

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

Fig. 11. Explicit algorithms to verify the confluence property.

"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.

Definition 2. 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. □

Just like termination, confluence can also be expressed in CTL, although in an extremely cumbersome way that we mention simply because, again, this guarantees that it is correct to verify it on the PN. Specifically, if we assume that each stable state s_i is the only one satisfying a special additional atomic proposition p_i, we can express confluence as the (enormous) conjunction over all stable states s_1, s_2, and s_3, with $s_2 \neq s_3$, of the CTL formulas

$$\neg\left(p_1 \wedge \mathsf{EX}(\mathit{unstable} \wedge \mathsf{E}\ \mathit{unstable}\ \mathsf{U}\ p_2) \wedge \mathsf{EX}(\mathit{unstable} \wedge \mathsf{E}\ \mathit{unstable}\ \mathsf{U}\ p_3)\right).$$

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

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

Fig. 12. Fully symbolic algorithm to verify the confluence property.

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 nondeterministic execution in the performing phase is the reason a system may violate confluence, checking the evolution from unstable states suffices.

The brute force algorithm *ConfExplicit* in Fig. 11 enumerates unstable states and generates reachable states only from unstable states using constrained saturation [24]. Then, it counts the stable states in the obtained set. We observe that, starting from an unstable state 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 **i** and, after generating the states reachable from **i** and verifying that they include only one stable state, it excludes all visited unstable states (Line 11). Furthermore, it starts only from states **i** in the frontier, i.e., unstable states reachable in one step from stable states (all other unstable reachable states are by definition reachable from this frontier). However, we stress that these, as for all symbolic algorithms, are heuristics, so they are not guaranteed to work better than simpler approaches.

Fig. 12 shows a fully symbolic algorithm to check confluence, It first generates the transition transitive closure set from $\mathcal{N}_{int}$ using constrained saturation [25], 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

Termination (time: sec, memory: MB)										
Model	$\lvert\mathcal{S}_{rch}\rvert$	BFS			Saturation					
		T_b	M_p^b	M_f^b	T_t	M_p^t	M_f^t	T_{sc}	M_p^{sc}	M_f^{sc}
PN_t	$2.66 \cdot 10^6$	8.11	91.85	35.28	0.01	36.25	6.52	7.54	48.41	39.49
PN_c	$2.61 \cdot 10^6$	0.01	31.85	4.28	0.03	32.41	4.48	0	0	0
PN_1	$8.99 \cdot 10^6$	7.98	96.23	36.03	0.04	41.44	6.62	8.08	51.68	41.99
PN_2	$1.78 \cdot 10^7$	15.55	183.04	70.45	0.04	64.00	8.97	16.49	90.14	75.97
PN_3	$2.61 \cdot 10^7$	106.29	554.29	291.11	0.04	155.55	17.29	81.04	356.04	354.94
PN_4	$5.02 \cdot 10^7$	39.33	118.88	41.21	0.04	67.69	8.08	14.54	74.77	58.95

Confluence (time: min, memory: GB, –: out of memory)							
Model	$\lvert\mathcal{S}_{rch}\rvert$	Best Explicit			Symbolic		
		T_{be}	M_p^{be}	M_f^{be}	T_s	M_p^s	M_f^s
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 of verifying the ECA rules for a smart home in Fig. 3.

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 the transitive closure is expensive [25], this approach avoids separate searches from distinct unstable states and is particularly appropriate when $\mathcal{S}_{unst}$ is large.

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 each method. For termination, it shows the time used to verify the property (T_t) and to find the shortest counterexample (T_{sc}). 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 in Fig. 3, and fails the termination check. Even though the state space is not very large, the saturation-based counterexample generation (T_{sc}) is computationally expensive [26] and consumes most of the runtime. The minimal counterexample generated by SMART has a long tail consisting of 1,863 states leading to the 10-state

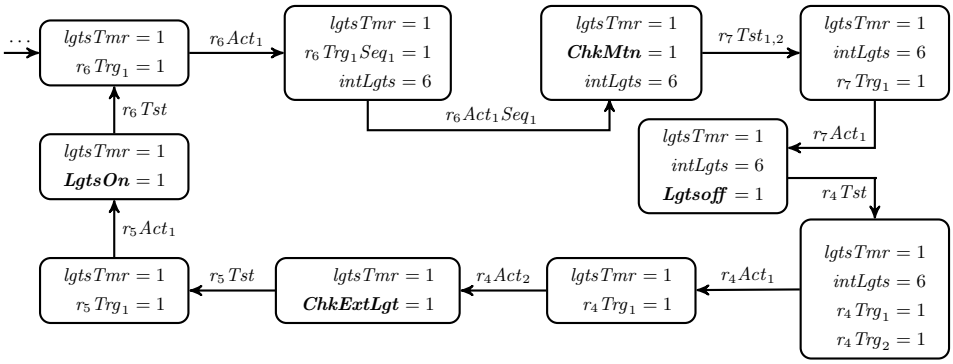

Fig. 13. A termination counterexample (related to rules r_4 to r_7).

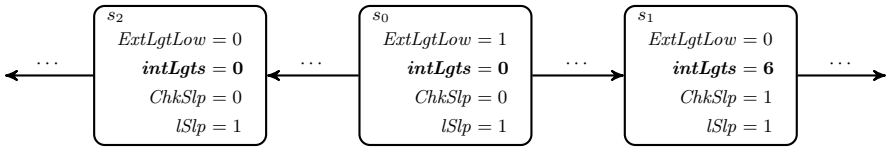

Fig. 14. A confluence counterexample (related to rules r_8 and r_9).

cycle of Fig. 13 (only the non-empty places are listed for each state, and edges are labeled with the corresponding PN transition). The BFS-based algorithm instead first computes set $\mathcal{S}_{EG}$ satisfying $\mathsf{EG}(\mathcal{S}_{unst})$, then it randomly chooses a state s_s in $\mathcal{S}_{EG}$, forms an EG witness [26] by randomly selecting successors also in $\mathcal{S}_{EG}$, and generates an EF shortest trace from $\mathcal{S}_{init}$ to s. This does *not* result in an overall shortest counterexample but, with a reasonable number of attempts, a counterexample with 1,883 states, 12 in the final cycle, is found.

Analyzing the shortest trace provided by the saturation-based algorithm, 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 must be investigated. Among the possible modifications, we choose to replace 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 that it satisfies termination. The experimental results for PN_c confirm that verifying termination does not consume much time for either BFS or saturation-based algorithms. However, for PN_t, the saturation-based algorithm finds a minimal counterexample in less time than the random BFS-based algorithm, which does not ensure minimality. In practice, minimal counterexamples are desirable when debugging.

We then run SMART on PN_c to verify confluence, and find 72,644 bad states. Fig. 14 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). Recall rule r_8: **on** *ExtLgtLow* **do** (**set** (*intLgts*, 6) **par activate** (*ChkSlp*) and rule r_9: **on** *ChkSlp* **if** (*lSlp* = 1) **do set** (*intLgts*, 0), in Fig. 3. We correct and replace 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 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_2 to **on** *SecElp* **if** ((*lgtsTmr* $\geq$ 1 **and** *lgtsTmr* $\leq$ 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 $[0, 3600]$. For termination, the saturation-based algorithm behaviors much better on the two largest models PN_3 and PN_4 for both time and peak memory consumption. For PN_2 the BFS algorithm happens to a minimal counterexample using less time, but at a cost of twice the peak memory consumption. For smaller models, the requirements for both algorithms are quite acceptable. However, we emphasize that the saturation algorithm guarantees counterexample minimality. We observe that, when verifying 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 the transitive closure is expensive. However, as the state space grows, enumerating the unstable states becomes unfeasible. 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 the original explicit approach, and 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 observe that larger state spaces might require fewer resources. With symbolic methods, this might happen if the corresponding MDD is more regular than the one for a smaller state space.

7 Conclusion

Verifying ECA rule bases for reactive systems is challenging due to their highly concurrent and nondeterministic nature. We proposed an approach that transforms an ECA rule base into a self-modifying Petri net with inhibitor arcs and priorities, then uses symbolic verification algorithms. Our approach is general enough to give precise answers to questions about several properties, certainly those that can be expressed in CTL, such as *termination*.

More importantly, we also presented a symbolic *confluence* algorithm, a property that is much more difficult to verify. In principle, this requires us to enumerate "stable" states and show that, from each of them, exactly one stable state can be reached when one or more rules are triggered, even if the sequence of "unstable" states that the system traverses while processing these rules exhibits nondeterminism due to the interleaving of asynchronously triggered actions. 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.

Acknowledgment

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-1442586.

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. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 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. In *Toward a Human-Friendly Assistive Environment*, volume 14, pages 106–113, 2004.
4. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems*, volume 985 of *LNCS*, pages 163–181. Springer, 1995.
5. E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM Transactions on Database Systems*, 25(3):269–332, Sept. 2000.
6. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.
7. 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.
8. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. In *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 78–97. Springer, 2003.

9. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, 2001.
10. G. Ciardo, Y. Zhao, and X. Jin. Ten years of saturation: a Petri net perspective. *Transactions on Petri Nets and Other Models of Concurrency*, V:51–95, 2012.
11. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
12. S. Comai and L. Tanca. Termination and confluence by rule prioritization. *IEEE Transactions on Knowledge and Data Engineering*, 15:257–270, 2003.
13. T. French. Decidability of propositionally quantified logics of knowledge. In T. Gedeon and L. Fung, editors, *AI 2003: Advances in Artificial Intelligence*, volume 2903 of *Lecture Notes in Computer Science*, pages 352–363. Springer Berlin Heidelberg, 2003.
14. T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
15. K. G. Kulkarni, N. M. Mattos, and R. Cochrane. Active database features in SQL3. In *Active Rules in Database Systems*, pages 197–219. Springer, New York, 1999.
16. E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, University of California, Berkeley, May 2007.
17. X. Li, J. M. Marín, and S. V. Chapa. A structural model of ECA rules in active database. In *Proc. of the Second Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*, MICAI '02, pages 486–493. Springer-Verlag, 2002.
18. D. McCarthy and U. Dayal. The architecture of an active database management system. *ACM Sigmod Record*, 18(2):215–224, 1989.
19. T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.
20. D. Nazareth. Investigating the applicability of Petri nets for rule-based system verification. *IEEE Trans. on Knowledge and Data Engineering*, 5(3):402–415, 1993.
21. I. Ray and I. Ray. Detecting termination of active database rules using symbolic model checking. In *Proc. of the 5th East European Conference on Advances in Databases and Information Systems*, pages 266–279. Springer-Verlag, 2001.
22. R. Valk. Generalizations of Petri nets. In *Mathematical foundations of computer science*, LNCS 118, pages 140–155. Springer, 1981.
23. D. Varró. A formal semantics of uml statecharts by model transition systems. In *Processings of ICGT 2002: International Conference on Graph Transformation*, pages 378–392. Springer, 2002.
24. Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proc. ATVA*, LNCS 5799, pages 368–381. Springer, 2009.
25. Y. Zhao and G. Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7(2):141–150, 2011.
26. Y. Zhao, J. Xiaoqing, and G. Ciardo. A symbolic algorithm for shortest EG witness generation. In *Proc. TASE*, pages 68–75. IEEE Comp. Soc. Press, 2011.