# eCAT: a Tool for Automating Test Cases Generation and Execution in Testing Multi-Agent Systems

## (Demo Paper)

Cu D. Nguyen, Anna Perini and Paolo Tonella
Fondazione Bruno Kessler
Via Sommarive, 18
38050 Trento, Italy
{cunduy, perini, tonella}@fbk.eu

## ABSTRACT

We introduce *eCAT*, a tool that supports deriving test cases semi-automatically from goal-based analysis diagrams, generates meaningful test inputs based on agent interaction ontology, and more importantly it can evolve and execute test cases automatically and continuously on a multi-agent system (MAS). Our experiments have shown that the proposed tool can exercise MAS more extensively and effectively than manual testing under the usual time constraints.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Verification

## Keywords

Testing tools, multi-agent systems

## 1. ECAT

Agents have been recognized as a promising technology to build next generation services. They appear in mobile phones or personal digital assistant equipments, helping their owners manage complicated work; agents facilitate e-learning, decentralized enterprise management. In the development process of such complex and critical systems, testing becomes crucial in order to ensure a satisfactory level of quality.

We propose an agent testing framework, called *eCAT* (http://sra.fbk.eu/people/cunduy/ecat) with following features:

- Generate test case skeletons from goal analysis diagrams produced using TAOM4E (http://sra.fbk.eu/tools-/taom4e). These skeletons can adopt agent interaction protocols and are ready to be completed with specific test inputs. Details are introduced in [6].
- Provides GUIs to help human testers specify test inputs easily.
- Generate test inputs based on agent interaction ontology: given an agent system and agents in such system communicate using an interaction ontology, *eCAT* can make use of the ontology in order to generate test inputs [7].
- Evolve and generate more test inputs by evolutionary-mutation or random testing techniques, and run these test inputs continuously to extensively test MAS [5].

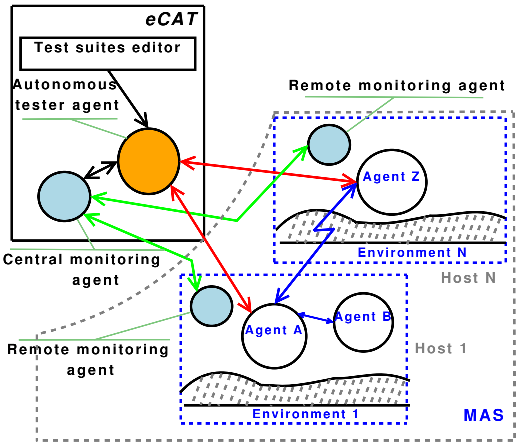

**Figure 1: eCAT framework**

Figure. 1 depicts a high level architecture of *eCAT* that consists of three main components: *Test Suite Editor*, allowing human testers to derive test cases from goal analysis diagrams; *Autonomous Tester Agent*, capable to automatically generate new test cases and to execute them on a MAS; and *Monitoring Agents*, that monitor communication among agents, including the *Autonomous Tester Agent*, and all events happening in the execution environments in order to trace and report errors. *Remote monitoring agents* are deployed with the environments of the MAS under test, transparently to the MAS, in order to avoid possible side effects. All the *remote monitoring agents* are under the control of the *Central monitoring agent*, which is located at the same host as the *Autonomous Tester Agent*. The monitoring agents overhear agent interactions, events, and constraint violations taking place in the environments, providing a global view of what is going on during testing and helping the *Autonomous Tester Agent* evaluate test results.

**Cite as:** eCAT: a Tool for Automating Test Cases Generation and Execution in Testing Multi-Agent Systems (Demo Paper), C. D. Nguyen, A. Perini and P. Tonella, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp.1669-1670.

## 2. TEST CASES GENERATION IN ECAT

Four test cases generation techniques are equipped to *eCAT*: *Goal-oriented*, *Ontology-based*, *Random*, and *Evolutionary mutation* (also called evol-mutation).

**GOAL-ORIENTED**. Goal-oriented test cases generation is a part of a methodology presented in [6] that integrates testing into *Tropos*, providing a systematic way of deriving test cases from *Tropos* output artifacts. *eCAT* can take these artifacts as inputs to generate test case skeletons that are aimed at testing goal fulfillment. Specific test inputs (i.e. message content), and expected outcome are partially generated from plan design (e.g. UML activity or sequence diagrams) and are then completed manually by testers.

**ONTOLOGY-BASED**. Agent behaviors are often influenced by messages received. Hence, at the core of test case generation is the ability to build meaningful messages that exercise the agent under test so as to cover most of the possible running conditions. *eCAT* can take advantage of agent interaction ontologies, which define the semantics of agent interactions, in order to automatically generate both valid and invalid test inputs, to provide guidance in the exploration of the input space, and to obtain a test oracle against which to validate the test outputs [7].

**RANDOM**. *eCAT* is capable of generating random test cases, following the random test data generation strategy [4]. First, the *Autonomous Tester Agent* selects a communication protocol among those provided by the agents platform, e.g. FIPA Interaction Protocol [2]. Then, messages are randomly generated and sent to the agents under test. The message format is that prescribed by the agent environment of choice (such as the FIPA ACLMessage [3]), 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 (simpler) form of sets of admissible data that are sampled randomly.

**EVOL-MUTATION**. This technique combines mutation [1] and evolutionary [9] testing for the automated generation of the test cases executed by the tester agent in a given multi-agent environment. Intuitively, we use the mutation adequacy score as a fitness measure to guide evolution, under the hypothesis that test cases that are better at killing mutants are also likely to be better at revealing real faults. The proposed technique consists of the following three steps:

**Step 0: Preparation**, given the MAS under test $M$, we apply mutation operators to $M$ to produce a set of mutants $\{M_1, M_2, \ldots, M_n\}$. One or more operators are applied to one or more (randomly chosen) agents in $M$.

**Step 1: Test execution and adequacy measurement**, the *Autonomous Tester Agent* executes the test cases $\{TC_1,\ TC_2, \ldots, TC_n\}$ on all the mutants. Initially, test cases are those derived from goal analysis by the user. The *Autonomous Tester Agent* then computes the adequacy of each test case (fitness value): $F(TC_i) = \frac{K_i}{N}$, where $K_i$ is the number of mutants killed by $TC_i$. To increase performance, the executions of the test cases on the mutants are performed in parallel (e.g., on a cluster of computers, with one mutant per node).

**Step 2: Test case evolution**, the procedure for generating new test cases is described as follows.

```
Select randomly whether to apply mutation or crossover
if Crossover is chosen then
    Select 2 test cases (i, j) with probability $F(TC_i), F(TC_j)$
    Apply crossover on $TC_i$ and $TC_j$
else
    Select a test case with probability of selection $F(TC_i)$
    Apply mutation
end if
Add the new test cases to the new set of test cases
```

The basic mechanisms used to evolve a given test case are mutation and crossover. *Mutation* consists of a random change of the data used in the messages exchanged in a test case, similarly to the random generation described above. *Crossover* consists of the combination of two test cases. Two good test cases are chosen, some data in the second test case replace the data used in the first one.

The algorithm stops when the number of generation exceeds a given maximum number of generation. Otherwise, we go back to Step 1 and keep on testing continuously [5]. When no improvement of the fitness values is observed for a number of evolutionary iterations, Step 0 (Preparation) is repeated.

*eCAT*'s performance and capability to reveal faults have been evaluated on two BDI agent case studies [8]. *eCAT* has been implemented as an Eclipse[1] plug-in. It supports testing agents implemented in JADE and JADEX, and the input ontology formats are those supported by Protégé[2] like OWL.

## 3. REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[2] FIPA. Interaction protocols specifications. http://www.fipa.org/repository/ips.php3, 2000-2002.

[3] FIPA. ACL Message Structure Specification. http://www.fipa.org/specs/fipa00061, 2002.

[4] H. D. Mills, M. D. Dyer, and R. C. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987.

[5] C. D. Nguyen, A. Perini, and P. Tonella. Automated continuous testing of multi-agent systems. In *The fifth European Workshop on Multi-Agent Systems*, December 2007.

[6] C. D. Nguyen, A. Perini, and P. Tonella. A goal-oriented software testing methodology. In *8th International Workshop on Agent-Oriented Software Engineering, AAMAS*, volume LNCS 4951, May 2007.

[7] C. D. Nguyen, A. Perini, and P. Tonella. Ontology-based Test Generation for Multi Agent Systems. In *Proc. of the International Conference on Autonomous Agents and Multiagent Systems*, 2008.

[8] C. D. Nguyen, A. Perini, and P. Tonella. Ontology-based test generation for multi agent systems. definition and evaluation. Technical Report FBK-IRST0108, FBK, 2008.

[9] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verifications, and Reliability*, 9:263–282, September 1999.

---

[1] `http://www.eclipse.org`

[2] `http://protege.stanford.edu`