A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems

Zoltán Micskei, Zoltán Szatmári, János Oláh, and István Majzik

Budapest University of Technology and Economics, Budapest, Hungary
{micskeiz,szatmari,olahj,majzik}@mit.bme.hu

Abstract. Autonomous systems are used nowadays in more and more sectors from vehicles to domestic robots. They can make decisions on their own or interact with humans, thus their robustness and safety are properties of crucial importance. Due to the adaptive and context-aware nature of these systems, the testing of such properties is especially challenging. In this paper, we propose a model-based testing approach to capture the context and requirements of such systems, to automatically generate test data representing complex situations, and to evaluate test traces and compute test coverage metrics.

Keywords: testing, autonomous systems, context modelling, test coverage

1 Introduction

An autonomous system (AS) can be defined as one that makes and executes decisions to achieve a goal without full, direct human control [1]. Notable characteristics shared by the different kinds of autonomous systems include reasoning, learning, adaptation and context-awareness. In many scenarios, several autonomous systems are working together to reach a common goal, thus communication and cooperation are also important, and the interacting partners form part of the context of an individual AS.

A typical example of an AS is an autonomous robot, which is working in a real, uncontrolled environment, possibly in the presence of humans. Even if the task of a robot is relatively simple, e.g., to pick up garbage from the ground, it should be able to differentiate and recognize numerous types of objects and be prepared to take into account the unexpected movements of humans. Thus, it shall be *robust* in order to be capable of handling unforeseen situations and *safe* to avoid harmful effects with respect to humans. The prerequisite of the application of autonomous systems is a thorough verification of these requirements.

In the R3-COP project[1] our work is focused on testing the context-aware behaviour of autonomous robots, especially the robustness and functional safety of their behaviour. Precisely speaking, robustness is an attribute of dependability, which measures the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. Thus, the goal of robustness

[1] Resilient Reasoning Robotic Co-operating Systems, http://www.r3-cop.eu/

Final version appeared in: KES-AMSTA 2012, LNAI 7327, Springer, pp. 504–513, 2012. DOI: 10.1007/978-3-642-30947-2_55. The final publication is available at link.springer.com

testing is the generation of these situations to detect the potential design faults that result in incorrect operation. Functional safety is defined as freedom from unacceptable occurrence of harm (characterized with a given probability and severity), which depends on the correct functioning of the system. Testing functional safety is performed by executing the system in a controlled way to demonstrate the absence of unsafe behaviour. In our work we restrict the focus of testing robustness and functional safety to the design faults affecting context-aware behaviour and do not deal with testing the effects of random operational faults.

Testing the robustness and safety in autonomous systems is especially challenging and it requires the development of new methods due to the following characteristics. First, the behaviour is *highly context-aware*: the actual behaviour of an AS depends not only on the events it receives, but also on the perceived state of the environment (that is typically stored as a context model in the AS). Second, the context is complex and there are a *large number of possible situations*: in real physical world the number and types of potential context objects, attributes and interactions that need to be specified can be large. Third, *adaptation to evolving context* is required: as most of the autonomous systems contain some kind of learning and reasoning capabilities, their behaviour can change in time based on feedback from the evolving environment.

These characteristics have also consequences on the specification of the requirements to be tested. Full behaviour specification can be impractical due to the complexity of the behaviour and the diversity of the system environments, and requirements should include the evolution of the environment. It means that the developed test cases should not only include the input signals and messages to the system under test (SUT), but they should contain a sufficiently detailed description about the environment of the system.

As stated in a recent paper [6], testing autonomous systems is still an unsolved key area. In the project, considering our goals, we identified the following insufficiencies of existing testing approaches and proposed the following solutions:

- *Lack of easy-to-use mechanisms to express and formalize context-aware behaviour* (although these mechanisms are a prerequisite of requirements-based automated test generation and test evaluation). Most noticeably, in existing standard test description languages there is no support to express changes in the context [2]. To overcome this problem of capturing test requirements, we defined a language which is based on *context models* and *scenario based behaviour specification*. The context model allows systematically capturing domain knowledge about the context of the system. Hierarchy in the modelling can efficiently support handling of the types of diverse objects in the environment. Context models may also represent the existence of cooperating partners and humans, as well as the explicit messages and commands from them. With respect to the dynamics of the system behaviour, the scenario based language provides a lightweight formalism (that is close to the engineers' way of thinking) to capture the behaviour (actions and messages) of a SUT in case of a test context. The application of modalities in the scenario language allows expressing prohibited behaviour (that would violate safety) and potentially allowed behaviour (this way capturing learning and adaptive behaviour).

- *Ad-hoc testing of stressful conditions and extreme situations*: Previous research focused first of all on producing high fidelity simulators [4] or executing excessive field testing [5] for the verification of AS. There exist methods for testing the physical aspects; however, not all behavioural aspects are well-covered [3]. Our proposed solution is based on context modelling: we included in the context models the constraints and conditions that determine the normal and exceptional situations and defined methods to *systematically generate* stressful test contexts by violating the constraints, reaching boundary conditions, and combining contexts from various requirements (to test the implementation in case of interleaving scenarios).
- *Lack of precise and objective test coverage metrics* that can characterize the thoroughness of the testing process. On the basis of context models we defined precise coverage metrics, especially *robustness related metrics* that refer to constraints and conditions (to be violated), and combinations of context elements and fragments. Different coverage criteria were used to define the test goals and search-based techniques were applied to generate the required complex test suite. The formalization of robustness and safety related requirements allows an automated evaluation of test executions (especially in a simulated environment) and thus deriving concrete success and coverage measures that characterize the testing process and help identifying weak points in testing.

The rest of the paper presents our model based testing framework that is being finalized and validated in the R3-COP project. Section 2 specifies the testing approach (the goals and the components of the framework). Section 3 presents how the test requirements are captured, i.e., the context and scenario modelling. Section 4 outlines the test generation approach, while Section 5 presents the test evaluation and the main categories of the applied coverage metrics. The detailed presentation of all technical details will be publicly available in an R3-COP project deliverable (D4.2.1).

2 Testing Approach

Our work focuses on testing the control software of an individual AS that determines how the AS reacts (i.e., what actions it sends to the actuators) to the events from the perception or the communication components representing the (changes in the) context of the AS. The control software is treated as a black-box and the testing is concentrated on the system-level behaviour: detecting misbehaviour with respect to safety and robustness properties in diverse situations. Although the scope of testing is the behaviour of an individual AS, cooperation among systems can also be represented in the requirements, since input messages and input events (with respect to the movements of other systems and humans) as well as output messages are part of the context model. However, the overall goal of a cooperative behaviour is not tested. As safety and robustness failures manifest when several typical situations combine in an unexpected manner (e.g., an autonomous vehicle receives a new direction just when it detects that a pedestrian crossing and another vehicle approaching from the next lane), our approach focuses on covering such problematic complex (combined) situations.

Testing is carried out in an interactive simulator environment, where different test situations can be prepared and the system under test can react to the changes in the context as obtained through its sensors.

Fig. 1 gives a high-level overview of the testing process. Starting from the specification of the AS and the knowledge of the application domain, the process evaluates the safety and robustness of the control software and computes coverage metrics. The key components (that are novel in our approach) are the context and scenario modelling, the search-based test data generation on the basis of robustness related coverage criteria, and methods for automated test evaluation. Related new tools are the test data generator and the test oracle generator.

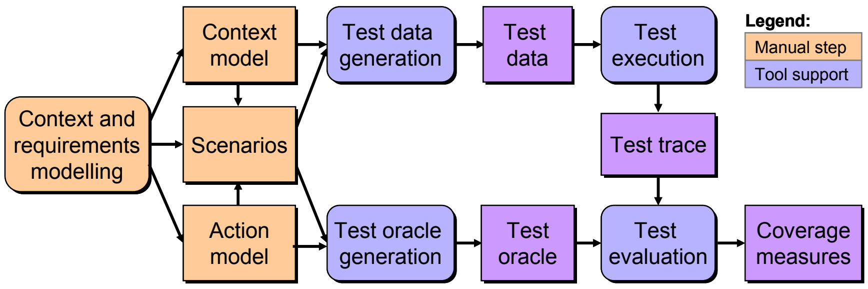

Fig. 1. Overview of the proposed approach

The rest of the paper walks through the steps of this testing process to show how the defined testing goals can be achieved by creating a context model, capturing requirements as scenarios, generating test data, executing the tests while recording test traces, and finally evaluating test traces with respect to the requirements.

To illustrate the developed testing approach a home *vacuum cleaner robot* is used as a running example. The robot is able to discover its surroundings when placed in a new room, create a map of its environment, and clean the room. The robot should avoid collision with living beings and should perform some limited surveillance tasks (e.g., detect unusual noises). Note that the purpose of the example is only to illustrate the testing concepts, and it is not intended to be a full system and test specification.

3 Formalizing application requirements

Usually application requirements are available as natural language specifications, as use case descriptions or as some text structured in tables. In order to use such requirements in test evaluation or in automated test data generation, a more structured format is needed. In general, to implement the testing, the formalization of at least two artefacts is required.

1. *Test data*. The input to the system under test should be specified. As described previously, in case of autonomous systems this should include (i) the initial state of the context of the system, (ii) its evolution in time, and (iii) the messages and

commands received by the SUT. These concepts are captured in a *context model*. Outputs of the SUT are included in a separate *action model*.

2. *Test oracle*. The responsibility of the test oracle is to evaluate the test outcome, the actions and output messages of the system. As specifying the exact outcome of every situation could be infeasible, a lightweight approach is used. The requirements are expressed as *scenarios* and are checked for every executed test (to detect potential safety and robustness failures).

3.1 Context modelling

In order to have a structured way of describing the test data, first a *context model* is created. It consists of two parts. The *static part* represents the environment objects and their attributes in a type hierarchy (in the vacuum cleaner example it includes concepts like room, furniture inside a room, humans or animals). The *dynamic part* contains events as distinguished elements to represent changes with regard to objects (i.e., an object appears, disappears) and their relations and properties (e.g., a relation is formed or a property is transformed). The events have attributes and specific relations to the static objects depending on the type of the event.

Several modelling languages exist to express such models; see e.g. [7] for context modelling approaches. An easy to use approach to capture domain concepts is the use of *ontologies*. Existing ontologies related to robots (like KnowRob [8]) can be reused when defining the context model. To ease the programmatic manipulation of these context models (which is needed when test data is generated), ontology models are transformed systematically to *metamodels* (see e.g. Fig. 2 in case of the vacuum cleaner robot) and *instance models* conforming to this metamodel. The metamodels are extended with domain-specific constraints that for example require specific attribute values or define restrictions with respect to the number of objects in a model.

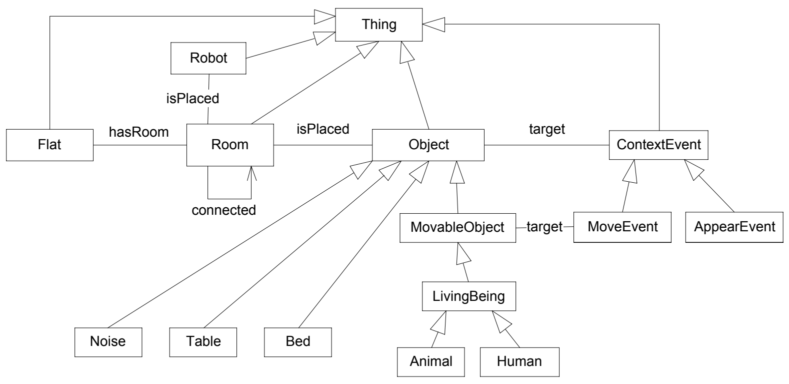

Fig. 2. Context metamodel for the vacuum cleaner robot

The structured context metamodel offers several advantages over an ad-hoc representation of context elements in the different requirements. It supports the automated generation of test data, including the completion and systematic combination of context model elements from different requirements. The domain-specific constraints allow the automated generation of extreme contexts (as test data) that violate these constraints. Moreover, it is also the basis of the definition of precise test coverage metrics. In our framework we construct an OWL2 based domain ontology then map it automatically to metamodels to be manipulated in the Eclipse Modeling Framework[2].

3.2 Scenario modelling

For test oracles, requirements are expressed as graphical scenarios in the form of extended UML 2 Sequence Diagrams that represent events/messages received and actions/messages sent by the SUT. Each diagram has two parts: (1) a trigger part (which means that the scenario is only relevant if its trigger part has been successfully traversed) that may have several fragments, like *opt* fragment for expressing optional behaviour, *alt* for expressing alternatives, *neg* for a negative fragment that should not happen and (2) an assert part that consist of an *assert* segment (that shall happen).

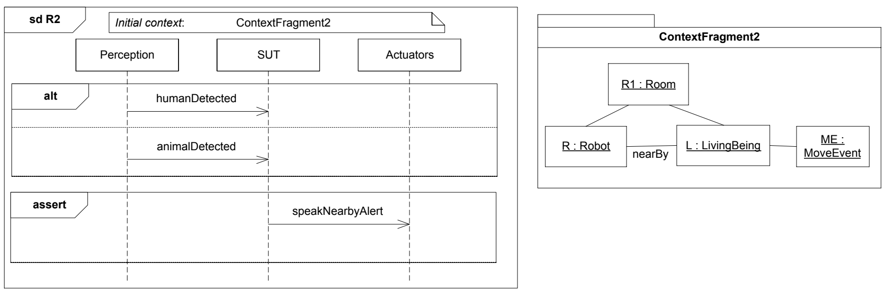

Fig. 3. Example scenario model R2: Alerting a living being

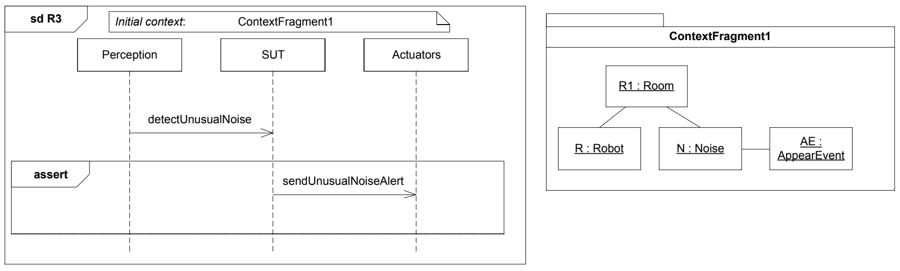

Fig. 4. Example scenario model R3: Detecting unusual noise

Language extensions were added to refer to (changing) contexts. Using these extensions, context model fragments are included as *initial context* of the scenario, *interim*

[2] http://www.eclipse.org/modeling/emf/

context in the trigger part that should occur during test execution, and *final context* (in the *assert* part) that is checked to determine the success of the test.

Fig. 3 and Fig. 4 present two requirements (R2 and R3) of the vacuum cleaner. R2 states that when a living being is detected nearby the robot then it has to be alerted. R3 states that if a noise is detected in the room then the robot should send a predefined alert. Here the requirements with respect to the initial contexts are described using context fragments (model instances conforming to the context metamodel).

4 Generating test data

The context models provide a mechanism to describe environments, which could be later represented in a test execution setup (e.g., in a simulator). Requirements expressed as scenarios provide a way to check later these executions and search for misbehaviours. However, it is still missing how the different test data describing interesting environments and situations are acquired. By creating these test data by hand, some of the situations can be covered, but there is no guarantee that every stressful context is represented in the test data, or that the combinations of situations are completely checked.

We proposed an automated *test data generation* approach, which uses search-based techniques [9]. The key ingredients for the application of a search-based technique are the representation of the potential solutions and the definition of a fitness function. According to our approach, the solutions, i.e., the potential test data, will be generated as model instances conforming to the context metamodel. We use model transformation to manipulate the instances (adding/removing context elements and fragments to/from the candidate solutions) and a fitness function to guide the search. The fitness functions are based on the context requirements and on the test strategies (corresponding to test coverage metrics, see Section 5) that include the following:

- Creating the minimal number of context objects that are needed to test the satisfaction of a requirement. Furthermore, completing the minimal number of context objects with more and more additional objects from the context metamodel, potentially instantiating all types of the context metamodel in the set of test data.
- Combining context models related to different requirements to test whether the combined contexts will violate these requirements.
- Creating context models that are at the boundary or even violate systematically selected domain-specific constraints for robustness testing.

Fig. 5 presents the minimal test data generated by combining the initial context fragments belonging to requirements R2 and R3 as a stressful situation for the SUT. Although this example is simple, the generation of test data for a given test strategy is a non-trivial task due to the high number of context model elements, their type hierarchy and relations, and the constraints that have to be fulfilled/violated to get meaningful test data.

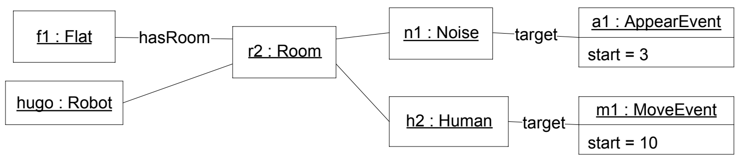

Fig. 5. Test data generated by combining the context fragments of requirements R2 and R3

A similar testing approach is presented in [10]: it utilizes so-called stakeholder soft-goals as quality functions and applies metaheuristic search-techniques to generate tests. However, there are essential differences in comparison with our framework. Their approach encodes the environment settings as a matrix of cells to represent object locations, while we propose the use of a high level metamodel to describe the possible contexts and use model instances conforming to this metamodel to represent concrete test data. We believe that this representation is more flexible and easy to use for domain experts. Another difference lies in the fitness calculation and execution of the tests. Their approach involves on-line testing: inserting the SUT into a test environment (made up of the generated test data) and calculating the fitness of potential test data based on the execution of the SUT. The test generation stops if the SUT violates the soft goal or there is no improvement observed after a number of generations and executions. On the contrary, we apply off-line test generation: we do not use the results of the execution of the SUT to assess the test data. In order to calculate the fitness of test data, we search for patterns that may trigger the formalized requirements and satisfy the test goals (coverage criteria). While our test data generation concept may not be adequate for the validation of every kind of soft goals, it does not depend on the domain and implementation of the SUT.

5 Evaluation of test results and coverage

To be able to include the generated test data in a simulator, a lot of simulator-specific information should be hard-coded in the test data (e.g., how is the position of an object encoded). To avoid this dependency from simulators, test data creation consists of two steps. First, so called *abstract test data* are generated and then a post-processing step produces *concrete test data* in a format dependent on the simulator. This way the formalized requirements use only general, abstract concepts and relationships (e.g., the robot is *near to* something). These are replaced in the post-processing step with compatible types defined in the simulator and the exact parameters (e.g., physical coordinates) are assigned.

After test data are generated, the following steps have to be executed. The simulator is fed with each generated test data (which describe the environment of the SUT) and then it processes the dynamic part of the test data (i.e., the evolution of the context, sending and receiving of messages etc.). During test execution detailed test traces are captured that record the events and actions of the SUT with their timing and also changes in the context. Finally, each captured test trace is checked against each of the scenarios to identify whether a scenario is triggered (checking the trigger part) and violation of any requirement is detected (in the "assert" part of the scenario). In this way, the proper handling of interleaving requirements can also be verified.

The test evaluation consists of two different aspects. On one hand, the matching of events and actions has to be checked. On the other hand, the actual context has also to be matched to the context fragment specified in the requirement.

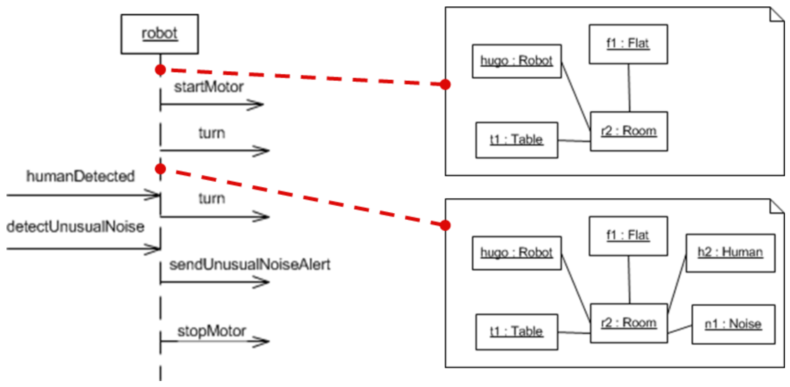

Fig. 6. Illustration of a test trace in case of testing the vacuum cleaner robot

Fig. 6 illustrates a test trace captured when the vacuum cleaner robot was tested using the test context presented in Fig. 5. According to this test data (the *AppearEvent* and *MoveEvent*) there was a change in the context: a human moved into the room and a noise appeared. When this test trace is evaluated with respect to requirements R2 and R3 (see in Fig. 3 and Fig. 4) the following can be derived: R3's initial context fragment can be matched with the second context, the message in its trigger part appears (there is a *detectUnusualNoise* message), and the action in its assert parts appears too, thus R3 is satisfied. In case of R2 the initial context fragment can be matched with the second context (when the distance between the position of the robot and the human satisfies the *nearBy* relation), the trigger part appears in the trace (there is a *humanDetected* message), but the action in the assert part does not appear in the trace. This way R2 is violated. This example demonstrated how the improper handling of stressful situations (i.e., interleaving of several potential scenarios) can be detected using a test context generated by combining the initial context fragments of different requirements. (Note that the matching of contexts and messages is obvious in this example due to the simplicity of the scenario, but in general it could be more complicated.)

Finally, when all the executed tests are evaluated, the quality of the testing should be assessed. One way to achieve this is to compute different coverage metrics. C*ontext related coverage metrics* measure what part of the context model has been covered during testing (e.g., whether there are objects, which have not been present in any test runs) and what combinations of initial context fragments from different requirements were covered. *Scenario related metrics* measure coverage on the scenarios, e.g., whether all scenarios have been triggered, or whether there were any violated requirements. *Robustness related metrics* measure the thoroughness of the generation of extreme contexts by considering the coverage of violated constraints and potential boundary values from the context model.

6 Summary and Future Work

This paper presented the challenges of testing autonomous systems and proposed a method to test the robustness and functional safety of the behaviour of the system's control. The proposed approach uses context modelling and graphical scenarios to capture the context and requirements of the system and automatically generates test data and test oracle to test complex or unforeseen situations. Once the test data is executed, the satisfaction of the requirements is checked and the coverage with respect to the context or scenarios is calculated. Currently we are working on the validation of the process using two real-world use cases, which involves refining the modelling notations, the test generation and the test evaluation algorithms.

Acknowledgements. This work was partially supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in frame of the R3-COP project.

References

1. Connelly, J., Hong, W., Mahoney, R., Sparrow, D.: Challenges in Autonomous System Development. In: Proc. of Performance Metrics for Intelligent Systems (PerMIS'06) Workshop (2006)
2. Waeselynck, H., et al.: Mobile Systems from a Validation Perspective: a Case Study, In: Proc. of 6th International Symposium on Parallel and Distributed Computing (ISPDC), IEEE Press (2007) doi: 10.1109/ISPDC.2007.37
3. Michelson, R. C.: Test and Evaluation for Fully Autonomous Micro Air Vehicles. In: ITEA Journal, 29.4, 367–374 (2008)
4. Scrapper, C., Balakirsky, S., Messina, E.: MOAST and USAR-Sim: A Combined Framework for the Development and Testing of Autonomous Systems. In: Proc. of SPIE 6230, 62301T (2006) doi: 10.1117/12.663898
5. Kelly, A., et al.: Toward Reliable Off Road Autonomous Vehicles Operating in Challenging Environments. In: The International Journal of Robotics Research, 25:5-6, 449–483 (2006) doi: 10.1177/0278364906065543
6. Weiss, L.G.: Autonomous Robots in the Fog of War. In: IEEE Spectrum, 48.8, 30–57 (2011) doi: 10.1109/MSPEC.2011.5960163
7. Baldauf, M., Dustdar, S., Rosenberg, F.: A Survey on Context-aware Systems. Int. J. Ad Hoc Ubiquitous Comput. 2:4, 263–277, (2007) doi: 10.1504/IJAHUC.2007.014070
8. Tenorth, M., Beetz, M.: KnowRob – Knowledge Processing for Autonomous Personal Robots. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, 4261–4266, IEEE Press, New York (2009) doi: 10.1109/IROS.2009.5354602
9. Szatmári, Z., Oláh, J., Majzik, I.: Ontology-Based Test Data Generation Using Metaheuristics. In: Proc. 8th Int. Conf. on Informatics in Control, Automation, and Robotics (ICINCO), Noordwijkerhout, The Netherlands (2011)
10. Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M.: Evolutionary Testing of Autonomous Software Agents. In: Proc. AAMAS (1) 521-528, (2009)