An Algebraic Framework for Modeling of Reactive Rule-Based Intelligent Agents

Katerina Ksystra, Petros Stefaneas, and Panayiotis Frangos

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

Abstract. As the use of intelligent agents in critical domains increases, the need for verifying their behavior becomes stronger. Reactive rules are the main reasoning formalism for intelligent agents. For this reason, we propose the use of the OTS/CafeOBJ method for the specification of reactive rules, which will permit the verification of safety properties for reactive rule-based intelligent agents.

Keywords: intelligent agents, reactive rules, CafeOBJ, Observational Transition Systems, rule-based system.

1 Introduction

Intelligent agents are a new paradigm for developing software applications. An intelligent agent is defined either as anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [1], or as a software that carries out some set of operations and acts on behalf of a user [2], or finally as a computational process that implements the autonomous functionality of an application [3]. Agent-based systems usually consist of many agents that communicate with each other and are known as multi-agent systems.

The use of rule-based systems as the main reasoning model of agents that are part of a multi-agent system has been proposed in early attempts. In this approach each agent includes a rule engine and is able to perform rule-based inference [4]. Thus, an intelligent agent is called rule-based, if its behavior and its knowledge are expressed by means of rules.

The task of verifying the behavior of rule-based agents is difficult because rules can interact during execution and this interaction can cause undesirable results [5]. For example, one rule may trigger another rule and cause a chain of rule triggerings. Also, changes to the rule base (add, remove, change rules) can introduce errors in the behavior of the system if the effects of the changes are not examined beforehand. Thus, using rules in critical systems implies that the system's behavior must be extensively analyzed. Formal methods provide powerful means for analyzing system's behavior and can prove really helpful for preventing design errors at an early stage of development. In this paper, we address the problem of formally analyzing reactive rule-based agents as follows:

V. Geffert et al. (Eds.): SOFSEM 2014, LNCS 8327, pp. 407–418, 2014.

- We present an algebraic framework for formally expressing Production and Event-Condition-Action rules (Section 3).
- We use the OTS/CafeOBJ method for the specification of reactive rule-based intelligent agents and the verification of their behavior (Section 3).
- We apply the framework to a supply chain management system and prove security properties in order to demonstrate its effectiveness (Section 4).

The proposed framework offers the ability to formally specify an intelligent agent whose behavior is expressed in terms of reactive rules, to verify its behavior and thus ensure its correctness. This work is in continuation of [6], where Observational Transition System (OTS) semantics were provided for reactive rules.

1.1 Related Work

A lot of research concerning analysis of rule-based systems exists in the area of active databases. For example, in [7] authors present an overview of processing rules in production systems, deductive and active databases. A larger survey on the different approaches of reaction rules can be found in [18]. Most of the approaches addressing formal analysis of such systems however deal with checking properties such as termination, confluence and completeness. One such attempt to verify rule-based systems can be found in [8], where authors use Petri-nets to analyze various types of structure errors such as inconsistency, incompleteness, redundancy and circularity of rules. Also, ECA-LP [15] which is based on a labeled transaction logic semantics, supports state based knowledge updates including a test case based verification and validation for transactional updates.

Few papers targeting the verification of the behavior of active rule-based systems/agents exist. More precisely, in [9] authors describe a reasoning framework for Ambient Intelligence that uses the Event Calculus formalism for reasoning about actions and causality. Also, an approach to verify the behavior of Event-Condition-Action rules is presented in [5] where a tool that transforms such rules to timed automata is developed. Then the Uppaal tool is used to prove desired properties for a rule-based application. This last work is the closest to ours with the difference that in [5] authors use model checking, while our approach uses theorem proving techniques.

Our framework focuses on verifying the behavior of rule-based agents, rather than proving correctness properties or handling problems with negations, mainly for two reasons; first, the verification of such properties has been studied in many other approaches [8] and second the OTS/CafeOBJ method does not study properties about the transitions of the system but analyzes their effects in the system's behavior. We believe that our framework has the following advantages over existing approaches; it can be used for the verification of complex systems due to the simplicity of the CafeOBJ language and its natural affinity for abstraction. Also, it has the ability to specify systems with infinite states (in contrast with approaches that use model-checking techniques) and it allows the reusability not only of the specification code but also of the proofs [17].

2 Observational Transition Systems and CafeOBJ

An Observational Transition System (OTS) is a transition system written in terms of equations [10]. Assuming that there exists a universal state space Y and that each data type we need to use, including their equivalence relationship, has been declared in advance, an OTS S is defined as the triplet $S = \langle O, I, T \rangle$ where:

1. O is a finite set of observers. Each $o \in O$ is a function $o : Y \rightarrow D$, where D is a data type that may differ from observer to observer. Given an OTS S and two states u_1, $u_2 \in Y$ the equivalence $u_1 =_s u_2$ between them with respect to S is defined as; $\forall o \in O$, $o(u_1) = o(u_2)$ i.e. two states are considered behaviorally equivalent if all the observers return for these states the same data values.
2. I is the set of initial states, such that $I \subseteq Y$.
3. T is a set of conditional transitions. Each $\tau \in T$ is a function $\tau : Y \rightarrow Y$ and preserves the equivalence between two states; if $u_1 =_s u_2$ then $\tau(u_1) =_s \tau(u_2)$. For each $u \in Y$, $\tau(u)$ is called the successor state of u wrt τ. The condition $c\text{-}\tau$ is called the effective condition of τ. Also, for each $u \in Y$, $c\text{-}\tau(u) = false \Rightarrow u =_s \tau(u)$. Finally, observers and transitions may be parameterized by data type values.

Observational transition systems can be described as behavioral specifications in CafeOBJ, an algebraic specification language and processor [11],[21]. In a CafeOBJ module we can declare sorts, operators, variables and equations. There exists two kinds of sorts; a visible sort denotes an abstract data type and a hidden sort denotes the state space of an abstract machine. Two kinds of behavioral operators can be applied to hidden sorts: action and observation operators. An observation operator can only be used to observe the inside of an abstract machine while an action operator can change its state. Finally, CafeOBJ system rewrites a given term by regarding equations as left-to-right rewrite rules.

CafeOBJ is used to specify OTSs [10]. The universal state space Y of an OTS is denoted in CafeOBJ by a hidden sort and an observer by an observation operator. Any initial state in I is denoted by a constant and a transition by an action operator. The transitions are defined by describing what the value, returned by each observer in the successor state, becomes when the transitions are applied in an arbitrary state u. Finally, for expressing the effective conditions, conditional equations are used.

3 An Algebraic Framework for Reactive Rules

The proposed framework aims to enhance reactive rules with verification capabilities. More precisely, it supports Event Condition Action and Production rules. This will allow proving desired safety properties about intelligent agents/systems whose behavior is expressed in terms of such rules. Because we are interested in proving application specific properties, additional characteristics (observers

and/or transitions) about the specific system will be required in order to specify its behavior. These characteristics will differ from application to application and thus the specification cannot become fully automated. This framework however will serve as the basis for specifying and verifying reactive rule based systems and most importantly for capturing the semantics of their rules.

3.1 Production Rules in CafeOBJ

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*. Some usual predefined actions supported by Rule Markup languages are: add, retract, update knowledge and generic actions with external effects [12].

A Production rule can be naturally expressed in our framework if we map the action of the rule to a transition which has as effective condition the condition of the rule. Also, since most of the actions correspond to changes of the knowledge base in order to describe their effects we need an observer that will observe the knowledge base (KB) at any given time. Thus, the observer $knowledge : Y \rightarrow Setof Bool$ which returns the set of boolean elements that belong to the knowledge base is needed. For expressing the functionalities of the KB, the following operators are required; `/in` which returns true if an element belongs to the knowledge base, | which denotes that an element is added to the KB and / which denotes that an element is removed from the KB. Formally, the definition of a set of Production rules as an OTS is presented below.

Definition 1. *Assume the universal state space Y and the following set of Production rules; $\{if\, C_i\ do\ A_i,\ i = 1, \ldots, n \in \mathbb{N}\}$, where without harm of generality we also assume that the conditions of the rules are disjoint. We define an OTS $S = \langle O, I, T\rangle$ from this set of rules as follows:*

- $O = \{O' \cup knowledge\}$
- $T = \{A_i\}$
- $I =$ *the set of initial states, such that* $I \subseteq Y$

In the above definition, O' denotes the rest of the system's observers. Transitions are the actions of the rules, $A_i : Y D_1 \ldots D_l \rightarrow Y$. They can be generic actions (with external changes) or the usual predefined actions $assert : Y Bool \rightarrow Y$ (add a fact to KB), $retract : Y Bool \rightarrow Y$ (remove a fact from KB), $update : Y Bool\, Bool \rightarrow Y$ (remove/add a fact) [20]. Facts are denoted by boolean-sorted CafeOBJ terms. Formally, the actions of Production rules are defined as transitions through the following steps;

1. The effective condition of an action A_i is defined as; `eq c-Ai(u,d1,...,dn) = Ci(d1,...,dn) /in knowledge(u)`.
2. If A_i is an assert action its effect on the knowledge observer is defined as; `knowledge(assert(u,ki(d1,...,dn))) = ki(d1,..,dn)|knowledge(u) if c-assert(u,ki(d1,...,dn))`.

3. 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(d1,..,dn) if c-retract(u,ki(d1,..,dn))`.
4. If A_i is an update action its effect on the knowledge observer is defined as; `knowledge(update(ki(d1,..,dn),kj(d1,..,dn))) = (knowledge(u) /ki(d1,..,dn))|kj(d1,..,dn) if c-update(ki(d1,..,dn),kj(d1,... ,dn))`.
5. If A_i is a generic action, we define; `knowledge(ai(u,d1,...,dn)) = ai(d1 ,...,dn)|knowledge(u) if c-ai(u,d1,...,dn)` and `oi(ai(u,d1,...,dn)) = vi if c-ai(u,d1,..,dn)`.

Step 1 declares that an action a_i can be successfully applied if the condition of the rule holds, i.e. belongs to the knowledge base[1]. Step 2 states that when a transition $assert(u, k_i(d_1, \ldots, d_n))$ is applied successfully in an arbitrary state u, k_i is added to the knowledge base. Where k_i is the fact being asserted. In step 3 it is stated that when the transition $retract(u, k_i(d_1, \ldots, d_n))$ is applied successfully in an arbitrary state u, k_i is removed from the knowledge base. When the transition $update(u, k_i(d_1, \ldots, d_n), k_j(d_1, \ldots, d_n))$ is applied successfully in an arbitrary state u, ki is removed and kj is added, as step 4 defines. Finally, step 5 states that when we have the application of a generic action we add to our KB the information that this action occurred. But generic actions may have side effects and in order to describe them we may have to use additional observers $o_i \in O$ and define how their values change when the action is applied successfully.

3.2 Event Condition Action Rules in CafeOBJ

In contrast to Production rules, Event Condition Action (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*. 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 [13].

In order to express ECA rules in our framework we need an observer 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 in the observer *event-memory* : $Y \rightarrow Nat$. Using *event-memory* 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. However, their semantics differs as the actions of ECA rules can be applied only if their triggering event has been detected first. Formally, the definition of a set of ECA rules as an OTS is presented below.

Definition 2. *Assume the universal 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*

[1] If we have negation-as-failure in the condition of the rule, i.e. if the condition cannot be proved, this is expressed in our framework as; if $c_i \notin knowledge(u)$, since this basically means that there is no information (in our knowledge base) about the condition.

also assume that for $i \neq j$; $E_i, A_i, C_i \neq E_j, A_j, C_j$, *respectively. The OTS* $S = \langle O, I, T \rangle$ *modeling these rules is defined as:*

- $O = \{O' \cup knowledge, event - memory\}$
- $T = \{E_i, A_i\}$
- $I =$ *the set of initial states such that* $I \subseteq Y$

Here, O' is the same as in definition 1. Transitions are the events, $E_i : YD_1 \dots D_n \rightarrow Y$ and the actions, $A_i : YD_1 \dots D_n \rightarrow Y$. Formally, the rule on E_i if C_i do A_i is defined in CafeOBJ terms through the following steps:

1. The effective condition of an event E_i is denoted as $c\text{-}ei(u, d1, \dots, dn)$ and states the conditions under which the system is able to detect the event.
2. The effects of the application of the event E_i in an arbitrary system state u is the following;

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

3. The effective condition of the action A_i, is defined as; `eq c-Ai(u,d1,..,dn) = Ci(d1,...,dn) /in knowledge(u) and ei(d1,...,dn) /in knowledge(u).`
4. The effects of the action A_i, if it is an assert action, is described through the following equations;

```
knowledge(assert(u,ki(d1,..,dn))) = ki(d1,..,dn)|knowledge(u)
/ei(d1,...,dn) if c-assert(u,ki(d1,...,dn)) .
event-memory((u,ki(d1,...,dn))) = null if
c-assert(u,ki(d1,...,dn)) .
```

The effects of the rest of the actions are defined in a similar way. Step 2 states that when the transition/event E_i is applied, the name of the occurred event (e_i) is added to the knowledge base as a fact if in the previous state the detection conditions of the event were true and event-memory was null (denoting that no events had occurred). Also, when the event is applied, event-memory stores the identification number of the event (here i). Step 3 declares that the action will be applied successfully, if the condition of the rule belongs to the KB and the triggering event of the action has been detected. In step 4 it is stated that when the action $assert(u, ki(d1, \dots, dn))$ is applied, the fact k_i is added to the knowledge base and its triggering event is consumed, i.e. its name is removed from the knowledge observer. Also, *event-memory* becomes null.

We must mention here that as we will see in the following section, sometimes the names of the events are not removed from the observer event-memory if they are required for the detection of complex events. Also, if many rules (either Production or ECA) can be executed at the same time, a selection function is

used from the inference engine of the system such as those presented in [14], [15]. It is quite straightforward to include this characteristic in our framework but is out of the scope of this paper.

One of the challenges we met while expressing these rules into our framework was the difference between events and actions, i.e. while events *can* occur at anytime and can be straightforwardly mapped to transitions, actions *must* be executed after the detection of their triggering events. To capture this difference we used the observer event-memory. Initially it returns the value null (meaning that no events have been detected) denoting that any event can occur, but when an event is detected then the only applicable transition in the system is the action of the detected event.

3.3 Complex Events Definition

Sometimes ECA rules react to the detection of complex events. Complex events are created by primitive event(s) and event operator(s). A typical set of event operators for defining complex events include the following; Xor (Mutually Exclusive), Disjunction (Or), Conjunction (And), Any, Concurrent (Parallel), Sequence (Ordered), Aperiodic, Periodic. In [14] definitions of such operators are presented in more details. In this section we will present how the basic event operators can be expressed in our framework.

Assume primitive events A_i and B_j defined as transitions with effective conditions $c\text{-}A_i$ and $c\text{-}B_j$ respectively. Complex event $Xor(Ai, Bj)$ means that either event Ai happens or Bj, but not both. The application of the complex event/transition $e_k : xor(u, Ai, Bj)$ to an arbitrary system state is defined as:

```
knowledge(xor(u,Ai,Bj)) = xor(Ai,Bj)|knowledge(u) if
Ai /in knowledge(u) xor Bj /in knowledge(u) .
event-memory(xor(u,Ai,Bj)) = k if Ai /in knowledge(u) xor
Bj /in knowledge(u) .
```

The above equations state that the complex event is detected (its occurrence is added to the KB) if its detection conditions are fulfilled, i.e. if we have detected either the primitive event Ai or event Bj. Also, the observer event-memory stores the id number k of the event (where xor is a built-in operator) if the same conditions hold. $Disjunction(Ai, Bj)$ means that either event Ai happens or Bj (or both). In a similar way, the application of the event $disjunction(u, Ai, Bj)$ is defined as; `knowledge(disjunction(u,Ai,Bj)) = disjunction(Ai,Bj)| knowledge(u) if Ai /in knowledge(u) or Bj /in knowledge(u)`.

$Conjunction(Ai, Bj)$ means that both events Ai and Bj occur in any order. The application of the event $conjunction(u, Ai, Bj)$ is defined as; `knowledge (conjuction(u,Ai,Bj)) = conjuction(Ai,Bj)|knowledge(u) if Ai /in knowledge(u) and Bj /in knowledge(u)`.

$Sequence(Ai, Bj)$ corresponds to the ordered execution of events Ai and Bj. The application of $sequence(u, Ai, Bj)$ is defined as; `knowledge(Bj(u))= sequence(Ai,Bj)|knowledge(u) if Ai /in knowledge(u) and event-`

`memory(u) = i`. This complex event is detected (its occurrence is added to KB) during the occurrence of event Bj, which can occur if in the previous state Ai had occurred, i.e. event-memory had stored i (and not if the memory is equal to null). By using the observer event-memory and declaring which event had occurred before we can avoid the unintended semantics these operators can have, which are caused because the events, in the active database sense, are treated as if they occur at an atomic instant. This problem is discussed in [15], [19] where also a solution is proposed by defining an interval-based effect semantics in terms of an interval-based event calculus formalization. The alternative interval-based semantics could be implemented in our framework by extending the definition of an event with the time of its occurrence and introducing the notions of event and time intervals. The rest event operators (Concurrent, Aperiodic and Periodic), which are used less often, cannot be straightforwardly expressed in our framework and an extension is required in order to include them as well.

4 Case Study: A Supply Chain Management System

To demonstrate the expressiveness of our framework we applied it to an industrial case study that uses Event Condition Action rules to control the activities of its agents. These activities are inter-enterprise business processes and thus their verification is an important task. In [16] authors present an integrated workflow-supported supply chain management system that was developed so that Nanjing Jin Cheng Motorcycle Corporation in China and its suppliers could handle better their inner processes. The proposed system consists of a set of business function agents whose tasks are to deal with outsourcing, production planning, sales, customer service, inventory, and so on. Each agent is an autonomic and independent entity. ECA rules are used to control the execution sequence of agents' activities. These rules are presented in table 1. A more detailed description of the system is presented in appendix A.

Table 1. ECA rules controlling the activities of the manufacturer

R1	On end(sales) Do st(charge)	R5	On end(ManufacturePlan) if isMaterialsEnough Do st(Manufacture)
R2	On end(sales) and end(charge) if payment $\geqq$ totalprice Do st(QueryInventory)	R6	On end(ManufacturePlan) if not isMaterialsEnough Do st(Outsource)
R3	On end(queryinventory) if IsGoodsEnough Do st(DeliverGoods)	R7	On end(Outsource) if ArrivedMaterials Do st(Manufacture)
R4	On end(queryinventory) if not IsGoodsEnough Do st(ManufacturePlan)	R8	On end(Manufacture) Do st(DeliverGoods)

4.1 Formal Specification and Verification of the System

Rules R1-R8 were expressed in our framework according to the previous definitions. For example, the first rule was defined in CafeOBJ using the transitions `endsales` and `stcharge`. The first transition represents the event part of the rule and the second the action. The definition of the transition `endsales` can be seen below:

```
-- endsales
op c-endsales : Sys department customer Nat -> Bool
eq c-endsales(S,Sales,C,N) = (order(S,Sales,C) = N)
and (event-memory(S) = null) .
ceq knowledge(endsales(S,Sales,C,N)) = (endsales|knowledge(S))
if c-endsales(S,Sales,C,N) .
ceq event-memory(endsales(S,Sales,C,N)) = 1
if c-endsales(S,Sales,C,N) .
```

The effective condition `c-endsales` denotes that the event endsales can be detected when the sales department receives an order from a customer and if no other event had been detected in the previous state. The observer `order` returns the cost of the order a department receives from a customer. When the event is successfully detected its name enters the knowledge base and event-memory stores its identification number. The transition `stcharge` is defined as follows:

```
-- stcharge
op c-stcharge : Sys Nat customer -> Bool
eq c-stcharge(S,N,C1) = (endsales /in knowledge(S)) and
(event-memory(S) = 1) .
ceq event-memory(stcharge(S,N,C1)) = null if c-stcharge(S,N,C1) .
eq knowledge(stcharge(S,N,C1)) = knowledge(S) .
ceq payment(stcharge(S,N,C1),C2) = pending if c-stcharge(S,N,C1)
and (C1 = C2) .
```

The effective condition `c-stcharge` denotes that the action stcharge will occur if endsales belongs to the KB and event memory contains the id number of the event. After the execution of the action, the observer event-memory becomes null, knowledge base stays the same (because the occurrence of endsales event is needed for the detection of the complex event end(sales) *and* end(charge) of R2) and the payment of the customer is pending until a receipt is received. The sixth rule was defined in CafeOBJ using the transitions `stoutsource` and `endmanufactureplan`. The definition of the transition `endmanufactureplan` is presented below;

```
-- endmanufactureplan
op c-endmanufactureplan : Sys bill inventory -> Bool
eq c-endmanufactureplan(S,B,I) = (materials(S,B,I) = computed)
and (event-memory(S) = null)  .
```

```
ceq knowledge(endmanufactureplan(S,B,I)) = (endmanufactureplan|
knowledge(S)) if c-endmanufactureplan(S,B,I) .
ceq event-memory(endmanufactureplan(S,B,I)) = 5 if
c-endmanufactureplan (S,B,I) .
```

The effective condition `c-endmanufactureplan` denotes that the event can be detected when it is computed if there are enough materials to produce goods for the order and if event-memory is null. When the event is detected the name of the event enters the knowledge base and the observer event-memory stores the number of the event, i.e. 5. The transition `stoutsource` is defined as follows;

```
-- stoutsource
op c-stoutsource : Sys bill inventory agent -> Bool
eq c-stoutsource(S,B,I,A) = endmanufactureplan /in knowledge(S)
and (event-memory(S) = 5) and (materials(S,B,I) < enough) .
ceq knowledge(stoutsource(S,B,I,A)) = (knowledge(S)/
endmanufactureplan) if c-stoutsource(S,B,I,A) .
ceq event-memory(stoutsource(S,B,I,A)) = null if
c-stoutsource(S,B,I,A) .
ceq list(stoutsource(S,B,I,A),A) = true if
c-stoutsource(S,B,I,A) .
```

The effective condition `c-stoutsource` declares that the action can be successfully applied if the event endmanufactureplan has been detected and the condition of the action holds, i.e. the materials are not enough. When the action occurs, the observer event-memory becomes null, the occurrence of the event is removed from the knowledge base and a list is sent to the outsourcing agent.

In a similar way we expressed all the rules in our framework. We also defined the transitions whose occurrence makes the detection conditions of the events true. For example, in order to detect the event endmanufacture, the products for the order must have been produced. Thus, we defined the transition `produceproducts`. When this transition is successfully applied, the value of the observer products becomes "produced", indicating that the event endmanufacture can be detected; `ceq products(produceproducts(S)) = produced if c-produceproducts(S) .`

In the above case study, the events may seem as simple propositional representations, or similar in format, but in the context of the whole specification they fully express the functionalities of the system (appendix A). In order to specify this manufacturer agent, 18 transitions (12 that correspond to events and actions and 6 external transitions) and 14 observers were needed.

The most important feature of the proposed framework is the ability to verify the behavior of reactive rule-based intelligent agents using the proof score methodology [10,17]. The type of properties that can be proved with the framework are safety properties, that hold in any reachable state of the system (called invariant properties), and liveness properties, which denote that something will eventually happen. For the supply chain system of the previous section, we

proved that the process of delivering the goods to the customer must not be activated if the payment of the customer does not cover the total cost of the order. This is an invariant property, important for the purpose of the system. Invariant 1, is defined in CafeOBJ terms as; `eq inv1(S,C) = not(not(payment(S,C) >= cost(S,C)) and (delivered(S,C) = true)).` Following the CafeOBJ/ OTS method [10,17] we successfully verified invariant 1 and two more invariants that were needed to conclude the proof (for more details see appendix B). The full specification, the proofs and the appendices can be found at http://cafeobjntua.wordpress.com.

5 Conclusions and Future Work

We believe that due to the fact that reactive rule-based intelligent agents are increasingly used in critical systems, there is a strong need for ensuring their intended behavior. This task is difficult because rules interact during execution and thus can have complex and unpredictable behavior. For this reason we have presented a framework for formally specifying reactive rules with the help of the OTS/CafeOBJ method. This framework can express complex systems while capturing the semantics of the underlying reactive rules, and can be used for the verification of safety properties reactive rule-based agents should meet. In order to demonstrate its effectiveness, we have applied it to a case study of a manufacturer business agent. In the future, we intend to develop a tool that will automatically translate a set of reactive rules, written in a Rule Markup language, to CafeOBJ and that will support online verification. Finally, this framework could be extended for modeling operational reactive systems that need to define an optimized proof-theoretic and operational semantics.

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. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 1st edn. Prentice Hall (1995)
2. Gilbert, D.: Intelligent Agents: The Right Information at the Right Time. IBM Intelligent Agent White Paper
3. FIPA (Foundation for Intelligent Physical Agents), `www.fipa.org`

4. Badica, C., Braubach, L., Paschke, A.: Rule-based Distributed and Agent Systems. In: 5th International Conference on Rule-Based Reasoning, Programming, and Applications, RuleML 2011, pp. 3–28. Springer (2011)
5. 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)
6. 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)
7. Vlahavas, I., Bassiliades, N.: Parallel, object-oriented, and active knowledge base systems. Kluwer Academic Publishers, Norwell (1998)
8. Xudong, H., Chu, C., Yang, H., Yang, S.J.H.: A New Approach to Verify Rule-Based Systems Using Petri Nets. Information and Software Technology 45(10), 663–669 (2003)
9. Patkos, T., Chrysakis, I., Bikakis, A., Plexousakis, D., Antoniou, G.: A Reasoning Framework for Ambient Intelligence. In: Konstantopoulos, S., Perantonis, S., Karkaletsis, V., Spyropoulos, C.D., Vouros, G. (eds.) SETN 2010. LNCS (LNAI), vol. 6040, pp. 213–222. Springer, Heidelberg (2010)
10. 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)
11. 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)
12. Paschke, A., Boley, H., Zhao, Z., Teymourian, K., Athan, T.: Reaction RuleML 1.0: Standardized Semantic Reaction Rules. In: Bikakis, A., Giurca, A. (eds.) RuleML 2012. LNCS, vol. 7438, pp. 100–119. Springer, Heidelberg (2012)
13. 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)
14. Paschke, A., Boley, H.: Rules Capturing Events and Reactivity. In: Giurca, A., Gasevic, D., Taveter, K. (eds.) Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, pp. 215–252. IGI Publishing (2009)
15. 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)
16. Liua, J., Zhangb, J., Hub, J.: A case study of an inter-enterprise workflow-supported supply chain management system. Information and Management 42, 441–454 (2005)
17. Futatsugi, K., Goguen, J.A., Ogata, K.: Verifying Design with Proof Scores. Verified Software: Theories, Tools, Experiments 4171, 277–290 (2005)
18. 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)
19. Teymourian, K., Paschke, A.: Semantic Rule-Based Complex Event Processing. In: Governatori, G., Hall, J., Paschke, A. (eds.) RuleML 2009. LNCS, vol. 5858, pp. 82–92. Springer, Heidelberg (2009)
20. Reaction RuleML, `http://ruleml.org/reaction`
21. Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical Foundations and Methodologies. Computing and Informatics 22, 257–283 (2003)