Formal Specification of Multi-agent Systems by Using EUSMs*

Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez

Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid. Spain
mgmerayo@fdi.ucm.es, {mn,isrodrig}@sip.ucm.es

Abstract. The behavior of e-commerce agents can be defined at different levels of abstraction. A formalism allowing to define them in terms of their *economic* activities, *Utility State Machines*, has been proposed. Due to its high level of abstraction, this formalism focuses on describing the economic *goals* rather on *how* they are achieved. Though this approach is suitable to *specify* the objectives of e-commerce agents, as well as to construct formal analysis methodologies, this framework is not suitable to define the *strategic* behavior of agents. In this paper we develop a new formalism to explicitly define the strategic behavior of agents in a modular way. In particular, we reinterpret the role of *utility functions*, already used in USMs in a more restrictive manner, so that they define strategic preferences and activities of agents. We apply the formalism to define the agents in a benchmark e-commerce agent environment, the *Supply Chain Management Game*. Since the strategic behavior of agents is located in a specific part of the formalism, different strategies can be easily considered, which enhances the reusability of the proposed specification.

Keywords: Formal specification of multi-agent systems, autonomous agents, e-commerce.

1 Introduction

One of the most interesting applications of agent-oriented computation is the area of *agent-mediated e-commerce* [1,2,3,4,5]. These agents can look for profitable offers, recommend products, negotiate with other agents, or, even autonomously, perform transactions on behalf of their respective users. However, users may be reluctant to delegate activities that dramatically affect their possessions. On the one hand, they may think that agents are biased by the manufacturer for commercial reasons. On the other hand, some users would never (voluntarily) delegate critical activities to *intelligent* entities.

* This research was partially supported by the Spanish MEC projects MASTER/TERMAS TIC2003-07848-C02-01 and WEST/FAST TIN2006-15578-C02-01, the Junta de Castilla-La Mancha project PAC06-0008, the Comunidad de Madrid project to fund research groups CAM-910606,‘ and the Marie Curie project MRTN-CT-2003-505121/TAROT.

The use of *formal methods* through the different stages of a system development increases the chance of finding and isolating mistakes in early phases of the system creation. Formal languages are used to define the critical aspects of the system under consideration, that is, to create a *model* of the system. Formal specification languages and formal semantics have been already used in the field of agent-oriented systems (see e.g. [6,7,8]). Then, some theoretical machinery allows to analyze the model and extract some relevant properties concerning the aspects included in the model. In this line we can mention *model checking* [9] and its application to agent systems [10]. We can also consider that the model is the *specification* of the system under construction. Thus, we can compare its behavior with that of a real system and *test* whether it correctly implements the model [11]. Such approach is taken in the agents field, for instance, in [12]. Regarding agent-mediated e-commerce, formal methods can be used to analyze the *high-level* requirements of agents. These requirements can be defined in economic terms. Basically, the high-level objective of an e-commerce agent is *"get what the user said he wants and when he wants it"*. Let us note that *what the user wants* might include not only what goods or services he wants (e.g., DVD movies) but also other conditions (e.g., he wants to keep his information private, he wants to perform only legal transactions, etc).

Following these ideas, a formalism allowing to *specify* the high-level behavior of autonomous e-commerce agents, as well as to test agents with respect to their specifications, is proposed in [13,14,15]. Users objectives are defined in terms of their preferences, denoted in turn by means of *utility functions*. A utility function associates a numerical value with each basket of resources, where a *resource* is any scarce good that can be traded in the market. Preferences may change as long as the time passes and objectives are fulfilled. Regarding the former, time is used as a parameter in utility functions, that is, if $\bar{x}$ and $\bar{y}$ are baskets of resources and f is a utility function, $f(\bar{x}, t) > f(\bar{y}, t)$ means that, at time t, $\bar{x}$ is preferred to $\bar{y}$. Concerning the latter, the formalism allows agents to behave according to different utility functions depending on the current *state* of the agent. We called *Utility State Machines* to our formalism. A USM is inspired in the notion of *finite state machine* but it is more powerful. A set of variables denote the current possessions of the user (i.e., the resources it can exchange with other agents) and each state is provided with a utility function denoting the objectives in the state. USMs evolve by changing their state, by performing an exchange with other USMs, or by waiting for some time. By performing these activities, they represent the economic behavior of an autonomous e-commerce agent. The work in USMs has also been supported by the specification of small case studies such as the specification of all the entities involved in the Kasbah system [16].

Even though there is a well-founded theory underlying USMs, the application of this formalism to the specification of complex agents has not been as successful as expected. In fact, our experimentation has detected some features of the original formalism that are almost not needed when specifying a wide range of different agents. In contrast, although USMs have a big expressive power, this experimentation has also revealed that there are several agents characteristics that can be expressed only in a very difficult and intricate way. Thus, we have decided to create a new formalism based on USMs but overcoming most of its drawbacks. We call this new specification language *extended utility state machines*, in short, `EUSM`. The term *extended* indicates not only that these new machines are an *extension* of the previous formalism; it also shows that these new machines are more powerful since the specification of some agent properties can be done in a more straight way. Among the several modifications and additions, we can remark the following structural contributions. First, utility functions allow now to specify not only short-term behavior but also strategic behavior. Second, `EUSMs` allow to create general patterns to build agents; by modifying some utility functions located in specific places, it is easy to define agents that, following the same behavior, present completely different characteristics.

In order to illustrate the features of our formalism, we apply it to the specification of a system that can be considered as a *benchmark* in the agent-mediated e-commerce community: The *Supply Chain Management Game* [17]. In a few words, in this system, agents face a simulated environment where they interact with suppliers and customers to stock up with components and sell constructed products, respectively. Next we briefly introduce the main components in the *Supply Chain Management Game*. We restrict ourselves to the most relevant aspects of the system. The system simulates an environment where PCs are traded. Thus, the system contains final consumers, vendors, and component providers. The global behavior is very simple: Clients buy PCs to vendors, which in turn buy the needed components to providers, ensemble them, and sell the final PCs to clients. While consumers and providers are simulated by the system, vendors have to be implemented by the different teams taking part in the game. As expected, the goal of these agents is to earn as much money as possible. Once these agents simulating vendors are added to the system, they have to take autonomous decisions. Each agent communicates with clients and component providers by following a simple protocol. Agents receive *requests for quotes* from the clients. Then, the agent sends quotes to the chosen clients (each agent can decide with which clients it wants to make business). Next, the clients accept the quotes that they like. Finally, the last communication is the transaction itself. It is worth to point out that each quote contains not only the price and the number of desired pieces, but also the reception date as well as the penalty to be paid in case of late delivery. The communication of agents with providers is similar, but exchanging in the previous dialogue the role of agent by the role of provider and the role of client by the role of agent. That is, an agent sends a request for quotes to the providers, and so on. In each game turn, the agents send to their *factories* the requests for assembling some PCs by using the available components that they bought in previous turns. The production capacity of each factory is not unlimited. Moreover, to store both components and PCs has an associated cost.

The rest of the paper is structured as follows. In the next section we relate the approach presented in this paper with some previous work. In Section 3, we introduce the language constructions to define `EUSMs`. We will use the Supply

Chain Management Game to show, along this section, the main `EUSMs` characteristics. In Section 4 we define the operational behavior of `EUSMs` in terms of their transitions. In Section 5 we show how to extend the concepts underlying `EUSMs` to deal with multi-agent systems. In Section 6 we present some general guidelines to define `EUSMs`. Finally, in Section 7 we present our conclusions and some lines for future work.

2 Related Work

Several approaches have been proposed to formally define the behavior of e-commerce systems. Most of them focus on providing models for checking the validity of *communications* in these systems (see e.g.[18,19]). Clearly, these aspects are critical for the reliability of these systems. For instance, features such as *authentication*, *non-repudiation*, *integrity*, and *privacy* are required in this kind of systems [20]. These requirements are critical in other system domains as well, so they are not *specific* to the e-commerce domain. Moreover, in these formal approaches entities are defined in terms of low level activities (e.g. messages), not in terms of their high level behavior (which, in this case, is the *economic* behavior). Thus, considering their behavior in terms of economic concepts such as *preferences*, *utilities*, *profit* or *Pareto equilibria* might be tricky or cumbersome. Similarly, other formalisms allowing to formally define the behavior of autonomous agents have been proposed (see e.g. AgentSpeak(L) [21] or 3APL [22]). Since they are generic and do not focus on any specific agent-oriented application domain, the decision-making procedures of entities are not defined in economic terms either. Some formalisms for defining and analyzing decision-making scenarios have been proposed. For example, we may consider *Markov Decision Processes* (MDP) [23] and *Partially Observable Markov Decision Processes* (POMDP) [24]. In these formalisms, the specifier totally or partially defines the consequences of the agent decisions. For example, we may specify that, in a given state s, the decision a will lead us to either a state s_1 with *profit* 2 or to a state s_2 with *profit* 3, with probabilities 0.25 and 0.75, respectively. Given this kind of information, optimal sequences of decisions can be computed in some cases. Let us note that composing these specifications requires to (partially or totally) know the actual (probabilistic) consequences of taking each decision. Such a condition may be unfeasible in several scenarios where the uncertainty about the environment is too high to approximately define its behavior. However, providing entities with a strategy is also required in these cases. In the approach presented in this paper, we consider that models are provided with a strategy, but we are not concerned with the data or the method followed to *choose* this strategy. On the contrary, the formalism will focus on analyzing the behaviors of systems where each entity is provided with a *given* strategy, regardless of its origin. That is, we will focus on checking the *consequences* of taking these strategies (though this feedback could be useful to choose them).

As we said before, the formalism presented in this paper, *extended utility state machines* (`EUSMs`), are constructed from a previous formalism presented in [13,14,15], *utility state machines* (`USMs`). One could think that, in order to consider the strategic behavior of agents, it is enough to reinterpret the meaning of the *utility functions* associated to states in `EUSMs` so that they denote *strategic* preferences rather than *true* preferences. For instance, let us suppose that I have two apples and that, according to my *true* preferences, I assign the same utility to one apple and to two oranges. Besides, let us suppose that, according to my knowledge, I suspect that the relative price of apples is raising. In this case, perhaps I should not accept *now* an exchange where I give one apple and I receive two oranges, despite of the fact that it is acceptable for my true preferences (note that my true utility would not decrease). Instead, I should act according to some *strategic* preferences. For example, strategic preferences could denote that now I give the same value to one apple and to *five* oranges.

Reinterpreting utility functions in this way in `USMs` is necessary but not sufficient to properly denote agents with strategic behaviors. `USMs` have other characteristics that have to be modified in order to provide a formalism allowing to denote strategic behaviors. First, `USMs` require that agents have a positive utility outcome before a given deadline is reached. Since utility functions will not denote *true* preferences in the new formalism, this is not suitable. Besides, in `USMs` variables only denote the amounts of *resources* owned by the agent and the *time*. However, in order to take strategic decisions, other variables (e.g., a historic file of last transactions) are required. In `USMs`, utility functions are used only to enable/disable exchanges of resources. However, in order to define strategies in the long term, utility functions should also be able to affect other decisions not directly involving transactions. In `USMs`, the only available kind of communication between agents is the *exchange of resources*, but it is not suitable for defining strategic scenarios where a transaction could be just the last step of a long term negotiation process. Actually, since in `USMs` only resource variables are modified in exchanges, it is not possible to affect other variables that could be relevant for the future strategy (e.g., the historic file). Finally, in `USMs` actions associated to transitions do not depend on the actual values that are taken to make conditions to hold, though registering these values could be useful for the strategy. As we will see in the next section, the new formalism, `EUSMs`, will overcome these problems by providing new constructions that enhance the expressivity in these cases.

3 Introducing Extended Utility State Machines

In this section we formally define the state machines that we propose as an appropriate formalism to define agents that incur in e-commerce activities. We put a special emphasis in the novelties with respect to USMs. Essentially, an *extended utility state machine* is a state machine where we introduce some additional features to help in defining e-commerce autonomous agents. Thus, we have several *states* and *transitions* among them. Transitions have an associated *condition*, that is, a transition can be performed only if the condition holds. Each transition also contains an *action*. This action can be either to change the value of some variables that the agent is controlling or to send a message to another agent. A particular case of action is associated with *exchanges of resources*, that is, two agents exchange messages, inducing a common transaction. The *configuration* of a `EUSM` is given by the current state and the current value of the variables. Finally, the *operational semantics* of our machines, indicating how such a machine evolves, is defined as follows:

– If there exists a transition from the current state where the condition holds then this transition can be performed.
– If there is more than one transition with the condition holding then there is a non-deterministic choice among all the possibilities.
– If there does not exist such a transition then the machine can let the time pass until one of the conditions hold.

The previous description of our state machines coincides with the classical notion. The important difference so far comes from the introduction of utility functions. In our framework, each `EUSM` has a function associating a utility function with each state. These functions have three parameters (the value of the variables V, an agent identifier id, and a time t) and return a real number indicating the *utility* of having the values of the variables V, at time t, after interacting with agent id. Thus, one of the main roles of utility functions consists in deciding whether a future exchange of resources with a certain agent will be *good* according to the current available information. So, utility functions are used to guide both current and future exchanges. However, we can give more sophisticated uses. For example, since the utility function takes as parameter all the available variables, we can decide to refuse a purchase at a low price because our record indicates that the prices are decreasing very fast. In this line, utility functions will be used to decide whether a vendor purchases components, at a given price, by taking into account the requests previously received. So, utility functions can be used to define complex strategies. This is a big advantage with respect to USMs [13,15]. In particular, in order to define agents as `EUSMs`, their strategies will be codified in the utility functions. Thus, by leaving the utility functions undefined, we are providing a *reusable pattern* to create new agents having a similar behavior but taking decisions in completely different ways.

We have already mentioned that utility functions will be taking as parameter a set of variables. Next, we describe the different types of variables as well as their roles in the framework of `EUSMs`. A distinguished set of variables will represent *resources*. The idea is that all the agents participating in a multi-agent system denote a specific resource with the same name; the rest of the variables will be considered *private* and they can have arbitrary names. A distinguished variable will be associated with the *buffer* (organized as a fifo queue) storing *incoming messages*. Symmetrically, another variable will refer to the *port* for *outgoing messages*. Any message sent to this port will be received in the buffer associated to the destination agent. The last distinguished variable is *time*.

In order to perform operations, agents will communicate with each other via *messages*. A special kind of message are *exchange* messages. We consider two types of exchange messages: *Proposal* (denoted by *expr*) and *acceptance* (denoted by *exac*). The agent sending a proposal does not modify its resources until it receives an acceptance from the partner. In both cases, these messages include the exchanged resources as well as the *id* of the agent proposing the exchange. This identifier is useful to keep a record of transactions and, in particular, to refer to deals that were reached but not concluded yet (e.g. the goods or the payment were not received yet). For example, if an agent receives the message $(id_4, expr(\{oranges =_\diamond 6, money =_\diamond -3\}))$, this means that the id_4 agent is offering 3 euros to purchase 6 oranges. Similarly, if the agent has as identifier id_7 and, after evaluating the exchange with the utility function corresponding to its current state, the deal is acceptable then the agent will send the message $(id_7, exac(\{oranges =_\diamond -6, money =_\diamond 3\}))$, indicating that it accepts the exchange.

Definition 1. Let M be a data type. A data type L is a *list data type for M* if it is defined as follows: $[\,] \in L$; if $l \in L$ and $m \in M$ then $m \cdot l \in L$.

Let $l \in L$. We consider that l_i denotes the i-th element of l. Besides, $tail(l)$ represents the result of eliminating l_1 in l, while $enqueue(l, m)$ represents the result of inserting m as the new last element of l.

Let $v_1, \ldots, v_n$ be different identifiers and $t_1, \ldots, t_n$ be data types. We say that $V = \{v_1 : t_1, \ldots, v_n : t_n\}$ is a *set of variables*.

Let R and A be disjoint sets of identifier names. We consider

$$S = \left\{ \{r_1 =_\diamond x_1, \ldots, r_k =_\diamond x_k\} \left| \begin{array}{l} \forall\, 1 \leq i \leq k : (r_i \in R \;\wedge\; x_i \in \mathbb{R}) \;\wedge \\ \forall\, 1 \leq i, j \leq k, i \neq j : r_i \neq r_j \end{array} \right. \right\}$$

Let $\texttt{ExchProposal} = \{expr(s) | s \in S\}$ and $\texttt{ExchAccept} = \{exac(s) | s \in S\}$. We say that $M = A \times \texttt{Info}$, for some type $\texttt{Info}$ such that $\texttt{ExchProposal} \cup \texttt{ExchAccept} \subseteq \texttt{Info}$, is a *messages data type* for *resources* R and *agents* A. □

Example 1. Let us consider the set of agents $A = \{Jimmy, Mary, Johnny\}$ and the set of resources $R = \{oranges, apples\}$. Any messages data type for R and A includes, e.g., the messages $(Johnny, expr(\{oranges =_\diamond 3, apples =_\diamond -2\}))$ and $(Mary, exac(\{apples =_\diamond 4\}))$. Moreover, depending on the definition of the type `Info` considered in the previous definition, other messages not involving an exchange proposal ($expr$) or an exchange acceptance ($exac$) could be included as well. For example, if $hello \in \texttt{Info}$ then $(Jimmy, hello)$ is also a message included in the messages data type. □

Next we introduce some concepts related to variables in `EUSMs`. An *extended set of variables* includes some special variables that must be present in the machine (time, buffer of incoming messages, port for outgoing messages). In this set, variable representing resources are explicitly marked. An *assignment* associates each variable identifier with its current value.

Definition 2. Let $V = \{v_1 : t_1, \ldots, v_n : t_n\}$ a set of variables. Let R and A be disjoint sets of identifier names, `Messages` be a messages data type for resources

R and agents A, and `MessagesList` be a list data type for `Messages`. We say that (V, R) is an *extended set of variables* if the following conditions hold:

- $\{v : \mathbb{R} | v \in R\} \subseteq V$. Each resource is represented by a variable in the set.
- $t : \mathbb{R}^+ \in V$ represents the *time*.
- $ib : \texttt{MessagesList} \in V$ represents the *input buffer*.
- $op : \texttt{Messages} \cup \{\perp\} \in V$ represents the message to be sent through the *output port*. The symbol $\perp$ indicates that no message is waiting to be sent.

Let us consider an extended set of variables $E = (V, R)$. We say that a set $\mathcal{V} = \{v_1 =_\diamond a_1, \dots, v_p =_\diamond a_p\}$ is an *assignment* for E if for all $1 \leq i \leq p$ we have $v_i \in V$ and $a_i \in t_i$. The set of all assignments is denoted by `Assign`.

Let $\mathcal{V} \in \texttt{Assign}$ and v be an identifier such that $v \in \{v' | \exists\, y' : v' =_\diamond y' \in \mathcal{V}\}$. The *update* of the variable v with the value x in V, denoted by $\mathcal{V}[v := x]$, is the substitution of the former value of v by x. Formally,

$$\mathcal{V}[v := x] = \{v' =_\diamond y' | v' =_\diamond y' \in \mathcal{V} \ \wedge \ v' \neq v\} \cup \{v =_\diamond x\}$$

We extend this operation to update k variables in the expected way:

$$\mathcal{V}[v_1 := x_1, v_2 := x_2, \dots v_k := x_k] = (\dots((\mathcal{V}[v_1 := x_1])[v_2 := x_2]) \dots [v_k := x_k])$$

Let $\mathcal{V}, \mathcal{V}' \in \texttt{Assign}$ be assignments. The *addition* of $\mathcal{V}$ and $\mathcal{V}'$, denoted by $\mathcal{V} + \mathcal{V}'$, is defined as

$$\begin{gathered}\{v' =_\diamond x' | v' =_\diamond x' \in \mathcal{V} \wedge \ \nexists\, y' : v' =_\diamond y' \in \mathcal{V}'\} \\ \cup \\ \{v' =_\diamond x' | v' =_\diamond x' \in \mathcal{V}' \wedge \ \nexists\, y' : v' =_\diamond y' \in \mathcal{V}\} \\ \cup \\ \{v' =_\diamond x' + y' | v' =_\diamond x' \in \mathcal{V} \ \wedge \ v' =_\diamond y' \in \mathcal{V}'\}\end{gathered}$$

Let A be a set of agent identifier names. A *utility function* is any function $f : A \times \texttt{Assign} \rightarrow \mathbb{R}^+$. The set of all utility functions is denoted by `UtilFuncs`. □

Example 2. The `EUSM` representing a vendor in the Supply Chain Management Game communicates with clients and suppliers by using some messages. Besides, they will keep some variables to control their interactions. Figure 1 shows the messages agents can send/receive to/from customers and suppliers, as well as the set of variables that agents keep. Vendors use some variables to register past interactions with clients and suppliers (requests for quotes to be attended, proposed quotes, actual commitments, and historical transactions). In this way, they know what to expect from them in subsequent turns. These variables are *sets* where each element keeps the corresponding interaction message as well as the sender/receiver of the message and other relevant data. Other variables represent resources, the time consumed in the current state, the remaining capacity of the factory in this turn, and all mandatory variables previously introduced. □

Types of messages:

$ClientRFQ$: Client requests a quote to agent
$crfq(pcModel, units)$
$AgentRFQ$: Agent requests a quote to supplier
$arfq(compModel, units)$
$AgentQ$: An agent sends a quote to a client
$aq(pcModel, units, price, deadline, penalty)$
$SupplierQ$: A supplier sends a quote to an agent
$sq(compModel, units, price, deadline, penalty)$
$AcceptAgentQ$: Client accepts quote from agent
$aaq(pcModel, units, price, deadline, penalty)$
$AcceptSupplierQ$: Agent accepts quote from sup.
$asq(compModel, units, price, deadline, penalty)$
$expr(E)$: The exchange E is proposed
$E \in$ `Assign`
$exac(E)$: The exchange E is accepted
$E \in$ `Assign`

Variables:

Resources:
$money$: money units
$pcModel_x$: units of model x PC
$compModel_y$: units of component y
Mandatory variables: t, ib, op
st: Time `EUSM` moved to the current state
$factoryCap$: # PCs the `EUSM` can still build
- Registering interactions with clients:
$RFQfromClients$: received RFQs
$(client, clientRFQ)$
$QtoClients$: quotes sent to clients
$(client, agentQ)$
$CwithClients$: commitments with clients
$(client, agentQ)$
$HwithClients$: past transactions with clients
$(client, agentQ, delay, finalPrice)$
- Registering interact. with suppliers:
$RFQtoSuppliers$: RFQs asked to suppliers
$(supplier, agentRFQ)$
$QfromSuppliers$: Qs from suppliers
$(supplier, supplierQ)$
$CwithSuppliers$: commitments with sup.
$(supplier, supplierQ)$
$HwithSuppliers$: past transactions with sup.
$(supplier, supplierQ, delay, finalPrice)$

Fig. 1. Messages and variables managed by vendors

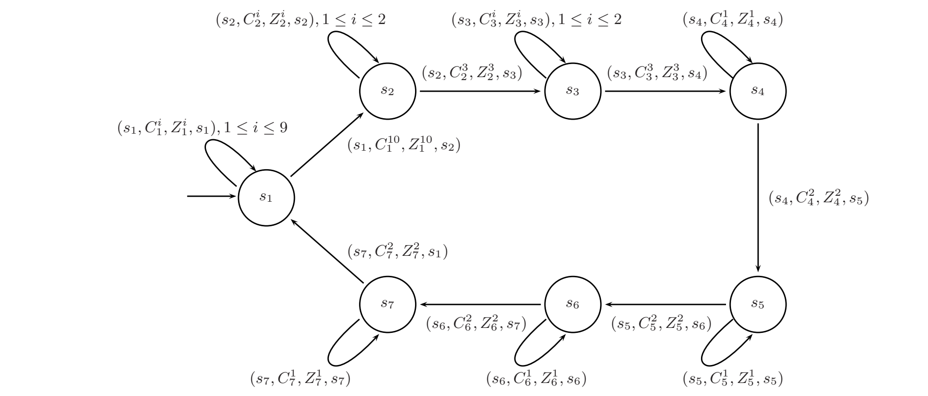
Next we introduce our notion of state machine. In the next definition we use the term *explicit transition* to denote all the transitions contained in the set of transitions. The idea is that there are other transitions that are not included in that set: Time transitions.

Definition 3. An *Extended Utility State Machine*, in the following `EUSM`, is a tuple $M = (id, S, E, U, s_0, \mathcal{V}_0, \mathcal{T})$ where:

– id is the (unique) *agent identifier* of M.
– S is the *set of states* and s_0 is the *initial state*.
– E is the *extended set of variables* and $\mathcal{V}_0$ is the *initial assignment*.
– $U : S \longrightarrow$ `UtilFuncs` is a function associating a utility function with each state.
– $\mathcal{T}$ is the set of *explicit transitions*. For all transition $\gamma = (s_1, C, Z, s_2) \in \mathcal{T}$ we have
  • $s_1, s_2 \in S$ denote the initial and final states of the transition, respectively.
  • C : `Assign` $\times$ `Extra` $\longrightarrow$ `Bool` is the transition *condition*, where `Extra` is a data type.
  • Z : `Assign` $\times$ `Extra` $\rightarrow$ `Assign` is the transition *transformation*.

A *configuration* of M is a tuple $(s, \mathcal{V})$ where $s \in S$ and $\mathcal{V}$ is an assignment for E. $\square$

We consider that all the variables belonging to the extended set of variables appear in the *initial assignment*. As we will see later in Definition 4 the C and



$(s_2, C_2^i, Z_2^i, s_2), 1 \le i \le 2$
$(s_3, C_3^i, Z_3^i, s_3), 1 \le i \le 2$
(s_4, C_4^1, Z_4^1, s_4)
(s_2, C_2^3, Z_2^3, s_3)
(s_3, C_3^3, Z_3^3, s_4)
s_2
s_3
s_4
$(s_1, C_1^i, Z_1^i, s_1), 1 \le i \le 9$
$(s_1, C_1^{10}, Z_1^{10}, s_2)$
s_1
(s_4, C_4^2, Z_4^2, s_5)
(s_7, C_7^2, Z_7^2, s_1)
s_7
s_6
s_5
(s_6, C_6^2, Z_6^2, s_7)
(s_5, C_5^2, Z_5^2, s_6)
(s_7, C_7^1, Z_7^1, s_7)
(s_6, C_6^1, Z_6^1, s_6)
(s_5, C_5^1, Z_5^1, s_5)

Fig. 2. EUSM denoting an agent

Z functions work as follows: If there exists e such that $C(\mathcal{V}, e)$ holds then $\mathcal{V}$ is substituted by $Z(\mathcal{V}, e)$. Let us note that there may exist several values of e such that $C(\mathcal{V}, e)$ holds. In these cases, the EUSM will be provided with different nondeterministic choices. Next we introduce the EUSM defining the behavior of a vendor in our case study. In particular, we show how C and Z can coordinate to find/use the extra parameter e.

Example 3. We consider that the interactions of the vendor with the environment are defined by using several states. Exchange proposals, on the basis of the utility function of the agent associated with the current state, are accepted/rejected. The objective of each state conforming the EUSM consists in dealing with a kind of messages or agent activities. The resulting EUSM will be denoted by $\mathcal{A}$. The graphical presentation of $\mathcal{A}$ is given in Figure 2.

In order to illustrate the behavior of the C and Z functions, let us consider the agent transition $tran_2^1 = (s_2, C_2^1, Z_2^1, s_2)$, which links the state s_2 with itself. This transition defines how the agent takes a client RFQ from its set $RFQfromClients$ and composes a quote q to be sent back to the client. For the sake of simplicity, in this approach the quote *penalty* (i.e., the price reduction due to a delayed delivery per time unit) will be 0. The transition condition C_2^1 requires that a client RFQ of the form $(client, crfq(pcModel_z, units))$ is found in $RFQfromClients$. Besides, C_2^1 requires finding a quote that is profitable for the agent, that is, a quote that would improve the utility in s_2. In order to do that, it searches for some values *price* and *deadline* such that exchanging *units* PCs of model $pcModel_z$ by *price* at time *deadline* would improve the utility of the agent, given by $U(s_2)$. If such values of *price* and *deadline* are found then C_2^1 returns True and these parameters are passed to function Z_2^1. The function Z_2^1 removes the concerned client RFQ to avoid processing it again. Besides, it composes a quote q with the considered parameters *price* and *deadline*. The quote is inserted in the set of quotes $QtoClients$ and it is written in the output port *op* so that it is sent to the client. Formally, we have

$$C_2^1\begin{pmatrix}\mathcal{V},\\(price,\\deadline)\end{pmatrix}=\begin{cases}\texttt{True} & \text{if } (client, crfq(pcModel_z, units)) \in RFQfromClients \\ & \qquad\wedge \\ & U(s_2)\left(client, \mathcal{V}+\begin{Bmatrix}pcModel_z =_\diamond -units,\\ t=_\diamond deadline,\\ money =_\diamond price\end{Bmatrix}\right) \\ & \qquad > \\ & U(s_2)(client, \mathcal{V}+\{t=_\diamond deadline\}) \\ \texttt{False} & \text{otherwise}\end{cases}$$

$$Z_2^1\begin{pmatrix}\mathcal{V},\\(price,\\deadline)\end{pmatrix}=\mathcal{V}\begin{bmatrix}RFQfromClients := \\ RFQfromClients\backslash(client, crfq(pcModel_z, units)),\\ QtoClients := QtoClients \cup \{(client, q)\},\\ op := (client, q)\end{bmatrix}$$

where q is defined as $aq(pcModel_z, units, deadline, price, 0)$.

Next we show a possible utility function to analyze the suitability of transactions in s_2. In the next expression, $prodCost_i$ denotes the estimated production cost of a model i PC. It can be calculated by taking into account $CwithClients$, $CwithSuppliers$, $HwithClients$, and $HwithSuppliers$. Besides, the utility function will discourage sending quotes that are clearly *unacceptable* for the other part, which would be a waste of time. A value $maxAcceptablePrice_{id,z}$ denotes the maximal price the agent estimates the client id would pay for a PC of model z. This value can be computed from $HwithClients$ by considering previous interactions with this or other clients. Let us note that sending a quote to an entity or accepting a quote from one of them (in our system, a supplier) are very different activities in terms of strategy. In the latter case, which is considered in the next state s_3, we would actually accept excessively profitable (low) prices, because the other part implicitly accepted the transaction by sending the quote.

$$U(s_2)(id,\mathcal{V}) = money + \sum_z min(maxAcceptablePrice_{id,z}, (prodCost_z+\delta))\cdot pcModel_z$$

□

Let us comment on the previous expression. It is the addition of some terms. The first term denotes the value given to money, while the rest ones denote the value given to PCs of each model. The relation between the value of money and the value of a given PC model is the key to decide whether an exchange (in this case, a sale) involving this model is acceptable or not. Let us note that the number of PCs of each model is multiplied by a factor. This multiplicative factor implicitly denotes the *exchange ratio* that is acceptable for the agent when PCs of this model are sold. For example, if this factor is equal to 1500, then 1500 units of the resource *money* (that is, \$1500) receive the same value than one PC of the model where this factor is applied. Thus, selling this PC at any price higher than or equal to this value will be acceptable, because in this case the value returned by the utility function will be at least the same as before. Finally, let us note that the multiplicative factor is the minimum between the maximal price the agent thinks the buyer would pay for a PC of this model and the cost of producing it (plus a small amount δ).

4 Evolutions in EUSMs

In order to define the operational behavior of a EUSM, as we said before, we consider two types of evolutions: *Explicit evolutions* (labelled by *exp*) and *temporal evolutions* (labelled by *tm*). An evolution is denoted by a tuple $(c, c')_K$, with $K \in \{exp, tm\}$, where c is the former configuration and c' is the new configuration. Let us note that, for a given c and K, there might exist several configurations c' such that $(c, c')_K$ is an evolution. Single evolutions of a EUSM can be concatenated to conform a *trace*.

Definition 4. Let $M = (id, S, E, U, s_0, \mathcal{V}_0, \mathcal{T})$ be a EUSM and $c = (s, \mathcal{V})$ be a configuration of M. Let us consider that $t =_\diamond time \in \mathcal{V}$. An *evolution* of M from c is a pair $(c, c')_K$, where $c' = (s', \mathcal{V}')$ and $K \in \{exp, tm\}$ are defined in such a way that one of the following conditions holds:

(1) (*Explicit evolution*) If there exists $(s, C, Z, s'') \in \mathcal{T}$ and e such that $C(\mathcal{V}, e) =$ `True` then $K = exp$, $s' = s''$, and $\mathcal{V}' = Z(\mathcal{V}, e)$.
(2) (*Passing of time*) If the condition of (1) does not hold then $K = tm$, $s' = s$, and $\mathcal{V}' = \mathcal{V}[t := \beta]$, where

$$\beta \le min\,\{\beta' \,|\beta' > time \,\wedge\, \exists\,(s, C, Z, s'') \in \mathcal{T}, e:\; C(\mathcal{V}[t := \beta'], e) = \texttt{True}\}$$

We denote by `Evolutions`(M, c) the set of evolutions of M from c.

Let $M = (id, S, E, U, s_0, \mathcal{V}_0, \mathcal{T})$ be a EUSM and $c_1, \ldots, c_n$ be configurations such that $c_1 = (s_0, \mathcal{V}_0)$ and for all $1 \le i \le n-1$ we have $(c_i, c_{i+1})_{K_i} \in$ `Evolutions`(M, c_i). Then, we say that $c_1 \Longrightarrow c_n$ is a *trace* of M. The set of traces of M is denoted by `Traces`(M, c_1). □

Example 4. Next we show an evolution in the context of our running example. We will suppose that the variable t denotes the time in days, and *money* denotes the amount of dollars owned by the agent. Let $c = (s_2, \mathcal{V})$ be a configuration where $pcModel_8 =_\diamond 10$, $money =_\diamond 200$, $t =_\diamond 5$, and $QtoClients =_\diamond \emptyset$ are in $\mathcal{V}$, that is, the agent owns 10 PCs of model 8 and \$200, it is day 5, and the set of quotes sent to clients is empty. Besides, let us consider $RFQfromClients =_\diamond \{(client_{15}, crfq(pcModel_8, 1)\} \in \mathcal{V}$, that is, client 15 requested a quote to buy one PC of model 8.

Besides, let $U(s_2)(client_{15}, \mathcal{V} + \{t =_\diamond 2, pcModel_8 =_\diamond -1, money =_\diamond 999\}) > U(s_2)(client_{15}, \mathcal{V} + \{t =_\diamond 2\})$, that is, the agent would accept to sell a PC of model 8 to client 15 by \$999 in day $5 + 2 = 7$. In this case, $tran_2^1$ can be taken: By setting the parameters *price* and *deadline* to 999 and 7, respectively, C_2^1 holds.

Let $\mathcal{V}' = Z_2^1(\mathcal{V}, (999, 7))$, i.e., the only variations between $\mathcal{V}$ and $\mathcal{V}'$ are that $RFQfromClients =_\diamond \emptyset$, $QtoClients =_\diamond \{(client_{15}, aq(pcModel_8, 1, 7, 999, 0))\}$, and $op =_\diamond (client_{15}, aq(pcModel_8, 1, 7, 999, 0))$. Let $c' = (s', \mathcal{V}')$. Then, we have $(c, c')_{exp} \in$ `Evolutions`$(\mathcal{A}, c)$ and $c \Longrightarrow c' \in$ `Traces`$(\mathcal{A}, c)$, being $\mathcal{A}$ the EUSM of our running example. □

5 Defining Multi-agent Systems with EUSMs

In our formalism, a *system* is just a tuple of agents. The evolutions of a system are defined from the ones corresponding to the individual agents by taking into account the following: If an agent can perform an explicit transition (that is, the condition of a transition from the current state holds) then the transition is performed. If the transition creates a message in the port of outgoing messages then the message is enqueued in the buffer of incoming messages of the corresponding agent. Afterwards, the message must be removed from the port of outgoing messages (this is simulated by setting the message to the $\perp$ value). Finally, if no condition holds then all the agents idle an amount of time less than or equal to the time needed for a condition to hold.

Definition 5. Let $M_1, \ldots, M_n$ be EUSMs such that for all $1 \leq i \leq n$ we have $M_i = (id_i, S_i, E_i, U_i, s_{0i}, \mathcal{V}_{0i}, \mathcal{T}_i)$ and $E_i = (V_i, R_i)$. If $R_1 = \ldots = R_n$ then we say that the tuple $S = (M_1, \ldots, M_n)$ is a *system* of EUSMs. A *configuration* of S is a tuple $(c_1, \ldots, c_n)$, where for all $1 \leq i \leq n$ we have that c_i is a configuration of M_i.

Let $S = (M_1, \ldots, M_n)$ be a system. Let $c = (c_1, \ldots, c_n)$ be a configuration of S, where for all $1 \leq i \leq n$ we have $c_i = (s_i, \mathcal{V}_i)$. An *evolution* of S from c is a pair $(c, c')_K$, where $c' = (c'_1, \ldots, c'_n)$ and $K \in \{exp, tm\}$ are defined in such a way that one of the following conditions hold:

(1) (*Explicit evolution*) If there exist $1 \leq i \leq n$ and $d = (s', \mathcal{V}')$ such that $(c_i, d)_{exp} \in$ Evolutions(M_i, c_i), then $K = exp$ and we consider the following possibilities:
  (1.a) (*No communication*) If $op =_\diamond \perp \in \mathcal{V}'$ then we have $c'_i = d$ and for all $1 \leq j \leq n$ with $j \neq i$ we have $c'_j = c_j$.
  (1.b) (*The agent communicates*) If $op =_\diamond (id_j, m) \in \mathcal{V}'$, for some $1 \leq j \leq n$, then $c'_i = (s', \mathcal{V}'[op := \perp])$, $c'_j = (s_j, \mathcal{V}_j[ib := enqueue(ib, (id_i, m))])$ and for all $1 \leq k \leq n$, with $k \neq i, j$, we have $c'_k = c_k$.

(2) (*Passing of time*) If there exists $newtime \in \mathbb{R}^+$ such that for all $1 \leq i \leq n$ we have $(c_i, c''_i)_{tm} \in$ Evolutions(M_i, c_i), with $c''_i = (s_i, \mathcal{V}_i[t := newtime])$, then for all $1 \leq i \leq n$ we have $c'_i = c''_i$ and $K = tm$.

We denote by Evolutions(S, c) the set of evolutions of S from c.

Let $S = (M_1, \ldots, M_n)$ be a system with $M_i = (id_i, S_i, E_i, U_i, s_{i0}, \mathcal{V}_{i0}, \mathcal{T}_i)$ for all $1 \leq i \leq n$. Besides, let $c_1, \ldots, c_n$ be configurations such that we have $c_1 = ((s_{10}, \mathcal{V}_{10}), \ldots, (s_{n0}, \mathcal{V}_{n0})$ and for all $1 \leq i \leq n-1$ we have $(c_i, c_{i+1})_{K_i} \in$ Evolutions(S, c_i). We say that $c_1 \Longrightarrow c_n$ is a *trace* of S. The set of traces of S is denoted by Traces(S, c_1). □

6 Defining EUSMs: General Guidelines

In this section we provide some general guidelines to properly define agents by means of EUSMs. First, let us note that our language focuses on splitting the behavior of each agent in two separate parts:

(a) *Tasks*: They are the actions the agent must perform. In order to properly define their behavior, *states*, *transitions*, and *variables* of the EUSM can be used.
(b) *Strategic decisions*: They refer to all the situations where the agent must choose among some choices in such a way that its decision may affect the future chances of success. This part of the agent should be defined by means of *utility functions* associated to states of the EUSM.

In general, the *strategy* of an agent is defined by the utility functions governing each state. Thus, the selection of suitable utility functions dramatically depends on the kind of strategic behavior we want to provide the agent with. In fact, if we leave undefined the utility functions associated to states, an EUSM provides a reusable framework to design agents with different strategy choices. For example, if we consider the specification described in the previous section and we remove utility functions, we obtain a generic reusable framework for defining vendor agents in the Supply Chain Management Game. In particular, the same specification allows to consider different strategies by just associating different suites of utility functions to states.

Tasks involving decision-making, optimization, etc. should be abstracted by means of the utility functions. Let us note that utility functions provide an implicit definition of what is preferable: In any situation where some choices are available, the best choice is, by definition, the one returning the highest utility. Sometimes, finding the values that maximize a utility function may be a hard task, specially if the form of utility functions is not constrained to a given form (in general, if the searching space is finite, it is an NP-hard problem). Hence, given a specification defined by means of an EUSM, an implementation should not be required to find the *optimal* choices in general. Depending on the time/optimality necessities, the method followed by an implementation to find *good enough* suboptimal choices should be based on a suitable tradeoff. Let us note that these issues do not concern the EUSM because they are *implementation* details.

Due to space limitations we cannot include the remaining parts of the case study that we are using along the paper. The objective of each state consists in dealing with a certain kind of messages by considering the agent preferences established by means of utility functions.

7 Conclusions and Future Work

In this paper we have presented a formalism called extended utility state machines, in short EUSMs. We have illustrated the behavior of these machines with a medium-size example: The specification of vendors in the Supply Chain Management Game. EUSMs are a big improvement with respect to the original USM formalism. In particular, strategic behavior can be now defined in an easier and more direct way. In this paper we tried to keep a balance between theory (by introducing a formal language) and practice (by applying the language to the specification of a benchmark in e-commerce agents). Our work needs a natural continuation in both directions. Regarding the more theoretical component, our previous results for USMs should be adapted to the new framework. Regarding the more practical component, we are assessing the formalism with other case studies that are not related to e-commerce. Specifically, we are continuing the work initiated in [25] by specifying in our formalism not only the *Interactive Driving System*, but the next layer of the system containing it.

References

1. Guttman, R., Moukas, A., Maes, P.: Agent-mediated electronic commerce: A survey. The Knowledge Engineering Review 13(2), 147–159 (1998)
2. Sandholm, T.: Agents in electronic commerce: Component technologies for automated negotiation and coalition formation. In: Klusch, M., Weiss, G. (eds.) CIA 1998. LNCS (LNAI), vol. 1435, pp. 113–134. Springer, Heidelberg (1998)
3. Ma, M.: Agents in e-commerce. Communications of the ACM 42(3), 79–80 (1999)
4. He, M., Jennings, N., Leung, H.: On agent-mediated electronic commerce. IEEE Trans. on Knowledge and Data Engineering 15(4), 985–1003 (2003)
5. Sierra, C.: Agent-mediated electronic commerce. Autonomous Agents and Multi-Agent Systems 9(3), 285–301 (2004)
6. Gruer, P., Hilaire, V., Koukam, A., Cetnarowicz, K.: A formal framework for multi-agent systems analysis and design. Expert Systems and Applications 23, 349–355 (2002)
7. Hilaire, V., Simonin, O., Koukam, A., Ferber, J.: A formal approach to design and reuse agent and multiagent models. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 142–157. Springer, Heidelberg (2005)
8. Cabac, L., Moldt, D.: Formal semantics for AUML agent interaction protocol diagrams. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 47–61. Springer, Heidelberg (2005)
9. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
10. Benerecetti, M., Cimatti, A.: Validation of multiagent systems by symbolic model checking. In: Giunchiglia, F., Odell, J.J., Weiss, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 32–46. Springer, Heidelberg (2003)
11. Myers, G.: The Art of Software Testing, 2nd edn. John Wiley and Sons, West Sussex, England (2004)
12. Núñez, M., Rodríguez, I., Rubio, F.: Testing of autonomous agents described as utility state machines. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, pp. 322–336. Springer, Heidelberg (2004)
13. Rodríguez, I.: Formal specification of autonomous commerce agents. In: SAC 2004, pp. 774–778. ACM Press, New York (2004)
14. Rodríguez, I., Núñez, M., Rubio, F.: Specification of autonomous agents in e-commerce systems. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, pp. 30–43. Springer, Heidelberg (2004)
15. Núñez, M., Rodríguez, I., Rubio, F.: Specification and testing of autonomous agents in e-commerce systems. Software Testing, Verification and Reliability 15(4), 211–233 (2005)

16. Chavez, A., Maes, P.: Kasbah: An agent marketplace for buying and selling goods. In: PAAM 1996. 1st Int. Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology, pp. 75–90 (1996)
17. Collins, J., Arunachalam, R., Sadeh, N., Eriksson, J., Finne, N., Janson, S.: The supply chain management game for 2005 trading agent competition. Technical Report CMU-ISRI-04-139, Carnegie Mellon University (2004)
18. Padget, J., Bradford, R.: A pi-calculus model of a spanish fish market - preliminary report. In: Noriega, P., Sierra, C. (eds.) AMET 1998 and AMEC 1998. LNCS (LNAI), vol. 1571, pp. 166–188. Springer, Heidelberg (1999)
19. Adi, K., Debbabi, M., Mejri, M.: A new logic for electronic commerce protocols. Theoretical Computer Science 291(3), 223–283 (2003)
20. Bhimani, A.: Securing the commercial Internet. Communications of the ACM 39(6), 29–35 (1996)
21. Rao, A.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS (LNAI), vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
22. Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.J.: Formal semantics for an abstract agent programming language. In: Rao, A., Singh, M.P., Wooldridge, M.J. (eds.) ATAL 1997. LNCS (LNAI), vol. 1365, pp. 215–229. Springer, Heidelberg (1998)
23. Feinberg, E., Shwartz, A.: Handbook of Markov Decision Processes, Methods and Applications. Kluwer Academic Publishers, Boston, MA (2002)
24. Cassandra, A., Kaelbling, L., Littman, M.: Acting optimally in partially observable stochastic domains. In: 12th National Conf. on Artificial Intelligence (1994)
25. Núñez, M., Pelayo, F., Rodríguez, I.: A formal methodology to test complex embedded systems: Application to interactive driving system. In: IESS 2005. IFIP TC10 Working Conf.: International Embedded Systems Symposium, pp. 125–136. Springer, Heidelberg (2005)