On Verifying Reactive Rules Using Rewriting Logic

Katerina Ksystra, Nikos Triantafyllou, and Petros Stefaneas

National Technical University of Athens,
Iroon Polytexneiou 9, 15780 Zografou, Athens, Greece
{katksy,nitriant}@central.ntua.gr,
petros@math.ntua.gr

Abstract. Rule-based programming has been gaining a lot of interest in the industry lately, through the growing use of rules to model the behavior of software systems. A demand for verifying and analyzing rule based systems has thus emerged. In this paper we propose a methodology, based on rewriting logic specifications written in CafeOBJ, for reasoning about structural errors of systems whose behavior is expressed in terms of reactive rules and verifying safety properties within the same framework. We present our approach through a simple but illustrative example of an e-commerce web site.

Keywords: Reactive Rules, Verification, Rewrite Theory Specification, Theorem Proving, Model Checking, Structure Errors, Safety properties.

1 Introduction

Reactive rule-based systems are an attractive paradigm to software engineering since they enable systems to react to events, or combinations of events, occurring in an arbitrary order. Additional characteristics supported by rules, like flexibility and expressivity, are highly desired especially when modeling industrial systems.

However, analyzing the behavior of reactive rule based systems presents many difficulties because rules can interact with each other during execution. Thus, changing, introducing or removing a single rule from a rule base can have undesirable side effects. For these reasons, their extensive and formal analysis is required. This need becomes stronger when the rule based system is complex and/or used in critical domains. Also, the existing tool support for reasoning about rules is limited. To this end, in this paper we present a formal framework that can support the specification of reactive rule based systems and the verification of their behavior. More precisely:

– We express Production and Event Condition Action rules as rewrite theory specifications of Observational Transition Systems written in CafeOBJ (section 2).

– We propose a methodology to detect structural errors, like confluence and termination, and to verify safety properties for reactive rule-based systems (section 3).
– We illustrate the proposed approach through a running example (sections 2, 3).

1.1 Overview of Reactive Rules and Motivation

Reactivity on the Web, the ability to detect events and respond to them automatically in a timely manner, is needed for bridging the gap between the existing, passive Web, where data sources can only be accessed to obtain information, and the dynamic Semantic Web, where data sources are enriched with reactive behavior [1]. The two main categories of reactive rules are Production and Event Condition Action (ECA) rules. The former have the syntax *If condition do action* and are at the basis of Business Rules Management Systems. They specify the execution of an action in case some conditions are satisfied, i.e. define reaction to states changes. The latter have the syntax *On event if condition do action*, specify a system's response to events and are used to describe more complex systems. More precisely, 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(s) is true [2].

The properties of interest to rule systems verification include both safety properties and structure properties, such as confluence and termination of the rules. These properties are briefly described below;

A safety property is an assertion that a desirable property holds in all reachable states (i.e. is an invariant) of the rule-based system and is specific to the purpose of the specified application. Confluence concerns whether the result of executing a set of triggered rules depends on the execution order of the rules or not. A rule program is considered confluent in other words when from any initial state, all program executions lead to the same final state. Termination analysis aims to ensure that a set of rules will eventually terminate (i.e. reach a final state) and will not continue to trigger each other infinitely. A system may never terminate due to circular triggering of rules for example.

The behavior of rule based systems depends on their operational semantics. These are determined through the semantics of a rule execution procedure, which is usually called rule engine. A simple rule engine that executes production rules, consists of the following steps [3]:

1. Set the working memory to the initial state.
2. Build the set of all applicable and eligible rules. This set is called the agenda of the rule engine.
3. If the agenda is empty, the execution ends.
4. Otherwise, use a conflict set resolution strategy and choose a rule r in the agenda.
5. Update the working memory by executing the action of r. If the rule action contains several assignments, execute them in sequence.
6. Go to step 2.

The purpose of the rule eligibility strategy (step 2) is to avoid trivial infinite loops caused by applying again and again the same rule. It defines what a trivial loop is, and avoids them by making some rules ineligible. The purpose of the conflict set resolution strategy (step 3) is to pick the next rule to execute from the agenda. Again, several such strategies exist. Assigning a priority to each rule is a commonly used strategy.

Commercial engines employ such strategies to support logging execution traces, to provide simulation capabilities, and finally test and debug a rule set. For example, the problem of confluence can be solved by using priorities. It has been argued however that using this approach can be iterative since after prioritizing rules, say *r1* and *r2*, a new pair of rules causing non-confluence may be identified [4]. The problem of termination can be solved by not allowing rules to trigger each other. However, this can reduce the usefulness of the language [4]. Even though most engines provide the support described above, they also present the discussed downsides and in addition they do not permit reasoning about the rule based system. We believe that formal methods can provide a feasible solution to this problem complementing the existing tools.

2 Formal Specification of Reactive Rules

Some first steps to use algebraic specification techniques in this research area were presented in [5] and [6], where we proposed the use of OTS/CafeOBJ method to prove safety properties about reactive rule-based systems. In particular we gave an Observational Transition System (OTS) semantics to Production and Event Condition Action rules so that verification of reactive rules can be supported. OTSs are state transition systems (or state machines) that have emerged as a subclass of behavioral specifications [7] and are used to model the behavior of distributed concurrent systems. OTSs are described as equational theory specifications in CafeOBJ and the OTS/CafeOBJ method ([8], [9]) is then used to theorem prove that systems (formalized as OTSs) have desired properties. This approach has been effectively used for the specification of various complex systems [10] and the verification of invariant and liveness [11] properties of them.

The framework proposed in [6] however cannot express naturally structure properties about reactive rules, such as confluence and termination. To this end, in this paper we extend the previous approach by adopting a different logical formalism, so that the behavior of reactive rules can be formally analyzed in a seamless manner. The extended framework and its theoretical foundations are presented in the sections below;

2.1 CafeOBJ Rewriting Logic Specification

CafeOBJ [12] is an algebraic specification language which is a modern successor of OBJ [13]. The basic building blocks of a CafeOBJ specification are modules. In the body of a module we can declare sorts, operators, variables and equations. A sort is a name given to a set of values. Sorts can be partially ordered, interpreted as subset relations among the sets corresponding to the sorts [14]. Operators are declared over sorts. Terms are inductively defined with operators and variables. Equations are used to define (standard) equivalence relations over terms. Operators may be (data) constructors [14]. Examples of constructors are as follows; `op true : -> Bool {constr}, op s : Nat -> Nat {constr}`. Bool is the sort given to the set of Boolean values. Operators with no arguments such as true are called constants. Given a natural number n, s(n) denotes the successor of n [14]. Finally, as with OBJ, CafeOBJ is executable by term rewriting [15] and uses equations as left to right rewrite rules.

Rewriting logic in CafeOBJ is based on a simplified version of Meseguer's rewriting logic [16] for concurrent systems which gives an extension of traditional algebraic specification towards concurrency. RWL incorporates many different models of concurrency in a natural, simple, and elegant way, thus giving CafeOBJ a wide range of applications. Unlike Maude [17], 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 RWL and the user defined transitions [15]. This predicate evaluates to true whenever there exists a transition from the left hand side argument to the right hand side argument [15].

For a ground term t, a pattern p and an optional condition c, CafeOBJ can traverse all the terms reachable from t wrt 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 [14].

CafeOBJ supports both equational theory and rewrite theory specification. 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 [14] an attempt to combine the above is presented. They describe a way to theorem prove that rewrite theory specifications of OTSs have invariant properties by proof score writing.

2.2 Reactive Rules and CafeOBJ

We present here how reactive rules can be formally defined as a set of rewrite theory specifications in CafeOBJ. In a rewrite theory, 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. By adapting the definition presented in [6], here we define a set of reactive rules as rewrite theory specifications of OTSs expressed as collections of observable values as follows;

Definition 1. *A production rule is expressed as a term of the form* R_i `= On` C_i `do` A_i *where* A_i *can either denote a variable assignment, or an assertion, retraction, update of the knowledge base (add/remove/update facts from the KB respectively) or some other generic action with side effects. In the case where* A_i *denotes a variable assignment this is expressed by a transition rule of the form:* `ctrans [Ri] (V: v0) D => (V: v1) D if Ci = true .`

Ri is the label of the transition rule and v_0, v_1 *are variables. Also, the keyword* `ctrans` *is used because the rule is conditional. The above rule states that the observable value V will become* v_1 *if the condition of the rule is true. Also, D denotes an arbitrary data type needed for the definition of the transition. When the result of action* A_i *is the assertion of the fact* k_i *to the knowledge base, its definition is the following;* `ctrans [assert ki] (knowledge: K) D => (knowledge : (ki U K) D if Ci /in K .`

In the above rewrite rule knowledge is the observable value corresponding to the knowledge base and it is defined as a set of boolean elements. U is an operator for adding elements in a set and /in is an operator that returns true when an element belongs to a set, here the knowledge base. When the result of action A_i *is the retraction of the fact* k_i *from the knowledge base, its definition is;* `ctrans [retract ki] (knowledge: K) D => (knowledge: K / ki) D if Ci /in K .`

Operator / denotes that an element is removed from a set. When action A_i *is an update action, its definition is the following;* `ctrans [retract ki] (knowledge: K) => (knowledge: (K / ki) U kj) if Ci /in K .`

Finally, if A_i *is a generic action and extra observable values* (o_i) *need to be used for its definition, we have;* `ctrans [ai] (oi: vi) D => (oi: vj) D if Ci = true .`

In order to express ECA rules as rewrite rules we need an observable value that will “remember” the occurred events. For this reason, in each event we assign a natural number and when an event is detected its number is stored the observable value event-memory. Using event-memory we can map events to transitions/rewrite rules [6].

Definition 2. *An Event Condition Action rule of the form* R_i := *On* E_i *if* C_i *do* A_i *is defined in CafeOBJ terms as two transitions.*

- The first one specifies the event E_i *and in particular the fact that the system after the detection of the event it stores its identification number in the observable value event-memory . This is defined as;* `ctrans [Ei] (event-memory: null) => (event-memory: i) if c-ei = true .`

In the above rewrite rule, the value null of event-memory, denotes that no other event is detected at the pre state and c-ei is a boolean CafeOBJ term denoting the detection conditions for E_i.

- The second transition rule specifies the action A_i. More precisely it defines that the system must respond to the detected event by performing the corresponding action, where A_i is again either a generic action or a predefined action of the rule language. The triggering of the action as a response to the event is simply defined by adding the condition that in the pre state the event memory will contain the index of the occurred event, i.e. `ctrans [ai] (event-memory: m) (oi: vi) => (event-memory: null) (oi: vj) if Ci = true and (m = i) .`

This ensures that only the guard of this transition rule will hold at the pre state and thus this will be the only applicable transition for that state of the system. Also after the occurrence of the action, event-memory will become null again denoting that the system is ready to detect another event.

Let us note here that in cases where the action of the rule may activate an internal event (say E_j), the observable value stores the id of the event, j, and becomes null again when the corresponding to the internal event action is applied (if it does not activate another internal event).

Running Example. To illustrate the definitions presented in this paper, we will use as a running example, a company's e-commerce web site [3]. This company has customers with registered profiles on the site, which contain information about the customers' age and their category (Silver, Gold, or Platinum). When a customer puts items in his/her shopping cart a discount is computed based on the pricing policy of the company, i.e. on the customer's profile and the value of the cart. The behavior of this system is defined by the following three production rules; (R1) The gold-discount rule implements a policy that increments the discount granted to Gold customers by 10 points, if their shopping cart is worth 2,000 or more. (R2) The platinum-discount rule implements a policy that increments the discount granted to Platinum customers by 15 points, if their shopping cart is worth 1,000 or more. (R3) The upgrade rule implements a policy that promotes Gold customers to the Platinum category, if they are aged 60 or more. These rules can be written as a set of rewrite transition rules in CafeOBJ according to definition 1. First, the state of our system is formally described in the module below;

```
mod! STATE { pr(TYPE + NAT)
[Obs < State]
-- configuration
op void : -> State {constr}
op _ _ : State State -> State {constr assoc comm id: void}
-- observable values
op category:_ : type -> Obs {constr}
op value:_ : Nat -> Obs {constr}
op age:_ : Nat -> Obs {constr}
op discount:_ : Nat -> Obs {constr} }
```

As we can see a state is defined as a set of the following observable values (category:) (value:) (age:) (discount:). The three last values are represented by natural numbers and for this reason the module imports the predefined module NAT. Also pr(TYPE) imports a previously defined CafeOBJ theory which specifies the various customer types, i.e. gold, platinum and silver. Next, the rules R1-R3 are defined in the module RULES as a rewrite theory.

```
mod! RULES { pr(STATE + EQL)
var G : type
vars V N M  : Nat

ctrans [gold] : (category: G) (value: V) (age: N) (discount: M)
=> (category: G) (value: V) (age: N) (discount: (M + 10)) if
((V >= 2000) and (G = gold)) .
ctrans [platinum] : (category: G) (value: V) (age: N) (discount: M)
=> (category: G) (value: V) (age: N) (discount: (M + 15)) if
((V >= 1000) and (G = platinum)) .
ctrans [upgrade] : (category: G) (value: V) (age: N) (discount: M)
=> (category: platinum) (value: V) (age: N) (discount: M) if
(G = gold) and (N >= 60) . }
```

The gold rewrite rule, states that if the observable value `category` is gold and the `value` is equal or greater than 2000 then the value `discount` will be increased by 10 points. The platinum rewrite rule, states that if the observable value `category` is platinum and the `value` is equal or greater than 1000 then the value `discount` will be increased by 15 points. The upgrade rewrite rule states that if the observable value `category` is gold and the `age` is 60 or more then the value `category` will become platinum.

3 Formal Verification of Reactive Rules

As discussed in [3], there is an ambiguity between the upgrade and discount rule. If a gold customer is eligible to both being granted the gold discount and being upgraded to the platinum category, then this customer may end up with either a 15 or 25 per cent discount, depending on the execution order of the rules. This can be a hazard for the business application implementing this set of rules. We will present how such structural errors can be detected using our approach.

In particular, in this section we define some CafeOBJ operators which allow us to reason about confluence and termination properties of reactive rule based systems specified as rewrite logic theories. Also we demonstrate how existing operators can be used together with the proposed formalization of reactive rules to verify invariant properties about them.

3.1 Proving Termination Properties

Termination in a rule based system concerns with the existence of a state of that system where no more rules are applicable. More precisely;

Theorem (Termination). A rule program's state s is terminating if and only if there is no infinite sequence $s \rightarrow s1 \rightarrow s2 \rightarrow ...$ In other words a state s is terminating if it leads to a state where no rules can be applied. That is, there exist two states such that; $s \rightarrow s'$ and $\neg(s = s')$ where s' is a final state. Based on this, we can check if a state terminates by defining the following predicate in CafeOBJ terms;

```
op terminates? : State -> Bool
terminates?(s) = s =(1,*)=>! (event-memory: M1) (value: V1) (o: N)
red notConfluent?(s) .
```

The expression `t1=(1,*)=>! t2` indicates that the term matching to t2 should be a different term from t1 to which no transition rules are applicable. Also, the term `(event-memory: M1) (value: V1) (o: N)` represents an arbitrary state and it depends on the observable values of the specified system. By reducing the above predicate, we ask CafeOBJ to find a *a final state* reachable from the state s. If true is returned (together with a final state) it means that the state s is terminating; if false is returned it means that in the state s, no transition can be applied. Finally, the CafeOBJ reduction may not terminate, indicating that s is not terminating. Using the above predicate, we can check if the whole rule based system terminates or not, by defining the search to be performed for the initial state of the system; `op init : -> State, red terminates?(init) .`

When the number of reachable states reachable from init is small enough, the whole reachable state space can be checked by, `init =(1,*)=>`, where $*$ denotes infinity. Otherwise, the bounded reachable state space whose depth is d may be checked by, `init =(1,d)=> .`

Running Example. Here we test the set of rules of the running example for termination. We must mention that in most real life applications the initial state of the system is explicitly defined during the design of the system. An e-shop site for example before the implementation could have the following characteristics; initially, no customer is registered at the site, when someone registers for the first time his/her category is silver, the discount is zero and so on.

It is possible however, for a system to be defined without explicitly defining its initial state. In such cases, we can still check the desired properties (confluence and termination) by defining an arbitrary initial state and then discriminate the cases based on the conditions of the transition rules. In our example these cases are; (age < 60 or age >= 60), (discount = gold or platinum), (value < 1000 or value >= 1000) and (value < 2000 or value >= 2000). For the last two only the following (value < 1000 or 1000 <= value < 2000 or value >= 2000) need to be checked. Here we present the most indicative cases;

```
(a) open RULES .
op s : -> State .
eq s = (category: gold) (value: 500) (age: 50) (discount: 0)
red terminates?(s) .
```

In this case CafeOBJ returns false and the following message, which is reasonable since no transition can be applied.

```
** No more possible transitions.
(false): Bool

(b) eq s = (category: gold) (value: 500) (age: 60) (discount: 0)
red terminates?(s) .
```

In this case where upgrade is the only applicable rule the CafeOBJ system returns true and the final state (category: platinum) (value: 500) (age: 50) (discount: 0) .

```
(c) eq s = (category: gold) (value: 2000) (age: 50) (discount: 0)
red terminates?(s) .
```

In this case the gold rule can be applied to s and to all reachable states from s. Thus in CafeOBJ the above reduction does not halt indicating that this initial state is not terminating. The same conclusion holds for the platinum rule as well. Having detected this issue we can correct the rule base by adding constraints to the application of these rules, for example `(discount: (M1 + 10) <= 100)` and `(discount: (M1 + 10) <= 100)` respectively, since the discount cannot surpass this value. In this way the rules will stop triggering when the discount reaches the maximum value. When the same case is tested after adding the above constraints CafeOBJ finds the final state; `(category: gold) (value: 2000) (age: 50) (discount: 100) .`

3.2 Proving Confluence Properties

After checking that a rule based system is terminating, it is important to be able to determine if it is confluent or not (if the rules do not terminate they will not be confluent either).

Theorem (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 states. That is, there exist two traces and three states such that; $s \overset{trace_1}{\longrightarrow} s1$ and s $\overset{trace_2}{\longrightarrow} s2$ and $\neg(s1 = s2)$, where $s1$ and $s2$ are final states. Based on this, we can check a state for non-confluence by defining the following predicate in CafeOBJ terms;

```
op notConfulent? : State -> Bool
notConfluent?(s) =(2,*)=>! (event-memory: M1) (value: V1) (o: N) .
red notConfluent?(s) .
```

The above reduction i.e. asks CafeOBJ to search if it can find starting from an arbitrary state s *two different final states* of the system. 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 two (two different states). If two such solutions are found it means that the state s is not confluent. Otherwise if false is returned and one solution is found, the state is confluent. To check a rule based system for confluence we perform the search for the initial state of the system, as before, using the command `red notConfluent?(init).`

Running Example. Here we test the set of rules of the running example for confluence. Again we can discriminate the cases for an arbitrary initial state. For example:

```
(a) open RULES .
op s : -> State .
eq s = (category: gold) (value: 500) (age: 60) (discount: 0)
red notConfluent?(s) .
```

In this case where upgrade is the only applicable rule CafeOBJ returns false, as it finds one final state meaning that the state s is confluent. Now let us consider the state which is defined by the following observable values; the value of the items of the cart is equal to 2000 dollars, the age of the customer is 60 years old and her/his category is gold. This is the state we mentioned at the beginning of the section, in which the customer is eligible to both being granted the gold discount and being upgraded to the platinum category.

```
(b) eq s = (category: gold) (value: 2000) (age: 60) (discount: 0)
red notConfluent?(s) .
```

CafeOBJ returns true as it finds two solutions, denoting that s is not confluent as we expected. In particular it returns;

```
** Found [state 25] (category: platinum) (value: 2000) (age: 60)
(discount: 90)
** Found [state 27] (category: platinum) (value: 2000) (age: 60)
(discount: 95)
```

Using the command `show path id` we can see the two transition paths that cause the problem (and then we can add constraints in the conditions of the rules as before to solve this issue by letting for example the upgrade rule to be applied first). Even though the presented example is quite simple it demonstrates that detecting such errors before the implementation of a rule based system can prove really helpful especially when designing complex critical systems.

3.3 Proving Safety Properties

The built-in CafeOBJ search predicate can also be used to prove safety properties for a system specified in rewriting logic (RWL). In this work, we are interested in invariant properties.

Definition (Invariant property). 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 [14]. This can be achieved using the following 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. This methodology is very effective for discovering (shallow) counterexamples. However, model checking does not constitute a formal proof and is complementary to theorem proving. Formal proofs are required when we are dealing with critical systems. In [14] a methodology to (theorem) prove safety properties of OTS specifications written in RWL is presented. This methodology can be used to reason about rule based systems expressed in our framework as we will demonstrate throughout the running example.

Running Example. For our rule based system an invariant safety property could be the following; *a customer cannot belong to the platinum category if his/her age is less than 60 years.* This is expressed in CafeOBJ terms as;

```
op isSafe : State -> Bool .
eq isSafe((category: G) (value: V) (age: N) (discount: M)) =
not ((G == platinum) and (N < 60)) .
```

The proof is done by induction on the number of transition rules of the system. First, the following operator is used [14];

```
vars pre con : Bool
op check : Bool Bool -> Bool
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 and returns true if the proof is successful and false if *pre implies con* does not reduce to true (this is why the built in == CafeOBJ operation is used, which is reduced to false iff the left and right hand side arguments are not reduced to the same term). Using this predicate the base case of the proof is successfully discharged using the following CafeOBJ code:

```
eq init = (category: gold) (value: 2000) (age: 50) (discount: 0) .
red check(true, isSafe(init))  .
```

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 following reduction [14]: `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 i s'[1]. 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 the *gold* transition rule is applied to s.

```
eq s = (category: gold) (value: 2000) (age: N) (discount: 0) .
red s =(*, 1)=>+ s' suchThat (not check(isSafe(s),isSafe(s'))) .
```

In the above equation (category: gold) (value: 2000) (age: N) (discount: 0) is an arbitrary state of the rule based system to which gold rule can be applied. CafeOBJ returns false, and thus the induction case is discharged. Consider the case where the *platinum* rule is applied;

```
eq s = (category: platinum) (value: 2000) (age: N) (discount: 0) .
red s =(*, 1)=>+ s' suchThat (not check(isSafe(s),isSafe(s'))) .
```

CafeOBJ returns false for this case, thus the induction case is discharged. Following the same methodology the induction case for the upgrade rule was discharged as well, and thus the proof concludes. The full specification of the e-commerce site, the reasoning about the structure properties and proof of the invariant can be found at [18].

4 Related Work and Discussion

In the area of active databases, a lot of research concerning analysis of rule-based systems exists [19]. A survey on the different approaches of reaction rules can be found in [20]. For example, ECA-LP [21] supports state based knowledge updates including a test case/integrity constraint based verification and validation for transactional updates. Previous attempts, e.g. [22] and [23], propose the visualization of the execution of rules to study their behavior where rules can be shown in different levels of abstraction.

More recent approaches related to the application of formal methods for analyzing rule based systems and relevant to ours, include the following; In [24] 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 CP Solver that derive 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 [3] authors analyze the behavior of Event Condition Action rules by translating

[1] To modularly verify each transition rule separately we usually, define for each such transition a new module which only contains one transition rule at a time.

them into an extended Petri net and verify termination and confluence properties of a light control system expressed in terms of ECA rules. [4] presents an approach to verify the behavior of Event Condition Action 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.

Our approach for the verification of rule programs is based on a different formalism; in particular it uses the OTS/CafeOBJ method and rewriting logic. 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 [14]. Compared to existing similar approaches, it has the following contributions.

First, compared to [3] where structure errors are formally analyzed, our methodology can be used for the verification of both structure (confluence and termination) and safety properties for the specified rule system. This extends our previous work [6] where only safety properties could be proved. Second, when proving safety properties both model checking and theorem proving techniques can be applied, in contrast to [4] and [24] 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 when 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 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 [25].

However the proposed methodology does not come without limitations. When a proof is constructed by humans, they might forget cases to consider or proofs of some lemmas in the proof [10]. To solve this issue, some tools have been developed such as Creme [26] and Gateau [27]. We are also working towards developing a tool that will automate the OTS/CafeOBJ verification method using the Athena proof system [28]. Another possible limitation could be the fact that researchers should be familiar with the CafeOBJ formalism in order to use the proposed approach. However, we believe that the mapping from reactive to rewrite rules is natural enough and the verification method has a clear structure, thus allowing non-expert users to adopt our methodology with minimum effort.

5 Conclusion

We have argued that even though most commercial rule engines provide some support to rule testing, the use of formal methods can be beneficial, even required at some cases, for ensuring the proper behavior of complex and critical rule based systems. For this reason we proposed a methodology for expressing a set of reactive rules into rewrite theories in CafeOBJ that can be used to; (1) formally specify reactive rules, (2) detect structure errors, like confluence and termination, and (3) prove invariant safety properties of the specified reactive rule based system, via theorem proving and model checking techniques. This diversity of options to verification (techniques and properties) offered by the resulted form of the reactive rules and the underlying logic, consists the main contribution of our work.

To conclude, this methodology allows the verification of the knowledge base against the specification of the reactive rule based system and thus formal proofs about its consistency can be obtained. As a future work we intend to conduct more case studies using the proposed methodology so that it can be extended to support most rule-based systems and apply it to a real world application. Another future direction is to develop a tool to automate (part of) the mapping from reactive rules to a rewrite specification written in CafeOBJ. Finally, we intend to use similar formal techniques in order to verify an open-source rule engine implementation. Thus together with the proposed framework, end-to-end validation will be enabled.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS.

References

1. Berstel, B., Bonnard, P., Bry, F., Eckert, M., Pătrânjan, P.-L.: Reactive rules on the web. In: Antoniou, G., Aßmann, U., Baroglio, C., Decker, S., Henze, N., Patranjan, P.-L., Tolksdorf, R. (eds.) Reasoning Web 2007. LNCS, vol. 4636, pp. 183–239. Springer, Heidelberg (2007)
2. Paschke, A.: 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 (2005)
3. Jin, X., Lembachar, Y., Ciardo, G.: Symbolic verication of ECA rules. In: International Workshop on Petri Nets and Software Engineering (PNSE 2013) and International Workshop on Modeling and Business Environments (ModBE 2013), pp. 41–59 (2013)
4. Ericsson, A., Berndtsson, M., Pettersson, P.: Verification of an industrial rule-based manufacturing system using REX. In: 1st International Workshop on Complex Event Processing for Future Internet, iCEP-FIS (2008)
5. Ksystra, K., Triantafyllou, N., Stefaneas, P.: On the Algebraic Semantics of Reactive Rules. In: Bikakis, A., Giurca, A. (eds.) RuleML 2012. LNCS, vol. 7438, pp. 136–150. Springer, Heidelberg (2012)
6. Ksystra, K., Stefaneas, P., Frangos, P.: An Algebraic Framework for Modeling of Reactive Rule-Based Intelligent Agents. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 407–418. Springer, Heidelberg (2014)
7. Goguen, J., Malcolm, G.: A hidden agenda. Theoretical Computer Science 245(1), 55–101 (2000)

8. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 170–184. Springer, Heidelberg (2003)
9. Ogata, K., Futatsugi, K.: Some Tips on Writing Proof Scores in the OTS/CafeOBJ method. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Goguen Festschrift. LNCS, vol. 4060, pp. 596–615. Springer, Heidelberg (2006)
10. Ogata, K., Futatsugi, K.: Proof Score Approach to Analysis of Electronic Commerce Protocols. Int. J. Soft. Eng. Knowl. Eng. 20(253), 253–287 (2010)
11. Ogata, K., Futatsugi, K.: Proof score approach to verification of liveness properties. IEICE Transactions E91-D, 2804–2817 (2008)
12. Diaconescu, R., Futatsugi, K.: CafeOBJ report: The language, proof techniques, and methodologies for object-oriented algebraic specification. AMAST Series in Computing. World Scientific, Singapore (1998)
13. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In: Software Engineering with OBJ: Algebraic Specification in Action. Kluwer (2000)
14. Ogata, K., Futatsugi, K.: Theorem Proving Based on Proof Scores for Rewrite Theory Specifications of OTSs. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Futatsugi Festschrift. LNCS, vol. 8373, pp. 630–656. Springer, Heidelberg (2014)
15. Diaconescu, R., Futatsugi, K., Iida, S.: CafeOBJ Jewels. In: Futatsugi, K., Nakagawa, A.T., Tamai, T. (eds.) CAFE: An Industiral-Strength Algebraic Formal Method, pp. 33–60. Elsevier (2000)
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
17. `http://maude.cs.uiuc.edu/`
18. `http://cafeobjntua.blogspot.com/`
19. Vlahavas, I., Bassiliades, N.: Parallel, object-oriented, and active knowledge base systems. Kluwer Academic Publishers, Norwell (1998)
20. Paschke, A., Kozlenkov, A.: Rule-Based Event Processing and Reaction Rules. In: Governatori, G., Hall, J., Paschke, A. (eds.) RuleML 2009. LNCS, vol. 5858, pp. 53–66. Springer, Heidelberg (2009)
21. Paschke, A.: ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language. In: Int. Conf. on Rules and Rule Markup Languages for the Semantic Web, Athens, Georgia, USA (2006)
22. Fors, T.: Visualization of rule behaviour in active databases. In: VDB, pp. 215–231 (1995)
23. Benazet, E., Guehl, H., Bouzeghoub, M.: A visual tool for analysis of rules behaviour in active databases. In: Sellis, T. (ed.) RIDS 1995. LNCS, vol. 985, pp. 182–196. Springer, Heidelberg (1995)
24. Berstel, B., Leconte, M.: Using Constraints to Verify Properties of Rule Programs. In: ICST Third International Conference on Software Testing, Verification and Validation, Paris, France (2010)
25. Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical Foundations and Methodologies. Computing and Informatics 22, 257–283 (2003)
26. Nakano, M., Ogata, K., Nakamura, M., Futatsugi, K.: Creme: An Automatic Invariant Prover of Behavioral Specifications. Int. J. Soft. Eng. Knowl. Eng. 17(6), 783–804 (2007)
27. Seino, T., Ogata, K., Futatsugi, K.: A toolkit for generating and displaying proof scores in the OTS/CafeOBJ method. ENTCS 147(1), 57–72 (2006)
28. `http://www.proofcentral.org/athena/`