Received 25 April 2016
Accepted 27 June 2016

Formal analysis and verification support for reactive rule-based Web agents

Katerina Ksystra and Petros Stefaneas
National Technical University of Athens, Athens, Greece

Abstract

Purpose – Reactive rules are used for programming rule-based Web agents, which have the ability to detect events and respond to them automatically and can have complex structure and unpredictable behavior. The aim of this paper is to provide an appropriate formal framework for analyzing such rules.

Design/methodology/approach – To achieve this goal, the authors give two alternative semantics for the basic reactive rules' families which allow us to specify reactive rule-based agents and verify their intended behavior. The first approach expresses the functionality of production and event condition action rules in terms of equations, whereas the second methodology is based in the formalism of rewriting logic. Both semantics can be expressed within the framework of CafeOBJ algebraic specification language, which then offers the verification support and have their advantages and downsides.

Findings – The authors report on experiences gained by applying those methodologies in a reactive rule-based system and compare the two methodologies.

Originality/value – Finally, the authors demonstrate a tool that translates a set of reactive rules into CafeOBJ rewrite rules, thus making the verification of reactive rules possible for inexperienced users.

Keywords CafeOBJ, Formal analysis, Reactive rules, Verification, Web agents

Paper type Research paper

1. Introduction

Reactivity on the Web is the ability to detect events and respond to them automatically in a timely manner through reactive programs. Reactivity plays an important role for upcoming Web systems such as online marketplaces, adaptive Web and Semantic Web agents and Web services (Bry *et al.*, 2006).

Such behavior is needed for bridging the gap between the existing, passive Web, where data sources can only be accessed to obtain information, and the dynamic Web, where data sources are enriched with reactive behavior (Berstel *et al.*, 2007). The need for changing and updating data on the Web has many reasons. New information comes in, calling for insertions of new data; information is out-of-date, calling for deletions and replacements of data. Such changes need to be mirrored by other Web systems whose data depend on the initial changes (Bry *et al.*, 2006).

Reactivity can be specified and realized by means of reactive rules. This has led to an increase in the development of the so-called reactive rule-based Web agents and their use in programming critical software systems.

Web agents can be defined as intelligent systems that carry out some set of operations on behalf of a user or another program, with some degree of independence or

International Journal of Web Information Systems
Vol. 12 No. 4, 2016
pp. 418-447

1744-0084
DOI 10.1108/IJWIS-04-2016-0024

autonomy, and, in so doing, use some knowledge or representation of the user's goals or desires (Gilbert).

The use of rule-based systems as the main method of implementing such agents has been proposed from the beginning. In this approach, each agent includes a rule engine and is able to perform rule-based inference (Badica *et al.*, 2011). Thus, a Web agent is called rule-based if its behavior and knowledge are expressed by means of rules.

In more details, in the case of rule-based Web agents used in this work, we adopt the definition of (Stavropoulos *et al.*, 2015), where the system's application layer hosts agents that automatically monitor and manage the infrastructure. Each agent integrates a knowledge base (KB) and a reasoning engine. The KB is filled with facts about the world, i.e. measured values of the environment, and also contains policies in the form of rules, entered by expert human users. After collecting the facts at each cycle, the agent performs reasoning to decide on a proper set of actions, according to the policies.

Reactive rule-based systems (i.e. Web agents whose behavior is defined through reactive rules) are an attractive approach, as they enable systems to react to events, or combinations of events, occurring in an arbitrary order. Additional characteristics supported by reactive rules, such as flexibility and expressivity, are highly desired especially when modeling industrial systems.

However, because of the ability of rules to interact during execution, such systems often present unpredictable behavior, and, thus, the task of analyzing their behavior can become difficult. Changing, introducing or removing a single rule from a rule base, for example, can have undesirable side effects (e.g. making the KB of the system inconsistent). For this reason, the extensive and formal analysis of reactive rule-based systems is required. This need becomes stronger when the system is complex or used in critical domains. Finally, the existing tool support for reasoning about such systems is limited. To address this shortage of reasoning and verification support for reactive rules, in this paper, we present a framework that:

- transforms a set of production and a set of event condition action (ECA) rules (i.e. the main reactive rules' families) into equational rules, written in CafeOBJ;
- transforms a set of production and a set of ECA rules into rewrite rules, written in CafeOBJ;
- provides reasoning about the specified rule-based system;
- verifies safety properties of the rules;
- detects termination and confluence errors of the rules; and
- provides a clear understanding of the specified rule-based system by simulating the execution behavior of the rules.

The rest of the paper is organized as follows: in Subsection 1.1, we briefly present related work and discuss differences with our approach. Section 2 describes the theoretical foundations of the methodologies. In Section 3, we present the proposed frameworks which formally express reactive rules as rewrite and equational transition rules (in CafeOBJ). In Section 4, we apply the proposed methodologies in a case study. In Section 5, we compare the two approaches and report on some lessons learned from this attempt to bring closer two different research areas, one of formal methods and second of reactive rules. Finally, we demonstrate a tool that automates the transformation

process from reactive into rewrite rules in Section 6. The last section concludes the paper and discusses future directions.

1.1 Related work

Recent approaches related to the application of formal methods for analyzing rule-based systems include the following. In Berstel and Leconte (2010), the authors propose a constraint-based approach to the verification of rule programs. They present a simple rule language, describe how to express rule programs and verification properties into constraint satisfiability problems and discuss some challenges of verifying rule programs using a constraint-based programming solver that derives from the fact that the domains of the input variables are commonly very large. Finally, they present how to detect structure properties of a simple rule-based system. In Jin *et al.* (2013), the authors analyze the behavior of ECA rules by translating them into an extended Petri net and verify termination and confluence properties of a light control system expressed in terms of ECA rules. Also, in Jin *et al.* (2014), the same authors extend their previous work with tool support and counterexamples to help debugging. Ericsson *et al.* (2008) present an approach to verify the behavior of ECA rules where a tool that transforms such rules to timed automata is developed. Then, the Uppaal tool is used to prove desired safety properties for an industrial rule-based application.

In Boukhebouze *et al.* (2011), authors present a rule-based approach which is built upon the ECA model and supported by a rule-based business process definition language. In this approach rules, which specify business processes, are represented using the Event-Condition-Action-Post-Condition-Event (ECAPE) model. This allows translating a process into a graph of rules that is used to check how flexible a business process is and estimating this process's cost of changes. In addition, the ECAPE model allows the translation of a process into a colored Petri net, called ECAPE net, to verify process functioning prior to any deployment. In Lukichev (2011), authors present a declarative approach to rule verification. They consider several anomalies, which may appear in rule bases with production rules and semantic constraints. The presented approach defines verifier rules, which derive facts when anomalous business rules are detected.

Our approach for the verification of rule programs is based on a different formalism; in particular, it uses the CafeOBJ system (Diaconescu *et al.*, 2003) and expresses the functionality of reactive rules in both equational and rewriting logic. CafeOBJ has been successfully applied for the verification of various complex systems. To the best of our knowledge, this is the first time it is used in the area of reactive rules. One motivation for this work was a recent advancement in the field and, in particular, the methodology to theorem prove rewrite theories (Ogata and Futatsugi, 2014). Compared to existing similar approaches, it has the following contributions:

- First, compared to Jin *et al.* (2013), where structure errors are formally analyzed, our methodology can be used for the analysis of both structure (confluence and termination) and safety properties for the specified rule system.
- Second, compared to (Berstel and Leconte, 2010; Lukichev, 2011), our approach supports both production and ECA rules and also provides tool support for transforming such rules into rewrite rule specifications in CafeOBJ.

- Third, when proving safety properties, both model checking and theorem proving techniques can be applied, in contrast to Berstel and Leconte (2010); Jin *et al*, 2013, 2014; Ericsson *et al.* (2008); and Boukhebouze *et al.* (2011), where only model checking support is provided. The combination of these two proving methods provides strong verification power. Model checking can be used to search the system for a state where the desired invariant property is violated (counter example), and, next, if no such state is discovered, theorem proving techniques can be applied to ensure that the system preserves the property in any reachable state. In this way, infinite state systems can be specified. Also, CafeOBJ and Maude (Clavel *et al.*, 1999) allow inductive data structures in state machines to be model checked, and few model checkers exist with this feature.
- Finally, our approach can be used for the specification and verification of complex systems due to the simplicity of the CafeOBJ language and its natural affinity for abstraction (Diaconescu *et al.*, 2003).

2. Preliminaries

2.1 Reactive rules

The two basic reactive rules' families and most commonly used for developing software systems are production rules and ECA rules.

A production rule is a statement of rule programming logic, which specifies the execution of an action in case its conditions are satisfied, i.e. production rules react to states changes. Their essential syntax is *if Condition do Action* (*if* Ci *do* Ai). Some usual actions supported by rule markup languages are: add, retract and update knowledge or generic actions with external effects, such as assignment of specific values in variables.

In contrast to production rules, ECA rules define an explicit event part which is separated from the conditions and actions of the rule. Their essential syntax is *on Event if Condition do Action* (*On* Ei *if* Ci *do* Ai). The ECA paradigm states that a rule autonomously reacts to actively or passively detected simple or complex events by evaluating a condition (or a set of conditions) and by executing a reaction whenever the event happens and the condition is true (Paschke, 2005).

The events of ECA rules can be combinations of atomic events activated by environmental or internal changes, and based on that, they are usually classified as external and internal. These changes are captured by environmental and local variables. More precisely, in Jin *et al.* (2013), external events are produced by sensors monitoring environment variables. This means that environmental variables are used to represent environment states that can be measured by sensors but not directly modified by the system. In this way, environmental variables capture the non-determinism introduced by the environment. Instead, local variables can be both read and written by the system. An external event can be activated when the value of an environmental variable crosses a threshold; on the other hand, internal events can only be activated by the actions of ECA rules. 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.

The condition part of an ECA rule is a Boolean expression on the value of environmental and local variables. The last part of a rule specifies which actions must be performed. Most actions are operations on local variables which do not directly affect environmental variables. Thus, environmental variables are read-only from the

perspective of an action. Also, actions can activate internal events. Finally, to handle complex action operations, the execution semantics can be sequential or parallel.

An example of an ECA rule, expressed in natural language, can be seen here: "if the external light intensity of a house drops below 5, and the person inside the house is not asleep set the internal light intensity to 6". This rule is written according to the syntax of ECA rules as follows: *On* ExtLgtLow *if* (Slp = false) *do* set (intLgts, 6). This is an external rule, as the event of the rule is activated when the environmental variable that captures the changes of the external light intensity crosses a threshold.

We should mention at this point that we have chosen to give formal semantics to reactive rules expressed in a generic style (such as the example above) and not written in a specific rule markup language (e.g. Reaction RuleML Paschke *et al.*, 2012), because, in this way, more languages can be covered. Besides, in most cases, the semantics of the rules are independent of the syntax of the specific language. Also, many case studies and related work in the literature express the rules in this way, so it is easier to test the proposed methodologies.

2.2 CafeOBJ: basic syntax and notation

CafeOBJ is an algebraic specification language and can be used for the specification and verification of complex software systems.

The basic units of CafeOBJ are its modules. In CafeOBJ modules, we can declare module imports, sorts, operators, variables and equations. Sorts denote the data type of each module, and they are declared inside brackets (Ksystra *et al.*, 2014). Operators are declared with the keyword *op* (or *ops* if there are many). Operators without arguments are called constants. Operators can have attributes, such as *comm*, that specify that the binary operator is commutative. The constructor operators of the sorts are declared with the attribute *constr*. The non-constructor operators, or some properties of the operators, are defined in equations (which are declared using the keyword *eq*). Conditional equations can also be declared inside a module (using the keyword *ceq*). Finally, the CafeOBJ processor uses equations as left-to-right rewrite rules to compute (or reduce) a given term. For more details about CafeOBJ system, we refer the reader to Futatsugi *et al.* (2012).

CafeOBJ is an expressive language and contains built-in operators for denoting logical connectives such as negation, conjunction, disjunction, implication and exclusive disjunction. Logical quantifiers are also supported by the system. In CafeOBJ, the universal quantifier is handled by free variables, i.e. each equation $E(x)$ containing an unbound variable x is semantically equivalent to $\forall\, xE(x)$. The existential quantifier is not straightforwardly supported. However, each equation containing an existential quantifier can be transformed into its equivalent Skolem normal form without such quantifiers.

2.3 CafeOBJ, equational and rewriting logic

CafeOBJ supports both equational theory and rewrite theory specifications. State transitions are described in equations in the former and in rewriting rules in the latter. Equational theory specification is used for interactive theorem proving, whereas for rewrite theory specification, CafeOBJ can conduct exhaustive searches. In Ogata and Futatsugi (2014), an attempt to combine the above is presented. They describe a way to

theorem prove that rewrite theory specifications have invariant properties by proof score writing.

In equational logic, the *transitions* between the states of the system are modeled with constructor operators. Assuming that there exists a universal state space Y, each transition is a function $t : Y\,D_{t1} \ldots D_{tn} \rightarrow Y$ that changes the state of the system (where $D_{t1} \ldots D_{tn}$ are other data types that may be needed for the definition of the transition). Transitions may have an effective condition, which is declared as $c\text{-}t : Y\,D_{t1} \ldots D_{tn} \rightarrow Bool$. The meaning of this condition is that the transition can be effectively applied if its corresponding effective condition holds.

The structure of a state is abstracted by the observation operators (or *observers*), each one returning an observable information about the state. More precisely, each observer is a function, $o : Y\,D_{o1} \ldots D_{om} \rightarrow Do$, that takes as input a state of the system (and maybe other datatypes) and returns a data type value that characterizes the state.

The meaning of an observer is formally described by means of (conditional) equations, depending on whether the transition has an effective condition. These equations define how the value of each observer changes after the application of a transition rule.

Rewriting logic in CafeOBJ is based on a simplified version of Meseguer's (1992) rewriting logic for concurrent systems, which gives an extension of traditional algebraic specification toward concurrency. Rewriting logic (RWL) incorporates many different models of concurrency in a natural, simple and elegant way, thus giving CafeOBJ a wide range of applications. Unlike Maude (Clavel *et al.*, 1999), CafeOBJ design does not fully support labeled RWL, which permits full reasoning about multiple transitions between states, but supports reasoning about the existence of transitions between states (or configurations) of concurrent systems via a built-in predicate (denoted $==>$) with dynamic definition encoding both the proof theory of rewriting logic and the user-defined transitions. This predicate evaluates true whenever there exists a transition from the left-hand-side argument to the right-hand-side argument. More precisely, for a ground term t, a pattern p and an optional condition c, CafeOBJ can traverse all the terms reachable from t with respect to transitions in a breadth-first manner and find terms (called solutions) such that they are matched with p and c holds for them. This can be done using the command: `red t = (k,d) =>* p [suchThat c]`, where k is the maximum number of solutions and d is the maximum depth of search. Also, a natural number (id) is assigned to each term visited by a search, and, then by using the command `show path id`, a transition path to the term identified by id is displayed. Typically, the command is used to display a transition path to a solution found by a search from t (Ogata and Futatsugi, 2014).

In rewriting logic, states can be expressed as tuples of values `<a1, a2, b1, b2>` or as collections of observable values `(o1[p1]: a1) (o1[p2]: a2) (o2[p1]: b1) (o2[p2]: b2)` (soups), where observable values are pairs of (parameterized) names and values. The main difference between the two expressions is the following: when the states are expressed as tuples, the state expressions must be explicitly described on both sides of each transition. But, when expressing states as soups, only the observable values that are involved in the transitions need to be described on both sides of each transition. Of course, when specifying a dynamic system, special care has to be taken to adequately define the axioms that describe its domain, also known as the frame problem

(Hayes, 1971), so that effective reasoning can be conducted about its behavior. For more details about rewriting logic in CafeOBJ, we refer readers to (Ogata and Futatsugi, 2014).

3. Proposed frameworks

In this section, we describe our methodologies that formally specify reactive rules using two different logical formalisms: equational and rewriting logic.

Older versions of the above methodologies and definitions have been separately presented in Ksystra *et al.* (2014, 2012) and explained via illustrating examples. In a nutshell, in this paper, the following contributions have been made:

- We apply both the proposed methodologies in the same ECA rule-based intelligent system (commonly used in the related literature for verifying reactive rule agents), demonstrate their effectiveness and compare them in terms of their easiness in adoption from non-experts in the area of formal methods users.
- We make necessary additions in the definitions to cover cases discovered by the case study (e.g. event-memory and ECA rules definition or operator for detecting loops and simulating rules' execution).
- We discuss about the semantic relationship of the two formalizations.
- We define how complex actions can be expressed in our framework (in previous work, we presented how complex events can be detected).
- We develop a tool that translates reactive into CafeOBJ rewrite rules (after concluding that, in most cases, the rewriting approach is preferred).

3.1 Reactive rules and equational logic

Reactive rules can be expressed as state transition rules in equational logic, using the definitions below.

3.1.1 Production rules. A production rule can be naturally expressed in terms of equational logic if we map the action of the rule to a transition which has as effective condition the condition of the rule. The effects of the action are described through appropriate observers. As some actions correspond to changes of the KB, to describe them, we need an observer that will observe the KB at any given time. Thus, the observer $knowledge : Y \rightarrow Set\ of\ Facts$ which takes as input a state of the system and returns the set of facts that belong to the KB in that state is needed. For expressing the functionalities of the KB, the following operators are required; $/in : Fact\ Set\ of\ Facts \rightarrow Bool$, which returns true if an element belongs to the KB, $| : Fact\ Set\ of\ Facts \rightarrow Set\ of\ Facts$, which denotes that an element is added to the KB, and $/ : Set\ of\ Facts\ Fact \rightarrow Set\ of\ Facts$, which denotes that an element is removed from the KB.

3.1.1.1 Definition 1. Assume the state space Y and the following set of production rules: $\{\text{if } C_i \text{ do } A_i, i = 1, \ldots, n \in \mathbb{N}\}$, where without harm of generality we also assume that the conditions of the rules are disjoint. We define the following observers (O) and transitions (T) from this set of rules:

$$O = \{O' \cup knowledge\},$$

$$T = \{A_i\}.$$

In the above definition, O' denotes additional observers that may be used for the definition of generic actions and their side effects. Transitions are the actions of the

rules. As we mentioned before, they can be generic actions with external changes, $A_i : YD \rightarrow Y$ (where $D = D1, \ldots, Dn$ are data types that may be needed for the definition of the actions of a specific rule based system) or some of the predefined actions *assert* : *Y Fact* $\rightarrow$ *Y* (add a fact to KB), *retract* : *Y Fact* $\rightarrow$ *Y* (remove a fact from KB), *update* : *Y Fact Fact* $\rightarrow$ *Y* (remove/add a fact), with the semantics those given by rule markup languages (Delzanno *et al.*, 2010). The production rules are defined as transitions through the following steps:

- *Step 1*: If A_i is an assert action, its effect on the knowledge observer is defined as `knowledge(assert(u,ki,d1, . . .,dn)) = ki|knowledge(u) if Ci(u,d1, . . .,dn) = true`,

  where `ki` denotes the fact that is being asserted in the KB, `Ci(u,d1, . . .,dn)` is the condition of the production rule, `u` denotes a state of the system and `d1 . . .` and `dn` are variables of other datatypes that may be needed for the definition of the rule:
- *Step 2*: If A_i is a retract action, its effect on the knowledge observer is defined as `knowledge(retract(u,ki,d1, . . .,dn)) = knowledge(u)/ki if Ci(u,d1, . . .,dn) = true.`
- *Step 3*: If A_i is an update action, its effect on the knowledge observer is defined as `knowledge(update(ki,kj,d1, . . .,dn)) = kj|(knowledge(u)/ki) if Ci(u,d1, . . .,dn) = true.`
- *Step 4*: If A_i is a generic action, we define; `oi(Ai(u,d1, . . .,dn)) = vi if Ci (u,d1, . . .,dn) = true`, $o_i \in O$.

In more details; Step 1 states that the transition *assert*($u, k_i, d_1, \ldots, d_n$) is applied successfully in an arbitrary state u, and the fact k_i is added to the KB if the condition of the rule holds. In Step 2, it is stated that when the transition *retract*($u, k_i, d_1, \ldots, d_n$) is applied successfully in an arbitrary state u, the fact k_i is removed from the KB. When the transition *update*($u, k_i, k_j, d_1, \ldots, d_n$) is applied successfully in an arbitrary state u, *ki* is removed and *kj* is added, as Step 3 defines. Finally, Step 4 states that when we have the application of a generic action, we describe its effects using additional observers o_i that define how their values change when the action is applied successfully. If, for example, action A_i sets a specific value, say v_i to a variable, we state that the observer that corresponds to the variable will take the value v_i, when the action will be successfully applied.

We should mention here that in cases where the condition of the production rule does not hold (i.e. `if not Ci(u, d1, . . . , dn) = true`), the action of the rule will not be applied.

3.1.2 Event condition action rules. To express ECA rules as an equational logic theory, we need an observer that will remember the occurred events. For this reason, when an event is detected, its name is stored in the observer *event-memory*: $Y \rightarrow Name$, where *Name* is a sort denoting the names of the events. In this way, we can map events to transitions. The actions of ECA rules are assert, retract, update or generic actions and are mapped to transitions, as before. One key difference is that now the actions of the rules can be applied only if their triggering event has been detected first. Another difference is that we can have actions that activate internal events. The definition of a set of ECA rules as equational transition rules is presented below.

3.1.2.1 Definition 2. Assume the state space Y and a finite set of ECA rules {on E_i if C_i do A_i, $i = 1, \ldots, n \in \mathbb{N}$}, where without harm of generality, we also assume that for $i \neq j$; $E_i, A_i, C_i \neq E_j, A_j, C_j$, respectively. We define the following observers (O) and transitions (T) from this set of rules:

$$O = \{O' \cup \textit{knowledge}, \textit{event-memory}\},$$

$$T = \{E_i, A_i\},$$

Where O' is the same as in Definition 1. Transitions are the *external* events, $E_i : Y\,D \rightarrow Y$, and the actions, $A_i : Y\,D \rightarrow Y$, of the rules ($D = D1, \ldots, Dn$ are data types that may be needed for the definition of the specific rule based system). Formally, the rule on E_i if C_i do A_i is defined in CafeOBJ terms through the following steps:

- *Step 1*: The application of the external event E_i in an arbitrary system state u and its effects on the observer event-memory are defined as:

  ```
  event-memory(Ei(u,d1, . . .,dn)) = Ei if c-ei(u,d1, . . .,dn) =
  true and event-memory(u) = null.
  ```

  where *c-ei*(u, $d1$, …, dn) is the effective condition of the external event E_i and states the conditions under which the system is able to detect the event. Also, `null` is a `Name`-sorted constant and denotes the empty event memory.

- *Step 2*: The effects of the action A_i, if it is an assert action, for example, are described through the following equations;

  ```
  knowledge(assert(u,ki,d1, . . .,dn)) = ki|knowledge(u) if
  Ci(u,
  d1, . . .,dn) = true and event-memory(u) = Ei.
  event-memory(assert(u,ki,d1, . . .,dn)) = null if Ci(u,d1, . . .,
  dn) = true and event-memory (u) = Ei.
  ```

- *Step 3*: The effects of the action A_i if it is a generic action that sets the value v_i to the variable o_i, on the observer o_i are described as follows:

  ```
  oi(Ai(u,d1, . . .,dn)) = vi if Ci(u,d1, . . .,dn) = true and event-
  memory(u) = Ei.
  event-memory(Ai(u,d1, . . .,dn)) = null if Ci(u,d1, . . .,dn) =
  true and event-memory (u) = Ei.
  ```

- *Step 4*: The effects of the action A_i on the observer event memory, if it activates an internal event, say E_j, are described through the following equations:

  ```
  event-memory(Ai(u,d1, . . .,dn)) = Ej if Ci(u,d1, . . .,dn) = true
  and event-memory(u) = Ei.
  ```

The effects of the rest of the actions are defined in a similar way. Step 1 states that when event E_i is applied, the name of the occurred event (E_i) is stored in the observer *event-memory* if in the previous state, the detection conditions of the event were true and event-memory was null (denoting that no other external event had been detected). Step 2 declares that the action *assert*(u, *ki*($d1$, …, dn)) will be applied successfully, if the condition of the rule holds and the triggering event of the action has been successfully detected in the previous state. As a result, the fact k_i is added

to the KB and its triggering event is consumed, i.e. *event-memory* becomes null. In Step 3, it is stated that when the action $Ai(u, d1, \ldots, dn)$ is applied, the value of the observer o_i becomes v_i and *event-memory* becomes empty. Finally, Step 4 declares that in cases where the action of the rule activates an internal event (say E_j), the observer *event-memory* stores the name of the activated event *Ej* and becomes null again when the corresponding to the internal event action is applied (if it does not activate another internal event).

Let us also note that when the condition of the action does not hold, the action is not applied and the name of the detected event is removed from the observer event-memory. Also, if we want the external events to be detected in a specific order, we use extra observers that constrain their detection conditions.

Finally, as we mentioned in Step 2, once the rule to be triggered has been determined (according to the detected event and the evaluation of the condition) and its corresponding action is executed, the name of the event is consumed to prevent multiple and erroneous triggering of rules. In cases where many rules (either production or ECA) can be executed at the same time, usually a selection function is used from the inference engine of the system such as those presented in Paschke and Boley (2009) and Paschke (2006). This characteristic can be easily included in our framework by defining a selection order (and using appropriate observers) for the triggered rules, but it is out of scope of this paper.

3.1.3 Complex actions. We present how complex actions can be expressed in our framework. When we have parallel semantics, i.e. two (or more) parallel actions that are being executed at the same time, we declare the effects of the parallel transitions in one step (with one transition), as if these transitions were simultaneously applied[1]. Assume that the first action sets the value v'_a to the observer o_a and the second sets the value v'_b to the observer o_b. The parallel actions are described through the transition $action_{a|b}$ as follows:

```
o_a(action_a|b(u,d1,..,dn)) = v_a' if c-a|b(u,d1,..,dn) = true.
o_b(action_a|b(u,d1,..,dn)) = v_b' if c-a|b(u,d1,..,dn) = true.
```

The transition $action_{a|b}$ is effectively applied and changes the value of the observer o_a to v'_a while, at the same time, sets the value of the observer o_b to v'_b, if the condition of the complex action holds.

When we have sequential semantics, meaning that the action of the rule consists of two (or more) parts that are being applied one after the other, we basically divide the complex action into two (or more) transitions, and we force the second transition to be applied only if the first transition has been applied first. This is expressed in our framework as follows:

```
o_a(action_a(u,d1,..,dn)) = v_a' if c-a-b(u,d1, . . .,dn) = true.
o_b(action_b(u,d1,..,dn)) = v_b' if c-a-b(u,d1, . . .,dn) = true
and o_a(u) = v_a'.
```

Transition $action_a$ will be effectively applied if the condition of the complex action holds, and it will set the value v'_a to the observer o_a. Transition $action_b$ will be effectively applied right after the application of transition $action_a$, as its effective condition contains also the constraint value of observer o_a to be v'_a (i.e. $action_a$ has been applied before).

Also, according to the definition, no other transition can be applied after that, except from $action_b$.

Expressing reactive rules as equational transition rules allows us to verify their behavior by (theorem) proving desired safety properties about the specified rule-based system, as we will see in details in Section 4.

3.2 Reactive rules and rewriting logic

We now present an alternative specification framework and define a set of production rules and a set of ECA rules as a set of rewrite transition rules.

In a nutshell, in our framework, rewriting logic specifications are created using:

- observable values which are pairs of (parameterized) names and values [Obs];
- states which are expressed as collections (soups) of observable values [Obs $<$ State]; and
- state transitions which are described in transition rules (rewriting rules).

3.2.1 Production rules. A production rule can be expressed in terms of rewriting logic, by mapping the actions of the rule into conditional rewrite transition rules, with the condition of the rule. To describe the effects of the actions, we use observable values. The basic observable value is called knowledge, which again observes the KB of the system:

```
knowledge:_ : SetofFacts -> Obs
```

3.2.1.1 Definition 3. Assume the following set of production rules: {if C_i do A_i, $i = 1, \ldots, n \in \mathbb{N}$}, where A_i can be either a generic action with side effects, such as variables' assignment or an assertion, retraction or update of the KB.

The actions of production rules are defined as rewrite transitions, expressed in CafeOBJ, through the following steps:

- *Step 1*: If A_i is an assert action, its effect on the observable value knowledge is defined as: `ctrans [assert ki] (knowledge: K) D => (knowledge: (ki|K) D if Ci(d1, . . .,dn) = true.`
- *Step 2*: If A_i is a retract action, its effect on the observable value knowledge is defined as: `ctrans [retract ki] (knowledge: K) D => (knowledge: K/ki) D if Ci(d1, . . .,dn) = true.`
- *Step 3*: If A_i is an update action, its definition is the following: `ctrans [update ki kj] (knowledge: K) => (knowledge: kj|(K/ki)) D if Ci(d1,…,dn) = true.`
- *Step 4*: Finally, if A_i is a generic action, we define `ctrans [Ai] (oi: vi) D => (oi: vj) D if Ci(d1,…,dn) = true.`

In the above definitions, inside the brackets (Ksystra *et al.*, 2014), we declare the label of the transition rule. The keyword `ctrans` is used, because the rule is conditional. `K` denotes an arbitrary value of the observable value knowledge in the previous state, and `ki, kj` are the facts being added/removed from the KB. Also, `D = D1, . . .Dn` denotes arbitrary data types that may needed for the definition of the transition (that depends on the specified system). Finally, `oi` are extra observable values that may needed for the definition of the rules and `vi, vj` are variables of appropriate sorts. In Step 4, for example, we state that the observable value will become `vj` if the condition of the rule is true, as this is the effect of the generic action *Ai*.

3.2.2 Event condition action rules. To express ECA rules in terms of rewriting logic, we map the actions and the events of the rules into rewrite transitions, and the basic observable values that we use are knowledge and event memory, with the same functionality as before:

```
event-memory:_ : Name -> Obs
```

3.2.2.1 Definition 4. Assume a finite set of ECA rules {on E_i if C_i do A_i, $i = 1, \ldots, n \in \mathbb{N}$ where A_i denotes either a generic action or a predefined action of the rule language.

The rule on E_i if C_i do A_i is defined in CafeOBJ terms as a rewrite theory through the following steps:

- *Step 1*: The rewrite transition rule that specifies the event E_i is defined as follows:

  ```
  ctrans [Ei] (event-memory: null) => (event-memory: Ei) if
  c-ei (d1, . . ., dn) = true.
  ```

  The detection condition of the event is denoted as *c-ei*($d1, \ldots, dn$), and its definition depends on the specific event.

- *Step 2*: If the action *Ai* is an assert/update/retract action, its effects are denoted as follows:

  ```
  ctrans [assert ki] (event-memory: Ei) (knowledge: K) D =>
  (event-memory: null) (knowledge: ki|K) D if Ci(d1, . . .,dn) =
  true.

  ctrans [retract ki] (event-memory: Ei) (knowledge: K) D =>
  (event-memory: null) (knowledge: K/ki) D if Ci(d1, . . .,dn) =
  true.

  ctrans [update ki kj] (event-memory: Ei) (knowledge: K) D =>
  (event-memory:   null)   (knowledge:   kj|(K/ki))   D   if
  Ci(d1, . . .,dn) = true.
  ```

In the definition of the actions above, the term (event-memory: Ei) ensures that only the guard of the action will hold at the pre state, and, thus, it will be the only applicable transition for that state of the system. After the occurrence of the action, event-memory will become null again (except from the cases where internal events are activated), denoting that the system is ready to detect another event. In a similar way, the rest of the actions are defined.

3.2.3 Complex actions. Parallel semantics, i.e. two (or more) actions that are being executed concurrently, are expressed as a rewrite transition rule as follows: assume that the first action sets the value $v_{a'}$ to the variable o_a and the second sets the value $v_{b'}$ to the variable o_b. The effects of these actions are then defined simultaneously, and the effect of the one step transition $a \mid b$, as shown below:

```
ctrans [a|b]: (o_a: v_a) (o_b: v_b) => (o_a: v_a') (o_b: v_b')
if c-a|b(d1, . . .,dn) = true.
```

The values of the two variables are successfully changed if the condition of the complex action holds. Complex actions with sequential semantics, i.e. two (or more) successive actions, are defined as follows:

```
ctrans [a]: (o_a: v_a) => (o_a: v_a') if c-a-b(d1, . . ., dn) =
true.

ctrans [b] : (o_a: v_a) (o_b: v_b) => (o_a: v_a) (o_b: v_b') D
if c-a-b(d1, . . .,dn) = true and v_a = v_a'.
```

The first transition changes the value of the variable o_a to v_a' if the condition of the complex action holds. The second transition changes the value of the variable o_b to v_b' if the condition holds and $action_a$ has been successfully applied before.

Expressing reactive rules as a rewrite theory specification allows us to verify them with regard to both their structure and their behavior, as we will discuss in details in Section 4.

In the definitions presented, we described how the system deals with the detection of atomic events. In cases where the system detects complex events (i.e. combinations of atomic events, e.g. $event_a$ *and* $event_b$), roughly speaking, the observer knowledge is used to keep track of the detection order of the atomic events, whereas event-memory stores an identification number of the complex event. For more details, we refer the interested reader to (Ksystra *et al.*, 2014) (Section 3.3 – Complex events definition).

Concerning the semantic relationship of the two formalizations, we would like to refer the readers to Zhang and Ogata (2009, 2012), where the authors present translations between rewriting logic specifications and equational specifications in ways that preserve semantically the properties of the specifications. Our methodologies meet the requirements of Zhang and Ogata (2009, 2012), and, thus, both approaches can be used to show the semantic equivalence between the two formalisms presented in this paper. However, as we do not propose a translation scheme between the two methodologies, such an analysis is outside the scope of this paper.

To demonstrate the effectiveness of the proposed methodologies, in the following section, we apply them in a case study and present the verification of a light-control system.

4. Case study: a light-control intelligent system

4.1 Informal description

In the following Table I, we can see the set of ECA rules that specify a light-control intelligent system which attempts to reduce energy consumption by turning off the lights in unoccupied rooms or in rooms where the occupant is asleep using sensors. The system also provides automatic adjustment for indoor light intensity based on the outdoor light intensity.

The values measured by the sensors are stored in environmental variables. The measure of a motion sensor that detects whether the room is occupied is expressed by the Boolean variable Mtn. A pressure sensor detects whether the person is asleep, and this information is stored in the Boolean environmental variable Slp. A light sensor, whose measure is expressed by the variable ExtLgt ($\in$1, …, 10), is used for monitoring the outdoor lighting.

MtnOn, MtnOff and ExtLgtLow are external events activated by the environmental variables discussed above. MtnOn and MtnOff occur when Mtn changes from false to true or from true to false, respectively. ExtLgtLow occurs when ExtLgt drops below 6.

Internal events model internal system actions. For example, internal event SecElp models the system clock, occurs every minute and changes the value of

Type of ECA rule part	Name of ECA rule part and details
Environmental variables	Mtn, ExtLgt, Slp
Local variables	lgtsTmr, intLgts
External events	MtnOn activated when Mtn = true MtnOff activated when Mtn = false ExtLgtLow activated when ExtLgt ≤ 5
Internal events	SecElp, LgtsOff, LgtsOn, ChkExtLgt, ChkMtn, ChkSlp
(R1)	When the room is unoccupied for 6 min, turn off the lights if they are on
ECA rule no.	*ECA rule in natural language and in rule syntax*
r1	on MtnOff if (intLgts > 0 and lgtsTmr = 0) do set (lgtsTmr, 1) par activate (SecElp)
r2	on SecElp if (lgtsTmr ≥ 1 and lgtsTmr < 6 and Mtn = false) do increase (lgtsTmr, 1)
r3	on SecElp if (lgtsTmr = 6 and Mtn = false) do set (lgtsTmr, 0) par activate (LgtsOff)
r4	on LgtsOff do (set (intLgts, 0) par activate (ChkExtLgt))
(R2)	When lights are off, if external light intensity is below 6, turn on the lights
r5	on ChkExtLgt if (intLgts = 0 and ExtLgt ≤ 5) do activate (LgtsOn)
(R3)	When lights are on, if the room is empty or a person is asleep, turn off the lights
r6	on LgtsOn do (set (intLgts, 6) seq activate (ChkMtn))
r7	on ChkMtn if (Slp = true or (Mtn = false and intLgts >= 1)) do activate (LgtsOff)
(R4)	If the external light intensity drops below 5, set the lights intensity to 6 and check if the person is asleep. If the person is asleep, turn off the lights
r8	on ExtLgtLow do set (intLgts, 6) par activate (ChkSlp)
r9	on (ChkSlp if (Slp = true) do set (intLgts, 0)
(R5)	If the room is occupied, set the lights intensity to 4
r10	on MtnOn do set (intLgts, 4) par set (lgtsTmr, 0)

Table I. ECA rules controlling the lights intensity of a house

lgtsTmr. Variable lgtsTmr is a timer for R1, to convert the continuous condition, "the room is unoccupied for 6 minutes" into 6 discretized SecElps events. Rule r1 initializes lgtsTmr to 1 whenever the motion sensor detects no motion and the lights are on, and activates SecELp. The timer then increases as minute elapses, provided that no motion is detected (Rule r2) until it reaches 6. Then, LgtsOff activated by Rule r3, turns the lights off and activates a check on outdoor light intensity through the internal event ChkExtLgt (Rule r4). ChkExtLgt activates LgtsOn if the external light intensity drops below 6 (Rule r5). Internal event ChkMtn, activated by Rule r6, activates LgtsOff if the room is unoccupied and all lights are on or if the room is occupied but the occupant is asleep (Rule r7). ExtLgtLow sets lights' intensity to 6 and activates internal event ChkSlp (Rule 8), which turns off the lights if the occupant is asleep (Rule 9) (Jin *et al.*, 2013).

4.2 Formal analysis using equational logic

4.2.1 Specification. First, to specify this system as an equational transition system using Definition 2, we will need the following observers. The five first observers are used to

observe the variables' (local and environmental) changes. The last two are used for the detection of the events (Box 1):

```
-- Observers
Mtn : State -> Bool
ExtLgt : State -> Nat
Slp : State -> Bool
lgtsTmr : State -> Nat
intLgts : State -> Nat
event-memory : State -> Name
Mtn-memory : State -> SetofNames

var S : State
ops MtnOn, MtnOff, ExtLgtLow, LgtsOff, ChkExtLgt, ChkSlp,
ChkMtn : -> Name
```

State is the sort denoting the state of the system, and *S* is a variable of the same sort. The names of the (internal and external) events are declared as constants of the sort Name. The transitions of the system (and also constructors of State) are the actions and the external events.

Rule 8, for example, can be described by the following conditional equations (Box 2):

```
event-memory(E8(S)) = ExtLgtLow if ExtLgt(S) <= 5 and event-
memory(S) = null .
```

The event ExtLgtLow is detected and stored in the observer event-memory whenever the external lights' intensity drops below 6 and if no other event has been detected in the previous state. The rest of the observers stay the same during the detection of the event E8.

The effects of the action of Rule 8 are defined as (Box 3):

```
intLgts(A8(S)) = 6 if Slp(S) = false and event-memory(S) =
ExtLgtLow .

event-memory(A8(S)) =ChkSlp if Slp(S) = false and event-
memory(S) = ExtLgtLow .
```

Internal lights intensity will be set to 6 if the occupant of the room is not asleep and the event ExtLgtLow has been detected in the previous state. The rest of the observers stay the same.

Rule 3 is triggered by the action of the previous Rule (r2), and, thus, we do not have to specify its event. The equational transition Rule A3 describes the effects of the action of the rule, as follows (Box 4):

```
ceq lgtsTmr(A3(S)) = 0 if Mtn(S) = false and lgtsTmr(S) = 6
and event-memory(S) = SecElp .

ceq event-memory(A3(S)) = LgtsOff if Mtn(S) = false and
lgtsTmr(S) = 6 and event-memory(S) = SecElp .
```

The timer (observer lgtsTmr) will be set to 0 if the effective condition of the rule holds. This means that in the previous state, the sensors had not detected any motion in the room, the timer had the value 6 and the event SecElp had been successfully detected. At the same time, the internal event LgtsOff will be activated and stored in the observer event-memory.

Transition Rule A4 describes the effects of the action of Rule r4 as follows: (Box 5):

```
ceq intLgts(A4(S)) = 0 if event-memory(S) = LgtsOff .
ceq event-memory(A4(S)) = ChkExtLgt if event-memory(S) =
LgtsOff .
```

Internal lights intensity will be set to 0 as a reaction of detecting event LgtsOff in the previous state. At the same time, the internal event ChkExtLgt will be activated and stored in the observer event-memory.

In an analogous way, we define the rest rules of the system. The full specification of the light-control system as an equational transition system can be found at cafeobj@ntua.blogspot.com.

4.2.2 Verification support. For the verification of the system, when it is expressed in equational logic, we use Cafe OBJ's theorem proving technique (Ogata and Futatsugi, 2013) to verify desired safety properties of the rules. We will explain the methodology through the running case study.

For our rule-based system, an invariant safety property could be the following: *the lights cannot be turned off if someone is in the room and he/she does not sleep.* To prove such properties, four steps need to be taken:

(1) The first step is to express the property as a predicate in CafeOBJ terms in a module, usually called INV:

```
inv1(S) = not((intLgts(S) = 0) and (Mtn(S) = true) and
(Slp(S) = false)).
```

(2) The next is to define the inductive step in a module (usually called ISTEP), i.e. a predicate which states that if the property holds in an arbitrary state, say s, then that implies that it holds in any successor state, say s':

```
istep1 = inv1(s) implies inv1(s').
```

(3) The third step is to ask CafeOBJ to prove (using the reduce command) if the property holds for an arbitrary initial state, using the following reduction:

```
red inv1(init).
```

(4) Finally, s' must be instantiated and then ask Cafe OBJ to prove the inductive step for each transition rule. For example, for the transition E8, the inductive step is declared as follows:

```
s' = E8(s).
red istep1.
```

This step has to be repeated for all the transitions in turn, i.e. for `s' = E1, A1, . . ., E10, A10.`

When we ask CafeOBJ's reduction algorithm to prove a property, through the reduce command, three results might be returned; true, false and a CafeOBJ term. If true is returned, the proof is successful. If a CafeOBJ term is returned, the user must intervene and split the case by stating that the returned term equals to true and false in turn to help CafeOBJ to reduce the term. Finally, in the case where CafeOBJ returns `false`, either the property does not hold for our system or the case that returned false is unreachable.

In our case, CafeOBJ returned `true` for the following initial state:

```
event-memory(init) = null.
Slp(init) = true.
Mtn(init) = true.
Mtn-memory(init) = null.
intLgts(init) = 1.
lgtsTmr(init) = 0.
ExtLgt(init) = 4.

red inv1(init).
```

In the inductive step of invariant 1 for the transition A4, for example, CafeOBJ could not reduce the effective condition to either true or false, and we had to split it in the following cases:

```
open ISTEP
-- effective condition false
eq (event-memory(s) = LgtsOff) = false.
s' = A4(s).
red istep1.
close

open ISTEP
-- effective condition true
eq event-memory(s) = LgtsOff.
s' = A4(s).
red istep1.
close
```

Although the case where the effective condition does not hold returned true, its symmetrical case required further case splitting. Continuing with the case splitting, we reached the case denoted by the following equations, where CafeOBJ returned false:

```
open ISTEP
eq event-memory(s) = LgtsOff.
eq intLgts(s) = 1.
eq Mtn(s) = true.
eq Slp(s) = false.
s' = A4(s).
red istep1.
close
```

In such a case, either we have discovered a counterexample for the property we are proving or we must formulate a lemma that discards the case that returned false, i.e. prove that this case is unreachable with respect to our specification. In our case, it was easy to understand that not all of the above equations can hold simultaneously (event LgtsOff cannot be detected if there is motion in the room), and, thus, we used the lemma inv2(S) = not(event-memory(s) = LgtsOff and Mtn(s) = true) to discard it. Of course, in such cases, the lemma used has to be proven as well. The proofs can be found at Ksystra *et al.* (2014).

4.3 Formal analysis using rewriting logic

4.3.1 Specification. A state in rewriting logic is described as a collection (soups) of observable values, as we have already mentioned. To specify the light-control system using Definition 4, we use the following observable values (Box 6):

```
event-memory:_ : Name -> Obs
Mtn-memory:_ : SetofNames -> Obs
intLgts:_ : Nat -> Obs
Mtn:_ : Nat -> Obs
lgtsTmr:_ : Nat -> Obs
Slp:_ : Nat -> Obs
ExtLgt:_ : Nat -> Obs
```

Thus, an arbitrary state of the system is defined as: (event-memory: e) (Slp: s)(Mtn: l)(Mtn-memory: m)(intLgts: i)(lgtsTmr: t)(ExtLgt: x), where e, s, l, m, i, t and x are predefined variables which denote arbitrary values for the corresponding sorts.

Rule *r*8 is defined in rewriting logic as follows (Box 7):

```
ctrans [E8] : (event-memory: null) (ExtLgt: x) =>
(event-memory: ExtLgtLow) (ExtLgt: x) if (x <= 5) .
```

Event ExtLgtLow is successfully detected and stored in the observable value event-memory when the sensor detects that external light density drops below 5 and if no other event has been detected in the previous state. The definition of the action of the rule can be seen below (Box 8):

```
ctrans [A8] : (event-memory: ExtLgtLow) (Slp: s) (intLgts: i)
=> (event-memory: null) (Slp: s) (intLgts: 6)  if (s = false) .
```

The action of the rule sets the internal lights intensity to 6 as a reaction to the detected event, if the person is not asleep. Also, event-memory becomes null.

Rule *r*3 can be defined as a rewrite transition rule in CafeOBJ terms, as follows (Box 9):

```
ctrans [A3] : (event-memory: SecElp) (Mtn: l) (lgtsTmr: t) =>
(event-memory: LgtsOff) (Mtn: l) (lgtsTmr: 0) if (l = false)
and (t = 6)
```

The timer is set to 0 and the internal event LgtsOff is activated if the condition of the rule holds.

Rule r4 can be defined as a rewrite transition rule in CafeOBJ terms, as (Box 10):

```
trans [A4] : (event-memory: LgtsOff) (intLgts: i) =>
(event-memory: ChkExtLgt) (intLgts: 0)
```

Internal lights intensity is set to 0 and the internal event ChkExtLgt is activated if in the previous state event LgtsOff has been detected.

In an analogous way, we define the rest rules of the system. The full specification of the light-control system as a rewrite transition system can be found at cafeobj@ntua.blogspot.com.

4.3.2 Verification support. This methodology allows us to verify both the structure and the behavior of the specified rule-based system. In respect to the structural properties, with the help of CafeOBJ's search engine and two operators we have defined, we can: detect termination, confluence errors for the system and simulate the execution behavior of the rules more precisely.

4.3.3 Termination. A rule program's state s is terminating if and only if there is no infinite sequence $s \rightarrow s1 \rightarrow s2 \rightarrow \ldots$ of states. In other words, a state s is terminating if there exist two states such that $s \rightarrow s'$ and $\neg(s = s')$, where s' is a final state, and there are no cyclic paths (Baba-hamed and Belbachir, 2007). Based on this, we can check if a state terminates using the following predicate we have defined in Ksystra *et al.* (2012);

```
eq terminates?(s) = s =(1,*)=>! s'.
```

The expression `s=(1,*) =>! s'` defines that s' should be a different term from s, and that no transition rules are applicable to s'. Thus, by reducing the above predicate, we basically ask CafeOBJ to find *a final state* reachable from the state s of the system. To check a rule-based system for termination, we must perform the search for all initial states of the system using the command `red terminates?(init).`

If true is returned and a final state, it means that this state of the system may terminate. If false is returned, it means that in this initial state, no transition can be applied or that the reachable state(s) from this state is(are) not final. Finally, the rewriting may not terminate, because CafeOBJ's rewriting system applies a transition rule on and on.

In the last case, there is no reason to check the system for confluence, because if a state is not terminating, it is not confluent either. In the first two cases, to get a clear picture about the behavior of the state, we proceed with checking it for confluence errors and possible loops.

Let us apply our methodology to the running case study. Suppose that the initial state[2] of the system is the following. The lights are on, the room is occupied, but no one is sleeping and the intensity of the external lights is low (below 5), and we wish to see if this system will terminate or not. We first define the required variables, constants and state (Box 11):

```
set trace on
init = (event-memory: null) (Slp: false) (Mtn: true) (Mtn-
memory: null) (intLgts: 1) (lgtsTmr: 0)(ExtLgt: 4) .
s' = (event-memory: e1) (Slp: s1) (Mtn: l1) (Mtn-memory: m1)
(intLgts: i1) (lgtsTmr: t1) (ExtLgt: x1) .
red terminates?(init) .
```

The command set trace on instructs CafeOBJ to show in details the performed rewrites. CafeOBJ for the above reduction returns the result: `true` and `found state 3`. This means that, state 3 is a final state reachable from the initial state of the system.

With the command `show path id`, we can see the applied transitions that lead to this state and their order. In our case, CafeOBJ returns the sequence of the transitions E8 and A8. This result is as expected, because according to the rules and our initial state, the external event `ExtLgtLow` will be activated (as the external light intensity is low), and as no one sleeps, the internal lights will be set to 6. This is a final state of the system, as no other transitions can be applied. We can now proceed with checking the state for confluence errors and cyclic paths.

4.3.4 Non-Confluence. A rule program's state s is non-confluent if there exist two traces $trace_1$ and $trace_2$ from this state that lead to distinct final states. That is, there exist two traces and three states such that; $s \xrightarrow{trace_1} s1$ and $s \xrightarrow{trace_2} s2$ and $\neg(s1 = s2)$, where $s1$ and $s2$ are final states (Berstel and Leconte, 2010). Based on this sufficient condition, we can check a state for non-confluence by using the following predicate we have defined in Ksystra *et al.* (2012); `eq notConfluent(s) = s =(2,*)=>! s'.`

To check a rule-based system for non-confluence, we perform the search for all the initial states of the system, as before, using red notConfluent?(init).

The above reduction instructs CafeOBJ to search, starting from the initial state of the system, for *two different final states*. For this reason, we use again the predicate with the exclamation mark at the end (final state), but in the number indicating the number of solutions we assign the value 2 (two different states). If two such solutions are found, it means that the state is not confluent.

To understand which rules cause the problem or in cases where the system's behavior remains unclear, we should continue the analysis of the rules by conducting one more test. This last check basically simulates the execution behavior of the rules and detects possible loops.

In our case study, if we use the command `notConfluent?(init),` Cafe OBJ will return the result; `found state 3` and `no more possible transitions.`

In that case, it is not clear how the initial state behaves, and further analysis is required. For this reason, we use the command, `red init =(*,*)=>* S,` which returns *all the reachable states* from the initial state of the system. This, in combination with the command show path id, simulates the execution behavior of the rules.

In our running case study, if we use the following reduction: `red init =(*,*)=>* (event-memory: e1) (Slp: s1) (Mtn: l1) (Mtn-memory: m1) (int-Lgts: i1) (lgtsTmr: t1) (ExtLgt: x1)` and also use the command `show path id` for each reachable state, we get the following result (Box 12):

```
found state 0 (init)
found state 1 (init - E8)
found state 2 (init -  E10)
found state 3 (init - E8 - A8)
found state 4 (init - E10 - A10)
found state 5 (init - E10 - A10 - E8)
```

This is graphically presented in Figure 1.

In this way, we get a clear perception of the execution of the rules. Now, because of the fact that CafeOBJ system visits each reachable state once, using this check, we can discover that the last state of the execution path can be either:

- final, i.e. no transition can be applied in it;
- in between, i.e. a transition can be applied in it and will lead to an already visited final state; and
- part of a loop, i.e. a transition can be applied in it, but it will lead to a non-final state already visited in the same state path.

Based on the above result, we can decide whether the initial state of the rule-based system behaves as expected[3]. Discovering a loop using the proposed methodology allow us to isolate the rule that is responsible for it and the condition which causes it. This useful information can be used to change the rules and redesign the system.

In our case study (Figure 1), state 3 is final, as the first check already showed. State 5 is an in-between state, because if Transition A8 will be applied in it, it will lead to state 3 again (that is why CafeOBJ does not apply A8). *So, this initial state behaves well, as it leads to the same final states*.

However, after checking other possible initial states and using the returned feedback, we changed Rules 8-10 as follows[4]:

r8′: on ExtLgtLow if (Slp = false) do set (intLgts, 6).
r9′: on ExtLgtLow if (Slp = true) do set (intLgts, 0).
r10′: on MtnOn if (Slp = false) do (set (intLgts, 4) par set (lgtsTmr, 0)).

Regarding now the behavior of the rules, the built-in CafeOBJ search predicate can also be used to prove safety properties for a system specified in rewriting logic. Ogata and Futatsugi (2014) propose a methodology for model checking and theorem proving rewrite specifications using CafeOBJ. We will describe this verification methodology using the running case study again.

4.3.5 Invariant properties. A desirable safety property p is an invariant for a rule-based system if it holds in each reachable state (R_s) of the system, i.e. $\forall s \in R_s.p(s)$.

For the verification of such properties, model checking and/or theorem proving can be used. An invariant property can be model checked by searching if there is a state reachable from the initial state such that the desirable property does not hold (Ogata and Futatsugi, 2014). This can be achieved using the expression: `red init =(1,*)=>* p [suchThat c].`

In the above term, c is a CafeOBJ term denoting the negation of the desired safety property. Thus, CafeOBJ will return true for this reduction if it discovers (within the given depth) a state which violates the safety property.

For example, suppose we are interested in the following invariant property; *the lights cannot be turned off if someone is in the room and he/she does not sleep*. Also, consider

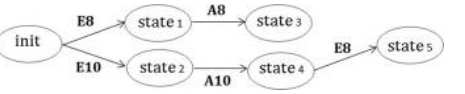

Figure 1.
Graphical representation of the reachable states from the initial state of the system

an initial state in which the room is occupied, the person sleeps, the lights are on and the intensity of the external lights is low. If we ask CafeOBJ to find a state reachable from this initial state, in which the lights are off, the room is occupied and the person does not sleep, it will return the result `false`.

This methodology is very effective for discovering (shallow) counterexamples. However, model checking does not constitute a formal proof and is complementary to theorem proving. Ogata and Futatsugi (2014) present a methodology to theorem prove safety properties of specifications written in rewriting logic. This methodology can be used to reason about rule-based systems expressed in our framework.

Suppose we are interested in the same invariant property; this is expressed in CafeOBJ using the rewriting approach as follows:

```
Eq isSafe((event-memory: e1)(Slp: s1)(Mtn: l1)(intLgts:
i1)(Mtn-memory: m1) (lgtsTmr: t1)(ExtLgt: x1)) = not ((i1 == 0)
and (l1 == true) and (s1 == false)).
```

The proof is done by induction on the number of transition rules of the system using the following operator (Ogata and Futatsugi, 2014); `eq check(pre, con) = if (pre implies con) == true then true else false fi.`

This operator takes as input a conjunction of lemmas and/or induction hypotheses and a formula to prove. It returns true if the proof is successful and false if *pre implies con* does not reduce to true.

For the base case, all we have to do is to check if the following term reduces to true: `red check(true, isSafe(init)),` which it does for our initial state.

The inductive step consists of checking whether from an arbitrary state, say s, we can reach in one step a state, say s', where the desired property does not hold. This can be verified using the reduction (Ogata and Futatsugi, 2014): `red s =(*, 1)=>+ s' suchThat (not check(isSafe(s), isSafe(s'))).`

When false is returned, it means that CafeOBJ was unable to find a state s' such that the safety property holds in s and it does not hold in s'[5]. If a solution is found, i.e. the above term is reduced to true, then either the safety property is not preserved by the inductive step or we must provide additional input to the CafeOBJ machine. In the second case, this input may be either in the form of extra equations defining case analysis or by asserting a lemma (in which case, the new lemma has to be verified separately).

Consider the inductive step where, for example, the transition Rule E8 is applied to s (Box 13):

```
eq s = (event-memory: null) (Slp: s1) (Mtn: l1) (intLgts : i1)
(Mtn-memory: m1) (lgtsTmr: t1) (ExtLgt: 4) .
red s =(*,1)=>+ s' suchThat (not check(isSafe(s),isSafe(s')))
```

CafeOBJ returns false, and, thus, this induction case is discharged. Consider now the inductive step where the transition Rule A8 is applied to s (Box 14):

```
eq s = (event-memory: ExtLgtLow) (Slp: false) (Mtn: l1) (int-
Lgts: i1) (Mtn-memory: m1) (lgtsTmr: t1) (ExtLgt: x1) .
red s =(*,1)=>+ s' suchThat (not check(isSafe(s),isSafe(s')))
```

CafeOBJ returns false, and, thus, the induction step for Rule 8 is discharged.

In the same way, the induction case for all the transition rules are discharged, and, thus, the proof concludes. The whole specification of the rule based system and the proof can be found in cafeobj@ntua.blogspot.com

5. Discussion

We present here some lessons learned throughout our research in formalizing reactive rule-based systems. A more general comment is that we believe that equational and rewriting logic is easier to learn than other logics, such as higher order, because they are similar to everyday life reasoning. Hence, we believe that the verification with CafeOBJ is easier to learn than those with other methods.

Also, transition rules expressed either in equational or in rewriting logic are closer to reactive rules in researchers' mindset (as their reasoning is similar to that of rules), thus, the transformation is more straight forward. But, that was not obvious from the beginning.

One of the difficulties we faced during our research in reactive rules was the semantic difference between events and actions, i.e. the fact that although events *can* occur at anytime and can be straightforwardly mapped to transitions, actions *must* be executed after the detection of their triggering events. This issue was addressed by mapping an ECA rule into two different transition rules and by introducing appropriate observers that can handle this difference (by helping us decide if the system must react to a transition or treat it as an incoming event).

Another difficult point was selecting the most appropriate definition of termination and confluence for rule-based systems that can also be expressed in Cafe OBJ terms. To overcome this, we conducted a thorough search in the related literature and more precisely we based our definitions on those of Berstel and Leconte (2010) and Baba-hamed and Belbachir (2007), which had the level of abstraction that met our purposes.

As to which of the proposed methodologies is better, we should mention that although the equational approach has better modeling capabilities as it provides succinct, composable specifications and provides stronger verification support, to decide which of the two is the most suitable, we have to take into account that these frameworks aim in bringing reactive rules' researchers closer to the area of formal methods. Thus, we have to mainly considerate their needs and focus. To this end, we believe that the equational approach should be adopted when the specified system is complex and critical, and we need a really expressive formalism for modeling the reactive rule-based system. However, the rewriting approach is better suited for the reactive rules researchers community, as it is more natural and easier to use. Also, the rewriting logic approach offers a seamless framework for verifying reactive rules, as both safety properties and structure errors can be checked. Finally, it supports both theorem proving and model-checking techniques. Thus, we believe that in most cases, the rewriting approach is preferred.

For this reason and to make the rewriting approach even more friendly for the potential users, we have developed a tool that automatically transforms a set of reactive rules into a set of rewrite rules in CafeOBJ.

Before introducing the tool, we briefly discuss about interesting work that exists in the literature, namely, abstract interpretation, abstract execution models and automatic program analyzers, applied to various formalizations of transition systems, for example, in the context of logic programs.

In a nutshell, abstract interpretation (Delzanno *et al.*, 2010) is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g. control flow and data flow) without performing all the calculations. Abstract execution (Gustafsson *et al.*, 2006) is a form of symbolic execution, which is based on abstract interpretation. Abstract execution executes the program in the abstract domain, with abstract values for the program, and abstract versions of the operators in the language. Finally, automatic program analyzers (Shankar, 2000) are used for analyzing properties of transition systems by combining tools for program analysis, model checking and theorem proving. The above methodologies are mainly used for formal static analysis, i.e. the automatic extraction of information about the possible executions of computer programs. The computation of execution time bounds, the calculation of loop bounds and the detection of infeasible paths are few examples of their possibilities.

Also, among other well-known formalisms that deal with state transitions, that can be used to describe the effect of actions, is the Situation Calculus. Situation Calculus Gu *et al.* (2006) is a logical language for representing changes. The basic concepts in Situation Calculus are situations, actions and fluents. Briefly, actions are what make the dynamic world change from one situation to another when performed by agents. Fluents are situation-dependent functions used to describe the effects of actions. Situation Calculus is mainly used for representing temporal domains and for performing reasoning about change and causality in dynamic domains.

Compared to the above methodologies, algebraic specifications, such as CafeOBJ, are more concerned with reasoning about the behavior of specified systems and verifying desired properties of them. The type of properties that can be verified are safety properties, which hold in any reachable state of the system (called invariant properties) and liveness properties (something will eventually happen).

6. Tool: from reactive rules to CafeOBJ rewrite rules

The developed tool is written in Java and takes as input a set of reactive rules, written using the generic style described in Section 2, and automatically produces a set of rewrite transition rules written in CafeOBJ, implementing the proposed definitions of the rewriting methodology (Section 3.2). The steps need to be taken to use the tool are the following:

- Choose and open a Reactive rules specification file.
- Press the 'Translate to CafeOBJ' button.
- Save the generated CafeOBJ specification.
- Verify the behavior of the reactive-rule based system using CafeOBJ and the proposed methodology.

The reactive rules specification has to obey in some syntactic guidelines. In more details, rules declaration should start with an identification number (e.g. r1). The definition of variables, events and rules should end with a fullstop (.), and, finally, external events should be accompanied with their detection conditions.

The tool comprises two main parts. The first part contains classes that provide the required functionality to parse a reactive rules specification and to store appropriately

the elements of the rules. The second part translates the set of rules into an executable CafeOBJ rewrite theory specification based on the proposed definitions.

For example, in the following piece of code, the external event of an ECA rule is translated into a rewrite transition rule (Box 15):

```
public String translate(Rule rule) {
        if (rule.getExternal() == true) {
                return "ctrans [E" + Eca2CafeIndex(rule) + "]:
                (event-memory: null) " + "(" + rule.getdetect
                variable() + ": " + rule.getEnvVar() + ") " +
                " => (event-memory: " + Eca2CafeEvent(rule)
                + ") " + "(" + rule.getdetectvariable() + ":"
                + rule.getEnvVar() + ") " + "if (" + rule.get
                EnvVar() + " " + rule.getdetectoperator() +
                " " + rule.getdetectvalue() + ") ." + '\n';
                ...
        }
}
```

Suppose that we have the external event $E1$: *MtnOn*, activated when $Mtn = true$. The outcome of the tool for this event will be the following CafeOBJ rewrite transition rule; ctrans [E1]: (event-memory: null) (Mtn: l) => (event-memory: MtnOn) (Mtn: l) if (l = true)[6].

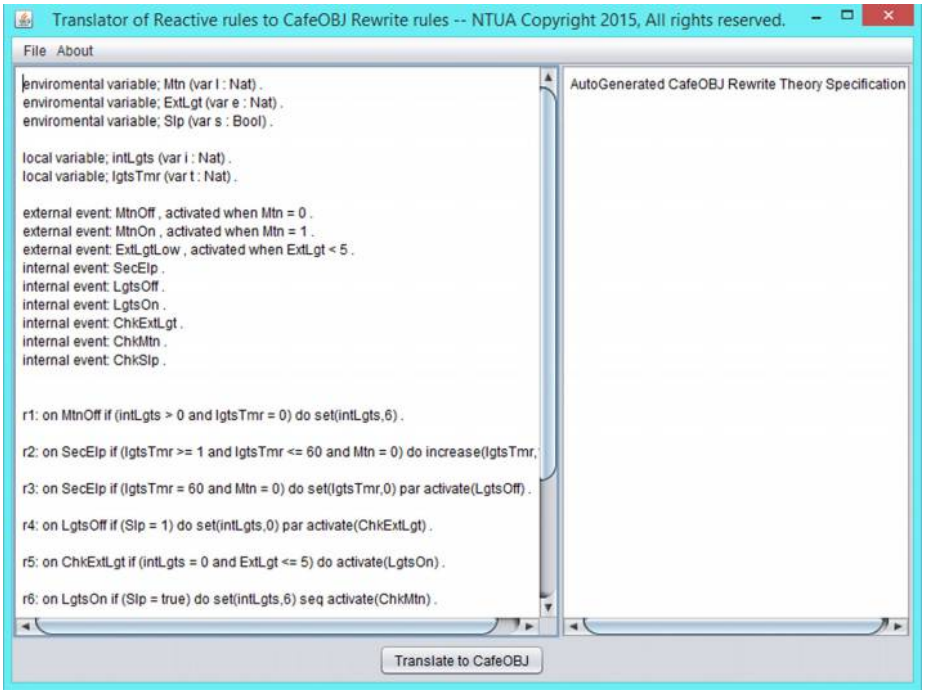

Figure 2.
ECA rules that define the intelligent light system

Eca2CafeEvent(rule) returns the name of the triggering event of the rule (here MtnOn) which is stored in the observer event-memory after the detection of the event. rule.getdetectvariable() returns the name of the variable which activates the event, here Mtn, and rule.getEnvVar() returns an arbitrary value for that variable, here l. Finally, the part after the if statement is basically the detection condition of the event, here if (l = true). In an analogous way, and based on the presented definitions, the rest types of events and the actions are translated into CafeOBJ rewrite rules.

For demonstration purposes, we present in the following screenshots, part of the transformation of the ECA rules of the running case study into CafeOBJ rewrite rules. In Figure 2, we can see part of the rules we wish to translate. Clicking on 'Translate to CafeOBJ' button will result in the outcome shown in Figure 3. As we can see, the developed tool hides the details of the translation.

Moreover, except from the automatically generated rewrite rules, the tool also creates the rest of the CafeOBJ specification which is required so that the output specification can be given as input to the CafeOBJ processor. In this way, there is no need for the user to add any additional information, such as module and sort declarations, observers, variables and so on (Figure 4). After the transformation, the user can use CafeOBJ directly to further analyze and verify the behavior of the rules.

Sometimes, it is difficult for inexperienced users to do interactive theorem proving especially for complex systems. This is another reason why we believe that

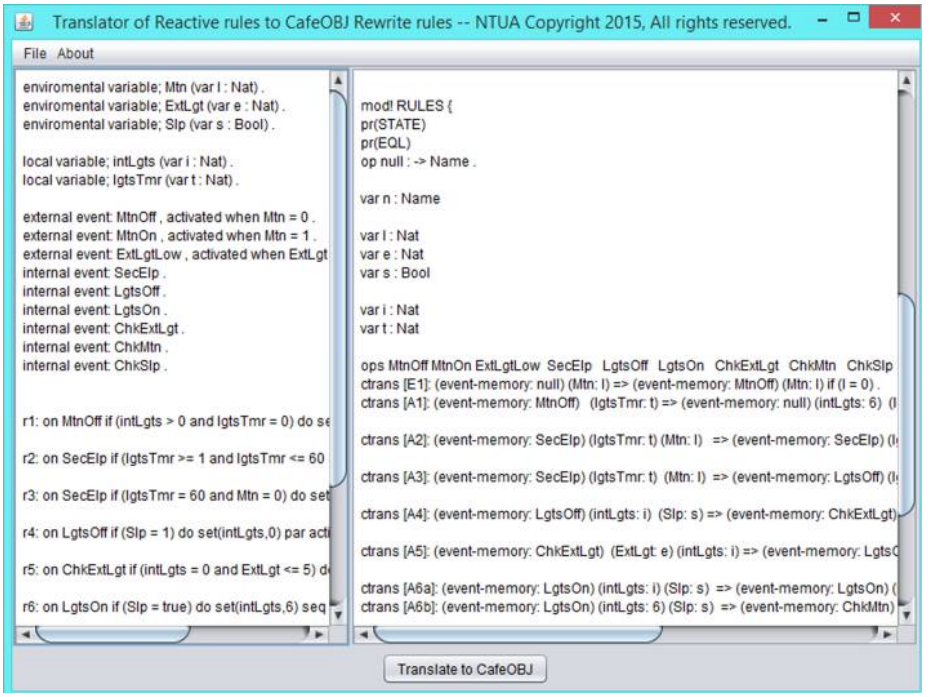

Figure 3.
Output of the tool – generated rewrite rules in CafeOBJ terms

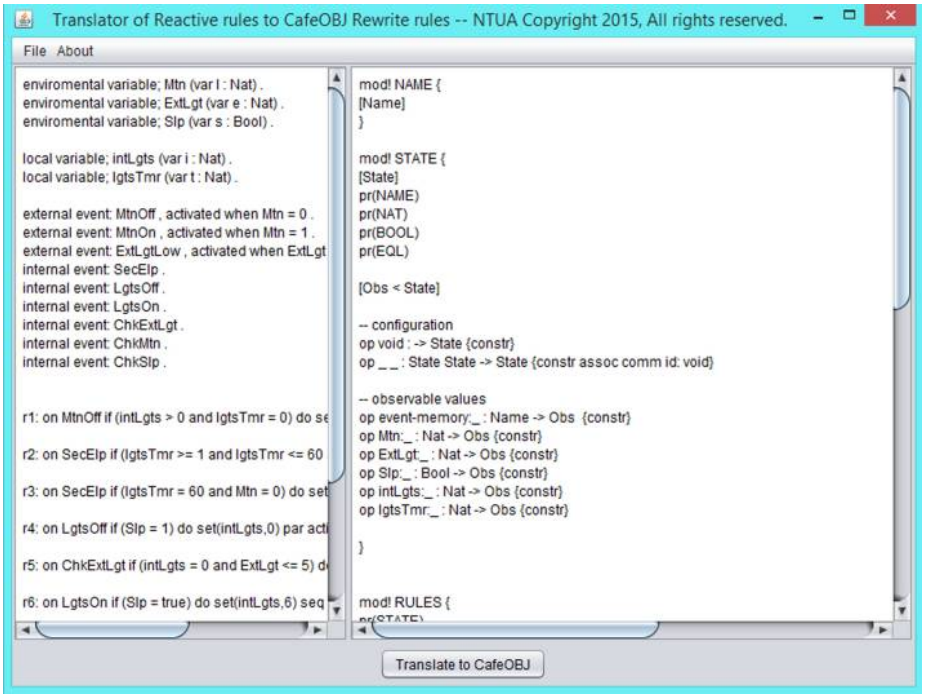

Figure 4.
Output of the tool – generated auxiliary CafeOBJ modules

the rewriting logic approach is better suited for verifying reactive rule-based systems, as it supports model checking as well, which is easier to learn and use. Also, we hope that through this paper, which presents how to verify reactive rules with simple steps, the verification process will become clear even for unexperienced users.

7. Conclusion and future work

Reactive rules are extensively used for the development of complex systems and are a promising approach, as they support flexibility, expressivity and are user-friendly. However, they can present unpredictable behavior because of their ability to interact during execution. Thus, the need for verifying their behavior has become clear.

To this end, we have presented a framework based in the CafeOBJ algebraic specification language that can be used to formally specify reactive rule-based systems in both equational and rewriting logic. This allows the verification of safety properties of the systems using model checking and theorem proving techniques, as well as the detection of confluence and termination errors. Moreover, it provides understanding of the specified systems and, thus, reliability in their development. Finally, we have developed a tool that automatically translates a set of reactive rules into a rewrite theory executable specification in CafeOBJ, thus making the proposed methodology easier for users without much experience in formal methods techniques.

In the future, we plan to integrate the proposed tool and methodologies with other specification systems, such as Maude, so as to have better tool support and broader use.

Notes

1. We assume that each transition is atomic and instantaneous.
2. In cases where the initial state of the system is not known beforehand, we can define a set of possible initial states by defining an arbitrary initial state and discriminating the cases based on the conditions of the transition rules.
3. In cases where CafeOBJ system does not apply a rule because it will lead to an already visited state, the check can be performed again with starting state, that is, the state where the previous check had stopped.
4. We could not explain in detail because of space limitations; for more details, we refer the reader to Jin *et al.* (2013) and cafeobj@ntua.blogspot.com
5. To modularly verify each transition rule separately, we define for each such transition a new module which only contains one transition rule at a time.
6. This rule denotes that in a state where event-memory is empty and the observer Mtn is equal to l, if l is true, then we derive a successor state where MtnOn is stored to event-memory.

References

Baba-hamed, L. and Belbachir, H. (2007), "The priority of rules and the termination analysis using Petri Nets", *The International Arab Journal of Information Technology*, Vol. 4 No. 2.

Badica, C., Braubach, L. and Paschke, A. (2011), "Rule-based distributed and agent systems", *5th International Conference on Rule-Based Reasoning, Programming, and Applications, RuleML, Springer, London*, pp. 3-28.

Berstel, B., Bonnard, P., Bry, F., Eckert, M. and Patranjan, P.L. (2007), "Reactive Rules on the Web", *Third International Summer School*, Vol. 4636, LNCS, Dresden, pp. 183-239.

Berstel, B. and Leconte, M. (2010), "Using constraints to verify properties of rule programs", *ICST Third International Conference on Software Testing, Verification and Validation, Paris*.

Boukhebouze, M., Amghar, Y., Benharkat, A. and Maamar, Z. (2011), "A rule-based approach to model and verify flexible business processes", *International Journal of Business Process Integration and Management*, Vol. 5 No. 4.

Bry, F., Eckert, M. and Patranjan, P.L. (2006), "Reactivity on the web: paradigms and applications of the language XChange", *Journal of Web Engineering*, Vol. 5 No. 1, pp. 003-024.

Clavel, M., Durn, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J. and Quesada, J. (1999), "Maude: specification and programming in rewriting logic", Maude system documentation.

Delzanno, G., Giacobazzi, R. and Ranzato, F. (2010), "Static analysis: abstract interpretation and verification in (Constraint logic) programming", A 25-Year Perspective on Logic Programming, Volume 6125 of the series Lecture Notes in Computer Science, pp. 136-158.

Diaconescu, R., Futatsugi, K. and Ogata, K. (2003), "CafeOBJ: logical foundations and methodologies", *Computing and Informatics*, Vol. 22, pp. 257-283.

Ericsson, A., Berndtsson, M. and Pettersson, P. (2008), "Verification of an industrial rule-based manufacturing system using REX", *1st International Workshop on Complex Event Processing for Future Internet, (iCEP08) Colocated with the Future Internet Symposium (FIS2008),Vienna, in* Anicic, D., Brelage, C., Etzion, O. and Stojanovic, N. (Eds), CEUR Workshop Proceedings, Vol. 412.

Futatsugi, K., Gaina, D. and Ogata, K. (2012), "Principles of proof scores in CafeOBJ", *Theoretical Computer Science*, Vol. 464 No. 1, pp. 90-112.

Gilbert, D. (1997), "Intelligent agents: the right information at the right time", IBM Intelligent Agent White Paper, IBM Corporation, Research Triangle Park, NC.

Gu, Y. and Kiringa, I. (2006), "Model checking meets theorem proving: a situation calculus based approach", *The Proceedings of the Eleventh International Workshop on Non-Monotonic Reasoning*, Lake District.

Gustafsson, J., Ermedahl, A., Sandberg, C. and Lisper, B. (2006), "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution", *RTSS '06 Proceedings of the 27th IEEE International Real-Time Systems Symposium*, *Rio de Janeiro*, pp. 57-66.

Hayes, P.J. (1971), "The frame problem and related problems in artificial intelligence", Technical Report, Stanford University Stanford, CA.

Jin, X., Lembachar, Y. and Ciardo, G. (2013), "Symbolic verication of ECA rules", International Workshop on Petri Nets and Software Engineering (PNSE'13) and International Workshop on Modeling and Business Environments (ModBE'13), *Milano*, pp. 41-59.

Jin, X., Lembachar, Y. and Ciardo, G. (2014), "Symbolic termination and confluence checking for ECA rules", *Transactions on Petri Nets and Other Models of Concurrency IX Lecture Notes in Computer Science*, Springer, Berlin, Vol. 8910, pp. 99-123.

Ksystra, K., Stefaneas, P. and Frangos, P. (2014), "An algebraic framework for modeling of reactive rule-based intelligent agents", *SOFSEM: Theory and Practice of Computer Science – 40th International Conference on Current Trends in Theory and Practice of Computer Science, LNCS*, *High Tatras*, pp. 407-418.

Ksystra, K., Triantafyllou, N. and Stefaneas, P. (2012), "On verifying reactive rules using rewriting logic", *RuleML, 67-81 6th International Symposium, RuleML*, Springer, Berlin, pp. 136-150.

Lukichev, S. (2011), "Improving the quality of rule-based applications using the declarative verification approach", *International Journal of Knowledge Engineering and Data Mining*, Vol. 1 No. 3, pp. 254-272.

Meseguer, J. (1992), "Conditional rewriting logic as a unified model of concurrency", *Theoretical Computer Science*, Vol. 96 No. 1, pp. 73-155.

Ogata, K. and Futatsugi, K. (2013), "Compositionally writing proof scores of invariants in the OTS/CafeOBJ method", *Journal of Universal Computer Science*, Vol. 19 No. 6, pp. 771-804.

Ogata, K. and Futatsugi, K. (2014), "Theorem proving based on proof scores for rewrite theory specifications of OTSs", *Specification, Algebra, and Software, Essays Dedicated to Kokichi Futatsugi, LNCS, 8373*, Springer, Berlin, pp. 630-656.

Paschke, A. (2005), "ECA-RuleML: an approach combining ECA rules with temporal interval-based kr event/action logics and transactional update logics", ECA-RuleML Proposal for RuleML Reaction Rules Technical Goup.

Paschke, A. (2006), "ECA-LP/ECA-RuleML: a homogeneous event-condition-action logic programming language", *International Conference on Rules and Rule Markup Languages for the Semantic Web*, *Athens, GA*.

Paschke, A. and Boley, H. (2009), "Rules capturing events and reactivity", in Giurca, A., Gasevic, D. and Taveter, K. (Eds), *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, IGI Publishing, Hershey, PA, pp. 215-252.

Paschke, A., Boley, H., Zhao, Z., Teymourian, K. and Athan, T. (2012), "Reaction RuleML 1.0: standardized semantic reaction rules", *6th International Symposium, RuleML, LNCS 7438*, Springer, Berlin, pp. 100-119.

Shankar, N. (2000), "Symbolic analysis of transition systems", From Abstract State Machines: Theory and Applications, Lecture Notes in Computer Science, No. 1912, pp. 287-302.

Stavropoulos, T.G., Kontopoulos, E., Bassiliades, N., Argyriou, J., Bikakise, A., Vrakas, D. and Vlahavas, I. (2015), "Rule-based approaches for energy savings in an ambient intelligence environment", *Pervasive and Mobile Computing*, Vol. 19 No. 1, pp. 1-23.

Zhang, M. and Ogata, K. (2009), "Modular implementation of a translator from behavioral specifications to rewrite theory specifications", *9th International Conference on Quality Software (QSIC)*, Beijing, pp. 406-411.

Zhang, M. and Ogata, K. (2012), "Invariant-preserved transformation of state machines from equations into rewrite rules", *19th Asia-Pacific Software Engineering Conference (APSEC)*, *Hong Kong*, pp. 551-556.

Corresponding author
Katerina Ksystra can be contacted at: katerinaksystra@gmail.com