On Generalized Reusable Verification Environment Based Testing

V. SUBEDHA  
Professor & Head  
Department of CSE  
Panimalar Institute of Technology  
Chennai

Dr.S.SRIDHAR  
Dean-Cognitive & Central Computing Facility  
RV College of Engineering  
Bangalore-560059

S.MALATHI  
Professor  
Department of CSE, Panimalar Engineering College  
Chennai

Abstract— Testing can be subdivided into defining or generating test inputs and test scenarios, specifying test oracles to judge testing results, and executing test cases. In this paper we introduce different techniques to tackle these problems, taking into account agent's properties and investigate automated ways to generating test inputs that can produce enormous number of different and challenging situations to exercise the agents under test. The automated generation to some extent helps dealing with the dynamic nature of the environments where the agents under test operate. In this paper we employ, three approaches to evaluate behaviours of software agents. As agents are autonomous, saying if an agent exhibits a correct behaviour or not is not as straightforward as traditional programs. A test evaluation, i.e. to evaluate test results in the first place as feedbacks from test results give important insights to guide the automated test input generation. Then, introduce monitoring as a way to collect data about test execution. The monitoring technique can deal with the distributed and asynchronous property of agent-based systems and provide a global view during test execution. Finally, we present different test generation methodologies to reveal the faults.

Keywords— Agent, Testing, Verification, Agent behaviour, MAS

I. INTRODUCTION

The strong connection between requirements engineering and testing is widely recognized.

1) Designing test cases early and in parallel with requirements helps discovering problems early [2,13], thus, avoid implementing erroneous specifications [9].

2) Requirements produce better tests. Moreover, early test specification produces better requirements because it helps to clarify ambiguities in requirements [12].

The link is so important that considerable effort has been devoted to what is called test-driven development. In such approach, tests are produced from requirements before implementing the requirements themselves. In terms of testing and verification, while some consider specification-based formal verification and other borrow testing techniques, taking advantage of a mapping of agent-oriented abstractions [15,16].

In this work, a testing methodology that exploits the link between requirements and test cases approach with reference to the hardware development methodology [7] and consider MAS as the target implementation technology. Specifically, the proposed methodology contributes to the existing hardware methodologies by providing:

1) A testing process model, which complements the development methodology by drawing a connection between goals and test cases [3] and

2) A systematic way for deriving test cases from goal analysis.

It is noticing that differently from goal-oriented test generation in the context of coverage testing, i.e., generation of test inputs to achieve a coverage goal, such as branch coverage [1], the goal-oriented hardware testing methodology exploiting goal analysis to derive systematically test suites and using the achievement of goals as criteria for testing. Inversely, the derived test suites provide feedback useful for refining the analysis, design and code artifacts to detect and solve problems.

Agents are autonomous, the same test inputs may result in different behaviours at different runs, and since agents might update their knowledge base between two runs [4] or they may learn from previous inputs, resulting in different decisions made in similar situations. Agents communicate through message passing. Traditional testing techniques, involving method invocation, cannot be directly applied. Environment and conventions are important factors that influence or govern the agent’s behaviours. An open MAS that allows third-party agents to come in and access to the resources of the MAS. Work in testing software agents and MAS can be classified into different testing levels: unit, agent, integration, system, and acceptance [5,6,8].

The testing objectives, subjects to test and activities of each level are described as follows:
1) Unit Test all units that make up an agent, including blocks of code, implementation of agent units like goals, plans, knowledge base, reasoning engine, rules specification and so forth, make sure that they work as designed.

2) Agent Test the integration of the different modules inside an agent, test agent’s capabilities to fulfill their goals [14] and to sense and effect the environment.

3) Integration or Group Test the interaction of agents, communication protocol and semantics [17], interaction of agents with the environment, integration of agents with shared resources, regulations enforcement [18]. Observe emergent properties, collective behaviours; make sure that a group of agents and environmental resources work correctly together.

4) System or Society Test the MAS as a system running at the target operating environment; test the expected emergent and macroscopic properties of the system as a whole; test the quality properties that the intended system must reach, such as adaptation, openness, fault tolerance, performance.

5) Acceptance Test the MAS in the customer’s execution environment and verify that it meets stakeholder goals, with the participation of stakeholders.

The active work on testing software agents and MAS, with classification is intended only to help easily understand the research work in the field. It is also worthwhile noticing that this classification is not complete in the sense that addresses testing in more than one level, but we put them in the level they mainly focus.

II. EVALUATION OF AGENT BEHAVIOURS

It considers three types of agent faults: faults related to constraints that restrict agent's behaviours, faults related to interaction semantics that define the semantics of agent interaction and faults related to user's requirements. Corresponding test oracles are defined to pinpoint these kinds of faults.

A. Constraint-based oracle

The behaviour of autonomous agents can change over time. This makes the evaluation of test results a non-trivial task. Often, it is impossible to give a fixed verdict to a test case based on the comparison of the returned message with a gold standard, because the returned message may be different, even for the same input, at different times. Similarly, mental states of an agent e.g., beliefs can change with respect to the same inputs, specifying some invariants as oracles for these variables can be non-trivial. It proposed to use constraints that restrict the behaviours of software agents as test verdicts. Constraint violations are considered as faults. Behavioural constraints are specified in terms of pre-, post- and invariant conditions. However, software agents are distributed programs that run at geographically different hosts, handling constraint violations.

B. Ontology-based oracle

1) Agent interaction ontology

In order for a pair of agents to understand each other, a basic requirement is that they speak the same language and talk about the same things. This is usually achieved by means of ontology, namely, interaction ontology. They provide tools for generating code from ontology documents, thus, reducing the development effort and for runtime binding of the message contents with concepts defined in an ontology. A common structure of interaction ontology involves two main concepts: (i) Concept and (ii) AgentAction. AgentAction define actions that can be performed by some agents, while define common concepts understandable by agents that interact.

2) Software agents and MAS testing

Software agents are computational programs that have the following properties: Reactivity, agents are able to sense environmental changes and react accordingly, proactivity, agents are autonomous that are able to choose which actions to take in order to reach their goals in given situations, social ability that is, agents are interacting entities, which cooperate, share knowledge, or compete for goal achievement. MAS are systems composed of multiple autonomous agents that interact with one another in an open environment to fulfill their goals and the goals of the systems as a whole. A MAS is usually a distributed and decentralized system, its agents can be located at geographically-different hosts and they communicate mainly through message passing. An agent might have to wait for other agents to fulfill its intended goals. An agent might work correctly when it operates alone but incorrectly when put into a community of agents or vice versa. MAS testing tools must have a global view over all distributed agents besides local knowledge about individual agents, in order to decide whether the whole system operate accordingly to the specifications. In addition, all the issues related to testing distributed systems are applied in testing software agent and MAS as well, with controllability and observability.

III. TEST SUITE DERIVATION

The derivations take place, when test suites are executed and goal-oriented test adequacy at each test level. It derive test suites contain four parts, as shown in Figure 1.

1) how to derive test suites for acceptance test from organizational and system goals
2) how system collaborative and agent goals are used to create system test suites
3) To exploit collaborative and agent goals create integration test suites in the development process to the agent interaction and capability design, and
4) Create test suites for agent plans, goals, and agents themselves.

Figure 1 Test suite derivations

A. System test
The transition from requirements to architectural design phase consists of identifying agents that realize the specified system actors, assigning system actor’s goals to agent’s goals and projecting system actor’s dependencies to agents dependencies and interactions. The artifacts (actors, goal models) obtained from the requirements phase, there are agents, their goals, roles, collaborative goals, agent’s dependencies for goals, resources, the dependencies between agents and the environment, regulations, constraints. System test suites should consider and make use of these artifacts. System tests suite derivation takes place in parallel with architectural design. Similar to acceptance test suite derivation, where it takes stakeholder actor’s goals as foundation concepts, uses system actor’s goals as foundations to create system test suites as they provide the system-level objectives and requirements. When the system as a whole is built so that the system actor’s goals are fulfilled, it is ready to be passed to the customer for acceptance test.

The procedure is described as follows: for each hardware actor, the goal analysis of the actor is analyzed to filter the leaf goals. For each leaf goal of a hardware actor, one has to create a test suite to test the achievement of the goal. Five creation steps are:
1) Identifying which agent(s) realize(s) the goal g,
2) Analyzing the goal model of each agent to identify goals related to the achievement of g,
3) Identifying environmental factors, pre-conditions, inputs that facilitate or trigger g,
4) Identifying fulfillment criteria for g,
5) Creating a test suite having a set of test cases for g that take inputs and identified from previous steps.

Since, hardware actors can have more goals than those delegated to the system by stakeholder, the number of system test suites is usually higher than the number of acceptance test suites. As a consequence, system test suites can reuse information from acceptance test suites, but much more details can be added, such as fulfillment criteria for goals and expected behaviours of agents involved [13].

B. Integration test
The aim of integration testing is to make sure that agents work together correctly, sharing tasks and resources to achieve collaborative or agent goals [10,11]. To obtain this objective, consider dependencies between agents for collaborative goals and dependencies between agents and resources. Integration test suite derivation takes place once have finished detailed design, so that it can make use of the interaction protocol design. The derivation for collaborative goals consists of the following steps:
/* create a test suite for g */
step1: identify agents involved
step2: identify interaction scenarios
step3: identify interaction protocols, ontologies
step4: identify fulfillment criteria for each scenario
step5: create a test suite for each scenario
step6: end for;

The procedure reads: in the architectural design of the system we identify a set of collaborative goals. For each of these goals we identify agents that are involved, interaction scenarios, protocols, and ontology. Finally, for each scenario can define a test suite making use of data identified, i.e. agents, protocols, criteria, and so on. A collaborative goal that involves all the cleaning agents go further into the detailed design of the agent, it determines two interaction scenarios: (1) one cleaning agent broadcasts information about its location; and, (2) the agent receives a message broadcast from another cleaning agent. An agent sends a request to the addresses of other cleaning agents. Once a list of agents is returned, the agent broadcasts a message containing situated information to all the agents.

Testing the integration of agents with the operating environment consists of testing their perception and affecting capabilities. There is a need to make sure that the agents under test are able to perceive changes regarding the resources are interested in. This is shown in Figure 2.
The following steps guide us when deriving test suites for testing the agent-environment interaction:
/ * for all agent do */
step1: identify related resources
step2: identify integration scenarios
step3: identify access policy, interaction protocol, ontology, and other related factors if any.
step4: identify fulfillment criteria for each scenario
step5: create one test suite for each scenario
step6: end for;

The procedure is described as follows: for each agent type in the system identifies resources that the agents of the type use. It identifies usage or interaction scenarios, access policies, protocols and other related factors. Finally, define criteria for each scenario and create a test suite for it, making use of the data identified. Test suites created for this objective should focus on providing necessary environment, so as to facilitate the agent interaction under test and on enforcing the rules that govern the behaviour of the agents under test. As with the other testing levels, integration test suites are aimed at two distinctive targets:

1) To refine the interaction design and solve integration problems as early as possible and,
2) To test the integration of the implemented agents with one another and with the environment, once these are available.

The target is realized during the integration test suite derivation, while the second can be started as soon as an agent or an environmental resource is implemented. Mock agents, which simulate behaviours of agents, can be used during integration testing so that do not need to wait until all the involved entities are implemented to start integration testing.

A. Unit and agent test

Unit testing consists of verifying agent units, e.g., goals, plans, beliefs, and events that agents are composed, it mainly plan, goal and agent testing. The proposed test suite is shown in table 1.

1) Plan testing

Different aspects related to plans and events testing introduced in the context, apply to an approach as well. The plans are achieve goals (ends), plans are triggered as a result of goals selection. Consequently, to test a plan, need to create test suites such that they satisfy all the pre-conditions of its end goal and pre-conditions of the plan itself. These conditions, among others, contain corresponding events or percepts that eventually trigger the plan. Then, have to evaluate the execution of the plan, its subsequent tasks. For each single plan, need to create a test suite that contains a set of test cases to challenge the plan with different inputs.

2) Agent goal testing

Goals are states of affair and one must do something in order to achieve a very natural way of testing the achievement of a goal is to check one's work or behaviour with respect to the goal. Similarly, to test a goal we have to check what the agent does to fulfill the goal.

<table>
<thead>
<tr>
<th>Suite Plan</th>
<th>Test Suite 1 Plan Class</th>
<th>Move</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Test Case 1.1</td>
<td>There is an event that requires the cleaning agent to move from position A(1,1) to position B(3,5) no obstacle is in the middle of the two points</td>
<td>The cleaning agent moves straight from A to B</td>
<td></td>
</tr>
<tr>
<td>User Test Case 1.2</td>
<td>Between A(1,1) and B(3,5), there is a static obstacle at point C(2,3)</td>
<td>The cleaning agent moves close to C, identifies the obstacle, avoids C before going to B</td>
<td></td>
</tr>
<tr>
<td>User Test Case 1.3</td>
<td>The agent is requested to move from (1,1) to (3,1)</td>
<td>The cleaning agent moves to the boundary nearest to (3,1)</td>
<td></td>
</tr>
</tbody>
</table>

The principal elementary relationships are depicted in Figure 2. These include:

1) Means-End between a plan and a hardgoal,
2) Contribution + between a plan and a softgoal,
3) Contribution - between a plan and a softgoal.

In order to test this kind of relationships, the execution of the plan corresponding to a goal is triggered and checked based on assertions and constraints on the expected behaviour. Developers derive test suites from goal diagrams by starting from the relationships associated with each goal. Each relationship gives rise to a corresponding test suite, consisting of a set of test cases that are used to check goal fulfillment (called positive test cases) and counter-fulfillment (called negative test cases). Positive test cases are aimed at verifying the fulfillment capability of an agent with regard to a given goal, negative test cases [4], on the other
hand, are used to ensure an appropriate behaviour of the agent under test when it cannot achieve a given goal.

The derivation steps are as follows:

1: for all g
2. leaf goals g
Do {
3: identify means/plans from elementary relationships.
4: identify the fulfillment criteria of g
5: identify possible plans execution orders or schedules
6: create one test suite for g
}
7: end for

The procedure can be described as follows: for each leaf goal we identify means plans from the elementary relationships related to the goal. The fulfillment of the goal and possible plan executions are then defined. Finally, a test suite should be created for the goal in which each test case addresses one possible execution scenario. For the intermediate goals (i.e. not leaf goal), test suites are derived by inspecting all relationships that lead to the considered goals. This ends up analyzing all elementary relationships and creating/reusing test suites derived for them. Once the results of these test suites are obtained, can reason about the achievement of the intermediate goals based on the decomposition and/or contribution analysis.

A. Agent testing

An agent is composed of smaller components, e.g., beliefs, goals, plans, events, reasoning module, and so forth. Testing at the agent level consists of integration testing of agent components, so one has to derive test suites to verify this integration. Agent-level test suites have a strong relation with test suites created for testing agent goals. Because, in most cases, testing a goal involves testing one or a number of plans, testing a plan involves events, percepts, and resources. So to some extent, testing a goal triggers some integration of plans, events, and so on. Hence, test suites derived to test agent goals are also effective to test the agent integration. However, at the agent level, need to test the integration of goals as well. Some goals have dependencies among them, such as priority or inhibition dependences; others may be maintained or achieved in parallel while sharing a resource. So, have to identify goal integration scenarios, create test suites for each, and look for integration problems such as dependency violations, deadlock, and livelock. For example, at a given moment in time, the cleaning agent can only move either to a recharging station, or to a waste bin, or to a new position for patrolling. The basic test adequacy requirement for an agent is that all the agent goals must be tested. The agent should be able to achieve its goals and behave correctly in the cases where its intended goal cannot be achieved. This adequacy requirement may or may not be sufficient to cover the agent components, i.e. plans, events, beliefs, etc. If some are never exercised by the test suites defined to reach the basic adequacy criterion (goal coverage), more test suites have to be defined to complete agent testing.

B. Test suite structure

The key elements of goal-oriented testing are goals, either organizational, system, collaborative, or agent goals. The underlying objectives at different testing levels consist of tackling goals fulfillment. Thus, derived test suites must be able to specify test target, i.e. goal, and test scenario, including inputs, conditions, and expected behaviour of the agent under test. To support specifying goal-oriented test suites, it proposes the structure as shown in Figure 3. Each Test Suite contains a set of Test Cases, each Test Case contains a test scenario in which Test Actions are specified. Each Test Suite targets one or more agents, goals, and/or plans. Each Test Suite or Test Case can contain Support Actions (e.g., setup testing environment, tear down when finished). Finally, pre- and post-conditions can be specified for a goal, a plan or a scenario.
cases can be read by testing tools. To this purpose, the contents of the elements Test Action, Support Action, Condition support user-defined data types. Developers can associate their machine-readable data with their own parser and grammar so that test suites can be executed automatically.

IV. TEST CASE GENERATION

A. Goal-oriented generation

Goal-oriented test cases generation is a part of a methodology, it integrates testing providing a systematic way of deriving test cases from output artifacts. Goal-based specification diagrams are used as inputs to generate test case skeletons to test goal fulfillment. Specific test inputs (i.e. message content) and expected outcome are partially generated from plan design and then completed manually by the tester according to some test scenarios. These scenarios can be user-defined or can be some particular interaction protocols.

B. Ontology-based generation

This technique concerns generating messages to test software agents. Agent behaviours are often influenced by messages received. An approach to test generation using agent interaction ontology, the approach exploits ontology that defines the semantics of agent interactions to generate test inputs and guide the exploration of the input space. The task of the ontology-based test generator consists of completing the message content to send to the agent under test. For each concept to be instantiated in the message, the generator either picks up an existing or creates a new instance of the required concept. No input value is generated by the test generator if the interaction protocol prescribes that a value from a previously exchanged message must remain the same. When new instances are generated, the test generator selects one from those available in the ontology. The selection is based on the number of usages of each instance, or aims at increasing the diversity of test inputs and exploring the input space more extensively. In the case when no ontology instances are available, valid test inputs can be still generated by using available information, such as rules and property data types, specified in the interaction ontology.

C. Random generation testing

Random testing proven to be very effective in revealing some types of faults, specially those that result in crashing or raising exceptions. In dynamic and open environments for MAS, random testing seems to be a natural choice because it can generate unpredictable scenarios, which likely happen in such environments. It is interested in two types of test inputs: (i) messages and (ii) environment settings.

An approach to random testing of software agents, a communication protocol is randomly selected among the standard ones provided by the agent platform, or those specified in a library by human tester. Then, messages that are required by the protocol are randomly generated and sent to the agents under test. The message format is the agent environment of choice, while the content is constrained by a domain data model. Such a model prescribes the range and the structure of the data that are produced randomly, either in terms of generation rules or in the form of sets of admissible data that are sampled randomly. The model of domain data can be specified by means of ontology as well, so ontology-based generation rules can be applied to generate message content. This is shown in Figure 4.

![Figure 4 Procedure of the random generation technique](image)

Randomly generated messages are then sent to the agents under test and it is the responsibility of our monitoring agent network to observe their responses, happening in the agent system. When a deviation from the expected behaviour is found it is reported to the development team. A limitation of random testing of MAS is that long and meaningful interaction sequences are hardly generated randomly. However, it is often the case that agent interaction protocols need only few trigger messages, or the agent under test needs only one message to trigger its goals. In these cases, random testing is a cheap and efficient technique that can reveal faults.

D. Evolutionary generation

The specific properties of software agents demand for a framework that supports extensive and possibly automated testing. Therefore, it proposes to apply ET (Evolutionary Testing) for testing software agents and define two methods to guide the evolution of test cases: mutation guided, and quality function guided. In this technique, the agents under test are free to evolve during testing, but, at the same time their behaviours are observed and used to guide the evolution of test cases, making them more challenging, to run again on the next cycle. Testing objectives, e.g., to see if an agent violates a constraint, are transformed into fitness functions to guide the evolutionary generation of test inputs. The testing procedure is presented in Figure 5.
Figure 5 Evolutionary testing procedure

It has the following steps:

1) Generate initial population
A set of test cases is called population. Each test case is an individual in the population. Initial population can be generated randomly or taken from existing test cases created by testers.

2) Execution and monitoring
Test execution means to put the autonomous agents under test into the testing environment so that they can operate, i.e. performing tasks or achieving goals, or to send messages to them. At the same time, a monitoring mechanism is needed to observe the behaviours of the autonomous agents. Relevant observed data are recorded. Many executions might need to be performed repeatedly in order to provide statistically sufficient data to measure fitness values. The agents under test might need a sufficient amount of time to perform their tasks.

3) Collect observed data and calculate fitness values
Cumulative data from all executions are used to calculate fitness values of selected test cases. The way of calculating fitness values depends on the stakeholder’s softgoal of interest and the problem domain. As calculated fitness values provide insights about the improvement towards the optimal ones, if no improvement is observed after a number of generations, the test procedure will stop. Otherwise step 4 will be invoked.

4) Reproduction
Two elite individuals are selected, and then crossover operation is used to produce two new offsprings. Finally, mutation is applied with certain probability on one or both offsprings. The two offsprings are then put back to the population and the next iteration is triggered.

V. CONCLUSIONS

In the research work, on testing software agent and MAS focuses mainly on agent and integration level. Basic issues of testing software agents like message passing, distributed/asynchronous have been considered, testing frameworks proposed to facilitate testing process. However, there is still much room for further investigations, such as,

1) A complete and comprehensive testing process for software agents and MAS.
2) Testing MAS at system and acceptance level how the developers do and the end-users build confidence in autonomous agents?
3) Test inputs definition and generation to deal with open and dynamic nature of software agents and MAS.
4) Deriving metrics to assess the qualities of the MAS under test, such as safety, efficiency and openness.
5) Reducing/removing side effects in test execution and monitoring because introducing new entities in the system, e.g., mock agent’s tester agents and monitoring agent as in many approaches, can influence the behaviour of the agents under test and the performance of the system as a whole.

REFERENCES


