Specification and Testing of E-Commerce Agents Described by Using UIOLTSs

Juan José Pardo, Manuel Núñez, M. Carmen Ruiz

Specification and testing of e-commerce agents described by using UIOLTSs*

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

[1] Departmento de Sistemas Informáticos
Universidad de Castilla-La Mancha, Spain
juanjose.pardo@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 expand our work in our specification formalism UIOLTSs. We present three implementation relations and provide alternative characterizations of these relations in terms of the tests that the implementation under test successfully passes. In addition, we present the main ideas to obtain an algorithm to derive complete test suites from specifications.

1 Introduction

During the software development process, it is very usual to apply structured methodologies, consisting of several phases such as analysis, specification, design, coding, and testing. Formal methods are a powerful tool that should be used along all the software development phases because they facilitate the descrition, analysis, validation and verification of software system. So developer can discover posible errors at the beginning of the development process.

Although it is very important to use formal methods to specify the behavior of the system, it is even more important to ensure that the implementation of the system is correct. In this line, *testing* is one of the most extended techniques to critically evaluate the quality of systems. Although testing and formal methods are considered rival, they are complimentary techniques that can profit from each other. The idea is that we have a formal model of the system (a specification), we check the correctness of the system under test by applying experiments and we match the results of these experiments with what the specification says and decide whether we have found an error. The formal description of the system allows to automatize most of the testing phases.

The main theory underlying formal specification and testing can be also applied to a specific kind of software like e-commerce agents. In this context, it is necessary to introduce new features in the formal language in order to express the high-level requirements of agents, which are usually defined in economic terms. In the literature, we can find several proposals to use formal methods to formalize multi-agent systems (see [1]).

* This research was partially supported by the TESIS project (TIN2009-14312-C02), by the Junta de Castilla-la Mancha project "Aplicación de métodos formales al diseño y análisis de procesos de negocio" (PEII09-0232-7745), and by the UCM-BSCH programme to fund research groups (GR58/08 - group number 910606).

The initial point of this paper can be found in one formalism previously developed within our research group [2]. In that paper we presented a formalism called *utility state machines*, which was based on finite state machines with a strict alternation between inputs and outputs and, where 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. The alternation between inputs and outputs is a ver strong restriction that we wanted avoid in our model but, this slightly complicates the semantic framework. In particular, we need to include the notion of *quiescence* to characterize states that cannot produce outputs and we have 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.

Our new formalism, called *Utility Input-Output Labeled Transition System* (in short, UIOLTS), was presented in [3]. In this paper we complete our previous work by defining three implementation relations that can be used to formally establish the conformance of a system under test with respect to a specification. One of them takes into account only the sequences of inputs and outputs produced by the system and the other two relations consider resources that the system has after an action is executed. We redefine the notion of test so that we can obtain more information from the system under test. In order to relate the application of tests and our implementation relations, we define an algorithm to derive complete test suites from specifications, that is, an implementation conforms to a specification if and only if successfully passes the test suite produced from the specification.

The rest of the paper is structured as follows. In Section 2 we introduce our formalism. 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 A framework to formally specify economic agents

In this section we present our formalism as defined in [3]. Basically, 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 add 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 $\mathbb{R}_+ = \{x \in \mathbb{R} \mid x \geq 0\}$. 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). $\square$

Our systems can perform two different types of actions. *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. *Input actions* are initiated by the environment and cannot be refused by the system, that is, we consider that our systems under test are *input-enabled* (specifications do not need to be input-enabled). The performance of an input action can increase the resources of the agent that performs it. In addition to these two types of actions we need a third type that we introduce for technical reasons to represent quiescence [4]. This special action is denoted by δ, and special transitions labeled by this same δ action. In the following definition we also introduce the notion of *configuration*. Usually, in order to clearly identify *where* a system is, it is enough to record the current state. In our setting, in order to record the current situation of an agent we use pairs where we keep the current state of the system and the current amount of available resources.

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 $\overline{v}_0$ 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: the set of imputs actions L_I which elements are preceded by ?, the set of output actions L_O which elements are preceded by ! and a set with one special ation δ that represents quiescence.
- T is the set of transitions that is partitioned into three pairwise disjoint sets: the set of input transitions T_I which elements are tuples $(s, ?i, \bar{x}, s_1)$ where $\bar{x} \in \mathbb{R}^n_+$ is the increase in the set of resources, the set of output transitions T_O which elements are tuples $(s, !o, \bar{z}, C, s_1)$ where $\bar{z} \in \mathbb{R}^n_+$ is the decrease in the set of resources, and C is a predicate on the set of resources and the set of quiescence transtions with tuples $(s, \delta, \overline{0}, C_s, s)$ where $C_s = \bigwedge_{(s,!o,\bar{z},C,s_1)\in T_O} \neg C$.
- $U : S \to (\mathbb{R}^n_+ \longrightarrow R_+)$ is a function associating a utility function to each state in S.

A *configuration* of M is a pair $(s, \bar{v})$, where s is the current state and $\bar{v}$ is the current value of V. We denote by $Conf(M)$ the set of configurations of M.

We say that M is *input-enabled* if for all $s \in S$ and $?i \in L_I$ there exist $\bar{x}$ and s_1 such that $(s, ?i, \bar{x}, s_1) \in T_I$. □

Now we can define the concatenation of several transitions of an agent to capture the different evolutions, from one configuration to another one, that an agent can carry out. These evolutions can be produced either by executing an input or an output action or by offering an *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 3. Let $M = (S, s_0, L, T, U, V)$ be a UIOLTS. We consider that M can evolve from the configuration $c = (s, \overline{v})$ to the configuration $c' = (s', \overline{v'})$ if one of the following options is possible:

1. If there is an input transition $(s, ?i, \bar{x}, s_1)$, then this transition can be executed. The new configuration is $c' = (s_1, \overline{v} + \bar{x})$.
2. If there is an output transition $(s, !o, C, \bar{z}, s_1)$ such that $C(\overline{v})$ holds then the transition can be executed. The new configuration is $c' = (s_1, \overline{v} - \bar{z})$.
3. Let us consider the transition associated with quiescence at s: $(s, \delta, C_s, \bar{0}, s)$. If $C_s(\overline{v})$ holds, that is, no output transition is currently available, then this transition can be executed. The configuration is not altered, that is, $c' = (s, \overline{v})$.
4. Let us consider again the transition associated with quiescence at s, that is, $(s, \delta, C_s, \bar{0}, s)$. If $C_s(\overline{v})$ holds, then we can offer an exchange. We represent an exchange by a pair $(\xi, \bar{y})$ where $\bar{y} = (y_1, y_2, \dots y_n) \in \mathbb{R}^n$ 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 will be willing to perform an exchange $(\xi, \bar{x})$ if $U(s, \overline{v}) < U(s, \overline{v} + \bar{x})$. If another agent is accepting the exchange, then the new configuration is $c' = (s, \overline{v} + \bar{y})$.

We denote an evolution from the configuration c to the configuration c' by the triple $(c, (a, \bar{y}), c')$, where $a \in L \cup \{\xi\}$ and $\bar{y} \in \mathbb{R}^n$. 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 sequence 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. Let $l = e_1, e_2, \dots, e_m$ be a trace of M where for all $1 \leq i \leq m$ we have $e_i = (c_i, (a_i, \bar{x_i}), c_{i+1})$. 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 occurrences of ξ. We sometimes represent this observable trace as $c_1 \stackrel{\sigma}{\Longrightarrow} c_{n+1}$. $\Box$

3 Implementation relations for UIOLTSs

In this section we introduce our implementation relations to 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 user of our methodology may consider that an implementation I of a specification S is good if the number of resources that I obtains after performing some actions is always greater than the one given by S while another user could be happy with an agent that obtains smaller amounts of resources as long as the utility returned by them is bigger than the one foreseen by the specification. Let us remind that implementations must be input-enabled while specifications might not be.

We have defined three different implementation relations. The first one is close to the classical **ioco** implementation relation [5] 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. The formal definition of our first implementation relation was presented in [3]

In order tu define our two new implementation relation we introduce some auxiliary notation

Definition 4. 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,

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

We use $M \;\texttt{after}\; \sigma$ as a shorthand for $c_0 \;\texttt{after}\; \sigma$, being c_0 the initial configuration of M. □

Intuitively, $c \;\texttt{after}\; \sigma$ returns the configuration reached from the configuration c by the execution of the trace σ.

Our firts new implementation relation is based on the **ioco** mechanism but we take into account both the resources that the system has and the actions that the system can execute. In order to define the new relation we need to define the set `out` of outputs. In this case we have 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 5. 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,

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

We extend this function to deal with sets of configurations in the expected way, that is, $\texttt{out}'(C) = \bigcup_{c \in C} \texttt{out}'(c)$.

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 for all output action $!o \in act(A)$ we have $\min(rec(A, o)) \geq max(rec(B, o))$, where $Act(X) = \{a | (a, \bar{y}) \in X\}$ and $rec(X, o) = \{r | (o, r) \in X\}$. □

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 6. 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 $\sigma \in Traces(S)$ we have that $\texttt{out}'(I \;\texttt{after}\; \sigma) \sqsubseteq \texttt{out}'(S \;\texttt{after}\; \sigma)$. □

Our new second implementation relation is again based on the **ioco** approach but we take into account both the utility value that the available resources provide and the actions that the system can execute. In order to define the new relation we need to redefine the set of immediately available outputs. In this case, our set of outputs has two components: The output action that can be executed and the value of the utility function after this action is executed. We also introduce an operator to compare sets of pairs (output,utility).

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,

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

We extend this function to deal with sets of configurations in the expected way, that is, $\texttt{out}''(C) = \bigcup_{c \in C} \texttt{out}''(c)$.

Given two sets $A = \{(o_1, u_1), \ldots, (o_n, u_n)\}$ and $B = \{(o_1, u_1), \ldots, (o_n, u_n)\}$, we write $A \sqsubseteq' B$ if $act(A) \subseteq act(B)$ and for all output action $!o \in act(A)$ we have $\min(util(A, o)) \geq max(util(B, o))$, where $act(X) = \{a | (a, y) \in X\}$ and $util(X, o) = \{u | (o, u) \in X\}$. □

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 value of the utility function 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 value of the utility function of implementation I is *better* than the value of the utility function in the specification.

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

4 Tests: Definition and application

A *test* represents an experiment that will be carried out on an implementation under test (IUT). Depending on the answers provided by the IUT we may conclude that it is behaving in an unexpected way. In our setting, a test can do three different things: It can accept an output action started by the implementation, it can provide an input action to the implementation, or it can propose a exchange of resources. If the test receives an output, then 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. Therefore, we might also detect errors if the amounts of resources differ from the ones that the test indicates.

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. Let us remark that the current notion of test is more involved that the one given in [3] since the latter did not include any mechanism to deal with the amount of resources available to the implementation under test.

Definition 9. Let $M = (S, s_0, L, T, U, V)$ be and UIOLTS, with $L = L_I \cup L_O \cup \{\delta\}$. A test for M is a UIOLTS $t = (S^t, s_0^t, L^t, T^t, \lambda, V)$ 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 `fail` $\neq$ `pass`.
- $\lambda : S \rightarrow \mathbb{R}_+^n$ is a function that assigns a tuple of real numbers to a state. This tuple represents the amount of each resource in this state.
- L^t is the set of actions where L_I is the set of outputs of M, L_O is the set of imputs of M, θ is a special action that represents the detection of quiescence and ξ is an special action that represents the proposal of an exchange.
- $T^t = T_e \cup T_\theta$ is the set of transitions, where
  - $T_e \subseteq S^t \times L_O \cup L_I \cup \{\xi\} \times \mathbb{R}^n \times S^t$ is the set of *regular* transitions.
  - $T_\theta \subseteq S^t \times \{\delta\} \times S^t$.

□

Our tests can compare the resources that the IUT has and the ones properly specified in the test. Therefore, it is suitable to *test* systems according to the ideas underlying the first two implementation relations. If we are interested only in the returned utility (regardless of the specific amounts of different resources), we have to replace the definition of λ by the following: $\lambda : S \rightarrow (\mathbb{R}_+^n \rightarrow \mathbb{R})$ is a function that assigns a utility function to each state in S.

We define configurations of a test in the same way that we used to define them for UIOLTSs, and we thus omit the definition.

Given an implementation I and a test t, running t with I is the synchronized parallel execution of both taking into account the peculiarities of the special actions δ, θ, and ξ.

A first notion of passing a test considers only that the actions that the IUT performs are the expected ones.

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 `fail` of t.

We say that an implementation I passes a test t if all test executions of t with I go to a `pass` state of t. □

Another more complex notion 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 σ of t with I reaches a `pass` state s of t and $rec(I \texttt{ after } \sigma) \geq rec(S \texttt{ after } \sigma)$. □

The previous definition can be modified to deal with the alternative notion of test discussed at the end of Definition 9 where we do not compare resources but only consider the utility returned by the available resources.

Definition 12. An implementation I passes$_u$ a test t if all test execution σ of t with I reaches a `pass` state s of t and $util(I \texttt{ after } \sigma) \geq util(S \texttt{ after } \sigma)$. □

These three notions can be easily extended to deal with set of tests in the expected way: If $\mathcal{T}$ is a test suite then we say that I passes$_x$ $\mathcal{T}$ if for all $t \in \mathcal{T}$ we have I passes$_x$ t.

After definition of test we need to define an algorithm to derive test from specifications. Due to space limitation we do not show the algorithm.

Our algorithm is non-deterministic in the sense that there exist situations where different possibilities are available, and we have different tests depending on the choice that we select. If we consider all the possible choices we will have a full test suite. We denote the test suite produced by the algorithm for a specification M by $Test(M)$. Now we can present results that relate, for a specification S and an implementation I, the application of test suites derived from the specification and the different implementation relations. we omit the proof of this theorem due to space-limitations.

Theorem 1. *Lets S, I be UIOLTSs. We have I $\mathbf{ioco}_*$ S if and only if I passes$_*$ $tests(S)$, where $*$ is r or u or nothing .*

5 Conclusions and Future work

We have recently defined a new formalism, called *Utility Input Output Labeled Transition Systems*, to specify the behavior of e-commerce agents. In this paper we have introduced a testing methodology, based on this formalism, to test whether an implementation of a specified agent behaves as the specification says that it behaves. We have defined three different implementation relations, a notion of test, and an algorithm to obtain, from a given specification, a set of *relevant* tests.

Concerning future work, 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 and would like to define the interaction between agents in order to test multi-agents systems. The second line is more practical since we would like to apply our formalism to real complex agents. In order to support this line of work, we are developing a tool to automatically generate tests from specifications and apply them to implementations.

References

1. M. Núñez, I. Rodríguez, and F. Rubio. Formal specification of multi-agent e-barter systems. *Science of Computer Programming*, 57(2):187–216, 2005.
2. 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.
3. J.J. Pardo, M. Núñez, and M.C. Ruiz. A novel formalism to represent collective intelligence in multi-agent systems. In *New Challenges in Computational Collective Intelligence*, volume 244 of *Studies in Computational Intelligence*, pages 193–204. Springer, 2009.
4. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.
5. J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.