A novel formalism to represent collective intelligence in multi-agent systems*

Juan José Pardo[1], Manuel Núñez[2], and M. Carmen Ruiz[1]

[1] Departmento de Sistemas Informáticos
Universidad de Castilla-La Mancha, Spain
jpardo@dsi.uclm.es, MCarmen.Ruiz@uclm.es
[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain,
mn@sip.ucm.es

Abstract. In this paper we introduce a new formalism to represent multi-agent systems where resources can be exchanged among different agents by maximizing the utility of the agents conforming the systems. In addition to introduce a formalism to specify agents, we provide a formal framework to test whether an implementation *conforms* to the specification of an agent of the system.

1 Introduction

Computational collective intelligence has as its main objective to extract information from the knowledge of the components of a group that could not be extracted by taking this knowledge in an isolated way. A very simple example, extracted from [15], is that if one agent knows that $a \leq b$ and another one knows that $b \leq a$, then we can extract from the system a new piece of information $a = b$. We can use a similar approach in multi-agent systems where agents can compete to obtain resources: If we do this in a collaborative, collective way, all the agents can profit. For example, let us consider that a seller is willing to sell a certain item by at least n money units while a buyer is willing to pay at most m money units for this item. By putting together this information, we know that a deal can be reached as long as the selling price is between n and m money units. Multi-agent systems have been clearly identified as one of the fields of application of computational collective intelligence. Actually, they have been used in different application domains, in particular, outside purely computer science problems. One of the areas where research and development activities have been particularly increasing, maybe due to its financial applications, is in e-commerce systems where agents are in charge of some of the computations that users would have to perform otherwise (see, for example, [5, 10, 21, 12, 22, 3]).

Formal methods are a powerful tool that allows the analysis, validation and verification of systems in general and of e-commerce systems in particular. In fact, due to the complexity of current systems, it is very important to use a formal approach already in the early development stages since the sooner the errors of the system under

* This research was partially supported by the WEST project (TIC2006-15578-C02) and the Junta de Castilla-la Mancha project “Aplicación de metodos formales al diseño y análisis de procesos de negocio” (PEII09-0232-7745).

development are detected the less harm, in particular in monetary terms, is done. In the context of multi-agent systems, formal methods can be used to express 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* includes not only what goods or services he wants but also other conditions, such as when and how to pay for the obtained goods and services, and these other conditions must be included in the specification of the system. There have been already several proposals to formalize multi-agent systems and to use existing formal methods within their scope (see, for example, [19, 18, 8, 1, 16, 11]).

We have emphasized the importance of formal methods to specify the behavior of the system. However, it is even more important to ensure that the current implementation of the system is correct. In this line, testing [14, 2], is one of the most extended techniques to critically evaluate the quality of systems. Traditionally, since the famous Dijkstra aphorism "*Program testing can be used to show the presence of bugs, but never to show their absence*," testing and formal methods have been seen as rivals. However, during the last years there is a trend to consider them as complimentary techniques that can profit from each other. In fact, work on formal testing is currently very active (see, for example, [20, 7, 9, 23, 6]). The idea is that we have a formal model of the system (a specification) and we check the correctness of the system under test by applying experiments: We match the results of these experiments with what the specification says and decide whether we have found an error. Fortunately, and this is where formal methods play an important role, having a formal description of the system allows to automatize most of the testing phases (see [24] for an overview of different tools for formal testing).

The initial point of the work reported in this paper can be found in one formalism previously developed within our research group [17, 13]. The idea underlying the original framework, called *utility state machines*, is to specify the high-level behavior of autonomous e-commerce agents participating in a multi-agent system and formally test the implemented agents with respect to the existing specifications. In this formalism, the user's preferences are defined by means of *utility functions* associating a numerical value to each possible set of resources that the system can trade. After using utility state machines to describe several existing systems we found that their internal inherent structure sometimes complicates the task of formally defining some types of specification. The problem that we found is that utility state machines are based on *finite state machines*, where a strict alternation between inputs and outputs must be kept. However, there are frequent situations where several inputs can be sequentially applied without receiving an output, or where an output can be spontaneously produced without needing a preceding input. Therefore, we have decided to consider a more expressive formalism where the alternation between inputs and outputs is not enforced. This slightly complicates the semantic framework. In particular, we need to include the notion of *quiescence* to characterize states of the systems that cannot produce outputs. We have also to redefine the notion of test and how to apply tests to systems. On the contrary, we have reduced some of the complexity associated with our previous formalism. Most notably, we formerly considered that agents could have *debts* that can be compensated with future exchanges. For example, an agent could offer a value greater than the one given by its utility function for an item if he could include this item as part of a future deal that would compensate the transitory lost. Even though this is a very interesting characteristic, it complicated too much the underlying semantic model and our experience shows that this feature was not used very often. In this paper we introduce a new formalism, called *Utility Input-Output Labeled Transition System* (in short, UIOLTS) that includes these enhancements. We have already used our new framework to formally specify an agent participating in the 2008 edition of the Supply Chain Management Game [4] but, due to space limitations, we could not include it in the paper.

In order to formally establish the conformance of a system under test with respect to a specification, we define two different implementation relations. The first one takes into account only the sequence of inputs and outputs produced by the system. While this notion would be enough to establish what a correct system is in terms of what the system does, it can overlook some faults due to the way resources are exchanged. Thus, we introduce a second implementation relation that also considers the set of resources that the system has after an action is executed.

The rest of the paper is structured as follows. In Section 2 we introduce our model. In Section 3 we define our implementation relations. In Section 4 we give the notion of test and how to apply tests to implementations under test. Finally, in Section 5 se present our conclusions and some lines for future work.

2 Modelling e-commerce agents by using UIOLTSs

In this section we present our language to formally define agents and introduce some notions that can be used for their ulterior analysis. Intuitively, a UIOLTS is a labeled transition system where we introduce some new features to define agent behaviors in an appropriate way. The first new element that we introduce is a set of variables, where each variable represents the amount of the resource that the system owns. In addition, we associate a utility function to each state of the system. This utility function can be used to decide whether the agent accepts an exchange of resources proposed by another agent. Intuitively, given a utility function u we have that $u(\overline{x}) < u(\overline{y})$ means that the basket of resources represented by $\overline{y}$ is preferred to $\overline{x}$.

Definition 1. *We consider* $\mathbb{R}_+ = \{x \in \mathbb{R} \,|\, x \geq 0\}$. *We will usually denote vectors in* $\mathbb{R}^n$ *(for* $n \geq 2$*) by* $\overline{x}, \overline{y}, \overline{v} \ldots$ *Given* $\overline{x} \in \mathbb{R}^n$, x_i *denotes its i-th component. We extend to vectors some usual arithmetic operations. Let* $\overline{x}, \overline{y} \in \mathbb{R}^n$. *We define the addition of vectors* $\overline{x}$ *and* $\overline{y}$, *denoted by* $\overline{x} + \overline{y}$, *simply as* $(x_1 + y_1, \ldots, x_n + y_n)$. *We write* $\overline{x} \leq \overline{y}$ *if for all* $1 \leq i \leq n$ *we have* $x_i \leq y_i$.

We will suppose that there exist $n > 0$ *different kinds of resources. Baskets of resources are defined as vectors* $\overline{x} \in \mathbb{R}^n_+$. *Therefore,* $x_i = r$ *denotes that we own r units of the i-th resource. A utility function is a function* $u : \mathbb{R}^n_+ \longrightarrow \mathbb{R}$. *In microeconomic theory there are some restrictions that are usually imposed on utility functions (mainly, strict monotonicity, convexity, and continuity).*

As we indicated in the introduction, we consider two different types of actions that a system can perform. On the one hand, output actions are initiated by the system and cannot be refused by the environment. We consider that the performance of an output action can cost resources to the system. In addition, the performance of an output action will usually have an associated condition to decide whether the system performs it or not. On the other hand, input actions are initiated by the environment and cannot be refused by the system (that is, we consider that our systems are *input-enabled*). In contrast with output actions, the performance of an input action can increase the resources of the agent that performs it (that is, it receives a transfer of resources from the environment to *pay* the agent for the performance of the action). Let us remark that what we call in this informal description the *environment* can refer to a centralizer overviewing the activities of the different agents or to another agent that sends/receives actions to/from our agent. In addition to these two types of actions we need a third type that we introduce for technical reasons. It is possible to have a state where the system patiently waits for an input action and it cannot execute any output action for a certain combination of the existing resources. These states are called *quiescent*. In order to represent quiescence, we include a special action, denoted by δ, and special transitions labeled by this same δ action.

Definition 2. *A Utility Input Output Labeled Transition System, in short UIOLTS, is a tuple $M = (S, s_0, L, T, U, V)$ where*

- *S is the set of states, being $s_o \in S$ the initial state.*
- *V is an n-tuple of resources belonging to R_+. We denote by $\bar{v}_0$ the initial tuple of values associated with these resources.*
- *L is the set of actions. The set of actions is partitioned into three pairwise disjoint sets $L = L_I \cup L_O \cup \{\delta\}$ where*
  - *L_I is the set of inputs. Elements of L_I are preceded by the symbol $?$.*
  - *L_O is the set of outputs. Elements of L_O are preceded by the symbol $!$.*
  - *δ is a special action that represents quiescence.*
- *T is the set of transitions. The set of transitons is partitioned into three pairwise disjoint sets $T = T_I \cup T_O \cup T_\delta$ where*
  - *T_I is the set of input transitions. An input transition is given by a tuple $(s, ?i, \bar{x}, s_1)$ where $s \in S$ is the initial state, $s_1 \in S$ is the final state, $?i \in L_I$ is an input action, and $\bar{x} \in \mathbb{R}^n_+$ is the increase in the set of resources. Since the system has to be input-enabled, we require that for all $s \in S$ and $?i \in L_I$ there exist $\bar{x}$ and s_1 such that $(s, ?i, \bar{x}, s_1) \in T_I$.*
  - *T_O is the set of output transitions. An output transition is given by a tuple $(s, !o, \bar{z}, C, s_1)$ where $s \in S$ is the initial state, $s_1 \in S$ is the final state, $!o \in L_O$, $\bar{z} \in \mathbb{R}^n_+$ is the decrease in the set of resources, and $C \in \mathbb{R}^n_+ \longrightarrow \{\texttt{true}, \texttt{false}\}$ is a predicate on the set of resources.*
  - *In addition, for each situation where a state cannot perform an output action we add a transition representing quiescence. This transition is a loop and does not need/receive resources to be performed. That is, $T_\delta = \{(s, \delta, \bar{0}, C_s, s) | s \in S \wedge C_s = \bigwedge_{(s,!o,\bar{z},C,s_1) \in T_O} \neg C \wedge C_s \not\leadsto \texttt{false}\}$. Let us remark that $\bigwedge \emptyset = \texttt{true}$.*
- *$U : S \to (\mathbb{R}^n_+ \longrightarrow R_+)$ is a function associating a utility function to each state in S.*

In order to record the current situation of an agent we use *configurations*, that is, pairs where we keep the current state of the system and the current amount of available resources.

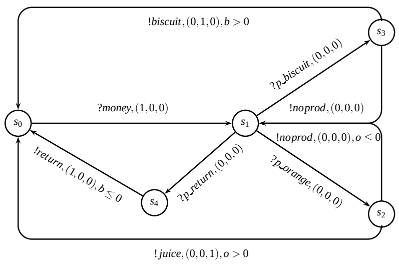

Fig. 1. Example of UIOLTS: A vending machine

Definition 3. *Let M be a UIOLTS. A configuration of M is a pair $(s, \overline{v})$ where s is the current state and $\overline{v}$ is the current value of V. We denote by $Conf(M)$ the set of possible configurations of M.*

In order to explain the main concepts of our framework, we use a simple running example.

Example 1. We consider a simple vending machine that sells biscuits and orange juice. The price of each of the items is one coin. This machine accepts four actions: Insert coin and press three different buttons, one for biscuits, one for orange juice and one for returning the money. In the initial state, after receiving a coin the machine waits for the user to press a button. If the pressed button is the return button the coin is returned and the machine returns to the initial state. If the user presses other button then if the machine has units of this product left, then it provides it; otherwise, it returns to the state to wait the user presses other button. Thus, the set of input actions for this machine is $L_i = \{?money, ?p\_orange, ?p\_biscuit, ?p\_return\}$ while the set of output actions is $L_O = \{!orange, !biscuit, !noprod, !return\}$. The tuple of resources has three variables $V = (m, b, o)$ that contain the amount of money (m), biscuits(b), and bottles of orange juice (o) currently in the machine. The utility function in each state takes a very simple form: $U(s_i)(V) = m + b + o$ that represents that every resource in the machine has the same value, meaning that the machine is *equally happy* with one coin or with one unit of a product.

If we consider that at the beginning of the day the machine has 10 bottles of orange juice, 10 packages of biscuits and it does not have money, the initial configuration of the system will be $c_0 = (s_0, (0, 10, 10))$.

In Figure 1 we graphically describe the system transitions. In order to increase readability, we have omitted trailing transitions labeled by δ and transitions that have to be added to obtain an input-enabled machine (for example, the transition $(s_0, ?p\_biscuit, (0,0,0), s_0)$).

Now we can define the concatenation of several transitions of an agent to capture the different evolutions of a system from a configuration to another one. These evolutions can be produced either by executing an input or an output action or by offering a *exchange of resources*. As we will see, exchanges of resources have low priority and will be allowed only if no output can be performed. The idea is that if we can perform an output with the existing resources, then we do not need to exchange resources.

Definition 4. *Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS. M can evolve from the configuration $c = (s, \overline{v})$ to the configuration $c' = (s', \overline{v'})$ according to one of these options*

1. *If there is an input transition $(s, ?i, \overline{x}, s_1)$ then this transition can be executed. The new configuration is $s' = s_1$ and $v' = v + \overline{x}$.*
2. *If there is an output transition $(s, !o, C, \overline{z}, s_1)$ such that $C(\overline{v})$ holds then the transition can be executed. The new configuration is $s' = s_1$ and $\overline{v'} = \overline{v} - \overline{z}$.*
3. *Let us consider the transition associated with quiescence at s: $(s, \delta, C_s, \overline{0}, s)$. If $C_s(\overline{v})$ holds, that is, no output transition is currently available, then this transition can be executed. The new configuration is $s' = s$ and $\overline{v'} = \overline{v}$.*
4. *Let us consider again the transition associated with quiescence at s, that is, $(s, \delta, C_s, \overline{0}, s)$. If $C_s(\overline{v})$ holds, then we can offer an exchange. We represent an exchange by a pair $(\xi, \overline{y})$ where $\overline{y} = (y_1, y_2, \dots y_n)$ is the variation of the set of resources. Therefore, $y_i < 0$ indicates a decrease of the resource i while $y_i > 0$ represents an increase of the resource i. M can do an exchange $(\xi, \overline{x})$ if $U(s, v) < U(s, v + \overline{x})$. If another agent is accepting the exchange, the new configuration is $s' = s$ and $\overline{v'} = \overline{v} + \overline{y}$.*

We denote an evolution from the configuration c to the configuration c' by the tuple $(c, (a, \overline{y}), c')$ where $a \in L \cup \{\xi\}$ and $\overline{y} \in \mathbb{R}^n$. We denote by $Evolutions(M, c)$ the set of evolutions of M from the configuration c and by $Evolutions(M)$ the set of evolutions of M from (s_0, v_0), the initial configuration.

A trace of M is a finite list of evolutions. $Traces(M, c)$ denotes the set of traces of M from the configuration c and $Traces(M)$ denotes the set of traces of M from the initial configuration c_0. Let $l = e_1, e_2, \dots, e_m$ be a trace of M with $e_i = (c_i, (a_i, \overline{x_i}), c_{i+1})$. The observable trace associated to l is a triple (c_1, σ, c_{n+1}) where σ is the sequence of actions obtained from $a_1, a_2, \dots, a_m$ by removing all occurrence of ξ. We sometimes represent this observable trace as $c_1 \overset{\sigma}{\Longrightarrow} c_{n+1}$.

Example 2. When the machine described in Example 1 is in its initial configuration, a user can insert one coin and the machine accept this action. After this action, the new configuration of the machine is $(s_1, (1, 10, 10))$. Now the user, who wants a pack of biscuits, press the biscuit button and the new configuration is $(s_3, (1, 10, 10))$ because by performing this action the set or resources does not change. In this configuration, the machine performs the output action $!biscuit$ and the user can take his pack. The new configuration of the machine is $(s_0, (1, 9, 10))$.

Let us suppose that after a set of interactions our machine has reached the configuration $(s_0, (15, 5, 0))$. In this situation, the person in charge of the machine would like to refill it and take as many coins as new items he adds. This action can be represented as a exchange of resources. The tuple that represents the variation of the resources is $(-15, 5, 10)$. After this exchange the new configuration is $(s_0, (0, 10, 10))$.

One of the traces corresponding to our vending machine is

$$(c_0, (?money, (1,0,0)), c_1), (c_1, (?p\_biscuit, (0,0,0)), c_2), (c_2, (!biscuit, (0,1,0)), c_3),$$
$$(c_3, (\xi, (-1,1,0)), c_4).$$

while the associated observable trace is $(c_0, (?money, ?p\_biscuit, !biscuit), c_4)$.

3 Implementation relations for UIOLTSs

In this section we introduce our implementation relations that formally establish when an implementation is correct with respect to a specification. In our context, the notion of correctness has several possible definitions. For example a person may consider that an implementation I of a system S is good if the number of resources always increases while another one considers that the number of resources can decrease sometimes.

We define two different implementation relations. The first one is close to the classical **ioco** implementation relation [23] where an implementation I is correct with respect to a specification S if the output actions executed by I after a sequence of actions is performed are a subset of the ones that can be executed by S. Intuitively, this means that the implementation does not *invent* actions that the specification did not contemplate. First, we introduce some auxiliary notation

Definition 5. *Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS, $c = (s, \overline{x}) \in Conf(M)$ a configuration of M, and $\sigma \in L^*$ be a sequence of actions. Then,*

$$c \ \texttt{after} \ \sigma = \{c' \in Conf(M) | c \overset{\sigma}{\Longrightarrow} c'\}$$

$$\begin{aligned}\texttt{out}(c) = \{!o \in L_O | \exists s_1, \overline{z}, C : (s, !o, C, \overline{z}, s_1) \in T \wedge C(\overline{x})\} \\ \cup \{\delta | \exists C_s : (s, \delta, C_s, \overline{0}, s) \in T \wedge C_s(\overline{x})\}\end{aligned}$$

Intuitively, $c \ \texttt{after} \ \sigma$ returns the configuration reached from the configuration c by the execution of the trace σ while $\texttt{out}(c)$ contains the output actions that the system can execute from the configuration c.

Definition 6. *Let I, S be two UIOLTSs with the same set of actions L. We write I* **ioco** *S if for all sequence of actions $\sigma \in L^*$ we have that $\texttt{out}(I \ \texttt{after} \ \sigma) \subseteq \texttt{out}(S \ \texttt{after} \ \sigma)$.*

Example 3. In figure 2 we show three UIOLTSs that model three possible implementations of our vending machine where, as before, we have removed trailing occurrences of input actions and of δ transitions.

Let us consider first the implementation I_1. In this case, the specification specifies that after ?*money* the machine does not execute any output actions.

So, $\texttt{out}(S \ \texttt{after} \ ?money) = \{\delta\}$. The implementation I_1 indicates that the machine can execute the output action !*return* after the input action ?*money*. Therefore, $\texttt{out}(I \ \texttt{after} \ ?money) = \{!return\}$. Since, $\texttt{out}(I \ \texttt{after} \ ?money) \not\subseteq \texttt{out}(S \ \texttt{after} \ ?money)$ we conclude that I_1 **ioco** S does not hold.

If we consider the implementation I_2 we can check that I_2 **ioco** S because the requested sets containment hold. In fact, this machine is *very good*, from the owner's point of view, since it requests two coins for each of the items. Similarly, I_3 **ioco** S, but this is a bad implementation because it provides two packets of biscuits per each coin. This shows the weakness in our framework of an implementation relation concentrating only on the performed actions.

Our second implementation relation is also based on the **ioco** mechanism but we take into account both the resources that the system has and the actions that the system

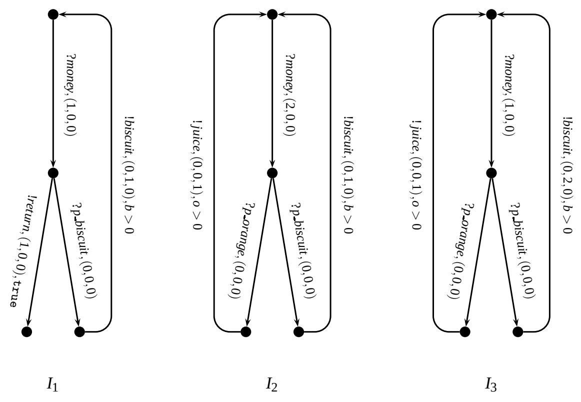

Fig. 2. Possible implementations of our vending machine

can execute. In order to define the new relation we need to redefine again the set `out` of outputs. In this case, our set of outputs has two components: The output action that can be executed and the set of resources that the system has. We also introduce an operator to compare sets of pairs (output, resources).

Definition 7. *Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS and $c = (s, \bar{x}) \in Conf(M)$ be a configuration of M. Then*

$$\begin{aligned}\texttt{out}'(c) = \{(!o, \bar{y}) \in L_O \times \mathbb{R}^n_+ \mid \exists s_1, \bar{z}, C : (s, !o, C, \bar{z}, s_1) \in T \wedge C(\bar{x}) \wedge \bar{y} = \bar{x} - \bar{z}\}\\ \cup \{(\delta, \bar{x}) \mid \exists C_s : (s, \delta, C_s, \bar{0}, s) \in T \wedge C_s(\bar{x})\}\end{aligned}$$

Given two sets $A = \{(o_1, \bar{y_1}), \ldots, (o_n, \bar{y_n})\}$ and $B = \{(o_1, \bar{x_1}), \ldots, (o_n, \bar{x_n})\}$, we write $A \sqsubseteq B$ if $act(A) \subseteq act(B)$ and $\min(rec(A, o)) \geq max(rec(B, o))$, where $Act(A) = \{a \mid (a, \bar{y}) \in A\}$ and $rec(A, o) = \{r \mid (o, r) \in A\}$.

The set $\texttt{out}'(c)$ contains those actions (outputs or quiescence) that can be performed when the system is in configuration c as well as the set of resources obtained after their performance. Next, we introduce our new implementation relation. We consider that an implementation I is correct with respect to a specification S if the output actions performed by the implementation in a state are a subset of those that can be performed by the specification in this state and the set of resources of implementation I is *better* than the set of resources in the specification.

Definition 8. *Let I, S be two UIOLTSs with the same set of actions L. We write I $\mathbf{ioco_r}$ S if for all sequence of actions $\sigma \in L^*$ we have that $\texttt{out}'(I \texttt{ after } \sigma) \sqsubseteq \texttt{out}'(S \texttt{ after } \sigma)$.*

Example 4. Let us consider the specification S given in Figure 1 and the implementation I_3 given in Figure 2. We have, according to the specification, that

$$\texttt{out}'(S \texttt{ after } ?money?p\_biscuit) = \{(!biscuit, (1, 9, 10))\}$$

while, according to I_3, we have that

$$\mathtt{out}'(I \ \mathtt{after} \ ?money?p\_biscuit) = \{(!biscuit,(1,8,10))\}$$

We can observe that the set of output actions is the same in both cases but the resources of the implementation are smaller than the ones of the specification. Then, we conclude that I_3 $\mathbf{ioco_r}$ S does not hold.

4 Tests: Definition and application

Essentially, a test is the description of the behavior of a tester in an experiment carried out on an implementation under test. In this experiment, the tester serves as a kind of artificial environment of the implementation. This tester can do four different things: It can accept an output action started by the implementation, it can provide an input action to the implementation, it can propose a exchange of resources, or it can observe the absence of output actions, so that it can detect quiescence. When the tester receives an output action it checks whether the action belongs to the set of expected ones (according to its description); if the action does not belong to this set then the tester will produce a fail signal. In addition, each state of a test saves information about the set of resources that the tested system has if the test reaches this state.

In our framework, a test for a system is modeled by a UIOLTS, where its set of input actions is the set of output actions of the specification and its set of output actions is the set of input actions of the specification. Also, we include a new action θ that represents the observation of quiescence. In order to be able to accept any output from the tested agent, we consider that tests are *input-enabled*, since its inputs correspond to outputs of the tested agent.

Definition 9. *Let $M = (S, s_0, L, T, V)$, with $L = L_I \cup L_O \cup \{\delta\}$. A test for M is a UIOLTS $t = (S^t, s_0^t, L^t, T^t, V)$ where*

- *S^t is the set of states where $s_0^t \in S^t$ is the initial state and there are two special states called* `fail` *and* `pass` *with* $\mathtt{fail} \neq \mathtt{pass}$. *We represent by $res(s)$ the information about resources saved in state s.*
- *L^t is the set of actions. $L = L_O \cup L_I \cup \{\theta, \xi\}$ where*
  - *L_O is the set of inputs (these are outputs of M).*
  - *L_I is the set of outputs (these are inputs of M).*
  - *θ is a special action that represents the detection of quiescence.*
  - *ξ is an special action that represents the proposal of an exchange.*
- *T^t is the set of transitions. Each transition is a tuple $(s, a, \bar{x}, s_1)$ where s is the initial state, s_1 is the final state, $a \in L_O \cup L_I \cup \{\theta, \xi\}$ is the label of the transition and $\bar{x} \in \mathbb{R}^n$ is the variation in the set of resources.*

We define configurations of a test in the same way that we defined them for UIOLTSs, and we thus omit the definition. In Figure 3 we present a test for the system described in Figure 1. Given an implementation I and a test t, running t with I is the parallel execution of both taking into account the peculiarities of the special actions δ, θ and ξ. Let $c = (s, \bar{v})$ and $c' = (s', \bar{v'})$ be configurations of I, and $c_t = (q_t, \bar{v_t})$ and $c_t' = (q_t', \bar{v'}_t)$

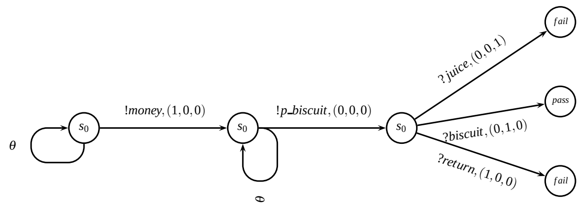

Fig. 3. Test for the vending machine.

be configurations of t. The rules for the composition operator, denoted by $|\lceil$, are the following

$$A)\ \frac{c \xrightarrow{(?a,\bar{x})} c', c_t \xrightarrow{(!a,\bar{x})} c'_t, a \in L_i}{c|\lceil c_t \xrightarrow{(a,\bar{x})} c'|\lceil c'_t} \qquad B)\ \frac{c \xrightarrow{(!a,\bar{x})} c', c_t \xrightarrow{(?a,\bar{x})} c'_t, a \in L_O}{c|\lceil c_t \xrightarrow{(a,\bar{x})} c'|\lceil c'_t}$$

$$C)\ \frac{c \xrightarrow{\delta}\ ,\ c_t \xrightarrow{\theta} c'_t}{c|\lceil c_t \xrightarrow{\theta} c|\lceil c'_t} \qquad D)\ \frac{c \xrightarrow{(\xi,\bar{x})} c', c_t \xrightarrow{(\xi,-\bar{x})} c'_t}{c|\lceil t \xrightarrow{(\xi,\bar{x})} c'|\lceil c'_t}$$

Definition 10. *A test execution of the test t with an implementation I is a trace of $i|\lceil t$ leading to one of the states* pass *or* f*ail of t.*

We say that an implementation I passes a test t if all test execution of t with I go to a pass *state of t.*

Another definition for passing a test, considering the resources administered by the system, is the following.

Definition 11. *An implementation I passes$_r$ a test t if all test execution of t with I go to a* pass *state s of t and the set $\bar{x}$ of resources in the configuration of I is greater than or equal to the set or resources $rec(s)$ that the test says the implementation may have if the test reaches the state s.*

The final step is to prove that the application of a certain set of tests has the same discriminatory power as the implementation relations previously defined. Due to space limitations we cannot include this proof. The idea is to derive tests from specifications using an adaption to our new framework of the algorithm given in [17].

5 Conclusions and Future work

In this paper we have introduced a new formalism, called Utility Input Output Labeled Transition Systems, to specify the behavior of e-commerce agents taking part in a multi-agent system. We have also defined a testing methodology, based in this formalism, to test whether an implementation of a specified agent behaves as the specification says that it behaves. In this theory we have defined two different implementation relations and a notion of test.

This paper is the first step in a long road that we expect to continue in the future. We currently focus on two research lines. The first one is based on theoretical aspects and we would like to extend our formalism in order to specify the behavior of agents that are influenced by the passing of time. The second line is more practical since we would like to apply our formalism to real complex agents. In this line, we have already developed a prototype tool that allows us to do automatic generation of tests, but we have to *stress test* the tool to check how it scales.

References

1. K. Adi, M. Debbabi, and M. Mejri. A new logic for electronic commerce protocols. *Theoretical Computer Science*, 291(3):223–283, 2003.
2. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
3. B.M. Balachandran and M. Enkhsaikhan. Developing multi-agent e-commerce applications with JADE. In *11th Int. Conf. on Knowledge-Based & Intelligent Information & Engineering Systems, KES'07*, pages 941–949. Springer, 2008.
4. J. Collins, A. Rogers, R. Arunachalam, N. Sadeh, J. Eriksson, N. Finne, and S. Janson. The supply chain management game for 2008 trading agent competition. Technical Report CMU-ISRI-08-117, Carnegie Mellon University, 2008.
5. R. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce: A survey. *The Knowledge Engineering Review*, 13(2):147–159, 1998.
6. R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal methods to support testing. *ACM Computing Surveys*, 41(2), 2009.
7. R.M. Hierons, J.P. Bowen, and M. Harman, editors. *Formal Methods and Testing, LNCS 4949*. Springer, 2008.
8. K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J.-J.C. Meyer. Formal semantics for an abstract agent programming language. In *4th Int. Workshop on Agent Theories, Architectures, and Languages, ATAL'97, LNAI 1365*, pages 215–229. Springer, 1998.
9. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
10. N.R. Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2):357–401, 1999.
11. I.A. Lomazova. Nested Petri Nets for adaptive process modeling. In *Pillars of Computer Science, Essays Dedicated to Boris Trakhtenbrot on the Occasion of His 85th Birthday, LNCS 4800*, pages 460–474. Springer, 2008.
12. M. Ma. Agents in e-commerce. *Communications of the ACM*, 42(3):79–80, 1999.
13. M.G. Merayo, M. Núñez, and I. Rodríguez. Formal specification of multi-agent systems by using EUSMs. In *2nd IPM Int. Symposium on Fundamentals of Software Engineering, FSEN'07, LNCS 4767*, pages 318–333. Springer, 2007.
14. G.J. Myers. *The Art of Software Testing*. John Wiley and Sons, 2nd edition, 2004.
15. N.T. Nguyen. Computational collective intelligence and knowledge inconsistency in multi-agent environments. In *11th Pacific Rim Int. Conf. on Multi-Agents, PRIMA'08, LNCS 5357*, pages 2–3. Springer, 2008.
16. M. Núñez, I. Rodríguez, and F. Rubio. Formal specification of multi-agent e-barter systems. *Science of Computer Programming*, 57(2):187–216, 2005.
17. M. Núñez, I. Rodríguez, and F. Rubio. Specification and testing of autonomous agents in e-commerce systems. *Software Testing, Verification and Reliability*, 15(4):211–233, 2005.

18. J.A. Padget and R.J. Bradford. A pi-calculus model of a spanish fish market - preliminary report. In *1st Int. Workshop on Agent Mediated Electronic Trading, AMET'98, LNCS 1571*, pages 166–188. Springer, 1998.
19. A.S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away, LNAI 1038*, pages 42–55. Springer, 1996.
20. I. Rodríguez, M.G. Merayo, and M. Núñez. $\mathscr{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.
21. T. Sandholm. Agents in electronic commerce: Component technologies for automated negotiation and coalition formation. In *2nd Int. Workshop on Cooperative Information Agents, CIA'98, LNCS 1435*, pages 113–134. Springer, 1998.
22. C. Sierra. Agent-mediated electronic commerce. *Autonomous Agents and Multi-Agent Systems*, 9(3):285–301, 2004.
23. J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.
24. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.