Auton Agent Multi-Agent Syst (2012) 25:260–283
DOI 10.1007/s10458-011-9175-4

Evolutionary testing of autonomous software agents

Cu D. Nguyen · Simon Miles · Anna Perini · Paolo Tonella · Mark Harman · Michael Luck

Published online: 20 May 2011

Abstract A system built in terms of autonomous software agents may require even greater correctness assurance than one that is merely reacting to the immediate control of its users. Agents make substantial decisions for themselves, so thorough testing is an important consideration. However, autonomy also makes testing harder; by their nature, autonomous agents may react in different ways to the same inputs over time, because, for instance they have changeable goals and knowledge. For this reason, we argue that testing of autonomous agents requires a procedure that caters for a wide range of test case contexts, and that can search for the most demanding of these test cases, even when they are not apparent to the agents' developers. In this paper, we address this problem, introducing and evaluating an approach to testing autonomous agents that uses evolutionary optimisation to generate demanding test cases. We propose a methodology to derive objective (fitness) functions that drive evolutionary algorithms, and evaluate the overall approach with two simulated autonomous agents. The obtained results show that our approach is effective in finding good test cases automatically.

C. D. Nguyen (✉) · A. Perini · P. Tonella
Fondazione Bruno Kessler, Via Sommarive, 18, 38050 Trento, Italy
e-mail: cunduy@fbk.eu

A. Perini
e-mail: perini@fbk.eu

P. Tonella
e-mail: tonella@fbk.eu

S. Miles · M. Luck
Department of Informatics, King's College London, Strand, London WC2R 2LS, UK

S. Miles
e-mail: simon.miles@kcl.ac.uk

M. Luck
e-mail: michael.luck@kcl.ac.uk

M. Harman
Department of Computer Science, University College London, Malet Place, London WC1E 6BT, UK
e-mail: m.harman@cs.ucl.ac.uk

Keywords Testing autonomous agents · Evolutionary testing · Quality requirements

1 Introduction

The concept of autonomy, as indicated by the origins of the word itself, where *auto* refers to *self* and *nomos* means *law*, refers to self-governance or freedom from external influence or authority. Such autonomy is key to establishing software agents as a distinctive class of computational system, and is also an enabling technology for building open, dynamic, and complex systems. Concerns about autonomy have appeared since the advent of artificial intelligence, because intelligent entities should be able, at least to some degree, to autonomously decide which actions to take. More recently, however, several different aspects related to agent autonomy, such as how to design and implement agents with different degrees and kinds of autonomy, have been discussed with greater intensity, such as by Hexmoor et al. [17].

As software agents with built-in autonomy are increasingly taking over control and management activities, such as in automated vehicles or e-commerce systems, testing these systems to make sure that they behave appropriately becomes crucial. The testing of traditional software systems, which have reactive (or input–output) style behaviour, is known to be non-trivial, but the testing of autonomous agents is even more challenging, because they have their own reasons for engaging in particular proactive behaviour. Thus, while a user might have certain concrete expectations of the behaviour of an agent, autonomy means that the agent's actual behaviour may differ from these yet remain appropriate; the same test input can give different results for different executions.

As an illustration of the issues involved in testing autonomous when compared to non-autonomous components, consider the following example. A component providing non–lossy compression functionality is tested according to two criteria: (1) the output of the compress function is smaller than the input targeted by the component (e.g., picture); (2) the output of the compress function, when given as input to the decompress function, returns the original input. In this simple non-autonomous testing example, the tester may not know the compression algorithm used by the component, and does not know the exact data that will be generated as output from the compress function. However, it can be expected that the output will remain the same for a given input over time; that is, *the run-time context in which the tests are performed does not change the test outcome*.

By comparison, the same tests can be applied by a client asking an *autonomous* agent to compress data, but there may be run-time variation in the actual behaviour of the agent. The agent may have its own internal goals and knowledge, both of which may change over time, and these can affect the result returned, if any. For example, the agent could choose different compression algorithms based on the resources available, or delegate the task to different subordinates over time, depending on which it currently considers to provide the best service. To properly test the autonomous agent thus requires more than the single test on the predictable component: it requires the same tests to be applied to the agent in a range of contexts. Ensuring that the range of contexts (in this example, the range of algorithms or subordinates available and chosen) tested against is adequate to declare the agent correctly functioning is, therefore, a hard but important task.

Existing work on agent testing dealing with agent autonomy (for example, Botía et al. [4] and Rodrigues et al. [27]), mainly use passive approaches, such as monitoring agent interactions and constraint enforcement. For Botía et al., agent interactions are observed to detect interaction problems such as missed communication. By contrast, Rodrigues et al. propose to exploit social conventions (i.e., norms or rules that prescribe permissions, obligations, and/or prohibitions of agents) in an open multiagent system for integration testing. During test execution these constraints will be checked, and violations reported.

In the context of agent autonomy, however, focussing only on violations (related to constraints, norms, and the like) is seldom sufficient. Confidence in the autonomous behaviour of an agent must be adequately established. We need to make sure that those autonomous agents under test are dependable. In addition, autonomous agents should be able to operate in open environments where different and even unpredictable circumstances may arise. Testing must also deal with this aspect.

In response to these concerns, in this paper we specify an evolutionary testing approach, guided by stakeholder quality criteria, to test autonomous agents. We describe the general testing procedure, evaluate it with a case study, and analyse the benefit it brings. Our approach can be outlined as follows. Stakeholder *requirements related to autonomy* (i.e. requirements that an autonomous agent may satisfy differently depending on its context) are transformed into *quality functions*. We then evolve increasingly demanding test cases using the quality functions as fitness measures, where the lower the quality the agent produces, the tougher we can infer the test case is, and the more likely the test case is to survive and be selected in the reproduction of test cases as the evolution progresses. By evolving steadily tougher test cases, we test the agent in a range of contexts, including those in which it is most vulnerable to poor performance. Since autonomous agents can behave differently under the same setting, statistical evaluation methods may be needed to properly evaluate each test case, for example, when the expected quality must be achieved with a high average probability.

It is worth noting that our technique focuses on the test case generation problem. The technique requires that the agents under test can run in an environment, therefore, we can apply it in late phases of software testing, e.g. integration, system, acceptance, or regression.

There are two primary motivations for our approach. First, we believe that quality functions—derived from requirements related to autonomy—can be used to evaluate autonomous agents as a means of building confidence in their behaviour, because meeting such requirements contributes to agent dependability. It is worth noting that our aim is to evaluate the *exhibited* performance of autonomous agents, not the mechanism underlying autonomy itself. Second, because it is automated, the use of *evolutionary* algorithms can result in thorough testing at low cost; a large number of challenging circumstances, generated by evolution, can be used to test agents, seeking to expose faults. That can complement other forms of testing, such as manual test case generation, which is often tiresome, error-prone and expensive.

In our previous work [24], we have applied evolutionary testing guided by a single *objective function* to a simulation of a cleaning agent, where the objective function dictated the fitness of the agents and was based on the requirements-derived quality functions. In this paper, we implement and evaluate evolutionary testing guided by multiple objective functions simultaneously. We aim to study the effectiveness of multi-objective evolutionary testing, under the observation that in reality undesirable behaviour arises only when agents (be they human or machine) are in tremendously difficult situations involving multiple factors. For example, a server ceases to operate quickly when it is suffering from overwhelming computation and denial-of-service attacks; similarly, humans may engage in illegal activities when they are under excessive work, family, financial and other pressures simultaneously. The results from our experiments with two different agents show that test cases found by multi-objective evolutionary testing, under some constraints, are indeed more effective than those found by single-objective evolutionary testing.

The remainder of the paper is structured as follows. Section 2 gives background notions about evolutionary testing and presents a motivating example. Section 3 introduces our approach, and Sect. 4 discusses our experiments and results obtained. Section 5 presents future work and Sect. 6 concludes.

2 Related work and motivating example

2.1 Background and related work

Agent autonomy can be classified into: *non-social* autonomy, or autonomy from the environment; and *social* autonomy, or autonomy as independence from or in collaboration with other autonomous agents [8]. In relation to non-social autonomy, agents need to be able to act proactively in an environment, on the basis of their local and contingent information. In relation to social autonomy, autonomy can be seen as *self-sufficiency*; that is, agents operate completely autonomously without any help or resources from other agents. Alternatively, autonomy can be found in collaboration in the sense of how autonomous it is when it operates in collaboration with others. In both cases, the agent behaviour cannot be completely determined based on current environmental settings due to internal constraints or representations produced by design, evolution, or learning, as well as the fact that agents do not simply receive inputs but perceive, interpret their environment and act in a goal-oriented way [8]. Autonomy, on the one hand, helps software agents deal with complex and open environments where changes (including those unknown at design time) can occur. On the other hand, it makes testing software agents a very challenging task.

Evolutionary testing [21,30] is inspired by evolutionary theory in biology, which emphasises natural selection, inheritance, and variability. In this view, individuals that are fitter have a higher chance of surviving and producing offspring, with favourable characteristics of individuals being inherited. In evolutionary testing, each test case is typically encoded as an individual in a population of candidate test cases. Then, in order to guide the evolution towards better test suites (sets of test cases), a fitness measure is defined as a heuristic approximation of the *distance* of these tests from achieving the testing goal (e.g., covering all statements or all branches in the program), where this distance should be minimised. Test cases with better fitness values have a higher chance of being selected for survival and reproduction. Moreover, mutation is applied to test cases during reproduction, thereby enhancing diversity, an important consideration in all evolutionary processes.

The key step in evolutionary testing is the transformation from testing objective to search problem. Specifically, through the fitness measure; different testing objectives give rise to different fitness definitions. For example, if the testing objective is to exercise code inside an *if* block, one can define a fitness function that gives lower values (considered as better) to test cases that are closer to making the conditions of the *if* statement true; the lowest (best) value is given to the test cases that make the conditions true, so that the code inside the *if* block is executed. Once a fitness measure has been defined, different optimisation search techniques, such as *local search, genetic algorithms, or particle swarms* [21] can be used to generate test cases aimed at optimising the fitness measure of the test cases.

The use of evolutionary search techniques for the automatic generation of test data has been receiving increasing interest from many researchers, reflecting a growing trend towards Search Based Software Engineering (SBSE) [16]. The SBSE approach is also bridging the technology transfer divide to industry. For example, Bühler and Wegener [7] applied evolutionary functional testing in two automatic systems: automatic parking and brake assistant systems for Daimler's Mercedes class cars. The systems investigated are automatic, not autonomous, but the results of evolutionary functional testing, which outperforms random and manual testing, show the potential of this technique.

More directly relevant to this work, Nguyen et al. [23] described the combination of evolutionary and mutation testing for testing autonomous software agents. For Nguyen et al., fitness is defined to be the mutant score, i.e. the number of mutants killed, where a mutant is a modified version of the original agent under test containing a single deliberately seeded fault. A mutant is said to be killed by a test input if the input causes the mutant to exhibit different behaviour to the original. Nguyen et al. approach the problem of testing for autonomy indirectly, by using constraints and the fact that while software agents are free to evolve, their behaviour must obey the *norms* and rules that govern the operation of the system in which agents are situated, or the constraints imposed on the behaviour. Test cases that kill more mutants are likely to reveal faults in the original agents, hence they have better fitness values.

Fulfilment (or violation) of norms and satisfaction (or otherwise) of requirements are directly comparable, as both define what *should* occur. However, both the aims and the approach taken here differ fundamentally from that of norm monitoring. Systems that detect the violation of norms provide tests (that just happen to occur at run-time) on agents: whenever a norm may be fulfilled or violated, this is determined and reported. However, the norm-monitoring approach does not attempt to rigorously test the agents involved. The only 'tests' performed are due to states of the system that happen to come about; the main concern is to check actual satisfaction of requirements (compliance with norms) at run-time. However, unless rigorous testing has occurred beforehand, the use of this approach alone could lead to a catastrophic violation, which would be detected but only dealt with after it has occurred (when it is too late). Our approach, on the other hand, attempts to cover a range of test cases *in preparation* for the agents under test to operate in a running system. The need for quality and variety of test cases is the most significant factor influencing our approach, as opposed to compliance in any single instance.

In the agent-oriented software engineering literature, a variety of research tackling different aspects of agent testing has been proposed. Coelho et al. [9] proposed a framework for unit testing of multi-agent systems based on the use of *Mock Agents*. Their work focuses on testing the roles of agents at the agent level. Mock agents that simulate real agents in communicating with the agent under test are implemented manually, each corresponding to one agent role. Sharing this inspiration from JUnit [14], Tiryaki et al. [29] proposed a test-driven multi-agent system development approach that supports iterative and incremental multi-agent system construction. A testing framework called SUnit, which is built on top of JUnit and Seagent [12], was developed to support the approach, allowing the writing of tests for agent behaviour and interactions between agents. Lam and Barber[19] proposed a semi-automated process for comprehending software agent behaviour. Their approach imitates what a human user, such as a tester, does in software comprehension: building and refining a knowledge base about the behaviour of agents, and using it to verify and explain the behaviour of agents at runtime. The ACLAnalyser [4] tool analyses runs on the JADE [28] platform, intercepting all messages exchanged among agents and storing them in a relational database. This approach exploits clustering techniques to build agent interaction graphs that support the detection of missed communication between agents that are expected to interact, unbalanced execution configurations, and overhead data exchanged between agents.

As software agent development emerges, testing software agents is receiving increased research attention. The above work focuses on agent interactions, which is reasonable, because agents communicate primarily through message passing. However, none of this previous work explicitly tackles agent autonomy, which is the objective of this paper.

Finally, Núñez et al. [25] introduced a formal framework to specify the behaviour of autonomous e-commerce agents. The desired behaviour of the agents under test is specified by means of a formalism, called *utility state machine*, that embodies users' preferences in its states. The operational traces of the agents under test are checked against these specifications in order to detect problems. Our work differs from Núñez et al. in that we investigate how to generate effective test cases based on the exhibited performance of the agents under test, not on their specification.

2.2 Motivating example

Throughout this paper we use an example, which we refer to as the *cleaner agent* example, to illustrate our testing methodology. Here, a cleaner agent (robot) is in charge of cleaning a square area at an airport, in which there can be obstacles, wastebins, waste, and charging stations located randomly or intentionally. In this environment, the agent needs to perform the following tasks autonomously.

- Locate important objects by exploring the environment.
- Look for waste and bring it to the nearest bin.
- Maintain battery life, with sufficient re-charging.
- Avoid obstacles by changing course when necessary.
- Exhibit *alacrity* by finding the shortest path to reach a specific location, while avoiding obstacles on the way.
- Exhibit *safety* by stopping gracefully if movement becomes impossible or if the battery level is too low.

Moreover, as *user requirements*, the agent must also exhibit good performance in terms of *robustness* and *efficiency*, by avoiding collision, consuming as little energy as possible, and collecting as much waste as possible.

These are the basic features of the cleaner agent. In addition, the agent can have global or local knowledge about the environment. In the former case, the agent is up-to-date with all changes, and there is no need for the exploratory task, while in the latter, the agent must sense and update its knowledge whenever it moves. Having global or local knowledge can thus influence the behaviour of the agent; we will discuss this further in our experiments.

3 Evolutionary testing of software agents

3.1 Evaluating software agents

Autonomous software agents differ from traditional software in that they have their own goals and operate in a self-motivated fashion. As a result, the response (also known as *test output*) we receive by stimulating an agent with an input can be different in different instances, even if the state of the agent (its capabilities) remains the same in those executions. For example, if an agent acquires different knowledge due to non-deterministic decision-making, this may result in different choices given the same subsequent situation (a point illustrated in the experiments presented later). In this case, a fixed oracle (also known as an evaluation *criterion*) to evaluate the output of an agent may have difficulty in dealing with such varied responses; instead, in testing software agents we need to have a kind of "soft" oracle. Moreover, in the evaluation of an autonomous agent, we are not only interested in the final results that arise from the behaviour of the agent, but also in its adaptation and its improvement during its lifetime, because we often expect agents to perform better (or at least in a constant manner) due to their autonomy, intelligence, and adaptivity. Therefore, the oracles used to evaluate an agent must do so based on the agent's progress over time and not just its fitness at a single instant.

In this paper, we propose to derive fitness measures from stakeholder requirements and use them as evaluation criteria for software agents. During test execution, agents under test are monitored; fitness values are, then, calculated based on monitored data and used to evaluate the agents.

3.1.1 Defining evaluation criteria

In requirements engineering, the importance of application domain stakeholder goals has long been recognised [20,22]. As such, the concept of *goal* has been considered central to a number of goal-oriented requirements engineering (GORE) approaches, such as described by Bresciani et al. [6] and by Dardenne et al. [10]. In GORE, *soft-goals* play a key role in representing non-functional or quality requirements, such as dependability, availability, and so forth, and thus denote important criteria for evaluating autonomy, as they allow us to assess the agent without dictating exactly what actions it must have taken. We propose to use stakeholder soft-goals as criteria for assessing the quality of autonomous agents, since satisfying quality criteria derived from these soft-goals is likely to indicate that the agents are reliable.

We illustrate this in relation to our motivating example of the *cleaner agent*. Figure 1, which adopts the notation proposed in the Tropos methodology [6], shows the goals of a specific stakeholder, the building manager, who decides to assign the goal of airport cleanliness to the cleaner agent. In this example, the agent must operate autonomously, with no human intervention, yet it must also be robust and efficient, as indicated in the two *soft-goals*, depicted as cloud shapes. Applying the proposed approach, these two soft-goals can be used as criteria for evaluating the quality of the cleaner agent. Thus the agent can be built with a given level of autonomy, using robustness and efficiency as two key quality criteria for evaluation. If the cleaner agent can perform tasks autonomously, but is not robust (for example, if it crashes), it is not ready to be deployed. Note, in addition, that in this example there are several sub-goals specified for the agent, such as *Recharging*, *Looking for charging station*, *Looking for waste*, and so on, as shown inside the balloon of the cleaner agent in Fig. 1, in which an analysis of goal decomposition and goal contribution is provided. Some of these contribute positively to the soft-goals related to the robustness and efficiency of the agent.

In order to assess whether an agent is good enough to be used in practice, a *threshold* is defined for each soft goal's quality function, such that an agent will not be put into use if any acceptability threshold cannot be reached—it is the function of the test oracle to check whether these thresholds are met. Apart from robustness, we can impose many other requirements related to autonomy on the cleaner agent: *stability, efficiency and safety*, for example. Stability demands that the agent should avoid dropping its goals too frequently. Efficiency requires the agent to finish cleaning an area after a specific amount of time, or to bring an amount of waste (for example, 2 kg) to the wastebins per hour. The safety requirement demands that the agent must switch to its 'safe mode' in undesirable circumstances, such as when its arms may be malfunctioning. In this paper, we concentrate on the sub-goals of the robustness soft-goal, *maintaining battery* and *avoiding obstacle*, to illustrate and evaluate our approach.

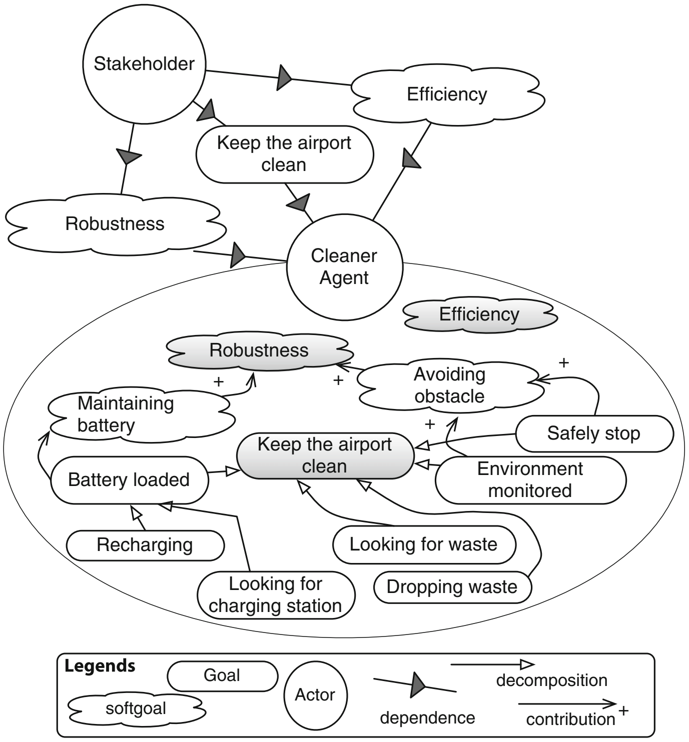


Fig. 1 Example of stakeholders' soft-goals

3.1.2 Analysing output data

In traditional software testing, we can set up some initial state, submit test inputs and then evaluate the outputs from the software. However, testing *autonomous* agents is more complex in that outputs arising from the same inputs can be different for different executions, so that it is important to evaluate not only the final results but also the *adaptation* of the agents under test over time. As a consequence, we need to collect sufficient and adequate data to properly evaluate an agent according to the defined criteria. Insufficient or inadequate test data can lead to incorrect conclusions.

To achieve this, we propose to monitor the behaviour of the agents under test during the test executions. Tentative monitoring can be very expensive, hence specifying *what* to monitor and *when* it needs to be done, prior to test execution is vital. However, since monitoring software agents is a topic that is beyond the scope of this paper, we assume that for each type of agent under test there is an adequate monitoring solution so that sufficient and adequate data is collected for testing.

Since the responses of an agent to a given input can be different in different instances, we propose to apply statistical methods to synthesise test output. Each test case execution must be repeated a number of times, and the statistical data obtained by monitoring compared to the correspondingly defined criteria, in order to give a verdict on the behaviour of such software agents. Taking the battery level of the cleaner agent as an example, in the same environment we may have to run the agent several (or a large number of) times in order to state whether the agents maintains its battery properly.

3.2 Evolutionary test case generation

Once evaluation criteria have been defined, following our approach presented above, we can use them to guide the generation of test cases automatically. The evolutionary testing methodology consists of the following top level steps:

1. *Representing stakeholder soft-goals as quality functions*. Relevant soft-goals that can be used to evaluate agent autonomy are transformed to evaluation criteria. We name these criteria *quality functions*; they refer to stakeholder satisfaction with respect to the agent under test. This transformation is domain specific and depends on the nature of the soft-goal. For example, a quality function that depicts the capability of the cleaner agent to maintain its battery may simply be a threshold with the constraint that the battery level must be greater than this threshold. However, this must be balanced against another quality function that captures the user expectations that efficiency should monotonically increase as a result of the agent's learning capability. The threshold constraint and the monotonic efficiency function are not necessarily always in conflict, but there may be situations in which satisfying one can make it harder to satisfy another; these are precisely those occasional interesting situations that make for challenging test cases.
2. *Evolutionary testing*. In order to generate varied tests with increasing levels of difficulty, we advocate the use of meta-heuristic search algorithms that have been used in other work on *search based software engineering* [16], and, more specifically, we advocate the use of evolutionary algorithms. Such algorithms are typically well-suited to complex multi-objective optimisation. The quality functions of interest are used as objective functions to guide the search towards generating more challenging test cases.

The evolutionary testing procedure is presented in Fig. 2. Its four steps are described as follows:

1. *Generate initial population*. A set of test cases is called a *population*. Each test case is an individual in the population, and represents a combination of states (i.e. values) of environmental inputs. The initial population can be generated randomly, or taken from existing test cases, created by testers.
2. *Execution and monitoring*. Test execution involves inserting the agents under test into the testing environment, made up of environmental inputs, so that they can operate (i.e. perform tasks or achieve goals). At the same time, a monitoring mechanism is needed to observe and record the behaviour of these agents. Multiple executions may be needed to be performed repeatedly (or in parallel) in order to provide sufficient data to statistically measure the fitness values used in the next steps.
3. *Collect observed data and calculate fitness values*. Cumulative data from all executions are used to calculate fitness values of selected test cases. The method for calculating fitness values depends on the stakeholder soft-goals of interest and the problem domain. Since these calculated fitness values provide some insight concerning improvement, if no improvement is observed after a specified number of generations (e.g., K generations), the test procedure stops. Otherwise Step 4 is invoked. The test process stops also when a maximum number of generations is reached, this maximum value is specified by the user.
4. *Reproduction*. Good individuals that have good fitness values are selected, and then the *crossover* operation (by which two individuals are combined to generate two more,

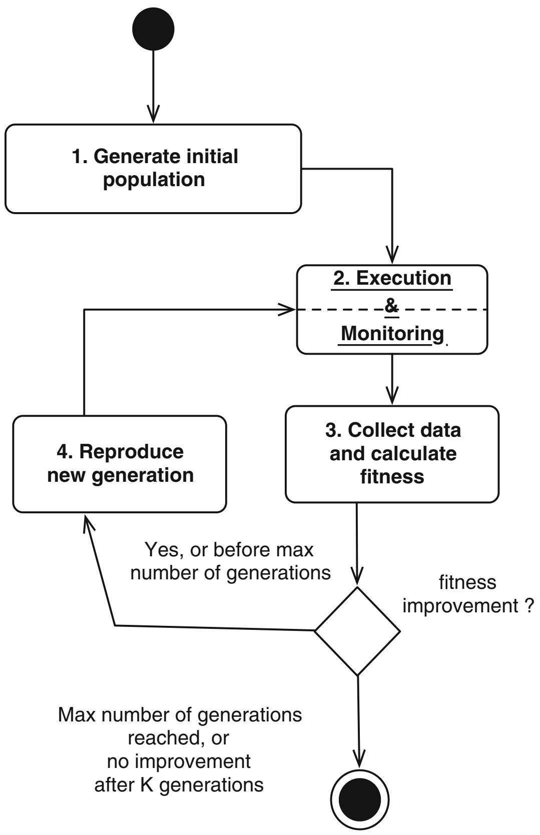

1. Generate initial population
2. Execution & Monitoring
3. Collect data and calculate fitness
4. Reproduce new generation
Yes, or before max number of generations
fitness improvement ?
Max number of generations reached, or no improvement after K generations

Fig. 2 Evolutionary testing procedure

inheriting the important material of their parents, as we describe later) is used to produce new offspring. Finally, *mutation* (by which some part of an individual may be changed to introduce some variation) is applied with a certain probability to some selected offspring. As with natural evolution, selection is biased in favour of fitter individuals.

When one fitness value (i.e., one soft-goal) is considered, a single-objective meta-heuristic algorithm is used to generate test cases. When more than one fitness value is considered, we can use a multi-objective meta-heuristic algorithm to guide the search for good test cases. For instance, the cleaner agent (see Fig. 1) can be evaluated using the *robustness* soft-goal or the *efficiency* soft-goal alone; but it can also be evaluated using the two soft-goals at the same time.

In the former case, when soft-goals are evaluated separately, this means that we test the autonomous agents with respect to the soft-goals separately. The expected outcomes are hard test cases that obstruct the agents under test from reaching the soft-goal under consideration. In the latter case, when multiple soft-goals are considered at once, we expect to obtain test cases that satisfy multiple hard-to-find conditions simultaneously. In the case of the cleaner, for example, we are interested in searching for test cases that require the agent to consume more power (subject to *efficiency*) and that cause it to be vulnerable to obstacles (subject to *robustness*). In such hard test cases, the cleaner may exhibit similar failures more quickly and frequently compared to those test cases that consider a single objective, or we may find other faults that are otherwise not revealed in the agents under test.

3.3 Encoding test inputs for evolutionary testing

In meta-heuristic (e.g. genetic) search algorithms, each individual is "genetically" encoded as a chromosome containing a set of genes. A gene is a set of elements, binary values (0, 1) for example. The task here is to produce successive generations of populations that give rise to fitter offspring, usually through specific operations. In this context, *crossover* for two individuals means that we cut the two corresponding chromosomes into parts, usually two, and exchange parts of one chromosome with those of the other to obtain offspring. *Mutation* means that a randomly-selected (or a small number of) gene element(s) of an offspring is forced to change, for example flipping 0 to 1.

In order to apply genetic search algorithms to generate test inputs, we must encode these test inputs such that we can easily perform crossover and mutation on them. Let us, first, examine the possible input space for autonomous agents.

Georgeff and Ingrand [15] present a minimal design of a reference architecture for BDI agents [5], that has been widely applied to build autonomous agents. In the architecture, agents perceive the outside world (from the agents' perspective) through a set of sensors, and make changes to the world through a set of effectors. Weyns et al. [31] complement the architectural picture of multiagent systems with a reference model for the environment, in which agents access the environment by employing either perception (sense and percept), action (make changes to the environment), or communication (send and receive messages). Though the two architectures appear to be slightly mismatched because of the communication element, they actually fit well with one another as agent communication involves environmental facilities—incoming (inbox) or outgoing (outbox) buffers—and receiving a message can be seen as perceiving the inbox, and sending one as placing it in the outbox. To this end, the outside world, from the perspective of an agent, consists of environmental artefacts and other agents, which are considered to be test inputs in testing software agents. Autonomous agents monitor these elements actively to detect and reply to events or changes in a timely fashion, or they can receive stimuli from these elements and react proactively.

In general, we can encode each environmental artefact by means of one *gene*, where each property of the artefact is encoded as one part of the gene. A test case, consisting of a set of all investigated artefacts, is encoded as a chromosome. For example, to apply evolutionary testing to test the cleaner agent, we consider each environment as a test case containing a set of objects, such as charging stations or wastebins. Since each of these objects has a specific location, by changing the locations of these objects we can obtain new test cases, and we can thus encode an environment (also known as a *test case*) as a set of locations. A genetic algorithm can then be used to search for good sets of locations; that is, to place the objects in specific locations, so that we can effectively pose demanding challenges to the cleaner agent.

It is worth noting that this method of encoding test inputs is generic, we can apply it to different agent systems to identify main elements of test cases, e.i artefacts and their properties. Then, for each artefact property, its domain-specific genetic encoding may be required. For example, we can apply the method to identify test cases composing of a set of objects, each has two properties: *location*, *size*—for testing a concrete agent. After that, we may encode *location* property using binary, integer, or real genetic encoding, depending on the domain of the agent.

4 Experiments

4.1 Testing scenario

In this section, we further elaborate the cleaner agent example introduced in Sect. 2.2, and build a simulation of a system composed of an artificial environment and this cleaner agent to evaluate the proposed approach. We describe, in detail, the functionality of the agent and the way we use soft-goals to guide test generation and, ultimately, evaluate the quality of the agent.

The artificial environment is a square area, A, in which there can be obstacles, wastebins, waste, and charging stations located randomly or intentionally. We define an *environmental setting* as a particular configuration of A, in which different numbers of obstacles, wastebins, waste, and charging stations are located at particular locations. Each such environmental setting comprises a test case, and different settings pose problems of different levels of difficulty for the cleaner agent: some might obstruct the movement of the agent or demand the agent to consume more power, while others might place wastebins in locations that make it easy for the agent to drop waste collected.

The simulation environment and the cleaner agent are implemented in Jadex [26], a software framework that facilitates the development of goal-oriented agents based on the belief-desire-intention (BDI) model. In fact, there is also a *cleaner agent* example provided with Jadex middleware, but its functionalities are much simpler in comparison to our implementation[1] in terms of the dynamism of the environment and the capabilities of the agent.

Our environment can be seen as a container in which there are observable objects, including obstacles, wastebins, charging stations and items of waste. At run-time, the properties of these objects (such as location) can change, and new objects of these kinds can appear in the environment. Figure 3 shows a snapshot of the environment at run-time. The name of the objects is placed beside it. In this environment there are one cleaner, two charging stations, one wastebin, a number of waste and obstacles.

In our experiments, two versions of the cleaner agent were implemented. The first version, which we refer to as *Simple Cleaner* or **s-Cleaner** for short, is provided with live up-to-date global knowledge about the environment, including the location of all the objects and their status. As a consequence, **s-Cleaner** can plan with global knowledge and does not need to explore the environment. The second version, which we refer to as *Exploring Cleaner* or **e-Cleaner**, is not provided with such information, and must itself explore the environment to sense other objects and detect waste. Thus, **e-Cleaner** may need to re-plan or change its course of movement when it encounters new objects. Both versions have a number of goals and associated plans, with goal deliberation based on conditions related to the current state of the agents (i.e. their location and charge) and the perceived state of other objects.

The detailed implementation of **e-Cleaner** as a BDI agent in Jadex can be summarized as follows. The agent has a set of beliefs regarding its location, profile (e.g. size, vision), and its knowledge about the surrounding environment. The agent has four root hard-goals: *maintain-battery*, *avoid-obstacle*, *cleanup*, *monitoring*; there are a number of sub goals that help achieving the root goals. In terms of agent plans, the agent has two plans *leastseenwalk* and *exploremap* that determine a place where to go; plan *moveto* prepares a trajectory and drives the agent to get to a target. Plan *stopandwaitforchanges* is triggered when the agent detects an obstacle; the agent may, then, wait for some seconds to see if the obstacle moves away and re-plan after that if continuing its original trajectory may lead to crashes. If a timeout

[1] Available at http://selab.fbk.eu/dnguyen/public/cleaner-agent.tgz.

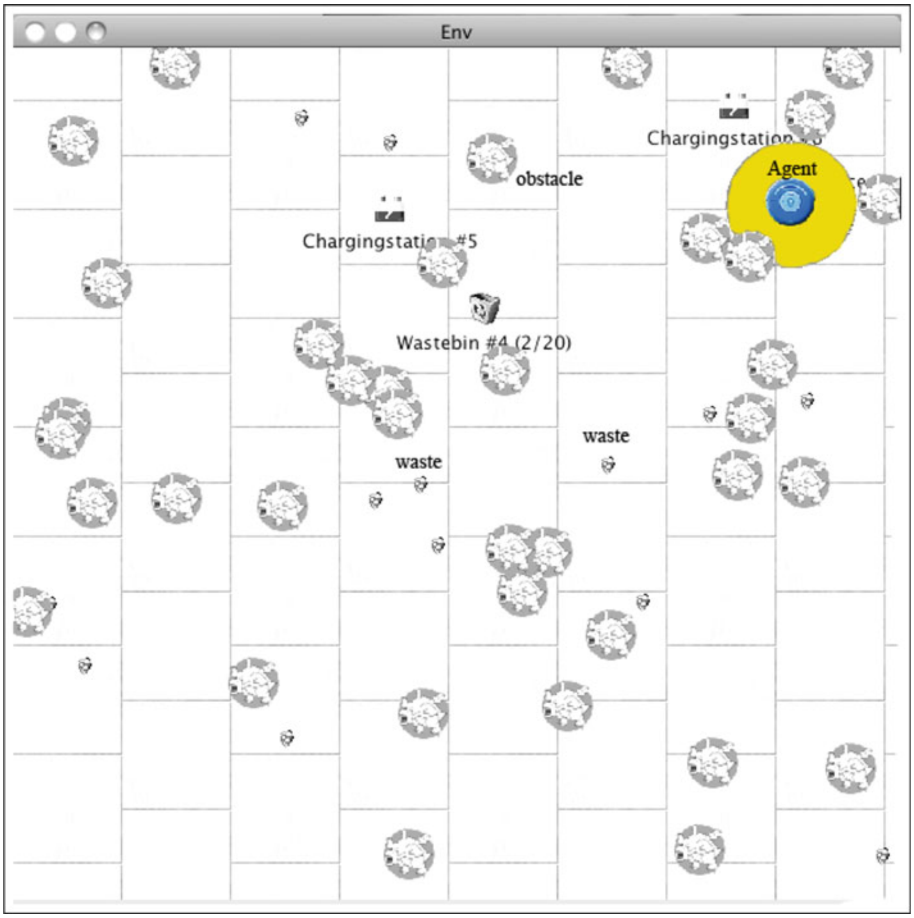


Fig. 3 A screenshot of the simulated environment at runtime

has occurred and no other possible path is found, plan *suspendforsafety* is triggered and the agent goes into 'sleep' mode. Finally, two other plans *loadbattery* and *cleanup* and their supplementary plans are responsible for recharging the agent's battery when it gets low and for cleaning up the discovered waste. Each plan has a trigger, which is a condition (or a set of conditions) or a reference to goals; the plan is executed when the required conditions are satisfied or when the referred goals are triggered. Detailed implementation of agent plans are in Java[2] code.

As mentioned in Sect. 3.1, we choose two soft-goals: *robustness* and *efficiency* to evaluate the quality of the cleaner agents. By analysing *robustness*, we can decompose from it two further goals: *maintaining-battery* and *avoiding-obstacles*. These sub-goals, and all other soft goals' sub-goals, should be taken into account when evaluating the quality of the cleaner agents. By so doing, each soft-goal can be used to derive a fitness function, which is then used by evolutionary testing to guide the generation of test cases.

[2] http://www.oracle.com/technetwork/java.

We apply the evolutionary generation steps introduced in Sect. 3.2 to generate test cases. More specifically, the NSGA-II algorithm with its ranking and genetic operations is used in our experiments. The algorithm was proposed by Deb et al. [11], it is a fast elitist multi-objective genetic algorithm that has been proved to be better in finding widely spread solutions (diversity if desirable) and offers better convergence to the optimum, in comparison to other algorithms. The algorithm uses a selection operator that creates a mating pool by combining the parent and offspring populations, and select the best solutions from them with respect to fitness and spread. In addition, the NSGA-II algorithm accepts encoding of real numbers, which fits well with our implementation of the location of objects in the testing environment.

In what follows, we detail the fitness functions and input encodings used in our experiments, and present the results obtained for the two agents, **s-Cleaner** and **e-Cleaner**.

4.2 Fitness functions

As described above, we have decomposed the *robustness* soft-goal into two sub-goals, *avoiding obstacles* and *maintaining battery*. In this section, we define fitness functions for these two goals.

The agent consumes battery power whenever it moves, picks up and drops waste; whenever the battery level dips below 20%, the agent moves to the nearest charging station to recharge the battery. In relation to battery power consumption, therefore, fitness is inversely proportional to the total power consumption of the agent throughout its lifetime, as follows:

$$f_{\text{power}} = 1/\text{Total power consumption}$$

The more power the agent consumes, the lower f_{power} will be. This will also be better from a testing perspective, since the test case forces the agent to consume more, potentially leading to fault revelation.

In relation to the *avoiding obstacles* soft-goal, the fitness function, f_{obs}, is similarly inversely proportional to the number of obstacles encountered during the lifetime of the agent. Encountering an obstacle means that it is close to the agent in the agent's direction of movement. The larger the number of obstacles encountered, the better f_{obs} will be for testing purposes.

$$f_{\text{obs}} = 1/\text{Number of obstacles encountered}$$

In preliminary work [24], we investigated the generation of test cases under which the agent is most vulnerable to obstacles by searching, based on the fitness function derived from the *avoiding obstacle* soft-goal, for environments in which the agent has a high chance of colliding with obstacles. In this paper we take that preliminary work further, extending it to consider both soft-goals, *avoiding obstacle* and *maintaining battery*, simultaneously. The search objective is to find test environments in which the cleaner agent is not only vulnerable to obstacles but also to consuming large amounts of battery power so that it may fail to maintain a sufficiently high battery level. In other words, we search for testing environments, subject to minimising both fitness functions, f_{power} and f_{obs}.

4.3 Test input encoding

A test case consists of obstacles, wastebins, waste, and charging stations. In our experiments, we control the quantity of these objects by fixing them a priori so that the test generation algorithm focuses only on the locations of the objects and converges faster. To apply our approach, we can relax this constraint and consider also object quantities as other variables for the evolutionary algorithm to find. We have done several pilot experiments with different object quantities and the results are, to some extent, similar to those reported in this paper. It is important, however, to notice that as the number of objects increases, the time for the test generation technique to find good test cases also increases.

In our experiments, there are two wastebins and two charging stations, while the numbers of obstacles and items of waste are higher (the precise quantity of these objects is discussed later in this paper). Consequently, one test case differs from others in the locations of the objects, and we must encode these locations to represent test cases in a suitable form for the evolutionary search algorithm.

We apply a real number encoding to specify the locations of objects. This allows an object to be placed anywhere in the environment. In this scheme, the location of an object consists of two coordinates x, y, where $0 \leq x \leq Width$ and $0 \leq y \leq Length$. In turn, *Width* and *Length* of the area to be cleaned are normalised to be 1. A test case, then, is encoded straightforwardly as a vector of real numbers as follows:

$$TC = < x_1, y_1, x_2, y_2, \ldots, x_N, y_N >$$

where (x_i, y_i) is the location of object i, and N is the number of objects of all types. The goal of the evolutionary testing we propose is thus to search for test cases (TCs), such that f_{power} and f_{obs} are optimal.

4.4 Experiments and results

During the development of the cleaner agents, before applying our evolution-based testing, we logged all the detected faults. In so doing, we discovered one subtle fault, which was found after many long human observations of the behaviour of the agents, that relates to the two goals of *maintaining-battery* and *avoiding-obstacles*. These two goals can be active at the same time, but when the battery level drops to too low a level, the agents favour the *maintaining-battery* goal regardless of the *avoiding-obstacles* goal, causing them to collide with obstacles if these appear on the paths of the agents. Figure 4 is an excerpt of agent code that contains the fault, lines 7–10: when the battery level of the agent is less than 3%, the goal *avoiding-obstacles* is inhibited by the goal *maintaining-battery*. This fault can be revealed under two simultaneous conditions: *the battery of the agents drops to too low a level* (specifically, 3%) so that the avoiding obstacle goal is ignored; and *there must be nearby obstacles in the path of the agents*.

In the following experiments we use the fault to evaluate the effectiveness of our proposed approach, evolutionary testing. We use the detection of the fault, and the value of the fitness functions introduced in the previous section, as a benchmark to assess our approach.

Regarding the parameters used with NSGA-II, we choose for all experiments the following values:

Population size	30
Max number of generation	100
Mutation probability	3%
Crossover probability	90%

4.4.1 Results with **s-Cleaner**

Using the NSGA-II algorithm [11], we performed evolutionary testing on **s-Cleaner** with three different fitness combinations: (i) f_{power} alone, (ii) f_{obs} alone, and (iii) both f_{power} and

```
 1  <maintaingoal name="maintainbattery" retry="true" recur="true" retrydelay="0">
 2          <deliberation cardinality="-1">
 3                  <inhibits ref="performlookforwaste" inhibit="when_in_process"/>
 4                  <inhibits ref="achievecleanup" inhibit="when_in_process"/>
 5                  <inhibits ref="achievepickupwaste" inhibit="when_in_process"/>
 6                  <inhibits ref="achievedropwaste" inhibit="when_in_process"/>
 7                  <!-- disable also the avoiding obstacle goal when battery is too low -->
 8                   <inhibits ref="avoidobstacles" inhibit="when_in_process">
 9                          $beliefbase.my_chargestate &lt; 0.03
10                   </inhibits>
11          </deliberation>
12          <!-- engage in actions when the state is below MINIMUM_BATTERY_CHARGE. -->
13          <maintaincondition>
14                  $beliefbase.my_chargestate > MyConstants.MINIMUM_BATTERY_CHARGE
15          </maintaincondition>
16          <!-- The goal is satisfied when the charge state is 1.0. -->
17          <targetcondition>
18                  $beliefbase.my_chargestate == 1.0
19          </targetcondition>
20  </maintaingoal>
```

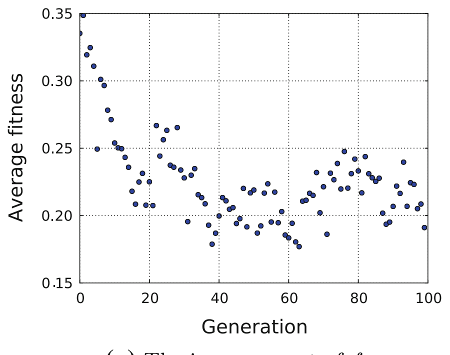
Fig. 4 Excerpt of agent code that contains a fault, lines 7–10. When the battery level of the agent is less than 3%, the goal *avoiding-obstacles* is inhibited

f_{obs}. In the first and second case only one fitness function is used to guide the evolutionary generation of test cases, but in the third case, both fitness functions are used. In this respect, the third case, which combines the two fitness functions, is a *multi-objective optimisation* problem.

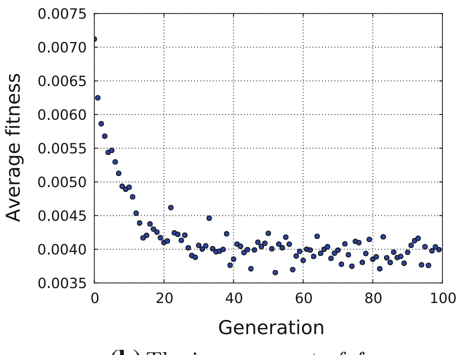
As described above, since **s-Cleaner** has global knowledge of the environment, it requires no exploration or sensing. Instead, at run-time, the agent decides what to do based on its battery level, number of items of waste collected, and number of items of waste that are not yet collected. Moreover, since the locations of all obstacles are known, **s-Cleaner** can plan its trajectory to avoid these obstacles. As a result, no collision should be expected. These experiments were therefore used to evaluate our evolutionary testing technique in terms of fitness values, in terms of how they are improved over time by the approach.

We performed these experiments with different settings of the environment in terms of quantity and types of object. For example, in one setting (which provides a clear perspective on the results), the quantity of the objects of each type in the environment is fixed — 4 obstacles, 4 waste items, 2 charging stations, and 2 wastebins — and the locations of the charging stations and wastebins are predefined. The starting location of the agent is at the bottom-left corner of the environment. (Other experimental settings for which we repeated the experiment gave similar results.) In this chosen setting, the aim of the evolutionary testing is thus to place items of waste and obstacles so that the fitness values reach their optima.

Figure 5 shows the improvement of fitness values over generations. At each generation we calculated the average fitness values of all individuals to see how the whole population evolves. In the sub-figures, the x axis represents time in terms of generations, while the y axis represents the averages of the fitness values. Overall we see that the average of fitness values in both sub-figures become smaller and smaller over time. This means that the fitness functions f_{obs} and f_{power} get better results over time or, alternatively, that the generated test cases are better and better in that they cause **s-Cleaner** to consume more power and increase the probability of encountering obstacles. When we ran the evolutionary testing guided solely by one fitness function—that is, by either f_{obs} or f_{power}—only one set of values improves, regardless of the other. Thus, the generated test cases satisfy only one condition at a time: either *causing* **s-Cleaner** *to consume more power* or *increasing the probability of encountering obstacles*.



(a) The improvement of f_{obs}



(b) The improvement of f_{power}

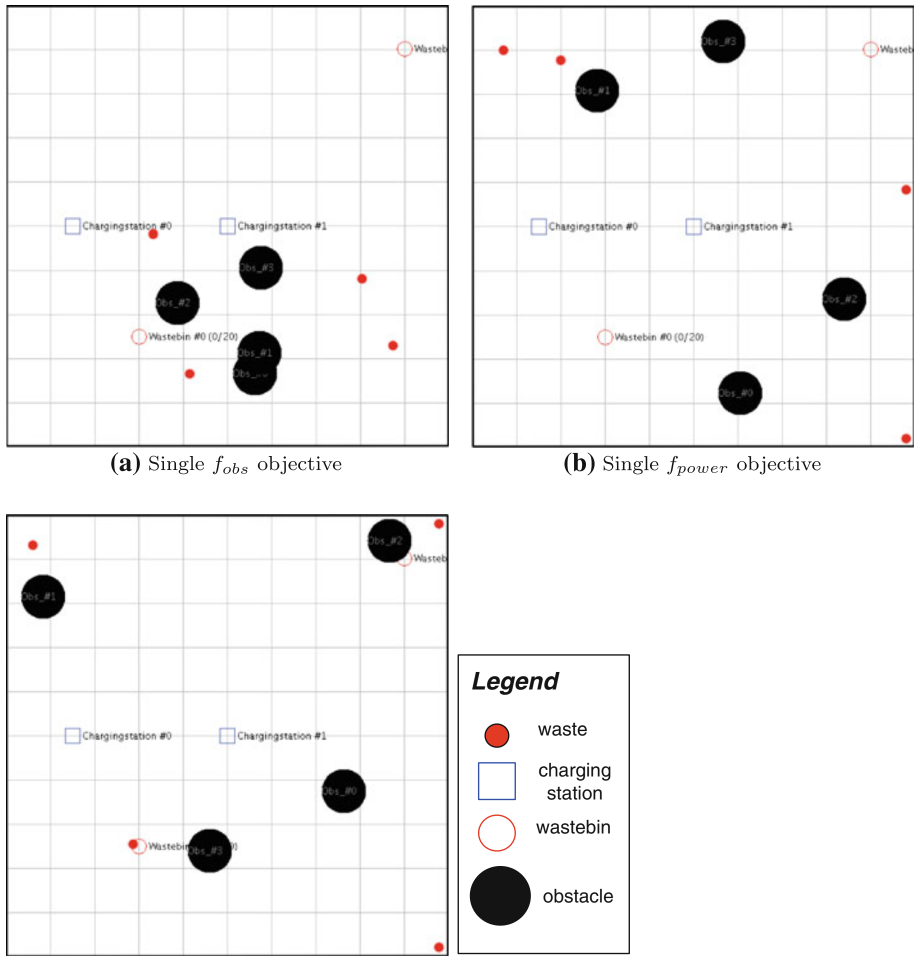
Fig. 5 The simultaneous improvement of fitness functions by multiple objective ET

Figure 6 illustrates three test cases found by evolutionary testing. Figure 6a, b are test cases found by single-objective evolutionary testing guided by f_{obs} and f_{power}, respectively. In Fig. 6a, obstacles (large filled circles) and waste items (small filled dots) are placed very close to each other, and they are also nearby two charging stations (squares) and the first wastebin (unfilled circle); in this setting, our execution trace showed that the agent encountered obstacles in almost every trajectory, indicating that the probability of crashing into an obstacle was high. In Fig. 6b, the waste items are placed in the far corners from the starting point of the agent, requiring the agent to consume more power to fulfil its cleaning goal. The locations of the obstacles are somewhat random. In Fig. 6c, the waste items are placed in the far corners, and the obstacles are placed on the paths the agent traverses to collect or drop waste, and to recharge its battery. In this way, the agent consumes a very high level of power and has a very high chance of colliding with obstacles.

In summary, the results obtained from this set of experiments provide evidence to support the claim that multi-objective evolutionary testing can search for test cases that satisfy multiple testing purposes, creating multiple conditions in which subtle faults can be revealed. The introduced fault manifests itself only when the battery level is very low *and* at that moment the agent encounters an obstacle. Thus, effective testing needs to synthesise separate stress conditions to automatically identify such fault revealing conjunctions of circumstances. This is why we believe that the agent testing problem naturally maps onto a multi-objective search based optimisation problem in the manner that we advocate herein. The chance of detecting the same fault when only one of these conditions holds regardless of the other (considering only a single objective) is lower because the probability that both conditions hold at the same time is naturally lower, too.

4.4.2 Results with **e-Cleaner**

In the previous results we have shown the capability of multi-objective evolutionary testing in searching for test cases that satisfy multiple conditions simultaneously. This is encouraging because these test cases have a good chance of finding subtle faults. In this section we investigate the application of multi-objective evolutionary testing on **e-Cleaner**. Since the **e-Cleaner** has only local knowledge of the area it explores, it may need to re-plan when discovering new objects such as obstacles in its way, unexpected items of waste, charging stations, and so on. Moreover, because the agent has no knowledge of its environment when



(a) Single f_{obs} objective **(b)** Single f_{power} objective

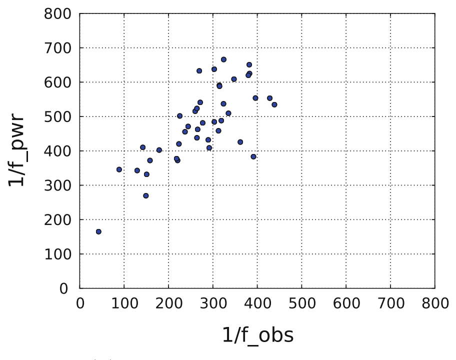
(c) Both f_{obs} and f_{power} objectives **(d)** Legend

Fig. 6 Examples of test cases found using single and multiple objective ET. The meaning of the objects in these figures is identical to those in Fig. 3. However, we make a new presentation to better visualise the test cases

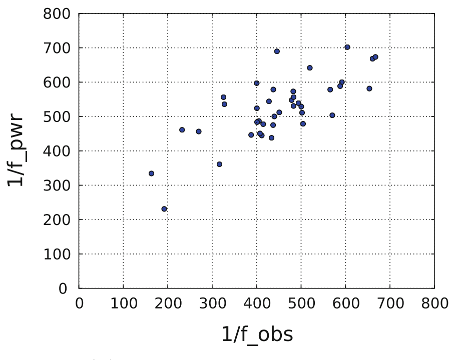
it starts, it must make random choices about where to explore. As a consequence, given the same test case, the agent acquires different knowledge and so can make different decisions in different runs, resulting in different behaviour. For each generated test case, therefore, we must repeat its execution several times in order to gather sufficient output data in order to correctly assess the *quality* of the test case.

In our experiments, each test case was executed repeatedly five times to measure the objectives f_{obs} and f_{power}. Each execution lasted 45 s[3] so that the agent has enough time to discover the environment and perform its tasks. Trial experiments revealed that with five executions and 45 s for each, the results of a test case converge and represent the overall result of the test case. To reduce testing time, five identical testing platforms were set up

[3] Execution time of the simulation should not be regarded as representative of execution time of a real cleaning robot, which is expected to be much longer for comparable behaviour.



(a) Inverses of fitness values of the 1^{st}generation



(b) Inverses of fitness values of the 70^{th}generation

Fig. 7 Improvement of test cases over generations (note that values increase as they represent the inverse of the fitness functions that decrease)

to undertake parallel executions. Similar to the experiments performed with *s-cleaner*, we also predefined the quantity of the objects in the environment, with 10 obstacles, 6 waste items, 2 charging stations at fixed locations, and 2 wastebins at fixed locations. We believe that these numbers are reasonable since these proportions of obstacles to wastebins provide evolutionary testing with adequate means to obstruct the movement of the agent. Within 45 s the agent can collect around 6 items of waste but must also recharge its battery once or twice.

In the same setting (that is, with the same number of objects of each type, and with the same initial populations), we executed single-objective evolutionary testing guided by f_{obs} and f_{power} individually, and then multi-objective evolutionary testing guided by both objectives. We compared the performance of these cases in terms of the number of instances in which the testing method caused the cleaner agent to collide with obstacles, where more collisions provide a better test case.

To illustrate how multi-objective evolutionary testing driven by f_{obs} and f_{power} improves test cases over time, consider Fig. 7, which shows two samples of the population at the 1st generation and at the 70th generation. The figure depicts a plot of $1/f_{\text{obs}}$ and $1/f_{\text{power}}$, so the larger the values, the better the population. In the 1st generation (Fig. 7a), the population is located in the lower-left area, while in the 70th generation (Fig. 7b), the population has improved significantly to the top-right area, where the fitness values are better.

Figure 8 shows the number of collisions over time. We use box-plots[4] to represent the statistical measure of collisions of every 20 consecutive generations, the 1st to the 20th , the 21st to the 40th , and so on. This provides an indication of the trend of the number of collisions over time. Looking at the plots in Fig. 8a, b, we can observe that the number of collisions caused by test cases found by f_{obs} alone (Fig. 8a) is high and stays relatively constant over time, while in the case of f_{power} (Fig. 8b), this is very small and without a clear pattern.

[4] A box-plot of a set of values depicts an effective visual representation of both central tendency and dispersion. It simultaneously shows the 25th, 50th (median), and 75th percentile values, along with the minimum and maximum values (called "whiskers"). The "box" of the box plot shows the middle or "most typical" 50% of the values, while the whiskers of the box plot show the more extreme values. The length of the whiskers indicates visually how extreme the outliers are.

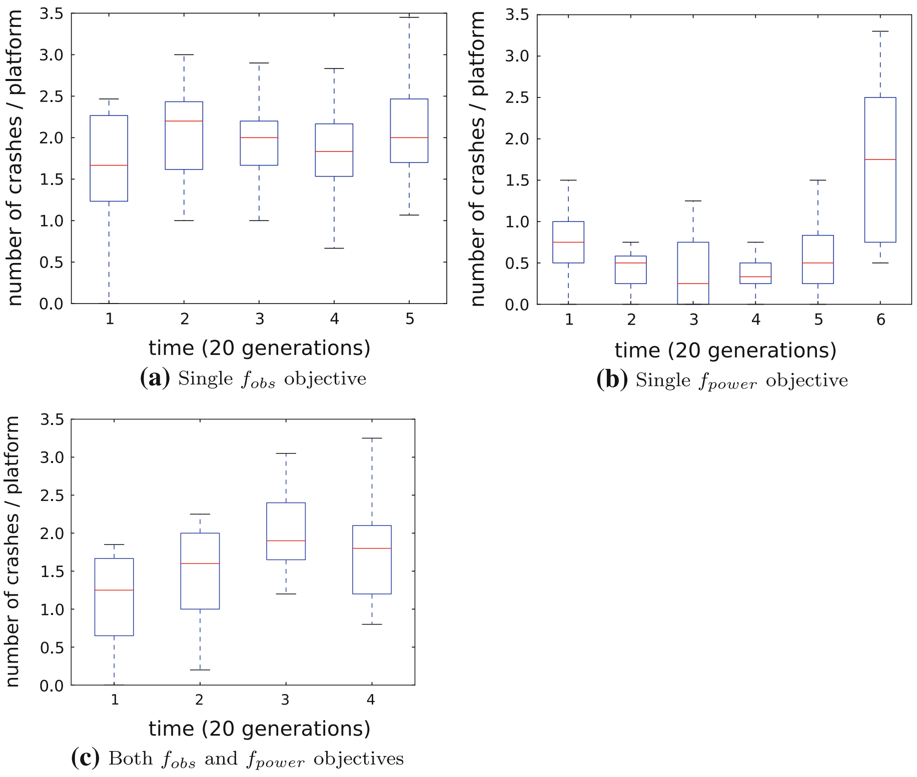


Fig. 8 Box-plots of statistical crash count per test execution (aka platform) over time

This can be explained by the fact that the fault is sensitive to the placement of obstacles; when evolutionary testing was guided by f_{obs} alone, it found test cases that facilitate obstacle encounters, and since the agent consumes power in any case, encountering more obstacles will increase the number of collisions. In contrast, when evolutionary testing was guided by f_{power} alone, it favoured test cases that led the agent to consume power regardless of obstacle encounters, so that the probability of the agent colliding with obstacles is low.

Finally in Fig. 8c, we observe an increased trend in the number of collisions. At the beginning this number is relatively low, perhaps because evolutionary testing must find test cases that compromise power consumption and obstacle avoidance. However, after some generations, multi-objective evolutionary testing performs better than single-objective evolutionary testing guided by f_{power}, and equivalently to single-objective evolutionary testing guided by f_{obs}. This leads us to conclude that multi-objective evolutionary testing can find test cases that satisfy multiple conditions simultaneously, even though its effectiveness may not always be better than well-chosen single-objective evolutionary testing. However, we do have a degree of guarantee that tough test cases can be found if they exist. If we had chosen only f_{power} in this experiment, we would have missed the opportunity to challenge the cleaner agent. In addition, we observe that the maximum number of collisions caused by multi-objective evolutionary testing is eventually higher than the maximum number of collisions caused by single f_{obs}, and always higher than the number caused by f_{power} alone.

5 Future work

There is a compelling case for much future research on SBSE for Agent Oriented Software Engineering. On the one hand agents' autonomous properties make their behaviour less predictable than traditional Software Engineering systems and this leads to emergent behaviour. This emergent property is one for which evolutionary optimisation algorithms are ideally suited; such algorithms seek to capture, control and exploit emergent behaviour to seek innovative solutions that may initially seem counter intuitive, yet which ultimately prove to be both optimal and valuable. One the other hand, the underlying model of computation used by multi-objective evolutionary computation is surprisingly similar to that used for Agent Oriented Software Engineering; both draw on the emergent properties of ensembles of individuals that work as a population to achieve an overall goal.

Future work will extend our techniques to test multiple agents. For example, we can define a fitness function related to power consumption as a sum of the power consumption of all the agents under test. There has been exciting recent work by White [32] on multi-objective SBSE for balancing functionality and power consumption on which this future research on search based agent testing might draw. As a result, evolutionary testing can also search for test cases that cause the agents, in total, to consume more energy, and this may lead to successful fault detection here, too. Following White [32], we may also seek to evolve agents that consume lower power and to explore the trade off between this objective and the other constraints and objectives set out for the agent's expected performance.

In this paper we focus on robustness of agent operation as an objective to be tested through search based optimisation. However, the approach we advocate is very general, based, as it is, on the very generic nature of SBSE. In almost any agent testing situation, there will be objectives and constraints that the agent must respect. It will often be possible to translate these into fitness functions. Having done so, the full power of search based optimisation can be brought to bear on the problem of developing and testing agents. Using the fitness functions to guide automated search, SBSE can be used both to optimise agent performance and, as we do here, to search for challenging environments that test agent behaviour against these objectives and constraints of interest.

It will also be interesting, in future work, to combine the search based optimisation of the agent's own BDI code with the search based optimisation of challenging test cases, using competitive co-evolution, an approach which has been proposed for several other applications of SBSE such as testing [1,18] and fault fixing [2,3]. Future work may use co-evolution to drive an 'arms race' that co-evolves ever better BDI agent code and, at the same time, even more demanding sets of test cases that test their behaviour thoroughly.

6 Conclusion

Autonomous software agents are goal-directed and self-motivated. Their behaviours are seldom determined from external perspectives. As a result, defining test cases to assess the quality of autonomous agents is challenging.

In this paper, we have proposed a systematic way of evaluating the quality of autonomous agents. First, stakeholder requirements are represented as quality measures. Autonomous agents need to meet a threshold under these measures in order to be accepted as ready for use by the stakeholders. Fitness functions, that represent testing objectives, are defined based on the quality functions, and guide our evolutionary test generation technique to generate test cases automatically. The longer the time for evolution is, the more challenging the evolved test cases are. Thus the autonomous agent is tested more and more extensively.

Fitness functions can be used individually, which guide the generation of test cases towards optimising a particular objective to bring the agents under test to a hard condition where faults might be revealed. On the other hand, they can be used together to put the agents under test into multiple hard conditions simultaneously, in which subtle faults can be revealed. These applications are complementary to each other; the final objective is to find as many faults and possible.

The significance of the test results presented in this paper is that evolutionary testing, following our approach, can test agents in a greater range of contexts than standard tests, thereby accounting for their autonomy to act differently in each such context. Testing an autonomous agent using a more traditional approach can only be effective if the range of contexts that influence the agent's behaviour is sufficiently limited that the developer can predict them all. However, when considering systems of any substantial complexity, of which a multi-agent system is certainly included, such a limited range is unlikely to occur. We can therefore argue that automated, search-based testing is essential to ensure complex system robustness and, as our tests show, evolutionary testing is an excellent candidate.

Multi-objective evolutionary testing can be guided by multiple fitness functions, including conflicting ones for which optimising one may constrain others (for example, maximising customer satisfaction and minimising expense). Multi-objective search techniques can find a broad range of test cases: some of them support of a particular fitness, while the others balance different ones. The former cases can be similar or comparable to those that can be found using single-objective techniques, while in the latter, the test cases may satisfy some balance conditions that lead to finding other faults. Therefore, multi-objective evolutionary testing has an advantage over single-objective evolutionary testing in terms of test execution (one multi-objective execution instead of multiple single-objective executions) and effectiveness.

When we have only partial or no control over the testing environment, for example when there is a third party agent participating in the test step, monitoring and statistical evaluation of the observed data can be used in testing. Evolutionary testing can actively search for the configurations of the part over which we have control, and because this part is affected during test execution by the rest of the system, an appropriate statistical evaluation may tell the whole picture. In the future, we will extend our experiments to validate that observation. Based on our results, we believe an evolutionary approach to testing autonomous agents is appropriate and effective, as it allows the agent to be thoroughly tested across a range of contexts that may affect its behaviour.

In addition, since our technique focuses on generating good test cases for the agents under test when they can be deployed on their environments, it can be applied in late phases of software testing, e.g. integration, system, acceptance, or regression. In regression testing, for example, we are interested in looking for failures or faulty behaviours that have occurred before as well as new ones when changes happen. (In fact, there are numerous research challenges in regression testing that have been surveyed in [13,33].) In the context of autonomous software agent, these issues become event more relevant because autonomous agents might expose new behaviours over time as results of learning and evolving. Fortunately, by nature, evolutionary algorithms used by our technique keep good individuals (i.e. test cases) with respect to fitness functions during evolution. As a result, test cases that are likely to be able to reveal faults are kept, and this fits the regression testing objective. We will further investigate this opportunity in our future work.

References

1. Adamopoulos, K., Harman, M., & Hierons, R. M. (2004). How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proceedings of the 2004 conference on Genetic and evolutionary computation (GECCO'04)* (Vol. 3103/2004, pp. 1338–1349). Seattle, Washington, USA: Springer. http://www.springerlink.com/content/77akkyyrjcdm400h/.
2. Arcuri, A., White, D. R., & Yao, X. (2008). Multi-objective improvement of software using co-evolution and smart seeding. In *Proceedings of the 7th international conference on Simulated evolution and learning (SEAL'08)* (pp. 61–70). Melbourne: Springer. doi:10.1007/978-3-540-89694-4_7.
3. Arcuri, A., & Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE congress on Evolutionary computation (CEC'08)* (pp. 162–168). Hongkong: IEEE. doi:10.1109/CEC.2008.4630793.
4. Botía, J. A., López-Acosta, A., & Gómez-Skarmeta, A. F. (2004). ACLAnalyser: A tool for debugging multi-agent systems. In *ECAI* (pp. 967–968). Valencia, Spain.
5. Bratman, M. E. (1987). *Intentions, plans and practical reason*. Cambridge, MA: Harvard University Press.
6. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., & Perini, A. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems, 8*(3), 203–236.
7. Bühler, O., & Wegener, J. (2008). Evolutionary functional testing. *Computers and Operations Research, 35*(10), 3144–3160. doi:10.1016/j.cor.2007.01.015.
8. Castelfranchi, C., & Falcone, R. (2003). Agent autonomy, chap.: From automaticity to autonomy: The frontier of artificial agents, pp. 103–136. In H. Hexmoor et al. (Eds.), *Proceedings of the 8th international working conference on Source code analysis and manipulation (SCAM'08)* (pp. 249–258). Beijing, China: IEEE. doi:10.1109/SCAM.2008.36.
9. Coelho, R., Kulesza, U., von Staa, A., & Lucena, C. (2006). Unit testing in multi-agent systems using mock agents and aspects. In *SELMAS'06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems* (pp. 83–90). New York, NY: ACM Press. doi:10.1145/1138063.1138079.
10. Dardenne, A., van Lamsweerde, A., & Fickas, S. (1993). Goal-directed requirements acquisition. *Science of Computer Programming, 20*(1–2), 3–50. citeseer.ist.psu.edu/dardenne93goaldirected.html.
11. Deb, K. D., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm : NSGA-II. *IEEE Transactions on Evolutionary Computation, 6*(2), 182–197. doi:10.1109/4235.996017.
12. Dikenelli, O., Erdur, R. C., & Gumus, O. (2005). Seagent: A platform for developing semantic web based multi agent systems. In *AAMAS'05: Proceedings of the fourth international joint conference on Autonomous Agents and Multiagent Systems* (pp. 1271–1272). New York, NY: ACM Press. doi:10.1145/1082473.1082728.
13. Engstrm, E., & Runeson, P. (2010). A qualitative survey of regression testing practices. In M. Ali Babar, M. Vierimaa, & M. Oivo (Eds.), *Product-focused software process improvement*. Lecture notes in Computer Science (Vol. 6156, pp. 3–16). Berlin: Springer.
14. Gamma, E., & Beck, K. (2000). JUnit: A Regression Testing Framework. http://www.junit.org.
15. Georgeff, M. P., & Ingrand, F. F. (1989). Decision-making in an embedded reasoning system. In *IJCAI* (pp. 972–978), Detroit, MI.
16. Harman, M. (2007). The current state and future of search based software engineering. In L. Briand & A. Wolf (Eds.), *IEEE international conference on Software Engineering (ICSE 2007), Future of Software Engineering* (pp. 342–357). Los Alamitos, CA: IEEE Computer Society Press.
17. Hexmoor, H., Castelfranchi, C., Falcone, R., Luck, M., D'Inverno, M., Munroe, S., Barber, K. S., Gamba, I., Martin, C. E., Braynov, S., Boella, G., Cohen, R., Fleming, M., Franklin, S., & McCauley, L. (2003). *Agent Autonomy*. Berlin: Springer.
18. Jia, Y., & Harman, M. (2008). Constructing subtle faults using higher order mutation testing. In *Proceedings of the 8th international working conference on Source Code Analysis and Manipulation (SCAM'08)* (pp. 249–258). Beijing: IEEE. doi:10.1109/SCAM.2008.36.
19. Lam, D. N., & Barber, K. S. (2005). *Programming multi-agent systems, Chap: Debugging agent behavior in an implemented agent system* (pp. 104–125). Berlin: Springer.
20. Lamsweerde, A. V. (2001). Goal-oriented requirements engineering: A guided tour. In *IEEE international conference on Requirements Engineering* (p. 249). Toronto, Canada. doi:10.1109/ISRE.2001.948567.
21. McMinn, P., & Holcombe, M. (2003). The state problem for evolutionary testing. In *Proceedings of the genetic and evolutionary computation conference* (pp. 2488–2498). Berlin: Springer.

22. Mylopoulos, J., Chung, L., & Yu, E. S. K. (1999). From object-oriented to goal-oriented requirements analysis. *Commun. ACM, 42*(1), 31–37.
23. Nguyen, C. D., Perini, A., & Tonella, P. (2007). Automated continuous testing of multi-agent systems. In *The fifth European workshop on Multi-agent systems*. Hammamet, Tunisia.
24. Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., & Luck, M. (2009). Evolutionary testing of autonomous software agents. In *8th International joint conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)* (pp. 521–528). Budapest, Hungary.
25. Núñez, M., Rodríguez, I., & Rubio, F. (2005). Specification and testing of autonomous agents in e-commerce systems. *Software Testing, Verification and Reliability, 15*(4), 211–233. doi:10.1002/stvr.v15:4.
26. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine, chap.: Multi-agent programming. Kluwer. http://vsis-www.informatik.uni-hamburg.de/projects/jadex.
27. Rodrigues, L. F., de Carvalho, G. R., de Barros Paes, R., & de Lucena, C. J. P. (2005). Towards an integration test architecture for open mas. In *1st workshop on Software Engineering for agent-oriented systems/SBES*. Uberlândia-MG, Brazil.
28. TILAB: Java agent development framework. http://jade.tilab.com (2002).
29. Tiryaki, A. M., Öztuna, S., Dikenelli, O., & Erdur, R. C. (2006). Sunit: A unit testing framework for test driven development of multi-agent systems. In *Agent-oriented software engineering VII, 7th international workshop, AOSE 2006* (pp. 156–173). Berlin: Springer.
30. Wegener, J. (2005). *Stochastic algorithms: Foundations and applications, chap.: Evolutionary testing techniques* (pp. 82–94). Berlin: Springer.
31. Weyns, D., Omicini, A., & Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems, 14*(1), 5–30.
32. White, D. R. (2009). *Genetic programming for low-resource systems*. Ph.D. thesis, University of York, UK. http://www.cs.york.ac.uk/~drw/papers/thesis/drwthesis.pdf.
33. Yoo, S., & Harman, M. (2010). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*. doi:10.1002/stvr.430.