2012 IEEE 36th International Conference on Computer Software and Applications

CATest: A Test Automation Framework for Multi-Agent Systems

Shufeng Wang
National Laboratory for Parallel and Distributed Processing
National University of Defense Technology
Changsha, 410073, China
Email: shufeng.wang@gmail.com

Hong Zhu
Dept of Computing and Communication Technologies
Oxford Brookes University
Oxford, OX33 1HX, UK
Email: hzhu@brookes.ac.uk

***Abstract*—Agents are difficult to test because it is notoriously complicated to observe their proactive, autonomous and non-deterministic behaviours and hard to judge their correctness in dynamic environments. This paper proposes a specification-based test automation framework and presents a tool called CATest for testing multi-agent systems (MAS). The agent-based formal specification language SLABS plays three roles in the framework. First, it is used to guide the instrumentation of the agent under test so that its behaviour can be observed and recorded systematically. Second, the correctness of agent's behaviours recorded during test executions are automatically checked against the formal specifications. Finally, the test adequacy is measured by the coverage of the specification and determined according to a set of adequacy criteria specifically designed for testing MAS. An experiment with the tool has demonstrated its capability of detecting faults in MAS.**

***Keywords*-Software test automation; Test adequacy criteria; Test oracle; Agent-oriented software; Specification-based test.**

I. Introduction

Testing is labour intensive and expensive. It is imperative to reduce the cost and improve the effectiveness of testing by automation. In the past decades, a great amount of research efforts has been reported and a significant progress has been made; C.f., [1]. The advent of agent-oriented development methodologies, which has been widely perceived as a promising new paradigm suitable for the Internet-based computing [2], raises the stakes for research on test automation. Agent-oriented systems are extremely difficult to test because they are poor on both controllability and observability aspects of software testability [3]. The challenge is: can automated testing tools deal with the complexity of agent-oriented systems? This paper proposes a new architecture of test automation framework (TAF) to meet the challenge and presents a caste level agent testing tools called *CATest* that realises the framework.

The paper is organized as follows. Section II reviews the current state of the art in the research on related topics. Section III outlines the proposed architecture of TAF. It is followed by technical details of CATest. Section IV reviews the formal specification language SLABS [4] on which CATest is based. Section V is devoted to the behaviour observation in testing agent-oriented software. Section VI discusses how to check the correctness of an agent's behaviour against a formal specification. Section VII defines a set of test adequacy criteria. Section VIII reports the implementation of the tool CATest. Section IX reports an experiment with CATest. Section X concludes the paper with a discussion of future work.

II. Current State of Art

Let's first briefly review the current state of the art.

A. Test Automation Frameworks

The current trend in the practice of software testing is to automate testing activities via the employment of TAFs. A large number of tools that support TAFs have been developed and widely used in the industry, which include *JUnit* for testing software written in Java, *CppUnit* for C++, *NUnit* for .NET, *RUnit* for Ruby, *PyUnit* for Python, *VbUnit* for Visual Basic, and *Selenium* for Web Services, just to mention a few. As Meszaros summarised [5], the common features of such TAFs are:

- associating each program unit (e.g. class) with a test unit that contains a collection of *test methods*, each for a test. For OO programming languages, the test unit is a declared as a subclass of the class under test and called *test class*.
- specifying the expected test results for each test in the form of calls to *assertion methods* in the test class;
- aggregating a collection of tests into *test suites* that can be run as a single operation by calling the test methods;
- executing test suites and reporting the results once the code of the program is revised.

The automated tool that supports a TAF sets up automatically the environment in which test methods and assertion methods are executed and enables test results to be reported. It can significantly reduce test costs and increase test efficiency, especially when the program code is revised frequently and testing is repeated for many times such as in agile development processes. Moreover, the test code becomes a valuable and tangible asset and can be sold to component consumers. Therefore, testing is no longer an activity that only consumes resources, but it also produces assets. This brings up a new economics of software testing.

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.24

It is no surprise that the adoption of TAFs in the IT industry has gained a rapid increase in the past years.

However, existing TAFs have some fundamental weaknesses, which include:

- *Manual coding of test classes*.

It relies on programmers to write test code to represent test cases in test methods and to translate specification into assertion methods. This is not only labour intensive, but also error prone.

- *Lack of support to the measurement of test adequacy*.

There is no facility in the existing TAFs that enables the measurement of test adequacy.

- *Weak in the support to correctness checking*.

The assertion methods can only access the local variables and methods of the unit under test. This means the correctness cannot be checked against the context in which the unit is called. Moreover, correctness checking cannot across multiple executions of the unit under test.

These weaknesses root deeply in the architecture of TAFs. Their applicability is limited to unit testing.

B. Testing Agent-based Software

In the past few years, research efforts have been reported in the literature on testing MAS in the context of various agent-oriented development methodologies and platforms. They have addressed the following aspects of testing MAS:

- correctness of interaction and communication [6]–[10];
- correctness of processing internal states [11]–[14];
- generation of test cases [12], [15]; and
- control of test executions [16]–[18].

There is little research on adequacy criteria for testing MAS. As far as we know, the work of Low *et al.* [14] is the only exception. They proposed a set of coverage criteria defined on the structure of plans for testing BDI (Belief-Desire-Intention) agents.

In the research on testing agent-oriented software systems, TAFs have also been proposed for various agent platforms by extending OO TAFs with agent-specific features. Works among the most well known are *SUnit* for Seagent by extending JUnit [17], *JAT* for Jade [7], the testing facility in INGENIAS for model-driven development of (MAS) [18], and [13] for model-based development in Prometheus methodology. However, these approaches inherited the fundamental weaknesses of OO TAFs. For the following reasons, the weaknesses become more serious for testing MAS.

First, different from objects, agents are autonomous, proactive, adaptive and context-aware. They often deliver the functionality through emergent behaviours that involve many agents. Thus, the correctness of their behaviours must be judged in the context of the dynamic and open environments and the histories that they have experienced in previous executions. Correctness checking replying on agent's internal information and the data at a single time point is insufficient.

Second, the specifications of the required behaviours are often hard to translate into assertion methods manually. It is highly desirable to use formal specifications of MAS directly to reduce the cost as well as the errors of manual translation.

Moreover, most MAS are continuous running systems. Given a test input, it is vital to determine when to stop the test execution in order to achieve good test adequacy. A support to measure test adequacy during testing executions is essential for testing MAS. Therefore, it is necessary to take a radical approach to TAF for agent-oriented software.

The architecture of TAF proposed in this paper takes a specification-based approach to overcome these weaknesses.

III. Overview of The Proposed Framework

The basic idea of the architecture is to provide a much stronger support to the observation of the software's dynamic behaviours through systematic instrumentation of the software under test. This is achieved by using formal specifications or models of agent-based systems to guide the instrumentation of the agent under test (AUT) so that its behaviour can be systematically observed and efficiently recorded. The recorded test behaviours are then automatically checked for correctness against the specification. Thus, the manual coding of assertion methods can be eliminated. The recorded behaviours are also used in the measurement of test adequacy by calculating the coverage of the formal specification or model. Such measurements can be performed dynamically during test executions and statically after the test executions. More importantly, test executions can be stopped automatically when a selected test adequacy criterion is met. We will demonstrate the feasibility of the approach by an automated testing tool CATest and show its fault detecting ability by an experiment using mutation analysis. It is also worth noting that the approach does not only apply to unit testing, but all level of agent testing, although this paper will only focus on agent unit testing.

A. Different Levels of Testing

To deal with the complexity in testing agent-based systems, we divide testing activities into four levels according to the objectives of testing.

1) Infrastructure level: Agent communications are often implemented by employing mechanisms and facilities different from what the traditional OO paradigm provides. For example, MAS commonly use subscribe-and-publish, group broadcast, brokers and mediators, and environment objects *etc.*, or a combination of them. The aim of infrastructure level testing is to validate and verify the correctness of the implementation of agent communication and interactions.

2) Caste level: Here, *caste* in agent-orientation is equivalent to class in object-orientation. Agents often exhibit complicated behaviours, contain complex internal states and situate in dynamic environments. The correctness of an agent's behaviour in updating its internal state and taking an action in response to an environment situation are fundamental to the correctness of the whole system. At this level, testing focuses on validating and verifying the correctness of each individual agent's behaviour.

3) Cluster level: In MAS, a group of agents often play various roles to achieve certain group goals while each agent may have its own individual goals. A group of agents may be regarded as one entity or a subsystem at a higher level of abstraction. It is important to test how agents form such groups, re-organise the group via role assignments, membership changes, behaviour adaptations, etc. Testing at cluster level aims at validating and verifying the correctness of the behaviours of a group of agents in interaction and collaboration processes.

4) Global level: Most MAS, such as ant colony optimisation systems, deliver their functionality as emergent behaviours, which are the most elusive feature of MAS. Testing whether a MAS has certain emergent behaviour cannot be achieved by just observing one or a few agents. It must take all agents in the system into consideration. This is what global level testing is concerned with. It aims at validating and verifying the correctness of the whole system's behaviour, especially the emergent behaviour.

At the different levels of agent testing, the specific aspects of behaviour to be observed and the means of observation are different. The specific test adequacy criteria and correctness conditions are also different. However, the proposed architecture is applicable for all these levels, although this paper will only give details of the testing tool CATest for caste level testing. CATest is only a part of our complete TAF called CATE-Test, which supports testing MAS at all levels.

B. Architecture of the Framework

The architecture of the proposed framework consists of the following key components.

- *Runtime facility for behaviour observation.*

A runtime facility in the form of a testing library provides support to the observation of the dynamic behaviours. Invocations of the library methods are inserted into the source code of the AUT so that when the agent under test is executed, its behaviour is observed and recorded systematically, which enables both correctness checking and adequacy measurement.

- *Test oracle.*

A test oracle takes a formal specification or model and a set of recorded dynamic behaviours of an implementation of the MAS as inputs and checks automatically the correctness of the recorded behaviours of the AUT against the formal specification.

- *Test coverage calculator.*

The framework employs a generic adequacy calculator to measure the test coverage according to a set of test adequacy criteria. It also takes a formal specification/model and a set of recorded behaviour as inputs, but it calculates the coverage of the specification/model while the correctness of AUT's behaviour is checked against its specification.

- *Test execution controller.*

The test executions of the MAS are controlled by the runtime execution controller. It stops the execution of the system on a test case according to a user selected elemental adequacy criterion by running the coverage calculator in parallel to the system under test. It also stops the whole testing process when the collective test adequacy is achieved.

Note that the same as all existing TAFs, the proposed framework leaves test case generation as an open issue. However, the architecture makes the integration of a test case generator with the framework simple and easy because its operation is completely independent of the test cases.

C. The Testing Process

Assume that test cases have been generated, e.g., by applying the existing techniques mentioned in Section II. Our testing process consists of the following steps.

1) Instrument the code of the AUT and the main class of the MAS system.
2) If the MAS is non-terminating (i.e., it is designed to execute forever), select an elemental adequacy criterion to determine when a test execution of the system on one test case will stop.
3) Select a collective adequacy criterion to determine when a set of test executions are adequate and thus the whole testing process can stop.
4) Execute the program on the test cases one by one and record the behaviour until the collective adequacy criterion is satisfied or all the test cases are executed. Here, each test execution stops either when the elemental adequacy criterion is satisfied, or the execution terminates, or the collective test adequacy criterion is satisfied.
5) Check whether the recorded behaviours are functionally correct w.r.t. to the specification.
6) Finally, check the probabilistic accuracy w.r.t. the specification.

Note that, first, we require that the elemental adequacy criterion can be always satisfied on every valid input. A simple example of elemental adequacy criteria is the length of time to execute on each test case. Another simple example of such elemental adequacy criteria is the number of rule firings to be made in each test execution. Second, if the

collective adequacy is not satisfied on a set of test cases, there are three possible reasons.

1) The elemental adequacy criterion is not strong enough. Then, it should be adjusted to execute the program for a longer time on each test case.
2) The set of test cases is inadequate. Then, new test cases should be generated.
3) The collective adequacy criterion is infeasible, e.g., it requires executing infeasible statements. This situation can be avoided if the adequacy criterion is finitely applicable. The test adequacy criteria proposed in this paper are all finitely applicable.

The caste level testing technique reported in the paper is specification-based. We assume the existence of rule-based specifications of MAS written in SLABS [4].

IV. Brief Review of SLABS

As in [4], an agent is defined as an active computational entity that executes in a designated environment. It encapsulates data, operations, behaviour rules and a description of its environment. These elements are specified in SLABS as follows.

- *State Variables*: The states that an agent can be in are represented by a set of state variables. They are divided into two kinds: visible and invisible. The value of a visible variable, indicated by an asterisk symbol before the identifier, can be viewed by other agents, but cannot be changed by any external entities.
- *Actions*: The actions are what the agent is capable of performing to change its state and to communicate with other agents. They are also classified into two kinds: visible and invisible. Taking a visible action produces an event that other agents in the system can observe.
- *Behaviour*: The behaviour of an agent is governed by a set of rules. It determines when the agent will take an action and which action to take, and when to change its state.
- *Environment description*: The environment in which an agent inhabits is a collection of other agents (or objects). An agent X is in the environment of agent A means that agent A can observe X's visible actions as events, and view the values of X's visible state variables.

A *type* of agents that have the same structure and behaviour is called a *caste*, which is equivalent to class in OO languages. However, different from class, an agent may have membership to multiple castes and such casteships may change at runtime through joining or quitting a caste. Fig. 1 shows the structure of caste specification in SLABS.

The environment of an agent is specified by a set of clauses in the following forms.

- $AgentID : CasteID$: The specific agent with the name $AgentID$ in the caste $CasteID$ is in the environment.

Caste $C \Leftarrow C_1, \cdots, C_k$; (* inheritance Relations *)
  Environment (* environment declaration *)
    $E_1, \cdots, E_w$;
  Var (* state variables declarations *)
    $*v_1 : T_1, \cdots, *v_m : T_m$; (* visible variables*)
    $u_1 : T'_1, \cdots, u_{m'} : T'_{m'}$; (* invisible variables*)
  Action (* action declarations *)
    $*A_1(p_{1,1}, \cdots, p_{1,n_1})$, (* visible actions*)
    $\cdots$,
    $*A_s(p_{s,1}, \cdots, p_{s,n_s})$;
    $B_1(q_{1,1}, \cdots, q_{1,n'_1})$, (* invisible actions*)
    $\cdots$,
    $B_t(q_{t,1}, \cdots, q_{s',n'_t})$;
  Behaviour (* specification of behaviour rules *)
    $R_1, \cdots, R_h$.
End C.

Figure 1. Structure of Caste Specification

- $AgentVar : CasteID$: An agent in the particular caste $CasteID$ determined by the variable $AgentVar$'s value is in the environment.
- $All : CasteID$: All the agents in the caste $CasteID$ are in the environment.

An agent's environment is dynamic in the sense that it may change during execution. For example, when its environment description contains clause $All : CasteX$, the environment changes if an agent joins or quits $CasteX$.

A behaviour rule is essentially a probabilistic *if-then* statement. In SLABS, it is in the following form.

$$RuleName : Pattern \mid Probability \rightarrow Action, \quad \textbf{if } Scenario, \textbf{where } Condition.$$

where $Pattern$ describes the agent's previous behaviour in terms of a sequence of state changes and actions. $Scenario$ specifies the context in the environment, i.e., the situations in the system outside the agent. The $Pattern$ and $Scenario$ form the condition to apply the rule, hence they are called the guard-condition of the rule. $Probability$ is an arithmetic expression that defines the probability that the rule is applied when the guard-condition is satisfied. $Action$ is the action to be performed if the rule is applied. The $where$-clause specifies the relationships between the internal state and the external scenario before and after firing the rule. The $Action$ and $where$-clause together defines the effect of the rule, which asserts the state of the agent after firing the rule. So, behaviour rules can be transformed into the following form:

$$\textbf{if } GuardCond \textbf{ then } PostCond \textbf{ with } Probability.$$

In SLABS, a pattern is in the form of $[E_1, E_2, \cdots, E_n]$, where $E_i, i = 1, \cdots, n$, are the most recent actions taken by the agent or predicates that specify the past states of the agent. A scenario expression is constructed from patterns

using quantifiers and logic connectors. Readers are referred to [4] for details.

Fig. 2 is an example of formal specification in SLABS. It specifies the *Salt World* system, in which a number of turtles collect the salt grains initially distributed randomly in a field and put them on piles of salt grains.

```
specification SaltWorld;
caste Turtle;
 environment all: Point;
 var *saltKind: integer; *position: Point;
  state: {FindingSalt,FindingPile,GettingAway};
 action *move; *pickSalt; *dropSalt;
 behavior
  <Move> [!state==FindingSalt & position==p
          & saltKind == 0]
  |-> move !position==p1 & state==FindingSalt
          & saltKind==0;
     if p : [!saltKind==sk & neighbour==neigh &
        (hasSalt==false | onPile(sk)==true)];
     where p1 in neigh;
  <Pick> [!state==FindingSalt & position==p
          & saltKind==0]
  |-> pickSalt !state==FindingPile &saltKind==sk
     & sk != 0;
     if p : [!saltKind==sk & hasSalt==true &
        onPile(sk)==false];
  <Search> [!state==FindingPile & position==p
           & saltKind == sk & sk != 0]
  |-> move ! position == p1 & state==FindingPile
     & saltKind == sk;
   if p : [!(onPile(sk)==false | hasSalt==true)
      & neighbour == neigh];
     where p1 in neigh;
  <Drop> [!state == FindingPile & position == p
          & saltKind == sk & sk != 0]
  |-> dropSalt !state==GettingAway &saltKind==0;
     if p: [!onPile(sk)==true & hasSalt==false];
  <GetAway> [!state==GettingAway & position==p]
  |-> move !position == p1 & state==FindingSalt;
end Turtle;
caste Point;
 environment all: Turtle;
 var *saltkind: integer; *hasSalt: boolean;
  *onPile(saltkind: integer): boolean;
  *neighbour: Set<Point>;
 behavior ... (*omitted for the sake of space*)
end Point;
caste Field
 var *width: 1..100; *height: 1..100;
  *member: Set<Point>;
end Field
end SaltWorld
```

Figure 2. Specification of Salt World

V. Runtime facility for Behaviour Observation

This section presents the runtime facility for observing and recording agent's dynamic behaviours.

A. Runtime Facility

We have developed a library of Java classes that supports recording agents' dynamic behaviours at runtime. The recorded data are strings in a data file that can be parsed and analysed by the correctness checker and adequacy calculator, and other parts of the framework. Fig. 3 gives the syntax definition of the data records in the format of ANTLR parser generator rules [19].

```
record   : ID ':'^ data;
data     : state | function | action;
action   : ID | ID params->^(ACTION ID params);
state    : ID '='^ value;
function : ID params '=' value
           -> ^(FUNCTION ID params '=' value);
params   : '('! valueList ')'!;
valueList: value (','! value)*;
value    : literal | ID
  | '-' literal -> ^('-' literal)
  | '<' valueList '>' -> ^(GROUP valueList)
  | '{' '}' -> ^(SET)
  | '{' valueList '}' -> ^(SET valueList)
  | '[' ']' -> ^(COMPLEXTYPE)
  | '[' valueList ']'->^(COMPLEXTYPE valueList);
literal  : booleanLiteral | integerLiteral
  | FLOAT | STRING;
```

Figure 3. Syntax Definition of The Data Recorded In Testing

The library consists of an interface and three classes as follows.

- `Instrumenter` interface provides four sets of methods for recording agent's behaviour during testing. Each set supports the behaviour observation at one level of testing. The following are the methods for caste level testing. The methods for other levels are omitted for the sake of space.

```
public void record_state(String agentId,
  String stateName, Object value);
public void record_cycle(int iteration);
public void record_action(String agentId,
  String actionName, Object...params);
public void record_function(String agentId,
  String funcName, Object value,
  Object...params);
```

- `InstrumentObject` class implements `Instrumenter`, but the virtual methods in the interface are left to be implemented by its subclasse.
- `FileInstrumenter` class inherits `InstrumentObject` and really implements the virtual methods defined in `Instrumenter`.
- `InstrumenterFactory` class is the factory class of `Instrumenter` for the creation of the records of the AUT. The main function of the class is the following method.

```
public static Map<String, Instrumenter>
  create(String FileName) throws IOException;
```

In summary, the data file is created and initialised by an `Instrumenter` object. It is in turn created by the factory class `InstrumenterFactory`.

B. Instrumentation Using the Runtime Facility

Given a specification of a caste C in SLABS, it can be implemented in Java or Java-based agent-oriented programming languages by a class in the structure shown in Fig. 4,

where the `Ai-stmt` statement in the body of the `Ai` method implements visible action A_i; similarly, `Bi-stmt` implements invisible action B_i; the `Body-stmt` in the method `cycle()` implements the behaviour rules. Each invocation of the `cycle()` method will search for one applicable behaviour rule and fire the rule if any.

```
import E1; ... ; import Ew;
public class C {
  ... /* infrastructure code */
public C(p1,..., pn)
  {Constructor for creating agents};
  public T1 v1;...; public Tm vm;
  private T'1 u1,...; private T'm' um';
  public void A1(p11,..., p1n1){A1-stmt}
    ...
  public void As(ps1,..., psns){As-stmt}
  private void B1(q11,..., q1n1){B1-stmt}
    ...
  private void Bs1(qt1,..., qtnt1){Bs1-stmt}
  public void cycle()
  { Body-stmt for behaviour rules}
};
```

Figure 4. OO Implementation of Caste

We use the runtime facility to instrument the implementation of the MAS as follows.

1) Setting up the global execution platform: In the main class of the system, import the `Instrumenter` interface and `InstrumenterFactory` class in order to link the runtime support library to the system under test. In the class of AUT, insert the public method `setInstrumenter` to setup the `Instrumenter` object and insert statements that invoke methods of the `InstrumenterFactory` class to create, initialise and finalise the `Instrumenter` object.

2) Setting up the local observation and recording platform: In the class of AUT, import the `Instrumenter` interface and `EmptyInstrumenter`, which is a stud class of `Instrumenter` interface, and declare a set of private variables for recording the state of test executions.

3) Instrumentation for recording AUT's behaviours: In the body of the constructor method of the class of AUT, insert statements that calls `record_state` methods to record the initial values of the agent's state variables. Insert a statement into the `cycle()` method before the `Body-stmt` for recording the the time point of observation. For each method that implements the visible action A_i and invisible action B_i, insert a statement that calls the `record_action` method into the body of the method after the `Ai-stmt` . For each visible state variable v_i and invisible variable u_i, insert a statement that calls the `record_state` method into the body of method `cycle()` after the `Body-stmt` to record the values of state variables.

4) Instrumentation for recording AUT's perception of its environment: For each visible state variable $v_{i,j}$ included in the scenarios of the behaviour rules, insert a statement that calls the `record_state` method into the body of method `cycle()` after the codes the agent perceives the state. For each action $A_{i,j}$ included in the scenarios of the behaviour rules, insert a statement that calls the `record_action` method in the body of `cycle()` after the codes the agent perceives the action.

```
ITERATION 0
turtle0 : state = FindingSalt
turtle0 : saltKind = 0
turtle0 : position = p1528
ITERATION 1
p1528 : caste = Point
p1528 : hasSalt = false
p1528 : saltKind = 0
p1528 : neighbour = {p1428, p1427, p1429, p1628, p1627, p1629,
p1527, p1529, p1528}
p1528 : onPile(0) = false
turtle0 : move
turtle0 : state = FindingSalt
turtle0 : saltKind = 0
turtle0 : position = p1627
ITERATION 2
p1627 : caste = Point
p1627 : hasSalt = false
p1627 : saltKind = 0
p1627 : neighbour = {p1527, p1526, p1528, p1727, p1726, p1728,
p1626, p1628, p1627}
p1627 : onPile(0) = false
turtle0 : move
turtle0 : state = FindingSalt
turtle0 : saltKind = 0
turtle0 : position = p1727
ITERATION 3
...
```

Figure 5. Example of Recorded Dynamic Behaviour: Turtle in Salt World

Fig. 5 gives a segment of the recorded dynamic behaviour of a `Turtle` with its view of the environment.

VI. Test Oracle

In this section, we discuss how to check the correctness of the behaviour of an AUT against a formal specification written in SLABS.

Existing work on specification based testing mostly treat the logic statements of specification as pre/post conditions on functional requirements. However, as mentioned above, a behaviour rule can be regarded as a probabilistic if-then statement. The condition in the *if* part should be considered as a guard-condition rather than a pre-condition. Checking the correctness of an agent's behaviour with regard to such a set of guard/post-condition pairs is significantly different from the pre/post-condition semantics. It is incorrect to treat such a rule as a pair pre/post-conditions. This is complicated by the non-determinism due to the possible overlaps between the guard conditions and the probabilistic feature of the rules. Therefore, we distinguish two types of correctness: functional correctness and probabilistic accuracy.

A. Functional Correctness

The guard-condition and the post-condition of a rule can be written in the form of $A_1 \wedge \cdots \wedge A_n$, where A_i $(i = 1, \cdots, n)$ is either an assertion of the agent's state or an assertion of the occurrence of an action. For example, for the `move` behaviour rule in Fig. 2, the guard-condition is:

$$state = FindingSalt \wedge position = p \wedge saltKind = 0 \wedge$$
$$p : [\ !saltKind = sk \wedge neighbour = neigh \wedge$$
$$(hasSalt = false \vee onPile(sk) = true)\ ],$$

and the post-condition is:

$$(move = true \wedge position = p_1 \wedge$$
$$state = FindingSalt \wedge saltKind = 0 \wedge p_1 \in neigh).$$

At any given time point in a test execution, a behaviour rule evaluates to true if both the guard-condition and the post-condition are true, or both of them are false. However, if the guard-condition and post-condition have different values, it does not necessarily imply that the rule is violated, because several behaviour rules may have overlapped guard-conditions. The behaviour rules are violated only in the following two situations.

- *Commission error*: If a post-condition is true at time moment $t+1$, but all of its corresponding guard-conditions are false at time moment t, this means an action is committed when it is not supposed to. Thus, there is a commission error.
- *Omission error*: If for the set of guard-conditions that are true at time moment t, none of its corresponding post-conditions is true at time moment $t+1$, this means an action is missed. Thus, there is an omission error.

For example, consider the Salt World system given in Fig. 2. Assume that when a `turtle` is in a state that $state = Findingsalt, position = p1528$ and $saltKind = 0$, and its environment has the condition $p.saltKind = 0, p.hasSalt = false$ and $p.onPile(0) = false$, it takes an action `move`, and changes the value of position to Point $p1627$, which is in adjacent to Point $p1528$, and the values of $state$ and $saltKind$ remain unchanged. The behaviour of the `turtle` at this time point is correct according to the behaviour rules in the specification. In particular, both of the guard-condition and post-condition of the `move` behaviour rule are true. For the `GetAway` behaviour rule, the evaluation of the guard-condition results in false, but the post condition is true. This is because its post-condition has overlap with the rule `move`. So, it does not imply that the behaviour is incorrect with respect to the rule `GetAway`. Its behaviour is correct with respect to all the other rules because for each of them, both the guard-condition and the post-condition are evaluated to false, which means, when the guard condition is not satisfied, the action is not taken by the agent.

Note that, sometimes, a predicate cannot be evaluated because a state variable in the rule has no value. In such cases, we define the value of the assertion to be false.

B. Probabilistic Accuracy

Assume that the AUT is tested on a number of test cases and a set of behaviours are recorded. We now evaluate the correctness of the AUT in terms of the accuracy of the probabilistic distributions with respect to the probability specified in the behaviour rules.

Suppose that we have a behaviour rule whose probability is r. Let a be the number of time points in the set of recorded behaviours at which the rule's guard-condition P is true and b be the number of time points at which both guard-condition g and the post-condition p are true. The behaviour rule is satisfied w.r.t. its probability specification r with a tolerable deviation δ, if $|r - b/a| \leq \delta$. The implementation of the AUT is probabilistically accurate w.r.t. the specification if it satisfies all the behaviour rules w.r.t. to the probability specifications with a tolerable deviation δ set by the user.

VII. Test Adequacy Criteria

This section is devoted to the test adequacy problem. We propose a set of coverage criteria to measure test adequacy according to the formal specifications in SLABS.

For the sake of convenience, in the sequel, we call a recorded dynamic behaviour during one test execution of the MAS a *test case*. In other words, a test case is a sequence of records of state changes and actions. We write $p_u(t)$, if predicate p is true at time point u on test case t. We write $p(t)$ if $p_u(t)$ for some time point u.

Let P be a set of predicates. Let T be a set of test cases. We say that T covers all predicates in P if for each $p \in P$, there exist at least one test case t in T such that $p(t)$ is true, if p is feasible, and there exist at least one test case t' in T such that $p(t')$ is false, if $\neg p$ is feasible.

Let p and q be the guard and post-condition of a behaviour rule R in Sp. We write $Cov(R)$ to denote the set $\{p \mapsto q, p \mapsto \neg q, \neg p \mapsto q, \neg p \mapsto \neg q\}$ of predicates, where each predicate represents one of the possible situations that the rule is evaluated during execution. In particular,

- $p \mapsto q$ means that the rule is applied;
- $p \mapsto \neg q$ means the rule is applicable but not applied;
- $\neg p \mapsto q$ means that the rule is not applicable, but the post-condition is satisfied for some reasons;
- $\neg p \mapsto \neg q$ means that the rule is not applicable and it is not applied.

We write $Guard(Sp)$ and $Post(Sp)$ to denote the sets of predicates that are the guard-conditions and post-conditions of the behaviour rules in Sp, respectively. We also write $Cov(Sp)$ for $Cov(Sp) = \bigcup_{R \in Sp} Cov(R)$.

Definition 1: (Rule Coverage)

A test set T is adequate in testing against specification Sp according to the *rule coverage criterion*, if T covers $Cov(Sp)$. □

The rule coverage criterion requires that testing exercises all of the four possibilities that a rule can be evaluated if it is feasible. This requirement on test adequacy is stronger than the coverage of the set of predicates $\{Guard(R) \mapsto Post(R) | R \in Sp\}$ in the traditional rule coverage, which is called *loose rule coverage* here.

Definition 2: (Condition Coverage)

A test set T is adequate in testing against specification Sp according to the *guard-condition coverage criterion*, if T covers the predicate in $Guard(Sp)$. It is adequate according to *post-condition coverage criterion* if T covers the predicate in $Post(Sp)$. □

Let P be any given set of predicates. $Clause(P)$ denotes the set of clauses contained in P. A clause is a predicate that does not contain any logic connectors. It can be either a state assertion or an action assertion.

Definition 3: (Clause Coverage)

A test set T is adequate in testing against specification Sp according to the *guard-condition clause coverage*, or *post-condition clause coverage*, or *rule clause coverage criterion*, if T covers $Clause(Guard(Sp))$, $Clause(Post(Sp))$, or $Clause(Cov(Sp))$, respectively. □

Let $Clause(p) = \{c_1, \cdots, c_n\}$ be the set of clauses in a predicate p. A combination b of the clauses in p is an expression in the form of $c'_1 \wedge \cdots \wedge c'_n$, where c'_i is either c_i or $\neg c_i$. In the sequel, we write $Comb(p)$ to denote the set of all combinations of the clauses of a predicate p, and also write $Comb(P)$ to denote the set $\bigcup_{p \in P} Comb(p)$ for a set P of predicates.

Definition 4: (Clause Combination Coverage)

A test set T is adequate in testing against specification Sp according to the *guard-condition clause combination coverage*, or *post-condition clause combination*, or *rule clause combination coverage criterion*, if T covers $Comb(Pre(Sp))$, $Comb(Post(Sp))$, or $Comb(Cov(Sp))$, respectively. □

The above adequacy criteria have the subsumption relations shown in Fig. 6. They have been implemented in CATest.

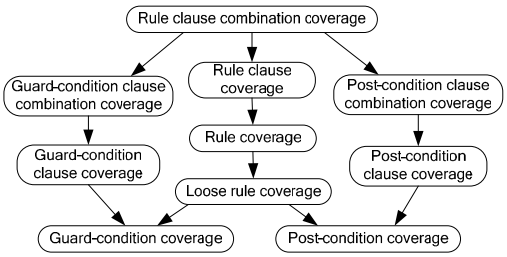

Figure 6. Subsumption Relations between Adequacy Criteria

VIII. The Tool CATest

We have developed an automatic testing tool, called CATest, implemented as an Eclipse plug-in, to support the testing process. As shown in Fig. 7, the tool consists of the following components.

- *Execution platform*: The instrumented program is executed on this platform with the support of a runtime library to record the dynamic behaviour of AUT.

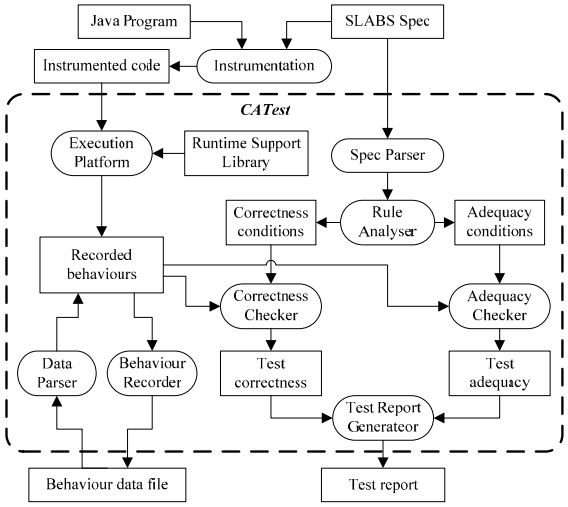

Figure 7. Architecture of CATest

- *Behaviour Recorder*: It enables recorded dynamic behaviours to be saved into a data file so that incremental testing results can be cumulated.
- *Data parser*: It enables previously saved dynamic behaviour records to be parsed and loaded into the system for the analysis of correctness and test adequacy.
- *Spec parser*: It parses specifications in SLABS and checks the syntax correctness of the specification.
- *Rule analyser*: It analyses the semantics of the rules in the specification and generates two sets of predicates: one for checking the correctness of observed behaviours and the other for calculating the test adequacy.
- *Correctness Checker*: It checks the correctness of the recorded dynamic behaviours according to the behaviour rules represented in the form of a set of predicates generated by the rule analyser.
- *Adequacy calculator*: It measures the test adequacy by evaluating the user selected adequacy criteria represented in the form of a set of predicates on the recorded dynamic behaviour.
- *Test report generator*: It generates a test report based on the results of correctness checker and adequacy checker and display it in GUI; see Fig. 9.

Fig. 8 shows the user interface of the CATest tool. It enables the user to set various parameters of the testing environment such as to select the adequacy criteria.

IX. Experiment

In order to investigate the fault detecting ability of the testing method and the adequacy criteria, we conducted an experiment using mutation testing [20].

In the experiment, we used two MAS that are reported in the literature on agent-oriented software development

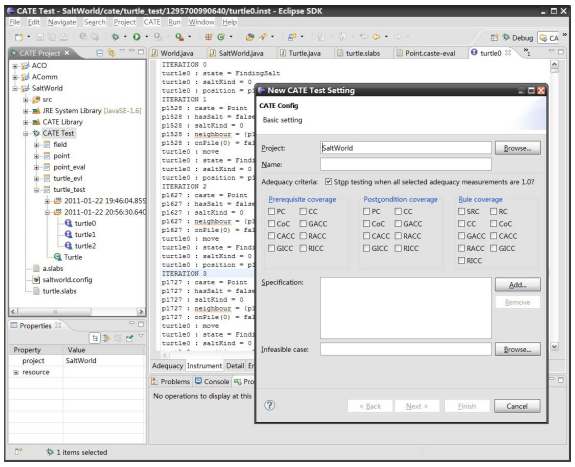

Figure 8. GUI of CATest: Setting Test Parameters

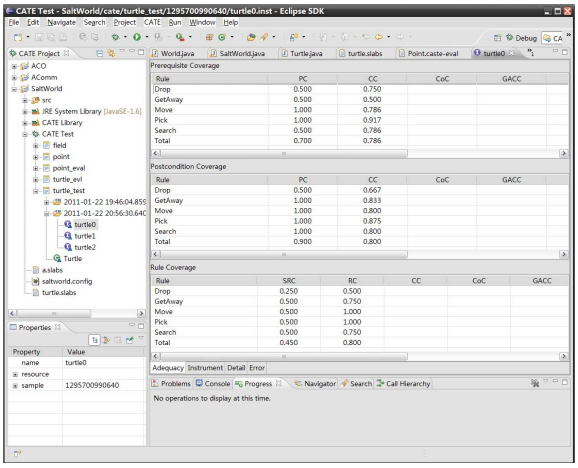

Figure 9. GUI of CATest: Displaying Test Results

methodologies. They are Salt World and Ant Colony [4]. Table I shows the data about their implementations in Java.

In the experiment, we tested `Turtle` and `Point` in the *Salt World*, and `Ant` in the *Ant Colony* as the AUT. For each of them, the experiment followed the process below.

1) *Generation of mutants*. The muJava testing tool [20] is used to generate mutants of the Java class that implements the caste under test.
2) *Analysis of mutants*. Each mutant is compiled and those contain syntax errors are deleted. Those equivalent to the original are also removed.
3) *Test on mutants*. The original class is replaced by the mutants one by one and tested using our tool. The test cases were generated at random. The test executions stop when the Rule Coverage Criterion is satisfied, or the execution stops abnormally when an interrupting exception occurs.
4) *Analysis of Data*. A mutant is regarded as killed if an error is detected, i.e. when the specification is violated. Otherwise, the mutant is regarded as alive. Note that, this is different from traditional definition of dead mutants, which does not work because the non-deterministic nature of the system.

The results are showed in Table II.

Table I
THE SUBJECTS OF THE EXPERIMENT

Class	#Methods	#Attributes	#Lines
Salt World			
Turtle	9	6	133
StateType	2	1	17
Point	19	7	150
World	14	12	221
SaltWorld GUI	7	19	339
GraphCanvas	2	7	58
Subtotal	43	62	918
Ant Colony			
Ant	8	11	233
ModeName	2	1	16
Node	13	7	95
Arc	8	3	52
Field	15	11	255
ACOPanel	10	20	395
ArcIcon	6	5	87
NodeIcon	9	5	101
Visualizer	15	15	283
Subtotal	86	78	1517

Table II
RESULTS OF THE EXPERIMENT

	Salt World	Ant Colony
#All mutants	41	305
#Live mutants	5	53
#Dead mutants	30	176
#Abnormal mutants	6	76
# Total valid mutants	35	229
Fault detecting rate	85.7%	76.9%

From the mutants that remain alive in the experiments, we identified the following kinds of mutants that caste level testing are not effective to kill.

- Mutants that change the code that initializes the AUT's states;
- Mutants that change the code that sends/receives messages to/from the others agents;
- Mutants that change the code inside the functions/ methods of actions;
- Mutants that change the infrastructure code.

These mutants correspond to faults that are either at a higher or a lower level than caste level. Thus, testing at other levels are necessary. However, mutants that represent faults at the caste level, such as in the behaviour rules, are detected 100% in our experiments using the rule coverage criterion. This implies that our specially designed adequacy criteria are highly effective. The more stregent adequacy criteria are only needed when the behaviour logic is highly complex.

X. Conclusion

The main contributions of the paper are as follows.

(A) *A novel architecture of TAF.*

This paper proposes a novel architecture of TAFs and presents an automated test tool CATest for testing MAS at caste level. In comparison with the existing TAFs, it has the following three distinctive features.

1) It automatically checks the correctness of software dynamic behaviours against formal specifications without the need to manually write assertion methods.
2) It fully supports automatic measurement of test adequacy and use the adequacy measurement to control test executions.
3) Test cases are not hard coded into the test code, and therefore, test case generation tools can be easily integrated to the framework.

This architecture overcomes the weaknesses of existing TAFs by providing a higher degree of automation and supporting a wider range of test activities. It applies to all levels of testing. In fact, we have developed a test environment called CATE-Test that supports all levels of agent test. CATest is a part of the complete test framework CATE-Test. Moreover, it can also be easily adapted for testing traditional OO software. Of course, the architecture manifests its benefits more clearly in testing MAS because of the complexity of agent-oriented systems.

(B) *A new hierarchy of test adequacy criteria.*

We have also proposed a new hierarchy of adequacy criteria for specification-based testing and implemented them in the CATest tool. In comparison with existing works, the unique features of these criteria include:

1) They treat guard-conditions differently from pre/post-conditions, thus reflect better the semantics of guard-conditions in testing.
2) They take full consideration of non-determinism of the system in the definition of the criteria.

It is also worth noting that the hierarchy of adequacy criteria is not only applicable to MAS, but also to all systems that are running continuously, non-deterministically and event-driven. In addition to MAS, other typical examples of such systems include distributed and service-oriented systems. They are normally specified by a set of behaviour rules with guard-conditions.

We are currently doing more experiments with the system to evaluate its fault detecting ability, usability and scalability.

For future work, it is important to develop techniques that automatically insert instrumentation code, to control the execution paths in a non-deterministic program, and to integrate a test case generation tool into the framework.

Acknowledgement

The work reported in this paper is partially funded by China Ministry of Science and Technology in the 863 Programme under grant 2005AA113130 and in the 973 Programme under grant 2005CB321800, and China Natural Science Foundation under grant 90612009.

References

[1] *Proceedings of The 6th IEEE/ACM International Workshop on Automation of Software Test (AST 2011)*. Waikiki, Hawaii, USA: IEEE CS Press, May 2011.
[2] N. R. Jennings, "On agent-based software engineering," *Artificial Intelligence*, vol. 117, pp. 277–296, 2000.
[3] J. Voas and K. Miller, "Software testability: the new verification," *IEEE Software*, vol. 12, no. 3, pp. 17–28, 1995.
[4] H. Zhu, "SLABS: A formal specification language for agent-based systems," *Int'l J. SEKE*, vol. 11, no. 5, pp. 529–558, 2001.
[5] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.
[6] C. Rouff, "Testing and monitoring intelligent agents," in *Proc. of WRAC'03*. LNCS 2564. Springer-Verlag, pp. 155–164.
[7] R. Coelho, E. Cirilo, U. Kulesza, A. von Staa, A. Rashid, and C. Lucena, "Jat: A test automation framework for multi-agent systems," in *Proc. of ICSM'07*, 2007, pp. 425–434.
[8] D. Poutakidis, L. Padgham, and M. Winikoff, "An exploration of bugs and debugging in multi-agent systems," in *Proc. of ISMIS'03*, 2003, pp. 628–632.
[9] M. Liedekerke and N. Avouris, "Debugging multi-agent systems," *Information and Software Technology*, vol. 37, no. 2, pp. 103–112, 1995.
[10] H. Nwana, D. Ndumu, L. Lee, and J. Collis, "ZEUS: a toolkit and approach for building distributed multi-agent systems," in *Proc. of Agents'99*, 1999, pp. 360–361.
[11] E. Ekinci, A. Tiryaki, O. Cetin, and O. Dikenelli, "Goal-oriented agent testing revisited," in *Proc. of AOSE'08*, 2008, pp. 85–96.
[12] C. Nguyen, A. Perini, and P. Tonella, "A goal-oriented software testing methodology," in *Proc. of AOSE'07*. LNCS 4951. Springer-Verlag, 2007, pp. 58–72.
[13] Z. Zhang, J. Thangarajah, and L. Padgham, "Automated unit testing for agent systems," in *Proc. of ENASE'07*, 2007, pp. 10–18.
[14] C. Low, T. Y. Chen, and R. Ronnquist, "Automated test case generation for BDI agents," *Autonomous Agent and Multi-Agent Systems*, vol. 2, no. 4, pp. 311–332, 1999.
[15] C. Nguyen, A. Perini, and P. Tonella, "eCAT: a tool for automating test cases generation and execution in testing multi-agent systems," in *Proc. of AAMAS'08*, 2008, pp. 1669–1670.
[16] G. Caire, M. Cossentino, A. Negri, A. Poggi, and P. Turci, "Multi-agent systems implementation and testing," in *Proc. of the 4th Int'l Symposium: From Agent Theory to Agent Implementation*, Vienna, 2004, pp. 14–16.
[17] A. Tiryaki, S. Oztuna, O. Dikenelli, and R. Erdur, "SUNIT: A unit testing framework for test driven development of multi-agent systems," in *Proc. of AOSE'06*. LNCS 4405. Springer-Verlag, 2006, pp. 156–173.
[18] J. Gomez-Sanz, J. Bota, E.Serrano, and J. Pavon, "Testing and debugging of MAS interactions with INGENIAS," in *Proc. of AOSE'08*. LNCS 5386. Springer-Verlag, 2008, pp.199–212.
[19] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, May 2007.
[20] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.