Why testing autonomous agents is hard and what can be done about it

Simon Miles[1] Michael Winikoff[2] Stephen Cranefield[2] Cu D. Nguyen[3]
Anna Perini[3] Paolo Tonella[3] Mark Harman[4] Michael Luck[1]

Intuitively, autonomous agents are hard to test. Their autonomy and their flexible, context-aware behaviour implies unpredictability, and their social ability intuitively leads to unpredictable emergent behaviour. On the other hand, software agents are programs just like those designed and developed in an object-oriented, procedural fashion, or under any other approach, and just because something is intuitive does not mean it is necessarily correct.

In their separate respective groups, the authors of this position paper have been examining exactly why agents may be harder to test than the kinds of software we would expect to be developed without an agent-oriented approach, quantifying this difficulty, and/or proposing means to address it. This position paper summarises and expands on our groups' work for the purposes of discussion. While we are aware that there are other relevant approaches to tackling agent testing by other groups (including some presented at the Agent-Oriented Software Engineering workshop series), we do not attempt to give a state of the art survey here, but instead refer to a forthcoming survey of this area by Nguyen et al. [12].

1 Why agent testing is hard

Following common definitions, agents can be characterised as being *autonomous*, *social*, *reactive*, and *pro-active* [21]. They are also often *goal-oriented*, *deliberative* and *reactive*, interleaving deliberation and *plan* enactment with processing percepts from their environment. We discuss below why some of these characteristics mean that agents are hard to test. In doing so, we will take procedural, object-oriented program testing as a point of comparison, i.e. giving us a way of clarifying what 'hard' means, and use agent program examples written in a rough approximation of AgentSpeak(L) [14], as it exhibits many of the characteristics listed above.

Figure 1 shows an illustrative agent program. It comprises six plans. Each plan starts with a *triggering event*, which is the addition of a goal (`+!predicate`), addition of a belief (`+predicate`), or removal of a belief (`-predicate`). The next section shows the plan's *context*, i.e. what the agent must believe for the plan to be applicable at this time. The final section is the enactment of the plan, i.e. subgoals added (`+!predicate`) or removed (`-!predicate`) or actions performed (`predicate`). The first three plans give ways to achieve the goal of reaching the ground floor of a building, either by lift, stairs (one level at a time), or by already being on the ground, depending on context. The next two plans specify how the agent reacts to a fire alarm turning on or off, i.e. by escaping or by abandoning the escape. The final plan states that if the agent wishes to achieve the goal of having escaped, they need to get to the ground floor then exit. We now analyse why an agent is hard to test from a variety of angles.

Assumptions and Architecture Agent programs execute within an architecure which assumes and allows the characteristics of agents listed above. For example, it is assumed that the agent may be internally initiated with certain goals, i.e. it is pro-active. In our illustration, the agent may begin with the goal

[1]Department of Informatics, King's College London, UK
[2]Department of Information Science, University of Otago, New Zealand
[3]Center for Information Technology FBK-IRST, Trento, Italy
[4]Department of Computer Science, University College London, UK

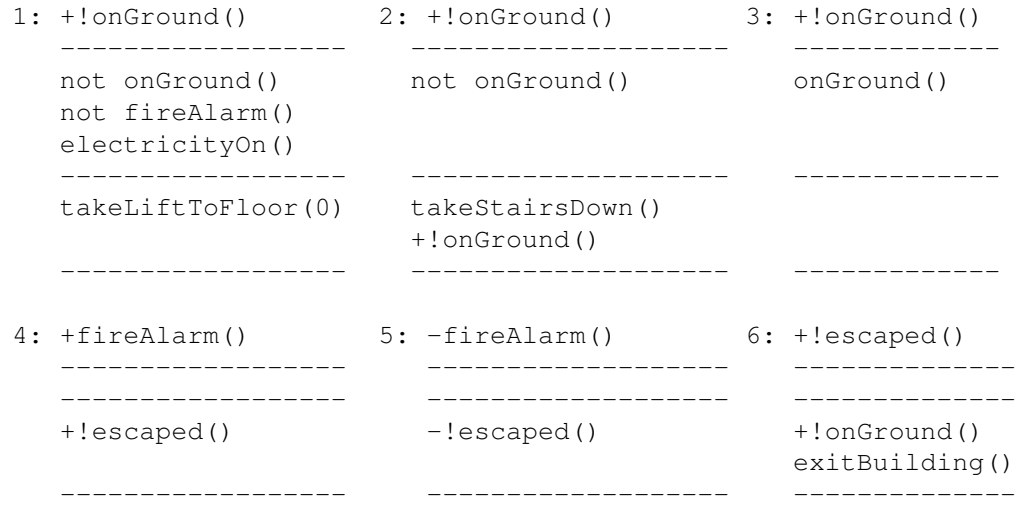

Figure 1: Example agent program for illustration

`!onGround()` and pro-actively pursue it. Similarly, it is assumed that the architecture will handle interleaving processing of incoming events (`fireAlarm()`) with acting towards existing goals. Finally, the architecture may handle removing sub-goals when their parent goal is removed, e.g. removing the goal of reaching the ground floor when the goal of trying to escape is removed. From an object-oriented perspective, this is equivalent to using an underlying framework which makes particular functionality easy to realise in a compact form, e.g. consider the functionality provided by an Enterprise JavaBeans container. As a side effect, the architecture makes the actual steps of execution rather opaque, as they depend on a complex mix of the architecture, the plans, and the environment (triggering events). In our example, the agent may only perform the action `takeStairsDown()` from the combined effect of an external event (`fireAlarm()`), the (possibly disparate) plans on how to handle this event and the `!escaped()` goal, and the fact way that the architecture processes goals taking the current context (`not onGround()`) into account. Establishing test cases is thus made more difficult, as it is hard to distinguish the behaviour which needs testing.

Frequently Branching Context-Based Behaviour If we considered an agent execution as a tree, we would see that choices between paths are made at very regular intervals (compared to that expected in a normal procedural program) to possibly extensive depths. This is because a goal or event can be pursued/handled by one of multiple plans, each applicable in a different context, and each plan can itself invoke subgoals. For example, if the agent has goal `!onGround()` and initially believes `not electricityOn()`, then it will take the stairs. At each level, it will reconsider its goal, including checking whether it has reached the ground. If during the journey, `electricityOn()` becomes true, the agent may take advantage of this and take the lift. Therefore, the agent program execution faces a series of somewhat interdependent choices. In recent work, Winikoff and Cranefield [20] quantified the number of possible paths in a typical agent program.

Reactivity and Concurrency Due to the reactivity of the agent, new threads of operation could be added to the agent's activity at regular points, caused by new inputs from the environment, e.g. `fireAlarm()`. This means that the choice of next action(s) made by an agent at every step depends not only on the plans applicable to the current goal/intention/event pursued, but also on the new inputs. Moreover, all threads followed (e.g. each intention held) generally share *all* of the same state, i.e. the agent's knowledge/belief base. This can be seen as equivalent complexity to that of concurrent OO programs with shared state. While the agent program execution may be entirely deterministic, i.e. we could in principle determine how actions would be interleaved for a given test input, the complexity of handling new events along with

existing intentions pursued in context-dependent ways means that it is effectively for a human test designer. For example, it is not apparent from the plan triggered by `+fireAlarm()` that the agent's choice of stairs or lift may be affected, nor does `-fireAlarm()` necessarily mean that the agent will cease to aim for the ground floor, as it may have been pursuing goal `!onGround()` before the fire alarm started. An arbitrarily interleaved or actually concurrent program is harder to test than a purely serial one.

Goal-Oriented Specifications Agent goals are comparable to OO method calls, in that both declare what should be done/achieved, while leaving it to a separate part of the code to actually perform the necessary steps. However, with a method call it is generally clear which part of code (the method implementation) will be executed on invocation, whereas a goal may trigger any of multiple different plans depending on context. This is reflected in the way in which they are respectively expressed. In an OO program, a call is most commonly expressed as a request to act, e.g. `compress` to execute a file compression method. On the other hand, a goal is often a state of the world which you want reached by whatever means are appropriate, e.g. `compressed`. Achieving a desired state of the world can be done in a wide range of ways, and may require no action (if the state already holds), or a number of actions. This makes it inherently harder for the debugging developer, and so for a developer to construct an appropriate test if starting from existing code. For example, to achieve `!onGround()`, the agent may start to head to the ground floor, but equally it may find it is already there and do nothing. A goal explicitly abstracts from what activity takes place, and so it can be harder to know what to test for in terms of unwanted side-effects of that activity.

Context-Dependent Failure Handling As with any form of interactive software, failures can occur in agents, and these need to be handled. For example, if the electricity fails while our agent is in the lift, it will need to find an alternative way to the ground floor. As this failure is handled by the agent, the handling is itself context-dependent, goal-oriented, potentially concurrent with other activity etc. Therefore, testing the possible branches an agent may follow in handling failures amplifies the testing problem. As well as showing that the number of test cases was large, Winikoff and Cranefield [20] demonstrated the dramatic increase due to consideration of such failure handling.

2 What can be done about it

Due to complexity described above, it is apparent that basic means of testing are inadequate for agent programs operating in environments exhibiting a relatively wide range of possible cases. For some applications, it may be possible to use domain expertise to limit the possible failures to a manageable subset of those which an arbitrary environment could present. For others, this is simply not enough, and other approaches must be considered.

First, we note that it could be possible to verify the abstract specification of the software in all or a bounded subset of cases, e.g. using model checking. Verification by itself does not provide assurance of the implementation's correctness, as the implementation may not follow the specification. We could guide verification by starting from tests on the implementation, and step-by-step, move to a more complete proof of correctness on a subset of the abstract specification, with assurance that the implementation tests and verification proofs provide the same results along the way. This idea is described in Section 2.1.

However, enumerating the appropriate test cases is itself not always possible. The complexity of the problem and size of the space of test cases means that determining what subtle fault may cause an agent program to fail is often non-trivial. In such cases, we realistically have to employ automatic search methods to discover those tests which will expose the agent's ability to achieve goals. In Section 2.2, we describe an evolution-based approach to testing autonomous agents whose goals are expressable as search-guiding quality functions.

2.1 Combined formal verification and testing

Formal verification and testing each have strengths and weaknesses [6], and so a natural question is whether they can be combined. The challenge is to find a way of using testing and formal verification together such

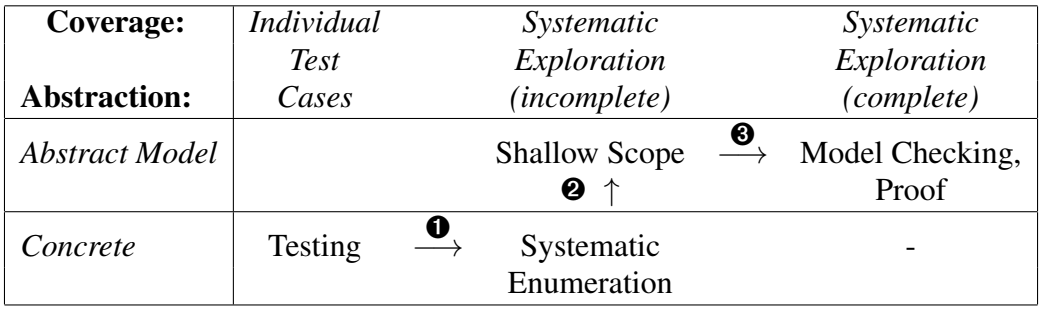

Figure 2: Approaches to Assurance: A two-dimensional taxonomy

that each compensates for the other's weaknesses, whilst retaining its strengths.

Abstractly, the idea of combining different techniques to obtain assurance of a system can be seen as a form of safety cases [1], where it has been argued that the assurance of a system should be done by providing *direct* evidence linking a system with the required properties, expressed in terms of the "real world" [8, 16]. There have been others who have argued for the combination of testing and proving (e.g. [10, 15, 5, 23, 13]). Space precludes a detailed discussion of these approaches, but we note that none provide a detailed and clear way of combining testing and proving in the way that is required.

In the remainder of this section, we sketch an approach for combining testing and formal verification. The key idea of the approach is that in order to link testing and formal verification we build a "bridge" using intermediate approaches. The "bridge" consists of a number of "steps" where we apply two approaches that differ in only one aspect. For example, given two different models (e.g. an abstract model and an implementation), we would subject them to the *same* assurance approach, in order to look for differences between the models. Since we use the same assurance approach, we can conclude that any difference found is actually due to the differences between the models. Alternatively, given the same model, we might subject it to two different assurance techniques.

In order for this approach to work, we need to define a number of assurance techniques that are "intermediate", i.e. are between formal verification and testing. Furthermore, these approaches need to provide a one-step-at-a-time "staircase" that links formal verification and testing.

We define these intermediate techniques using a two-dimensional taxonomy (see Figure 2, ignore the arrows for now). The first dimension is *abstraction*: we distinguish between executable models ("concrete") and non-executable ("abstract") models. The second dimension is the *coverage* of the method: is it covering only particular selected points in the space of inputs/behaviours? ("individual test cases"); all points within a sub-space? ("incomplete systematic exploration"); or the entire space? ("complete systematic exploration").

An example of how we might use these intermediate assurance techniques to build a stepwise bridge between testing and formal verification is the following (the numbers correspond to the numbered arrows in Figure 2):

❶ We subject the implementation to testing (with selected test cases) and systematic enumeration (of a subset of the input or behaviour space), i.e. same model, different techniques. If we find the same issues with both approaches, then this gives us some confidence that the selected test cases provide good coverage. If the systematic enumeration finds additional issues, then this is evidence that the test cases are insufficient. On the other hand, if the selected tests cases find errors that the systematic enumeration misses then this is evidence that the scope within which enumeration is performed is too limited.

❷ We apply the same technique (systematic generation of test cases within a limited scope) with two different models (concrete and abstract). Finding the same issues in both cases gives us some confidence that the models are equivalent.

❸ We apply systematic generation within a limited scope and formal verification (complete systematic

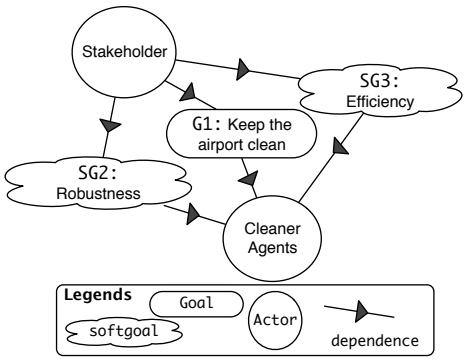

Figure 3: Example of stakeholders' soft goals

exploration) to the same (usually abstract) model. If the same issues are found by both techniques, this provides evidence that the limited-scope systematic enumeration did not miss anything.

Taken together, these steps build a bridge that links proving and testing. At each step along the way we change only one thing which allows for conclusions to be drawn from differences, or lack thereof.

A more detailed discussion of this approach, along with a small case study, can be found in [17]. It does need to be noted that the approach is still immature and further development is needed. A key question is how to quantify the level of confidence that can be gained from a given step (e.g. doing both individual tests and systematic exploration on the same model).

2.2 Evolutionary and search-based testing

For agent-based programs operating within complex environments, the size of possible test cases can be too large to feasibly cover every distinct possible input sequence. Due to the complexity described in Section 1, it is often difficult to discern which test cases thoroughly test an agent. To be confident of the agent's reliability, we therefore require 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.

We (Nguyen et al.) addressed this problem in a prior publication [11] by introducing and evaluating an approach to testing autonomous agents that uses evolutionary optimisation to generate demanding test cases. We argue that quality functions derived from requirements can be used to evaluate autonomous agents as a means of building confidence in their behaviour, because meeting such requirements contributes to agent dependability. Because it is automated, the use of evolutionary algorithms can result in more thorough testing at comparatively lower cost than other forms of testing, such as manual test case generation, which is tiresome, error-prone and expensive. That is, large numbers of challenging circumstances, generated by evolution, can be used to test agents, with each test case seeking to expose any possible faulty behaviour.

The evolutionary testing methodology consists of the following top level steps. First, relevant soft goals that can be used to evaluate agent autonomy are transformed into evaluation criteria, i.e. *quality functions*. This transformation is domain specific and depends on the nature of the soft goal. Then, in order to generate varied tests with increasing level of difficulty, we advocate the use of meta-heuristic search algorithms that have been used in other work on search based software engineering [7], 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.

As an example of an applicable domain, Figure 3 illustrates the goals of a specific stakeholder in an airport organization, namely the building manager, who decides to assign her goal of keeping the airport clean to cleaner agents. In this example, the agents must operate autonomously, so that no human intervention is required. The agents must be robust and efficient as stated by the two stakeholder soft-goals, depicted as

two cloud shapes. Applying the proposed approach, these two soft-goals can be used as criteria to evaluate the quality of the cleaner agents: the agents can be built with a given level of autonomy. Robustness and efficiency are two key quality criteria to evaluate them. If cleaner agents can perform tasks autonomously, but are not robust, e.g. they crash, they cannot gain the trust necessary to be deployed.

Test cases are encodings of the environments as strings comprising multiple 'chromosomes'. An initial test case population is generated, the tests are executed to determine their difficulty (the agent's lack of ability to achieve its goals), the fitness value of each test case compared, and the more successful (harder) test cases reproduced and randomly mutated. Ultimately, this automatically leads towards a rich set of challenging test cases which the human developers may not have considered. If the agent under test is deemed from these tests to be inadequate, it can then be improved to better meet the test cases and the evolutionary testing applied again.

2.3 Other Approaches

The above approaches can be seen as complementary, in that a subtle (failed) test case found through evolution could then be transformed to reach a proof that the abstract model of the (repaired) agent under test will not fail that case or a family of cases in which it is included.

Another way to approach the problem is to try to mitigate the need for testing in advance. We should aim to use notations and design/implementation approaches that make certain classes of errors impossible to make. An analogy is the use of automated garbage collection, rather than manual memory management. In the context of multi-agent systems, one place where we currently use 'manual' methods is developing agent interaction in terms of sending and receiving (asynchronous) messages. This can be argued to be analogous to using a 'goto' statement (but, even worse, involves concurrency), and is error-prone [3]. Instead, we could work to develop design and programming constructs for agent interaction that work at a higher level (such as social commitments [22, 9, 19], or some form of shared goals [2]). A key property is that we aim to make certain errors impossible to make. For example, under certain conditions interactions designed in terms of social commitments are robust to agents perceiving messages having arrived in different orders [18, 19, 4].

References

[1] P. Bishop, R. Bloomfield, and S. Guerra. The future of goal-based assurance cases. In *Proceedings of Workshop on Assurance Cases. Supplemental Volume of the 2004 International Conference on Dependable Systems and Networks*, pages 390–395, 2004.

[2] Christopher Cheong and Michael Winikoff. Hermes: Designing flexible and robust agent interactions. In Virginia Dignum, editor, *Multi-Agent Systems – Semantics and Dynamics of Organizational Models*, chapter 5, pages 105–139. IGI, 2009.

[3] Amit K. Chopra and Munindar P. Singh. An architecture for multiagent systems: An approach based on commitments. In *Workshop on Programming Multiagent Systems (ProMAS)*, 2009.

[4] Amit K. Chopra and Munindar P. Singh. Multiagent commitment alignment. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 937–944, 2009.

[5] David L. Dill. What's between simulation and formal verification? (extended abstract). In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 328–329, New York, NY, USA, 1998. ACM.

[6] Jimin Gao, Mats Heimdahl, David Owen, and Tim Menzies. On the distribution of property violations in formal models: An initial study. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. IEEE Computer Society, 2006.

[7] Mark Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *IEEE International Conference on Software Engineering (ICSE 2007), Future*

of Software Engineering, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society.

[8] Daniel Jackson. A direct path to dependable software. *CACM*, 52(4):78–88, April 2009.

[9] Rob Kremer and Roberto Flores. Using a performative subsumption lattice to support commitment-based conversations. In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael Wooldridge, editors, *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 114–121. ACM Press, 2005.

[10] M. Lowry, M. Boyd, and D. Kulkami. Towards a theory for integration of mathematical verification and empirical testing. In *13th IEEE International Conference on Automated Software Engineering*, pages 322–331, 1998.

[11] Cu D. Nguyen, Simon Miles, Anna Perini, Paolo Tonella, Mark Harman, and Michael Luck. Evolutionary testing of autonomous software agents. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 521–528, Budapest, Hungary, May 2009. IFAAMAS.

[12] Cu D. Nguyen, Anna Perini, Carole Bernon, Juan Pavn, and John Thangarajah. Testing in multi-agent systems. In *Proceedings of the Tenth International Workshop on Agent Oriented Software Engineering (AOSE 2009)*, 2009.

[13] David R. Owen. *Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy*. PhD thesis, West Virginia University, Lane Department of Computer Science and Electrical Engineering, 2007.

[14] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, LNAI 1038, 1996.

[15] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, dec 1985.

[16] John Rushby. A safety-case approach for certifying adaptive systems. In *AIAA Infotech@Aerospace Conference*, apr 2009.

[17] Michael Winikoff. *Assurance of Agent Systems: What Role should Formal Verification play?*, chapter 12, pages 353–383.

[18] Michael Winikoff. Implementing flexible and robust agent interactions using distributed commitment machines. *Multiagent and Grid Systems*, 2(4):365–381, 2006.

[19] Michael Winikoff. Implementing commitment-based interactions. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 873–880, 2007.

[20] Michael Winikoff and Stephen Cranefield. On the testability of bdi agents. In *Proceedings of the Eighth European Workshop on Multi-Agent Systems (EUMAS 2010)*, 2010.

[21] Michael Wooldridge and Nick Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[22] Pınar Yolum and Munindar P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534, 2002.

[23] M. Young and R.N. Taylor. Rethinking the taxonomy of fault detection techniques. In *11th International Conference on Software Engineering*, pages 52–63, 1989.